Sailfish: Towards Improving the Latency of DAG-based BFT

Nibesh Shrestha
n.shrestha@supraoracles.com
Supra Research

Rohan Shrothrium
rohan.shrothrium@duke.edu
Duke University

Aniket Kate
aniket@purdue.edu
Purdue University / Supra Research

Kartik Nayak
kartik@cs.duke.edu
Duke University

ABSTRACT

The traditional leader-based BFT protocols often lead to unbalanced work distribution among participating parties, with a single leader carrying out the majority of the tasks. Recently, Directed Acyclic Graph (DAG) based BFT protocols have emerged as a solution to balance consensus efforts across parties, typically resulting in higher throughput compared to traditional protocols.

However, 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. Furthermore, we extend Sailfish to Multi-leader Sailfish, which facilitates multiple leaders within a single round and commits all leader vertices in a round with a latency of one RBC round plus 1δ. Through experimental evaluation, we demonstrate that our protocols achieve significantly better latency compared to state-of-the-art DAG-based protocols, with slightly better throughput.

1 INTRODUCTION

Byzantine fault-tolerant state machine replication (BFT SMR) protocols form the core underpinning for blockchains. At a high level, a BFT-SMR enables 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 in practice, there has been a lot of recent progress in improving the key efficiency metrics namely, latency, communication complexity, and throughput under various network conditions. Assuming the network is partially synchronous, existing SMR protocols can commit with a latency overhead of 3δ (where δ represents the actual network delay) [8, 9, 15] and also achieve linear communication complexity [23, 34] 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 uneven scheduling of work among the parties. While the leader is sending a proposal, the other parties' processors and their network is not used leading to uneven resource usage across parties. Second, in typical leader-based protocols progress stops if the leader fails and until it is replaced. Several techniques proposed in the literature can potentially mitigate these concerns. These include the use of erasure coding techniques [2, 26] or the data availability committees [17, 18, 32] to disseminate the data more efficiently.

Recently, a novel approach known as DAG-based BFT has emerged [4, 12, 19, 21, 22, 29, 30]. 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 [13, 29]. However, existing DAG-based protocols incur a high latency compared to their "leader-heavy" counterparts [9, 15, 20, 23, 34]. Is high latency inherent for such DAG-based protocols? This paper works towards addressing this question.

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) [7] to disseminate the vertices; this ensures non-equivocation and guaranteed delivery [21, 28, 29]. Depending on whether a communication-optimal [14] or latency-optimal [1] 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 [29, 30], Shoal [28], Cordial miners [22] and Mysticeti [3]. 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 that share a round with previous leader require a minimum of 4 RBCs.

A recent work, Shoal introduced a "pseudo-pipelining" technique to reduce the commit latency of non-leader vertices by employing 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. When Bullshark fails to commit, Shoal requires an extra two RBCs to initiate a new instance. Additionally, when dealing with alternating adversarial and honest leaders, both Bullshark and Shoal struggle to make progress, compromising Shoal's ability to ensure a leader vertex in every round. Shoal also inherits a latency of 2 RBCs for committing the leader vertex.

Cordial Miners recently improved the latency of DAG-based BFT protocols by using best-effort broadcast (BEB) instead of RBC. They achieved a commit latency of 3δ for leader vertices and 6δ for non-leader vertices that coincide with the leader round, with the leader round repeating every 3 rounds. Building on this, Mysticeti [3] adds support to accommodate multiple leaders within a single round. Despite these improvements, both protocols maintain a communication complexity of $O(n^4)$ per round in the presence of Byzantine failures (where n is the number of parties) and lack modularity. In contrast, DAG-based protocols that utilize RBC can offer a range of communication complexities and commit latencies by leveraging existing RBCs from the literature.

In this work, we concentrate on modular DAG-based BFT protocols that use RBC. To the best of our knowledge, existing modular DAG-based protocols do not truly support a leader vertex in every (RBC) round and necessitate a minimum of 2 RBCs to commit the leader vertex. To address these concerns, we 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 [1], Sailfish incurs only 3δ to commit the leader vertex, effectively matching the best latency achieved by classical approaches [9] and DAG-based BFT not relying on RBC [3, 22]. When using a communication-optimal RBC [14], our protocol incurs 5δ latency to commit the leader vertex. Compared to the state-of-the-art DAG-based BFT that rely on RBC, Sailfish improves the latency for committing leader vertices by at least 1δ (when using [1]) and 3δ time (when using [14]). Additionally, compared to DAG-based protocols relying on BEB, Sailfish improves the latency to commit the non-leader vertices by at least 1δ (when using [1]).

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 improving the commit latency to 1RBC plus 1δ for leader vertices. In a typical RBC protocol [7, 14, 25], 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.

Towards supporting multiple leaders in a round. In order to improve the latency for multiple vertices, we extend Sailfish to support multiple leaders within a single round. We categorize these leaders as the main leader and secondary leaders. The primary role of the main leader remain identical to that of Sailfish: its leader vertex must either establish a path to all leader vertices from the previous round or contain a proof that some missing leader vertices cannot be committed; thus maintaining the safety invariant. We term this extended version Multi-leader Sailfish.

The commit rule closely resembles that of Sailfish, with some additional constraints. In an ideal scenario where all leaders are

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

	RBC Used	LV Commit Latency	NLV Commit[(1)] Latency	Communication Complexity	Leader Frequency	Multiple Leaders	NLV Latency [(2)] Under Failure	Modular?
Bullshark [29, 30]	Das et al. [14]	8δ	$+8\delta$	$O(n^3)$	1/2	✗	$+(8\Delta+8\delta)$	✓
Shoal [28]	Das et al. [14]	8δ	$+4\delta$	$O(n^3)$	1	✓	$+(8\Delta+4\delta)$	✓
Cordial Miners [22]	None	3δ	$+3\delta$	$O(n^4)$	1/3	✗	$+6\Delta$	✗
Mysticeti [3]	None	3δ	$+3\delta$	$O(n^4)$	1/3	✓	$+6\Delta$	✗
Sailfish	Das et al. [14]	5δ	$+4\delta$	$O(n^3)$	1	✓	$+(8\Delta+2\delta)$	✓
Sailfish	Abraham et al. [1]	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. [14] 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. [1] to achieve a commit latency of 4δ for leader vertices and additional 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 previous leader vertex; the commit latency of these vertices is the maximum among non-leader vertices between two leader rounds. (2) The column lists the increase in latency to commit non-leader vertices when a single Byzantine failure occurs between honest leaders.

honest, the respective leader vertices can be committed with a latency of one RBC plus 1δ.

Evaluation. We implement and evaluate the performance of Sailfish, comparing it with state-of-the-art DAG-based protocols. In our evaluation, we observe that Sailfish achieves significantly better latency than Bullshark [29, 30] and Shoal [28], with slightly better throughput. This improvement stems from Sailfish's support for a leader vertex in each round and its ability to commit the leader vertex with one RBC plus 1δ, and an additional RBC to commit the non-leader vertices.

We also evaluate the performance of Multi-leader Sailfish. Our results indicate that in failure-free cases, the average latency of the protocol reduces with the increase in the number of leaders in a round as more vertices are committed with a latency of one RBC plus 1δ.

Organization. The rest of the paper is organized as follows: In Section 2, we present the system model and preliminaries for our work. Section 3 presents Sailfish, the first DAG-based BFT that supports a leader vertex in each round and achieves a commit latency of one RBC plus 1δ to commit the leader vertex and an additional RBC to commit the non-leader vertices. We further enhance this protocol in Section 4 to accommodate multiple leaders within a single round, aiming to achieve improved average latency. An evaluation of our protocols is presented in Section 5, followed by a comprehensive discussion of related works in Section 6.

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. [16]. Under this model, the network starts in an initial state of asynchrony during which the adversary may arbitrarily delay messages sent by honest parties. However, after an unknown time called the *Global Stabilization Time* (GST), the adversary must ensure that all messages sent by honest parties 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 [29], we focus on the Byzantine Atomic Broadcast (BAB) problem as defined below:

Definition 2.1 (Byzantine atomic broadcast [21, 29]). Each honest party $P_i \in \mathcal{P}$ can call $\text{a\_bcast}_i(m, r)$ and output $\text{a\_deliver}_i(m, r, P_k)$, $P_k \in \mathcal{P}$. 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 $\text{a\_deliver}_i(m, r, P_k)$ before $\text{a\_deliver}_i(m', r', P_\ell)$, then no honest party P_j outputs $\text{a\_deliver}_j(m', r', P_\ell)$ before $\text{a\_deliver}_j(m, r, P_k)$.

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.

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. 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 RBC to ensure non-equivocation and guarantee all honest parties eventually deliver them.

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 [29], 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 L_r is referred to as the round r leader vertex 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 (see Line 36). We note that including a reference to the round $r-1$ leader vertex serves as a "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 $\langle \text{timeout}, r-1 \rangle_i$ to all other parties (see Line 38). In addition, an honest party P_j in round $r' \leq r-1$ also multicasts $\langle \text{timeout}, r-1 \rangle_j$ messages if it receives $f+1$ distinct round $r-1$ timeout messages (see Line 40). 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 (see Line 36). 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 a quorum of 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.

Jumping rounds. Apart from advancing the rounds sequentially, our protocol allows 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 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).

The commit rule is presented in Figure 3. 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) 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

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_leader_vertex(r)

  return get_vertex(L_r, r)

procedure a_bcast$_i$(b, r)

  $blocksToPropose$.enqueue(b)

procedure broadcast_vertex(r)

  $v \leftarrow$ create_vertex(r)

  try_add_to_dag(v)

  r_bcast$_i$(v, r)

procedure order_vertices()

  while $\neg leaderStack$.isEmpty() **do**

    $v \leftarrow leaderStack$.pop()

    $verticesToDeliver \leftarrow \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v, v') \wedge v' \notin deliveredVertices\}$

    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 [21, 29].

Local variables:

$round \leftarrow 1$; $buffer \leftarrow \{\}$

upon r_deliver$_i$(v, r, p) **do**

  if $v.source = p \wedge v.round = r \wedge |v.StrongEdges| \geq 2f + 1 \wedge$ is_valid(v) **then**

    if $\neg$try_to_add_to_dag(v) **then**

      $buffer \leftarrow buffer \cup \{v\}$

    else

      for $v' \in buffer : v'.round \leq r$ **do**

        try_to_add_to_dag(v')

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**

  advance_round($r + 1$)

upon timeout **do**

  multicast $\langle$timeout, $round\rangle_i$

upon receiving $f + 1$ distinct $\langle$timeout, $r\rangle_*$ such that $r \geq round$ **do**

  multicast $\langle$timeout, $r\rangle_i$

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

  multicast $\mathcal{TC}_r$

procedure create_new_vertex(r)

  $v.round \leftarrow r$

  $v.source \leftarrow P_i$

  $v.block \leftarrow blocksToPropose$.dequeue()

  $v.strongEdges \leftarrow DAG_i[r - 1]$

  set_weak_edges(v, r)

  if $\nexists v' \in DAG_i[r - 1] : v'.source = L_{r-1}$ **then**

    $n.tc \leftarrow \mathcal{TC}_{r-1}$

    if $P_i = L_r$ **then**

      $v.nvc \leftarrow \mathcal{NVC}_{r-1}$

  return v

procedure try_to_add_to_dag(v)

  if $\forall v' \in v.strongEdges \cup v.weakEdges : v' \in \bigcup_{k \geq 1} DAG_i[k]$ **then**

    $DAG_i[v.round] \leftarrow DAG_i[v.round] \cup \{v\}$

    if $|DAG_i[v.round]| \geq 2f + 1$ **then**

      try_commit($v.round - 1, DAG_i[v.round]$)

    $buffer \leftarrow buffer \setminus \{v\}$

    return true

  return false

procedure advance_round(r)

  if $\nexists v' \in DAG_i[r - 1]| : v'.source = L_{r-1}$ **then**

    send $\langle$no-vote, $r - 1\rangle_i$ to L_r

  if $P_i = L_r$ **then**

    wait until $\exists v' \in DAG_i[r - 1] : v'.source = L_{r-1}$ or $\mathcal{NVC}_{r-1}$ is received

  $round \leftarrow r$; *start timer*

  broadcast_vertex($round$)

Figure 2: Sailfish: DAG construction protocol for party P_i

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 strong 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

```
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)
    v ← get_leader_vertex(r)
    votes ← {v′ ∈ S | strong_path(v′, v)}
    if votes ≥ 2f + 1 then
        commit_leader(v)

procedure commit_leader(v)
    leaderStack.push(v)
    r ← v.round − 1
    v′ ← v
    while r > committedRound do
        v_s ← get_leader_vertex(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

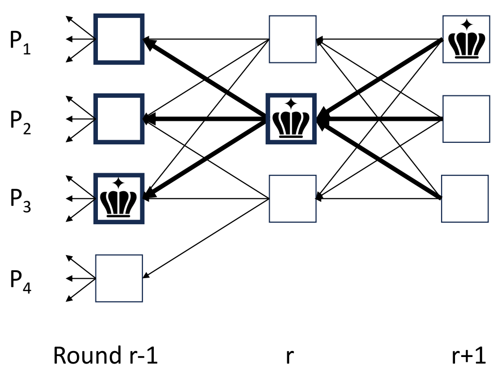


Figure 4: Depicts the view of one of the parties committing blocks from rounds r and $r-1$. When the party observes $2f+1$ votes on the round r leader vertex, it commits that vertex and deterministically orders non-leader vertices from earlier rounds (bold vertices). We assume round $r-1$ leader vertex was committed earlier. Note that performing this commit requires only the receipt of round $r+1$ vertices; they do not need to be RBC-delivered. Moreover, since the round r leader does not refer to P_4's round $r-1$ vertex, it is not ordered as yet.

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 [1, 2, 7, 25] 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. [2]. 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. [14] 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 $\langle \text{no-vote}, r \rangle$ 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. [14], 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. [14]).

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 does not 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 faulty 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 $\mathcal{TC}$ and $\mathcal{NVC}$. The size of these certificates is $O(1)$ assuming threshold signatures [6] ($O(n)$ without threshold signatures). In each round, each party propagates a single vertex via RBC. The RBC protocol of Das et al. [14] incurs $O(n^2)$ 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 (when using [14]).

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

We present detailed security analysis in Appendix A.

4 MULTI-LEADER SAILFISH

In Sailfish, the latency to commit the leader vertex is shorter compared to the non-leader vertices. To improve the latency for multiple vertices, we extend Sailfish to support multiple leaders within a single round. In the best-case scenario, when all of these leaders are honest, the respective leader vertices can be committed with a latency of one RBC plus 1δ.

Multiple leaders in a round. In this protocol, multiple leaders are chosen within a round based on the round number. One of these leaders serves as the main leader, while the others are designated as secondary leaders. The vertex proposed by the main leader is referred to as the main leader vertex, and the vertices proposed by the secondary leaders are termed secondary leader vertices. The responsibilities of the main leader in Multi-leader Sailfish are consistent with those in Sailfish: either the main leader vertex must have a strong path to all leader vertices from the previous round or the main leader must collect a no-vote certificate for any missing leader vertices.

To determine the multiple leaders in a given round, we define a deterministic function, get_multiple_leaders(r), which returns an ordered list of leaders for round r. The first leader in this list serves as the main leader, while the subsequent leaders are designated as secondary leaders. Analogous to Sailfish, the main leader for round r is denoted as L_r. We use $\mathcal{ML}_r$ to denote the ordered list of leaders provided by get_multiple_leaders(r). $\mathcal{ML}_r[x]$ denotes the x^{th} element in the list. Additionally, $\mathcal{ML}_r[:x]$ represents the first x leaders, while $\mathcal{ML}_r[x+1:]$ denotes the leaders in the list excluding the first x leaders.

```
89:  upon timeout do
90:      if ∄v′ ∈ DAG_i[r] : v′.source = L_round then
91:          multicast ⟨timeout, round⟩_i
92:  procedure advance_round(r)
93:      ML ← get_multiple_leaders(r − 1)
94:      for p ∈ ML do                                  ▷ iterate over ML in order
95:          if ∄v′ ∈ DAG_i[r − 1]| : v′.source = p then
96:              send ⟨no-vote, p, r − 1⟩_i to L_r
97:      if P_i = L_r then
98:          wait until ∃v′ ∈ DAG_i[r − 1] : v′.source = p ∀p ∈ ML[: x] or
             NVC^{p′}_{r−1} is received, where p′ = ML[x + 1]
99:      round ← r; start timer
100:     broadcast_vertex(round)
101: procedure create_new_vertex(r)
102:     v.round ← r
103:     v.source ← P_i
104:     v.block ← blocksToPropose.dequeue()
105:     v.strongEdges ← DAG_i[r − 1]
106:     set_weak_edges(v, r)
107:     if ∄v′ ∈ DAG_i[r − 1] : v′.source = L_{r−1} then
108:         v.tc ← TC_{r−1}
109:     if P_i = L_r then
110:         ML ← get_multiple_leaders(r − 1)
111:         for p ∈ ML do
112:             if ∄v′ ∈ DAG_i[r − 1]| : v′.source = p then
113:                 v.nvc ← NVC^p_{r−1}
114:             break
115:     return v
116: procedure try_commit(r, S)
117:     CLS ← [ ]
118:     ML ← get_multiple_leaders(r)
119:     for p ∈ ML do
120:         v ← get_vertex(p, r)
121:         votes ← {v′ ∈ S | strong_path(v′, v)}
122:         if votes ≥ 2f + 1 then
123:             CLS ← CLS || v
124:         else break
125:     commit_leaders(CLS)
126: procedure commit_leaders(cls)
127:     leaderStack.push(cls)
128:     v′ ← cls[0]
129:     r ← v′.round − 1
130:     while r > committedRound do
131:         CMV ← [ ]
132:         ML ← get_multiple_leaders(r)
133:         for p ∈ ML do
134:             v ← get_vertex(p, r)
135:             if strong_path(v′, v) then
136:                 CMV ← CMV || v
137:             else break
138:         if CMV ≠ [ ] then
139:             v′ ← CMV[1]                             ▷ main leader vertex for round r
140:         leaderStack.push(CMV)
141:         r ← r − 1
142:     committedRound ← cls[0].round
143:     order_vertices()
144: procedure order_vertices()
145:     while ¬leaderStack.isEmpty() do
146:         CMV ← leaderStack.pop()
147:         for v ∈ CMV do                              ▷ iterate over CMV in order
148:             verticesToDeliver ← {v′ ∈ ⋃_{r>0} DAG_i[r] | path(v, v′) ∧ v′ ∉ deliveredVertices}
149:             for every v′ ∈ verticesToDeliver in some deterministic order do
150:                 output a_deliver_i(v′.block, v′.round, v′.source)
151:                 deliveredVertices ← deliveredVertices ∪ {v′}
```

Figure 5: Multi-leader Sailfish

DAG construction protocol. The basic data structures are identical to Sailfish. In order to accommodate multiple leaders in a round, we modify how parties advance rounds. The modified protocol is presented in Figure 5.

Recall that in Sailfish, each party P_i waits for the round r leader vertex until a timeout. If the leader vertex is not delivered before the timeout, P_i sends $\langle \text{timeout}, r \rangle$ message. Upon receiving either the round r leader vertex or $\mathcal{TC}_r$ (along with $2f+1$ round r vertices) P_i advances to round $r+1$. When P_i advances to round $r+1$ via $\mathcal{TC}_r$, it sends $\langle \text{no-vote}, r \rangle$ to L_{r+1}. Additionally, L_r must collect $\mathcal{NVC}_r$ before proposing a round $r+1$ leader vertex.

In Multi-leader Sailfish, P_i sends $\langle \text{timeout}, r \rangle$ only when it does not deliver the round r main leader vertex before the timeout; it does not send timeout messages when the secondary leader vertices are not delivered.

To handle multiple leaders, various strategies can be employed for advancing through rounds. For instance, party P_i could wait for all leaders in $\mathcal{ML}_r$ or $\mathcal{TC}_r$ (along with $2f+1$ round r vertices) before advancing to round $r+1$. Upon advancing to round $r+1$, P_i sends $\langle \text{no-vote}, p, r \rangle$ for all $p \in \mathcal{ML}_r$ from which P_i did not deliver the corresponding round r leader vertex. In the ideal scenario, when all leaders in $\mathcal{ML}_r$ are honest and after GST, all honest parties will responsively receive all round r leader vertices and move to round $r+1$. However, a single faulty leader can cause the protocol to await its leader vertex, thereby slowing down the protocol.

Alternatively, each party P_i could choose to wait solely for the round r main leader vertex or $\mathcal{TC}_r$ (along with $2f+1$ round r vertices) before progressing to round $r+1$. Subsequently, P_i would send $\langle \text{no-vote}, p, r \rangle$ for all $p \in \mathcal{ML}_r$ from which P_i did not receive the round r leader vertex by the time it advances to round $r+1$. While this approach prioritizes the fastest leaders in $\mathcal{ML}_r$ for voting, it may result in slow leaders not being voted on, potentially causing the the slow leaders to not achieve the best possible latency.

We adjust the constraint on the main leader vertex as follows: The round $r+1$ main leader vertex must establish strong paths to all leader vertices corresponding to leaders in $\mathcal{ML}_r[: x]$ (for some $x > 0$) and include a quorum of $\langle \text{no-vote}, p, r \rangle$ (referred to as $\mathcal{NVC}^p_r$), where $p = \mathcal{ML}_r[x+1]$ (see Line 98). If the main leader vertex has strong paths to all leader vertices corresponding to leaders in $\mathcal{ML}_r$, it is not required to include $\mathcal{NVC}_r$ for any round r leaders. The constraint on other round $r+1$ vertices remain unchanged; specifically, the round $r+1$ vertex must include $\mathcal{TC}_r$ only if it lacks a strong path to the round r main leader vertex. The is_valid() function is also appropriately updated to ensure that these constraints are met.

Committing and ordering the DAG. Similar to Sailfish, only the leader vertices are committed, and the 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 illustrated in the order_vertices function (refer to Line 144).

In this protocol, an honest party P_i directly commits a round r leader vertex v_k corresponding to $\mathcal{ML}_r[x]$ when it observes $2f+1$ "first messages" (of the RBC) for round r+1 vertices with strong paths to the vertex v_k and when all round r leader vertices corresponding to leaders in $\mathcal{ML}_r[: x-1]$ have been directly committed. If v_k fails to be directly committed, party P_i refrains from committing the leader vertices corresponding to the leaders in $\mathcal{ML}_r[x+1 :]$, even if there are $2f+1$ round $r+1$ vertices with strong paths to the leader vertices corresponding to the leaders in $\mathcal{ML}_r[x+1 :]$. We will shortly explain why it is necessary to skip committing leader vertices corresponding to leaders in $\mathcal{ML}_r[x+1 :]$ in this case. The commit rule is presented in try_commit() function (see Line 126).

In addition to the above commit rule, Multi-leader Sailfish also enables party P_i to directly commit round r leader vertex v_k corresponding to $\mathcal{ML}_r[x]$ when it delivers (via RBC) $2f+1$ round $r+1$ vertices that have strong paths to v_k and when all round r leader vertices corresponding to leaders in $\mathcal{ML}_r[: x-1]$ have been directly committed.

Upon directly committing the main leader vertex v_m in round r, P_i first indirectly commits leader vertices corresponding to $\mathcal{ML}_{r'}[: y]$ (for some $y > 0$) in an earlier round $r' < r$ such that there exists strong paths from v_m to all leader vertices corresponding to $\mathcal{ML}_{r'}[: y]$. Subsequently, this process of indirectly committing leader vertices of earlier rounds is repeated for leader vertices that have strong paths from leader vertex corresponding to $\mathcal{ML}_{r'}[1]$ (i.e., the main leader vertex of round r') until it reaches a round $r^* < r$ in which it previously directly committed a leader vertex (see Line 126). When round r' leader vertices corresponding to leaders in $\mathcal{ML}_{r'}[: y]$ are directly committed, we ensure that any future main leader vertex has a strong path to these round r' leader vertices. This ensures that these leader vertices will be (directly or indirectly) committed by honest parties who missed directly committing these leader vertices.

The order_vertices() function is also appropriately modified to handle multiple leaders in a round (see Line 144).

Intuition behind skipping leaders in $\mathcal{ML}_r[x+1 :]$ when $\mathcal{ML}_r[x]$ is not directly committed. As mentioned earlier, if P_i does not directly commit a leader vertex v_k corresponding to $\mathcal{ML}_r[x]$, it also refrains from committing the leader vertices for the leaders in $\mathcal{ML}_r[x+1 :]$, even if there are sufficient votes for these leader vertices. This precaution is taken because the main leader vertex of a higher round $r'' > r$ may still have a strong path to v_k. When this main leader vertex from round r'' is committed, the leader vertices corresponding to $\mathcal{ML}_r[: y]$ (for some $y > 0$) are also indirectly committed in order, provided there are strong paths from the round r'' main leader vertex to the leader vertices corresponding to $\mathcal{ML}_r[: y]$. If $y > x$, v_k would be committed before the leader vertices corresponding to the leaders in $\mathcal{ML}_r[x+1 :]$. By skipping the commit of leader vertices corresponding to $\mathcal{ML}_r[x+1 :]$, we ensure the total order property during the indirect commit.

Additional conditions required for committing the secondary leader vertices. We note two additional conditions required for committing the secondary leader vertices. First, to commit leader vertices corresponding to $\mathcal{ML}_r[x+1 :]$, the leader vertex corresponding to $\mathcal{ML}_r[x]$ must be committed beforehand. When $\mathcal{ML}_r[x]$ is faulty, all leader vertices corresponding to leaders in $\mathcal{ML}_r[x+1 :]$ fail to be committed, despite having sufficient votes for these leader vertices. To address this concern, leader reputation schemes [28, 33] can be employed to elect multiple leaders with a good reputation for a given round.

Secondly, recall that parties send $\langle \mathsf{timeout}, r \rangle$ messages only when the round r main leader vertex is not delivered in a timely manner. The requirement for a round r vertex to include $\mathcal{TC}_{r-1}$ when it lacks a strong path to the round $r-1$ main leader vertex (say v_k) can only prevent the Byzantine parties from proposing the round r vertex without a strong path to v_k. This ensures that sufficient honest parties vote for v_k in round r and v_k is committed by round r, after GST. However, this does not prevent Byzantine parties from "not voting" for the secondary leader vertices and send round r vertices with strong path only to v_k. With the help of $f+1$ honest parties who vote for the secondary leader vertices, the adversary can cause the protocol to advance to a higher round $r' > r$ while f honest parties are lagging behind in some lower round $r'' \leq r-1$. The adversary can then deliver $2f+1$ round r' vertices along with round r' main 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 vertex. This prevents the round $r-1$ secondary leader vertices from being directly committed. This issue can potentially be addressed by introducing a timeout certificate for each leader in a round and requiring a round r vertex to include a timeout certificate for each missing round $r-1$ leader vertex; however the solution is less practical due to added synchronization overhead and increase in size of a vertex.

In this context, Multi-leader Sailfish ensures that the round r secondary leader vertices are committed by round $r+1$ only under an "optimistic condition" where at least $2f+1$ parties (including Byzantine parties) "vote" for the proposed secondary leader vertices. Under normal conditions, these vertices will be committed in the next round when the round $r+1$ leader vertex is committed. We also note that these conditions apply to Mysticeti [3], although they did not explicitly state the latter requirement.

4.1 Efficiency Analysis

Commit latencies. We analyze the commit latencies under the optimistic condition where all parties vote for all proposed leader vertices. If parties wait for all leader vertices corresponding to $\mathcal{ML}_r$, and all leaders in $\mathcal{ML}_r$ are honest, the corresponding leader vertices can be committed with a latency of one RBC plus 1δ. However, a single faulty leader can cause the protocol to await its leader vertex, resulting in a latency of $O(\Delta)$.

Alternatively, when parties wait solely for the round r main leader vertex before advancing to the next round, the subsequent main leader needs to collect $\mathcal{NVC}_r$ for leaders for which it lacks strong paths to the corresponding leader vertices. This incurs an additional 1δ time. Thus, the commit latency for the leader vertices is one RBC plus 2δ, while the non-leader vertices require an additional RBC. Under this strategy, when the RBC protocol of Das et al. [14] is used, as long as $x > \frac{n-f+4}{4}$ leader vertices are directly

Table 2: Ping latencies (in ms) between GCP regions

	Destination*				
Source	us-e-1	us-w-1	eu-n-1	as-ne-1	au-se-1
us-east1-a	0.75	66.14	114.75	160.28	197.98
us-west1-a	66.15	0.66	158.13	89.56	138.33
europe-north1-a	115.40	158.38	0.69	245.15	295.13
asia-northeast1-a	159.89	90.05	246.01	0.66	105.58
australia-southeast1-a	197.60	139.02	294.36	108.26	0.58

*Region names are abbreviated versions of the source regions.

committed in a round, the average latency is still better compared to Sailfish.

Communication complexity. In Multi-leader Sailfish, unlike Sailfish, each party can send a no-vote message for every leader in $\mathcal{ML}_r$ to the subsequent leader L_{r+1}. Even with a linear number of leaders in a round, sending these no-vote messages incurs only $O(n^2)$ bits. Additionally, although $\mathcal{NVC}_r$ can exist for multiple leaders in round r, the main leader vertex of round $r+1$ has to incorporate a single $\mathcal{NVC}_r$. Therefore, the communication complexity of Multi-leader Sailfish remains the same as that of Sailfish.

We present detailed security analysis in Appendix B.

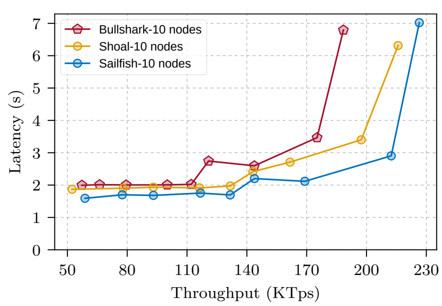


Figure 6: Throughput vs. end-to-end latency at $n = 10$ and varying input

5 EVALUATION

In this section, we evaluate the performance of Sailfish and Multi-leader Sailfish, comparing their throughput and latency with modular DAG-based BFT protocols: Bullshark and Shoal, across different system sizes and under failure scenarios.

Implementation details. Our implementation is a modification of the open-source implementation of Bullshark [27]. We made modifications to the core consensus logic to create Sailfish and Multi-leader Sailfish. Additionally, we created a custom implementation of Shoal (since their implementation is not publicly accessible) which guarantees a leader in every round and commits the leader vertex with two RBCs. This customized Shoal implementation is better as it does not require reinterpreting the DAG.

In the Bullshark implementation, the system consists of distinct clients, workers, and consensus nodes. Each consensus node is equipped with a number of workers. The client dispatches a configurable number of transactions to its designated worker. The workers then aggregate these transactions to form a batch, which is subsequently forwarded to the workers of other consensus nodes. Upon receiving the batch, a worker sends an acknowledgment back to its originating worker. Once a worker collects a quorum of acknowledgments, it sends the batch digest to its associated consensus node. The consensus node then incorporates this digest into its subsequent proposal.

Experimental setup. We carried out our evaluations on the Google Cloud Platform (GCP), distributing nodes evenly across five distinct GCP regions: us-east1-b (South Carolina), us-west1-a (Oregon), europe-north1-a (Hamina, Finland), asia-northeast1-a (Tokyo), and australia-southeast1-a (Sydney). We employed e2-standard-32[1] instances, each featuring 32vCPUs, 128GB of memory, and up to 16Gbps network bandwidth[2]. All nodes ran on Ubuntu 20.04, and we summarize round-trip latencies in Table 2. We used ED25519 signatures for authentication.

In our setup, one client and one worker is co-located within the consensus node. Each transaction is composed of 512 random bytes, and the batch size is configured to 500KB. We set the timeout parameter, τ to 3 seconds. Each experimental run spans 180 seconds, and the data presented in the graphs represents an average across three independent runs. For latency, we measured the average time between the creation of a transaction (or a vertex) and its commit by all (non-faulty) nodes to compute the end-to-end (or consensus) latency. Throughput is measured as the number of committed transactions per second.

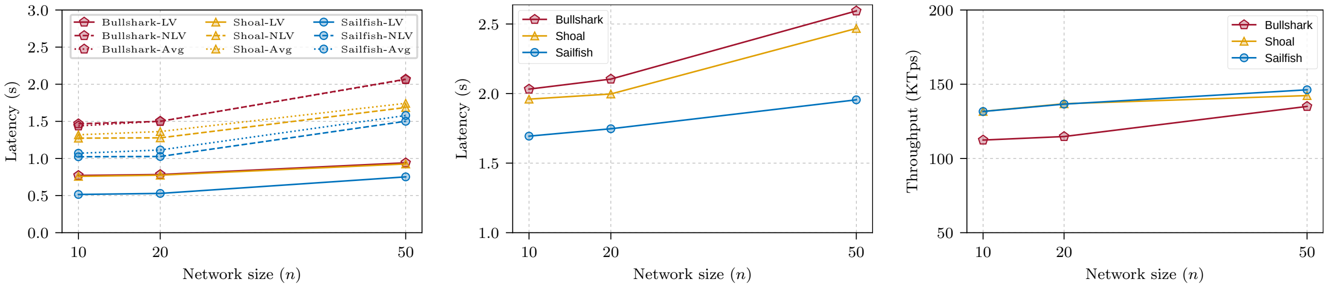
Methodology. In our evaluations, we gradually increased the input transactions. As depicted in Figure 6, the throughput increases with increasing load without increasing latency up to a certain point before reaching saturation. After saturation, the latency starts to increase while the throughput either remains consistent or slightly increases. In the subsequent figures, we report the throughput and latency just before reaching this saturation point.

Performance of Sailfish under fault-free case. We initially assess the performance of Bullshark, Shoal, and Sailfish under fault-free scenarios across system sizes of 10, 20, and 50 nodes. The consensus latencies are presented in Figure 7a, while the corresponding end-to-end latencies and throughput are illustrated in Figure 7b and Figure 7c respectively.

In Figure 7a, LV represents the average latency to commit the leader vertices, NLV represents the average latency to commit the non-leader vertices a round before the leader vertex and Avg represents the average latency for all vertices (including those from prior rounds that were ordered when the leader vertex was committed). For Bullshark, the NLV latency is the average latency to commit the two layers of non-leader vertices before the leader round.

Consistent with our theoretical analysis, Sailfish significantly outperforms both Bullshark and Shoal in terms of these latencies. While Bullshark and Shoal achieve similar latencies for the leader vertex, Bullshark's additional layer of non-leader vertices results in higher latency compared to Shoal for non-leader vertices. The improvement in consensus latencies directly translates to an improvement in the overall end-to-end latency. As depicted in Figure 7b, Sailfish reduces the end-to-end latency by approx. 20% compared to Bullshark and Shoal across all system sizes. Furthermore, due to

[1]https://cloud.google.com/compute/docs/general-purpose-machines
[2]https://cloud.google.com/compute/docs/network-bandwidth



(a) n vs. consensus latencies. **(b) n vs. end-to-end latency.** **(c) n vs. throughput.**

Figure 7: Performance in the absence of failures across different system sizes.

reduced latency of Sailfish, it achieves improved throughput before experiencing a latency spike as depicted in Figure 6 and Figure 7c.

Table 3: Consensus latencies (in ms) under failures at $n = 10$

	Leader vertices	Non-leader vertices	Average
Sailfish	754	2592	3234
Shoal [28]	1175	3003	6829
Bullshark [29]	1169	4960	7005

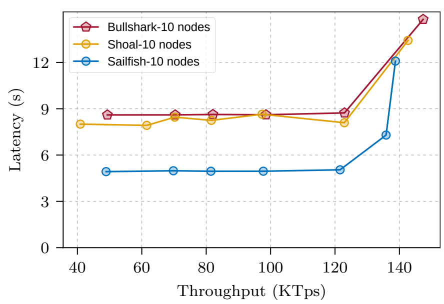


Figure 8: Throughput vs. end-to-end latency at $n = 10$ with 3 failures and varying input

Performance of Sailfish under failures. We subsequently evaluated the performance under f crash failures at $n = 10$, distributing the failed leaders across consecutive odd rounds. In the case of Sailfish, this translates to a leader failure occurring every other round over $2f$ rounds. Meanwhile, as Bullshark designates leaders exclusively in odd rounds, this equated to f consecutive leader failures. Consequently, Bullshark fails to commit for the first $2f$ rounds, with this pattern repeating every n rounds. Additionally, as Shoal relies on a Bullshark instance to commit some vertex before initiating a new Bullshark instance, Shoal does not start a new Bullshark instance until $2f$ rounds has passed.

The latency to commit the leader vertex increases slightly for all protocols compared to the fault-free scenario, as shown in Table 3. In fault-free cases, protocols commit with the fastest $2f + 1$ nodes. However, with f failures, the protocol must wait for all nodes, resulting in increased commit latency for the leader vertex. Additionally, rounds corresponding to the failed leader incur τ time, and the non-leader vertices of the failed rounds are only committed when the leader vertex of the following round is committed. This leads to an increase in the average latency to commit the non-leader vertices for all protocols.

In the case of Bullshark and Shoal, the vertices of the first $2f$ rounds are only committed after $2f$ rounds, resulting in worse average latency. As Sailfish supports a leader in every round, it can commit every other round, resulting in approx. 50% lesser average latency. We present the corresponding throughput and end-to-end latency in Figure 8. With the increased average commit latency, the end-to-end latency for both Bullshark and Shoal worsens compared to Sailfish, while the throughput remains (almost) the same as the failure-free case.

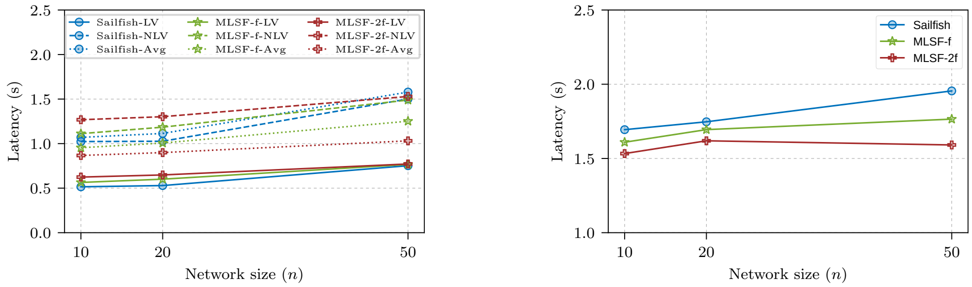
Performance of Multi-leader Sailfish under fault-free case. We also evaluated the performance of Multi-leader Sailfish in failure-free scenarios, exploring configurations with both f and $2f$ leaders in a round. To simplify implementation, we adopted the strategy where nodes wait for all leader vertices before advancing to the next round. The corresponding consensus and end-to-end latencies are presented in Figure 9. In Figure 9, MLSF-f represents Multi-leader Sailfish with f leaders, while MLSF-2f represents Multi-leader Sailfish with $2f$ leaders.

As depicted in Figure 9a, the latency to commit the leader vertex (and the non-leader vertices) increased slightly due to the necessity of waiting for all leader vertices in a round. Nonetheless, the average commit latency exhibits significant improvement as more vertices are committed with reduced latency (i.e., one RBC plus 1δ), which aligns with our theoretical analysis. This improvement in consensus latencies also translates to improved end-to-end latency. As illustrated in Figure 9b, we observe improved end-to-end latencies as the number of leader vertices increases.

6 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. Our protocol supports multiple leaders in a round. When employing the RBC protocol by Das et al. [14], our protocol requires 5δ to commit the leader vertex and an additional 4δ to commit the non-leader vertices. Moreover, it maintains a communication complexity of $O(n^3)$ per round.

Asynchronous DAG-based BFT. Hashgraph [4] builds an unstructured DAG, with each vertex containing two references to



(a) n **vs consensus latencies.**

(b) n **vs. end-to-end latency.**

Figure 9: Latency comparison of Multi-leader Sailfish in the absence of failures across different system sizes.

previous vertices. Parties then run an inefficient binary agreement protocol on this DAG, leading to an expected exponential time complexity. Aleph [19] 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 [21] 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 [13] is an implementation based on DAG-Rider.

Very recently, GradedDAG [11] and LightDAG [10] improve the latency of asynchronous DAG-based BFT protocols by using weaker primitives such as consistent broadcast [31] 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. Blockmania [12] employs a modified version of PBFT [9] for vertex dissemination and constructs a structured round-based DAG. Their protocol is specifically designed for owned objects [5], and it does not inherently ensure the total ordering of these vertices. Bullshark [29, 30] builds upon DAG-Rider to improve the commit latency during the synchronous period. 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 [28] proposed a pseudo-pipelining approach to reduce the latency of non-leader vertices in Bullshark. In their protocol, they execute multiple instances of the Bullshark sequentially to ensure a leader 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. When Bullshark fails to commit, Shoal requires an additional two rounds to commit some vertex and start a new Bullshark instance. This limitation compromises Shoal's ability to consistently guarantee a leader vertex in each round. Furthermore, Shoal's support for multiple leaders in a round hinges on executing multiple instances of Bullshark sequentially, each with a different leader. As Bullshark is inherently designed as a single-leader protocol which ensures the commitment of only one leader vertex per round (after GST), reinterpreting the existing DAG with a different leader does not guarantee the new leader will be committed, even if the new leader is honest.

In a private conversation with the Aptos team, we learned that they are also concurrently working on extending Shoal to improve the latency of the leader vertex to one RBC plus 1δ. However, following Shoal, their new protocol still does not support a leader vertex in each round in a true sense.

In a recent work, Cordial Miners [22] introduced a DAG-based BFT protocol that uses BEB instead of RBC to propagate vertices, aiming to reduce latency. While their protocol achieves a commit latency of 3δ for the leader vertex, it still requires 6δ to commit non-leader vertices aligned with the leader round. Extending the work of Cordial Miners [22], Mysticeti [3] introduces support for multiple leaders within the same round and enhances the speed of committing owned objects [5]. However, both the protocols suffer from a high communication complexity of $O(n^4)$ per round and lacks modularity. Additionally, both protocols incur higher latency in the event of leader failure as they need to wait in each (non-RBC) round. In comparison, the protocols that rely on RBC can employ a single wait for multiple steps of RBC in a round, resulting in reduced latency. Furthermore, while Mysticeti supports multiple leaders in a round and specifies the commit rule to commit multiple vertices, it does not detail the necessary conditions required to ensure these leaders are committed.

BBCA-chain [24] also addresses the challenge of supporting leaders in each round. They rely on a traditional leader-heavy BFT protocol inherently capable of accommodating a leader in each round. At the end of each round, each party sends a block of transactions (via BEB) along with the commit status of the current round and the commit certificate for the highest round it have observed. This message serves as the view-change message in traditional protocols [9]. The next leader aggregates a quorum of these messages in its new proposal; thus forming a DAG. The leader uses single-shot PBFT [9] instance to propose its block. However, in their protocol, 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; placing a heavier burden on the leader. In comparison, in our protocol (and DAG-based BFT protocols in general), each party performs the same amount of work.

ACKNOWLEDGEMENTS

We thank George Danezis, Lefteris Kokoris-Kogias, Oded Naor, Ehud Shapiro, and Alexander Spiegelman for their helpful comments on an earlier version of this manuscript.

REFERENCES

[1] 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.

[2] 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.

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

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

[5] Mathieu Baudet, George Danezis, and Alberto Sonnino. Fastpay: High-performance byzantine fault tolerant settlement. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 163–177, 2020.

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

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

[8] 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.

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

[10] 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.

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

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

[13] 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.

[14] 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.

[15] Isaac Doidge, Raghavendra Ramesh, Nibesh Shrestha, and Joshua Tobkin. Moonshot: Optimizing chain-based rotating leader bft via optimistic proposals. In *International Conference on Dependable Systems and Networks (DSN)*, 2024.

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

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

[18] Ethereum. Data availability | ethereum.org, 2024. Accessed on March 20, 2024.

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

[20] 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.

[21] 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.

[22] 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.

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

[24] Dahlia Malkhi, Chrysoula Stathakopoulou, and Maofan Yin. Bbca-chain: One-message, low latency bft consensus on a dag. In *International Conference on Financial Cryptography and Data Security*, 2024.

[25] 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.

[26] Kartik Nayak, Ling Ren, Elaine Shi, Nitin H Vaidya, and Zhuolun Xiang. Improved extension protocols for byzantine broadcast and agreement. In *34th International Symposium on Distributed Computing*, 2020.

[27] Facebook Research. Bullshark github repository. https://github.com/facebookresearch/narwhal/tree/bullshark. [Online; accessed 20-April-2024].

[28] Alexander Spiegelman, Balaji Aurn, Rati Gelashvili, and Zekun Li. Shoal: Improving dag-bft latency and robustness. In *International Conference on Financial Cryptography and Data Security*, 2024.

[29] 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.

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

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

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

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

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

A SECURITY ANALYSIS OF SAILFISH

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 $v \in DAG_i[r] \land v' \in DAG_j[r]$ 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 Q of $2f + 1$ vertices in $DAG_i[r + 1]$ that included a reference to vertex v_k. Let $\mathcal{H} \subset Q$ be the set of vertices proposed by honest parties in 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 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 leader vertices 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 $r_i \leq r_j < r'_i$. By Claim 2, both P_i and P_j will (directly or indirectly) commit the same leader vertex in the round $\min(r'_i, r'_j)$. Assume $\min(r'_i, r'_j) = r'_i$. By Fact 1, both DAG_i and DAG_j will contain v'_k and all vertices that have a path from v'_k in DAG_i.

By the code of commit_leader, after (directly or indirectly) committing the leader vertex v'_k, parties try to indirectly commit leader vertices in smaller round numbers until they reach a round in which they previously directly committed a leader vertex. Therefore, both P_i and P_j will indirectly commit all leader vertices from $\min(r'_i, r'_j)$ to $\max(r_i, r_j)$. Moreover, due to deterministic code of commit_leader, both parties will commit the same leader vertices between rounds $\min(r'_i, r'_j)$ to $\max(r_i, r_j)$ in the same order. □

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

Corollary A.1. *Honest parties commit the same leader vertices in the same order.*

Lemma A.2 (Total order). *The protocol in Figures 1 to 3 satisfies Total order.*

Proof. By Corollary A.1, honest parties commit the same leader vertices in the same order. By the code of order_vertices, parties iterate on the committed leader vertices 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 A.3 (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 A.2, 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 A.4 (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. [14] 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 $\langle$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 $\langle$timeout, $r\rangle$ and receive $\mathcal{TC}_r$. In addition, all honest parties will also send $\langle$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 $\langle$timeout, $r\rangle$ before $t + 8\Delta$ due to its round timer expiring. Accordingly, no honest party sends $\langle$timeout, $r\rangle$ due to receiving $f + 1$ $\langle$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 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 invokes r_bcast on its 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 A.5. *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 A.6 (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 A.5, 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 A.3, all honest parties will eventually invoke a_deliver(b, r, p_i). □

B SECURITY ANALYSIS OF MULTI-LEADER SAILFISH

We say that a *leader vertex v_i is committed directly* by party P_i if P_i invokes commit_leaders($\mathcal{CLS}$) and $v_i \in \mathcal{CLS}$. Similarly, we say that a *leader vertex v_j is committed indirectly* if $\mathcal{CMV}$ is added to *leaderStack* (in Line 140) and $v_j \in \mathcal{CMV}$. In addition, we say party P_i consecutively directly commit leader vertices in rounds r and $r' > r$ and does not directly commit any leader vertex between r and r'.

To commit a round r leader vertex in Multi-leader Sailfish, at least $f + 1$ round-$(r + 1)$ vertices proposed by honest parties must have a strong path to the round r leader vertex, which is identical to that of Sailfish. Additionally, the main leader vertex of Multi-leader Sailfish is identical to the leader vertex of Sailfish. Thus, the proof of the following claim (Claim 9) remains identical to Claim 1.

Claim 9. *If an honest party P_i directly commits a leader vertex v_k in round r, then for every main leader vertex v_ℓ in round r' such that $r' > r$, there exists a strong path from v_ℓ to v_k.*

Similarly, the indirect commit rule of a main leader vertex in Multi-leader Sailfish is identical to the indirect commit rule of the leader vertex in Sailfish. Thus, the proof of the following claim (Claim 10) remains identical to Claim 2 except Claim 9 needs to be invoked (instead of Claim 1).

Claim 10. *If an honest party P_i directly commits the main leader vertex v_k in round r and an honest party P_j directly commits the main leader vertex v_ℓ in round $r' \geq r$, then P_j (directly or indirectly) commits v_k in round r.*

Claim 11. *If an honest party P_i directly commits all leader vertices corresponding to $\mathcal{ML}_r[:x]$ (for some $x > 0$) and an honest party P_j directly commits the main leader vertex v_ℓ in round $r' > r$, then P_j indirectly commits all leader vertices corresponding to $\mathcal{ML}_r[:x]$ in round r.*

Proof. Given that P_i directly committed all leader vertices in $\mathcal{ML}_r[:x]$, by Fact 1 and Claim 9, there are strong paths from the main leader vertex of any round higher than r to all leader vertices corresponding to $\mathcal{ML}_r[:x]$ in DAG_j.

By the code of commit_leaders(), after directly committing the main leader vertex v_ℓ in round r', P_i tries to indirectly commit all leader vertices corresponding to $\mathcal{ML}_{r''}[:y]$ (for some $y > 0$) in an earlier round $r'' < r'$ such that there exists strong paths from v_ℓ to all leader vertices corresponding to $\mathcal{ML}_{r''}[:y]$. This process of indirectly committing multiple leader vertices of an earlier round is repeated for leader vertices that have strong paths from the main leader vertex of round r'' (i.e., $\mathcal{ML}_{r''}[1]$), 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 all leader vertices in $\mathcal{ML}_r[:x]$ in round r. Otherwise, by inductive argument and Claim 9, party P_j must have indirectly committed all leader vertices in $\mathcal{ML}_r[:x]$ when directly committing the main leader vertex of round r^*. □

Claim 12. *Let an honest party P_i consecutively directly committed in rounds r_i and r'_i. Also, let an honest party P_j consecutively directly committed in rounds r_j and r'_j. Then, P_i and P_j commits the same leader vertices 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 $r_i \leq r_j < r'_i$. Also, assume $\min(r'_i, r'_j) = r'_i$. Let $\mathcal{ML}_{r'_i}[:x]$ be the list of multiple leader vertices directly committed by party P_i in round r'_i for some $x > 0$. If $r'_i = r'_j$, by Claim 10, party P_j commits at least $\mathcal{ML}_{r'_i}[1]$ in round r'_i. Otherwise, by Claim 11, party P_j indirectly commits all leader vertices in $\mathcal{ML}_r[:x]$ in round r'_i.

Moreover, by Fact 1, both DAG_i and DAG_j will contain $\mathcal{ML}_{r'_i}[1]$ (i.e., the main leader vertex in round r'_i) and all vertices that have a path from $\mathcal{ML}_{r'_i}[1]$ in DAG_i. By the code of commit_leaders(), after (directly or indirectly) committing $\mathcal{ML}_{r'_i}[1]$, parties try to indirectly commit multiple leader vertices in a smaller round number $r'' < r'_i$ that have strong paths from $\mathcal{ML}_{r'_i}[1]$. And, this process is repeated by indirectly committing leader vertices of earlier round with strong paths from $\mathcal{ML}_{r''}[1]$ until it reaches a round $r^* < r$ in which it previously directly committed a leader vertex. Therefore, both P_i and P_j will indirectly commit all leader vertices from $\min(r'_i, r'_j)$ to $\max(r_i, r_j)$. Moreover, due to deterministic code of commit_leaders, both parties will commit the same leader vertices between rounds $\min(r'_i, r'_j)$ to $\max(r_i, r_j)$ in the same order. □

By inductively applying Claim 12 between any two pairs of honest parties we obtain the following corollary.

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

The proof of the following total order lemma (Lemma B.2) remains identical to Lemma A.2 except Corollary B.1 needs to be invoked (instead of Corollary A.1).

Lemma B.2 (Total order). *Multi-leader Sailfish satisfies Total order.*

Agreement. The agreement proof remains identical to Lemma A.3 except Lemma B.2 needs to be invoked (instead of Lemma A.2).

Integrity. The integrity proof remains identical to Lemma A.4.

Validity. We again rely on GST to prove validity and utilize the RBC protocol from Das et al. [14].

Claim 13. *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, 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 either round $r - 1$ main leader vertex (say v_k) or received $\mathcal{TC}_{r-1}$ along with $2f + 1$ round $r - 1$ vertices. By Property 2, all honest parties must have delivered $2f+1$ round $r-1$ vertices by $\max(t_g, t)+2\Delta$. If P_i delivered v_k, by Property 2, all honest parties must have delivered v_k by $\max(t_g, t) + 2\Delta$. Otherwise, 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.

Having delivered v_k or received $\mathcal{TC}_{r-1}$ (along with $2f+1$ round $r-1$ vertices), an honest party P_j sends $\langle$no-vote, $P_k, r-1\rangle$ for all $P_k \in \mathcal{ML}_{r-1}$ if P_j did not deliver its corresponding leader vertex by then. If no honest party delivered the leader vertex corresponding to P_k by $\max(t_g, t) + 2\Delta$, then all honest parties (including L_r) will send $\langle$no-vote, $P_k, r-1\rangle$ to L_r. Thus, L_r will receive $\mathcal{NVC}_{r-1}^{P_k}$ by time $\max(t_g, t) + 3\Delta$. On the other hand, if at least one honest party delivered the leader vertex corresponding to P_k by $\max(t_g, t) + 2\Delta$, by Property 2, L_r will deliver the leader vertex corresponding to P_k by $\max(t_g, t) + 4\Delta$. Thus, L_r will either deliver a leader vertex corresponding to P_k or receive $\mathcal{NVC}_{r-1}^{P_k}$ for all $P_k \in \mathcal{ML}_{r-1}$ by time $\max(t_g, t)+4\Delta$. Since L_r waits for leader vertices corresponding to $\mathcal{ML}_{r-1}[: x]$ and $\mathcal{NVC}_{r-1}^{p}$ where $p = \mathcal{ML}_{r-1}[x+1]$, L_r enters 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 14. *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 13, all honest parties will enter round r' or higher. Otherwise, all honest parties are in round r. Observe that all honest parties will r_bcast round r vertex when entering round r. Thus, all honest parties will deliver $2f+1$ round r vertices. Furthermore, if an honest party (except L_{r+1}) delivers the round r main leader vertex (say v_k), it will advance to round $r+1$.

Alternatively, if no honest party delivered v_k by the time their round r timer expires, due to the timeout rule, all honest parties will multicast $\langle$timeout, $r\rangle$ and subsequently receive $\mathcal{TC}_r$. Having delivered v_k or received $\mathcal{TC}_r$, an honest party P_j send $\langle$no-vote, $P_k, r\rangle$ for all $P_k \in \mathcal{ML}_r$ if P_j did not deliver its corresponding leader vertex by then. If no honest party delivered the leader vertex corresponding to P_k by the time they delivered v_k or received $\mathcal{TC}_r$, then all honest parties will send $\langle$no-vote, $P_k, r\rangle$ to L_{r+1}. Thus, L_{r+1} will receive $\mathcal{NVC}_r^{P_k}$. On the other hand, if at least one honest party delivered the leader vertex corresponding to P_k, by Property 2, L_{r+1} will deliver the leader vertex corresponding to P_k. Thus, L_r will either deliver a leader vertex corresponding to P_k or receive $\mathcal{NVC}_r^{P_k}$ for all $P_k \in \mathcal{ML}_r$. Since L_{r+1} waits for leader vertices corresponding to $\mathcal{ML}_r[: x]$ and $\mathcal{NVC}_r^{p}$ where $p = \mathcal{ML}_r[x+1]$, L_{r+1} will advance to round $r+1$. ◻

The proof of the following claim (Claim 15) remains identical to Claim 8 except Claim 13 needs to be invoked (instead of Claim 4).

Claim 15. *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 main leader vertex.*

By the commit rule and Claim 15, the following corollary follows.

Corollary B.3. *If the first honest party to enter round r does so after GST and L_r is honest, all honest parties will directly commit the round r main leader vertex.*

The proof of the following validity lemma (Lemma B.4) remains identical to Lemma A.6 except Corollary B.3 needs to be invoked (instead of Corollary A.5).

Lemma B.4 (Validity). *Multi-leader Sailfish satisfies Validity.*

As demonstrated in Claim 15, a round r main leader vertex (proposed by an honest leader) is always committed by round $r+1$ (after GST). We now establish that the round r secondary leader vertices will receive votes from at least $2f+1$ round $r+1$ vertices under an "optimistic condition" when at least $2f+1$ parties (including Byzantine parties) vote for the proposed secondary leader vertices. Consequently, all leader vertices corresponding to $\mathcal{ML}_r[: x]$ will be committed by round $r+1$ when all leaders in $\mathcal{ML}_r[: x]$ are honest (after GST).

Claim 16. *If the first honest party to enter round r does so after GST and $\mathcal{HML}_r \subseteq \mathcal{ML}_r$ be the set of honest round r leaders, then under an optimistic condition where all parties vote for the proposed vertices, there exists at least $2f+1$ round $r+1$ vertices with strong paths to round r leader vertices corresponding to parties in $\mathcal{HML}_r$.*

Proof. Let t be the time when the first honest party (say P_i) entered round r. Observe that no honest party sends $\langle$timeout, $r\rangle$ before $t+8\Delta$ due to its round timer expiring. Accordingly, no honest party sends $\langle$timeout, $r\rangle$ due to receiving $f+1$ $\langle$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 13, 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 the round r main leader vertex. 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 the round r main leader vertex. Moreover, under the optimistic condition, the round $r+1$ vertices must have strong paths to all other round r leader vertices corresponding to parties in $\mathcal{HML}_r$.

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

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 13, 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_bcast its 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, round r main leader vertex will be delivered by $t+8\Delta$. Thus, all honest parties will receive $2f+1$ round r vertices along with leader vertices corresponding to parties in $\mathcal{HML}_r$ by $t + 8\Delta$.

When honest parties advance to round $r + 1$, their round $r + 1$ vertex will have a strong path to v_k. □