# Angelfish: Consensus with Optimal Throughput and Latency Across the Leader-DAG Spectrum

Qianyu Yu
The Hong Kong University of
Science and Technology (Guangzhou)
qyu100@connect.hkust-gz.edu.cn

Nibesh Shrestha Supra Research n.shrestha@supra.com

#### **Abstract**

To maximize performance, many modern blockchain systems rely on eventually-synchronous, Byzantine fault-tolerant (BFT) consensus protocols. Two protocol designs have emerged in this space: protocols that minimize latency using a leader that drives both data dissemination and consensus, and protocols that maximize throughput using a separate, asynchronous data dissemination layer. Recent protocols such as Partially-Synchronous Bullshark and Sailfish combine elements of both approaches by using a DAG to enable parallel data dissemination and a leader that paces DAG formation. This improves latency while achieving state-of-the-art throughput. Yet the latency of leader-based protocols is still better under moderate loads.

We present Angelfish, a hybrid protocol that adapts smoothly across this design space, from leader-based to Sailfish-like DAG-based consensus. Angelfish lets a dynamically-adjusted subset of parties use best-effort broadcast to issue lightweight votes instead of reliably broadcasting costlier DAG vertices. This reduces communication, helps lagging nodes catch up, and lowers latency in practice compared to prior DAG-based protocols. Our empirical evaluation [49] shows that Angelfish attains state-of-the-art peak throughput while matching the latency of leader-based protocols under moderate throughput, delivering the best of both worlds. The implementation is open-sourced and publicly available.

#### 1 Introduction

State machine replication (SMR) [26] is a foundational building block in distributed computing, enabling a set of parties to jointly simulate a single, reliable state machine despite failures and malicious attacks. In particular, blockchain systems use the SMR approach to provide open, permissionless access to shared virtual machines that users can program using smart contracts.

Giuliano Losa Stellar Development Foundation giuliano@stellar.org

Xuechao Wang\*
The Hong Kong University of
Science and Technology (Guangzhou)
xuechaowang@hkust-gz.edu.cn

Applying the SMR approach in a permissionless environment is challenging because the parties participating in an SMR protocol cannot be trusted and may behave maliciously. A core building block to implement SMR in such adversarial environments is a Byzantine fault-tolerant (BFT) consensus protocol [29]. BFT consensus protocols allow parties to agree on the sequence of commands, called transactions in a blockchain context, that the state machine must execute, and they are resilient to malicious attacks by some fraction of the parties executing the protocol.

As the consensus protocol is in the critical path of user requests to an SMR system, its throughput and latency are crucial. A pragmatic approach to devise high-performance consensus protocols is to use the partial-synchrony model [13]. This model formalizes the observation that practical networks are mostly synchronous, which allows achieving high performance, but may suffer from bounded periods during which message delays are unpredictable. Among partially synchronous protocols, two competing approaches have emerged. The key difference lies in how much the two main tasks of a consensus protocol, namely the dissemination of transaction blocks and their ordering, are decoupled.

On the one hand, BFT consensus protocols [8, 11, 47] inspired by traditional partially-synchronous consensus protocols [13, 27] rely on a rotating leader which is responsible for both disseminating transactions and proposing an ordering. This approach, which tightly couples consensus with data dissemination, can achieve optimal theoretical latency (i.e., 3 message delays [28]) and low latency in practice. Moreover, while the leader's bandwidth constituted a throughput bottleneck in earlier approaches, protocols like DispersedSimplex [36] use asynchronous information-dispersal schemes (AVID) [9] that rely on erasure codes to harness the bandwidth of all parties to help a leader disseminate its transaction block. This holds the promise of eliminating the leader bottleneck while keeping latency low. However, erasure-code AVID has computational costs, synchronization costs, and data expansion costs; for example, in DispersedSimplex, erasure-coding increases block size by a factor of roughly 3. These overheads

<sup>\*</sup>Corresponding author.

impact both throughput and latency.

On the other hand, some protocols choose to decouple data dissemination from ordering: an asynchronous data-dissemination layer (sometimes called a mempool) reliably stores transaction blocks, and consensus only orders references to stored transaction blocks. Examples include Jolteon [15], which stores blocks individually, Narwhal-Hotstuff [12], which arranges blocks in a DAG and allows committing full causal histories at once, or Autobahn [16], which arranges batches in per-node chains to reduce waiting for and transmitting block references. These protocols achieve high throughput because all parties can work to produce and store blocks in parallel and at network speed, and there is no need for erasure-coding and its associated overheads. However, latency suffers as it becomes the sum of the data-dissemination latency plus the consensus latency.

Recent research strives to close the latency gap between the two approaches. DAG-based consensus protocols in the Bullshark [42] family (i.e., Bullshark, Shoal [40], Shoal++ [3], Sailfish [39], etc.) improve latency by interpreting the DAG structure to reach consensus and by introducing timeouts in the DAG. While all parties still build the DAG in parallel, like in previous DAG-based protocols, DAG building is not asynchronous: a rotating leader sets the pace, and DAG formation stalls until a timeout if the leader fails. This approach has successfully reduced the good-case latency of DAG-based protocols while keeping their throughput higher than what leader-based protocols, even when using AVID, can promise. In particular, Sailfish and Shoal++ achieve the theoretical best-case latency lower bound and state-of-the-art throughput.

However, under workloads that do not saturate bandwidth, even the latency of Sailfish remains higher in practice than that achieved by leader-based protocols. This is due to the DAG-formation process, which requires all nodes to wait for enough previous vertices and perform costly vertex-proposal operations that must keep up with the leader's pace.

In this work, we propose Angelfish, a new hybrid protocol that bridges the last latency gap between DAG-based protocols and leader-based protocols, achieving the best of both worlds. In Angelfish, parties build a DAG like in previous DAG-based protocols, but, except for the round leader, parties can abstain from proposing DAG vertices via reliable broadcast (RB) and instead only use the cheaper best-effort broadcast (BEB) primitive to send lightweight vote messages. Like in leader-based protocols, these vote messages contain no transaction blocks and at most one reference.

At one extreme, when only the leader proposes a DAG vertex in a round, Angelfish becomes similar to a leader-based protocol and achieves low latency; at the other extreme, when every party proposes a DAG vertex in a round, Angelfish behaves like the state-of-the-art Sailfish protocol [39] and matches its throughput and latency. Angelfish is able to continuously adapt between those two extremes in order to obtain the best possible latency given the incoming transaction

throughput. This design poses significant algorithmic challenges, but it offers compelling advantages.

Advantage 1: Improved commit latency in practice under moderate load. Theoretically, in the best case Angelfish achieves the optimal leader commit latency of three message delays  $\delta$ , as do Sailfish [39] and Shoal++ [3] (see Table 1 for a comparison of theoretical latency with other major DAGbased protocols). In practice, however, Angelfish achieves lower latency when a portion of DAG vertices are replaced by much cheaper BEB votes, which enables parties to advance rounds faster. Of course, BEB votes contain no transactions and so make no contribution to throughput. However, even in popular production DAG-based systems such as Sui (which uses the Mysticeti protocol [5]), the network often operates under moderate load, with most proposed vertices containing only a single transaction or none at all [1]; with Angelfish, nodes with few transactions to propose can use votes instead of creating DAG vertices, which improves system-wide latency without decreasing throughput.

In Section 5, we evaluate Angelfish [49] experimentally on a geo-distributed testbed with 50 nodes. Compared to Sailfish, Angelfish achieves a reduction in latency of at least 35% (from roughly 0.8 s to 0.5 s) in the moderate-throughput regime (20,000 to 100,000 transactions per second (tps), with 512 byte transactions), under a vertex-proposal rate of 40% (i.e., when 60% of the parties are casting votes instead of creating DAG vertices). This is only about twice the 220 ms latency of our lower-bound baseline: the optimal-latency (3δ) leader-based protocol Hydrangea [38], which we push to the physical limits of the testbed by using empty blocks (and no mempool). Moreover, our experiments confirm that Angelfish's latency gains come at no cost to throughput, as peak throughput matches Sailfish.

In addition, we introduce Multi-leader Angelfish, which supports multiple leaders within the same round and further reduces commit latency in practice. Our evaluation shows that, with 10 leaders, Multi-leader Angelfish [50] achieves a latency of roughly 300 ms at a propose\_rate of 40% while serving 25,000 tps; this latency is only 36% slower than Hydrangea when it orders empty blocks.

The implementation is open-sourced and publicly available. **Advantage 2: Improved communication complexity.** Votes

also incur lower communication complexity than vertices because only a few different votes can be cast in a given round, so we can aggregate them using multi-signatures [2, 20].

The fifth column of Table 1 compares the communication complexity of Angelfish and other DAG-based protocols under the assumption that RBC proceeds optimistically, like in Narwhal [12], incurring only  $O(\kappa n^2)$  complexity. Angelfish achieves a per-round cost of  $O(\kappa xn^2 + n^3)$ , where  $\kappa$  is the security parameter of the cryptographic primitives used, and x is the average number of parties proposing vertices per round (with each vertex containing x references). The  $n^3$  term corre-

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

	Certified DAG	LV Commit Latency	NLV Commit <sup>(1)</sup> Latency	Multiple Leaders	Bits per Round <sup>(2)</sup> in the Good Case	Bits per Round <sup>(3)</sup> in the Bad Case
Bullshark [41,42]	✓	4δ	+2δ	×	$O(\kappa n^3)$	$O(\kappa n^4)$
Shoal [40]	$\checkmark$	4δ	$+2\delta$	✓	$O(\kappa n^3)$	$O(\kappa n^4)$
Shoal++ [3]	$\checkmark$	3δ	$+2\delta$	✓	$O(\kappa n^3)$	$O(\kappa n^4)$
Sailfish [39]	$\checkmark$	3δ	$+2\delta$	✓	$O(\kappa n^3)$	$O(\kappa n^4)$
Cordial Miners [24]	×	3δ	$+3\delta$	×	$O(\kappa n^4)$	$O(\kappa n^4)$
Mysticeti [5]	×	3δ	$+3\delta$	✓	$O(\kappa n^4)$	$O(\kappa n^4)$
Angelfish	$\checkmark$	3δ	$+2\delta$	$\checkmark$	$O(\kappa x n^2 + n^3)$	$O(\kappa x n^3)$

LV stands for leader vertex. NLV stands for non-leader vertex. In Angelfish, x denotes the average number of vertices proposed by honest parties per round. (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) This column lists the communication complexity in the good case when RBC's optimistic path succeeds. We use RBC as in Narwhal [12], whose optimistic case incurs 2 communication steps and  $O(\kappa n^2)$  communication complexity to propagate  $O(\kappa n)$ -sized message, where  $\kappa$  is security parameter. (3) This column lists the communication complexity in the bad case where parties must initiate fetching missing data. This still incurs 2 communication steps to obtain availability certificates but  $O(\kappa n^3)$  communication complexity to deliver  $O(\kappa n)$ -sized message.

sponds to votes and vote certificates using multi-signatures. Note that the best other protocols achieve  $O(\kappa n^3)$ .

The last column of Table 1 shows communication complexity assuming bad-case RBC complexity; in this case, the  $O(n^3)$  complexity of votes is hidden by the cost of RBC of vertices. The per-round communication complexity of other DAG-based protocols reaches  $O(\kappa n^4)$  while Angelfish achieves  $O(\kappa x n^3)$ . This is the case even in Cordial Miners [24] and Mysticeti [5], where parties disseminate vertices using BEB, and to ensure liveness, each party must forward all the vertices it receives, as highlighted by Starfish [35].

Advantage 3: Efficient catch-up for lagging nodes. In most DAG-based protocols [3, 19, 39, 41], one condition for a party to advance to a new round is receiving at least n-f vertices, and parties typically progress sequentially through rounds. To mitigate delays, protocols such as Bullshark [41] and Sailfish [39] allow lagging parties to skip rounds and jump to a future round if they can obtain n-f vertices from that round. However, in practice, honest parties running on slow machines may fail to collect these n-f vertices in time, causing them to stall and preventing them from proposing new vertices. Such parties may then be perceived by others as crashed, effectively reducing the fault tolerance. In Angelfish, by contrast, a lagging non-leader party can catch up to a higher round with only f + 1 messages in total—vertices and votes combined—and can still participate by sending simple vote messages. This enables slow parties to catch up more easily, continue participating in consensus, and contribute to system security and fault tolerance.

**Roadmap.** We start in Section 2 by giving a high-level technical overview of Angelfish and the challenges in its design, followed by an in-depth presentation in Section 3 and a high-level presentation of Multi-leader Angelfish in Section 4. Section 5 presents empirical results and evaluation of both of

these protocols, pitching them against two state-of-the-art protocols, Sailfish [39] and Jolteon [15]. Finally, we discuss related work in Section 6. A detailed presentation of Multileader Angelfish appears in Section B, rigorous security analyses of both protocols appear in Sections C and D.

#### 2 Technical Overview

In this section, we first review technical background on DAGbased protocol and Sailfish in particular, we present the key idea to reduce latency and the challenges towards its realization, and we finally present our solution.

**DAG-Based BFT Consensus Protocols.** In DAG-based consensus protocols, each party creates and disseminates so-called vertices, where each contains a transaction block and cryptographic hashes providing immutable references to other vertices. This process creates a growing directed acyclic graph (DAG) where each vertex determines the whole subgraph reachable from it (called its causal history). Each party then forms its local, partial view of the DAG as the largest set of received vertices such that no references are missing.

To allow parties to order the vertices of their local DAGs in a consistent way, the protocol repeatedly commits DAG vertices, with the guarantee that any two committed DAG vertices (even if committed by two different parties) are connected (by a directed path) in the DAG. This ensures that each party's local DAG admits a total order, constructed by deterministically ordering the causal history of successive committed vertices, yielding a prefix of the same global total ordering of the DAG.

Different protocols follow this blueprint in different ways. Most protocols build the DAG in rounds where each party creates one vertex per round including references to at least n-f vertices from the previous round (where n is the number

of parties and  $f < \frac{n}{3}$  the maximum number of faulty parties). This creates a so-called layered DAG and allows quorum-intersection arguments in the DAG structure to ensure that committed leader vertices are connected.

Among protocols using a layered DAG, protocols like DAG-Rider [21] or Narwhal-Tusk [12] let parties create vertices as fast as the network allows (DAG formation is asynchronous); others, like Bullshark [41] and Sailfish [39], improve common-case latency by requiring parties to wait for their round's leader vertex until they time out (DAG formation is partially synchronous).

Finally, many protocols [3, 21, 39–41] choose to use Reliable Broadcast or Consistent Broadcast to disseminate vertices and corresponding availability certificates, yielding so-called certified DAGs. This ensures that parties can progress from one round to the next without blocking on data retrieval as long as they obtain an availability certificate. In practice, this has been shown crucial to obtaining robust performance under even moderate network instability [3].

Our starting point to design Angelfish is thus the state-of-the-art protocol Sailfish [39]. Sailfish uses a partially-synchronous certified DAG, achieves theoretically optimal best-case latency, and can commit a new leader vertice every round. It is on this solid foundation that we endeavor to improve latency even further.

**The key idea.** In certified DAGs like Sailfish, each party must reliably broadcast large vertices (containing transactions and at least n-f references) and wait to deliver at least n-f vertices before entering a new round. This is a major source of latency compared to classic leader-based protocols.

Angelfish reduces this overhead by allowing non-leader parties to avoid creating DAG vertices and instead cast lightweight votes that either reference only the leader vertex or contain no references at all, and that are disseminated using best-effort broadcast. Since there may be fewer than n-f vertices created per round, Angelfish also dispenses with the requirement that DAG vertices must contain n-f references to vertices from the previous round.

Since some parties may use votes instead of creating DAG vertices, Angelfish modifies the leader commit rule of Sailfish to count both votes and DAG vertices: a leader vertex is committed when the number of votes for it plus the number of DAG edges pointing to it from the next round is at least n-f.

The key technical challenge. Naively removing the requirement of n-f references in Sailfish makes the protocol unsafe. To understand why, let us briefly explain the relevant aspects of Sailfish. In Sailfish, each round, a rotating leader creates a so-called leader vertex that may possibly be committed. A leader vertex is committed when n-f parties create vertices in the next round that reference it, and the leader vertex of the next round must reference it unless it has proof it cannot be committed; together with the condition that each vertex references at least n-f vertices from the previous round and

that n > 3f, a standard quorum-intersection argument ensures that every two committed leader vertices are connected by a directed path even if there are rounds with no committed leader vertex in between.

By allowing parties to use lightweight votes that may only contain one or no references to previous vertices, we cannot invoke the quorum-intersection argument and, if there are rounds where no leader vertex is committed, we lose the guarantee that committed leader vertices are connected. Finally, without guaranteed paths between committed leader vertices, locally linearized DAGs may fork.

To overcome this challenge, Angelfish introduces special leader edges in the DAG, i.e., direct links between leader vertices that may be more than one round apart, together with new timeout-certificate requirements to guarantee safety across rounds. Getting this right is the main algorithmic contribution of the paper. The next paragraphs give an overview of the techniques we employ.

Ensuring paths between leader vertices. Like Sailfish, Angelfish uses a rotating round leader, and it requires the round leader to create a vertex (and not send a vote instead). To ensure that every two committed leader vertices are connected, if a round r leader vertex  $v_r$  does not reference round r-1 leader vertex, it must instead include a leader edge to some prior leader vertex  $v_{r'}$  from a round r' < r-1, along with TCs (timeout certificates) for rounds from r'+1 to r-1 showing that enough honest parties did not vote for any leader vertex between  $v_r$  and  $v_{r'}$ . This ensures that the skipped leaders cannot be committed; hence, it is safe for  $v_r$  to connect to  $v_{r'}$  while bypassing the intermediate leader vertices. Thanks to using reliable broadcast for vertices, for each round we are guaranteed to eventually receive the leader vertex or a timeout certificate, which ensures leaders can make progress.

Pacing round progression. Parties should now increment their round upon hearing from n-f parties in the current round (if they have the leader vertex), whether through votes or vertices, since they cannot wait for more when up to f parties may be faulty. Consequently, some DAG vertices may be delivered too late compared to votes, and thus vertices in the next round may not reference them. As in many DAG protocols (e.g. DAG-Rider [21]), we use special weak edges to ensure those vertices eventually get included. However, this substantially increases latency for those vertices. Angelfish therefore introduces propose flags that can be included in messages to indicate that the sender plans to propose a vertex in the next round. Parties in the next round then wait for vertices from those parties (minus f of them to accommodate failures). While we chose this propose-flag design, another option is to follow Shoal++ [3] and add a small timeout to wait for more vertices before leaving a round. Finally, because we do not require n - f references per vertex or vote, lagging nodes can catch up upon receiving only f + 1 messages from a round higher than their next round, versus n - f in most other DAG-based protocols.

Reducing communication with vote certificates. To avoid the need to forward all votes to promote synchronized round progression (since votes do not use RBC), Angelfish relies on parties that complete a round to disseminate certificates summarizing the number of votes they received. We use multisignatures to keep bit complexity low, which must be done carefully since there are different possible votes in each round.

The cherry on top: multiple leaders per round. To further reduce average latency, Angelfish supports multiple leaders within a single round. Each round, on top of the existing main leader, we use a predetermined list of secondary leaders; we require a main leader that has a leader edge to another main leader of a round r to include, for each secondary leader of r, either an edge or a certificate proving that the secondary leader cannot commit. Secondary leaders that did not time out are then linearized in order and immediately following the main leader. As explained in the Sailfish paper [39], there are multiple possible strategies to handle secondary leaders that may not commit, depending on how much waiting for secondary leaders is acceptable. To balance waiting and including more secondary leaders, we choose to implicitly skip any secondary leader after the first one that fails to commit.

# 3 Angelfish in Depth

## 3.1 Preliminaries

We consider a system  $\mathcal{P} := P_1, \dots, P_n$  consisting of n parties out of which up to  $f < \frac{n}{3}$  parties can be Byzantine, meaning they can behave arbitrarily. A party that is not faulty throughout the execution is said *honest* and follows the protocol.

We consider the partial synchrony model of Dwork et al. [13]. In this model, the network initially operates in an asynchronous state, allowing the adversary to arbitrarily delay messages sent by honest parties. However, after an unknown point in time, known as the *Global Stabilization Time* (GST), the adversary must ensure that all messages from honest parties are delivered to their intended recipients within  $\Delta$  time of being sent (where  $\Delta$  is known to the parties). We use  $\delta$  for the actual message delay after GST, noting that  $\delta \leq \Delta$ . 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 authenticate messages and prevent spoofing or replay attacks. We use  $\langle x \rangle_i$  to denote a message x digitally signed by party  $P_i$  using its private key.

We assume that each party receives transactions from external clients and periodically groups them into blocks that it submits to a Byzantine atomic broadcast service, implemented by Angelfish, that delivers blocks in the same total order at all parties.

**Definition 3.1** (Byzantine atomic broadcast [22, 41]). Each honest party  $P_i \in \mathcal{P}$  can call a\_bcast<sub>i</sub>(b) to submit a block b. Each party  $P_i$  may then outputs a\_deliver<sub>i</sub>(b,  $P_k$ ), where  $P_k \in \mathcal{P}$  represents the block creator. A Byzantine atomic broadcast protocol must satisfy the following properties:

- **Agreement.** If an honest party  $P_i$  outputs a\_deliver $_i(b, P_k)$ , then every other honest party  $P_j$  eventually outputs a\_deliver $_i(b, P_k)$ .
- **Integrity.** For every block b and party  $P_k \in \mathcal{P}$ , an honest party  $P_i$  outputs a\_deliver<sub>i</sub> $(b, P_k)$  at most once and, if  $P_k$  is honest, only if  $P_k$  previously called a\_bcast<sub>k</sub>(b).
- **Validity.** If an honest party  $P_k$  calls a\_bcast $_k(b)$  then every honest party eventually outputs a\_deliver $(b, P_k)$ .
- **Total order.** If an honest party  $P_i$  outputs a\_deliver $_i(b, P_k)$  before a\_deliver $_i(b', P_\ell)$ , then no honest party  $P_j$  outputs a\_deliver $_j(b', P_\ell)$  before a\_deliver $_j(b, P_k)$ .

The definition for RBC can be found in Section A.

# 3.2 Protocol Description

**Round-based execution.** Our protocol advances through a sequence of consecutively numbered *rounds*, starting from 1. In each round r, a designated leader, denoted as  $L_r$ , is selected using a deterministic method based on the round number.

Basic data structures. At a high level, the communication among parties is structured as a combination of a DAG and vote messages. In each round, a leader proposes a single vertex containing a (possibly empty) block of transactions along with references to vertices proposed in an earlier round. Additionally, the leader vertex references a leader vertex from some previous round. Meanwhile, In each round, a non-leader party either proposes a vertex—if it has transactions to include—or sends a vote. A vertex includes a block of transactions and references to earlier vertices, while a vote references the previous round's leader vertex or contains no reference. The references of vertices serve as the edges in the DAG. All proposed vertices are propagated using RBC to ensure non-equivocation and guarantee all honest parties eventually deliver them. The vote messages are propagated using BEB.

The basic data structures and utilities of Angelfish are presented in Figure 1, with our modifications over Sailfish [39] highlighted in blue. Each party maintains a local copy of the DAG, and while different honest parties may initially have distinct views of it, the reliable broadcast of vertices ensures eventual convergence to a consistent DAG view. The local DAG view for party  $P_i$  is represented as  $DAG_i$ . Each vertex is uniquely identified by a round number and a sender (source). When  $P_i$  delivers a vertex from round r, it is added to  $DAG_i[r]$ , where  $DAG_i[r]$  can contain up to n vertices.

A vertex proposed by  $L_r$  is termed the round r leader vertex, while all other round r vertices are classified as non-leader

```
Local variables:
                                                                                                                                                                                                                                   > The struct of a vertex in the DAG
            struct vertex v:
                        v.round - the round of v in the DAG
                        v.source - the party that broadcasts v
                        v.block - a block of transactions
                        v.propose - if the party will propose a vertex in the next round
                        v.strongEdges - a set of vertices in v.round - 1 that represent strong edges
                        v.weakEdges - a set of vertices in rounds < v.round - 1 that represent weak edges
                        v.leaderEdge - a leader vertex in round < v.round - 1 that represent leader edge (for leader vertices)
                        v.tc - a set of timeout certificates for round \leq v.round - 1 (if any)
                                                                                                                                                                                                                                                             ▶ The struct of a vote
           struct vote vt:
                        vt.round - the round of vt
                        vt.source - the party that broadcasts vt
                        vt.sig - signature of vt
                        vt.propose - if the party will propose a vertex in the next round
                        vt.strongEdge - leader vertex in vt.round - 1 that the party votes for; otherwise, \bot
           struct propose vote certificate \mathcal{PVC}:
                                                                                                                                                        ▶ The struct of aggregated BLS signatures from votes with propose = true
                        \mathcal{PVC}.sig - a set of signatures of \mathcal{PVC}
           struct no-propose vote certificate \mathcal{NPVC}:
                                                                                                                                                       ▶ The struct of aggregated BLS signatures from votes with propose = false
                        \mathcal{NPVC}.sig - a set of signatures of \mathcal{NPVC}
           struct commit vote certificate \mathcal{CVC}:
                                                                                                                                         ▶ The struct of aggregated BLS signatures from votes referencing the leader vertex
                        CVC.sig - a set of signatures of CVC
            timeout certificate \mathcal{TC}- An array of sets, each containing a quorum of timeout messages
            DAG_i — An array of sets of vertices
            votes [ - An array of sets of votes
           sigPropose | - An array of sets of signatures of parties that propose vt and will propose a vertex in the next round
           sigNoPropose[] — An array of sets of signatures of parties that propose vt and will not propose a vertex in the next round
            sigCommit[] An array of sets of signatures of parties that propose vt and vote for leader vertex in vt.round-1
            vertexProposers [] - An array of sets of proposers that propose vertices
           blocksToPropose - A queue, initially empty, P_i enqueues valid blocks of transactions from clients
           leaderStack \leftarrow initialize empty stack
 1: procedure path(v, u)
                                                                                                                                               ▷ Check if exists a path consisting of strong, weak and leader 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, \text{ and } \forall j \in [2,..,k] : v_j \in \bigcup_{r > 1} DAG_i[r] \land (v_j \in v_{j-1}.weakEdges \cup v_{j-1}.strongEdges \cup v_{j-1}.leaderEdge)
 3: procedure leader_path(v, u)
                                                                     \triangleright Check if exists a path consisting of leader edges and strong edges from leader vertex v to leader vertex u in the DAG
              return exists a sequence of k \in \mathbb{N}, leader vertices v_1, \dots, v_k s.t.
                            v_1 = v, v_k = u, and \forall j \in [2,..,k] : v_j \in \bigcup_{r > 1} DAG_i[r] \land v_j \in v_{j-1}.leaderEdge \cup v_{j-1}.strongEdges
                                                                                                                                                         18: procedure broadcast_vertex(r)
                                                                                                                                                         19:
                                                                                                                                                                        v \leftarrow \text{create\_new\_vertex}(r)
5: procedure set_weak_edges(v, r)

    Add edges to orphan vertices

                                                                                                                                                         20:
                                                                                                                                                                        try_add_to_dag(v)
            v.weakEdges \leftarrow \{\}
6:
                                                                                                                                                         21:
                                                                                                                                                                        r_bcast_i(v, r)
7:
            for r' = r - 2 down to 1 do
                                                                                                                                                         22: procedure multicast_vote(r)
                   for every u \in DAG_i[r'] s.t. \neg path(v, u) do
8.
                                                                                                                                                         23.
                                                                                                                                                                         \langle vt, r \rangle_i \leftarrow \text{create\_new\_vote}(r)
9:
                           v.weakEdges \leftarrow v.weakEdges \cup \{u\}
                                                                                                                                                                        votes[r] \leftarrow votes[r] \cup \{vt\}
                                                                                                                                                         24.
10: procedure get_vertex(p,r)
                                                                                                                                                         25:
                                                                                                                                                                        multicast \langle vt, r \rangle_i
               if \exists v \in DAG_i[r] s.t. v.source = p then
11:
                                                                                                                                                         26: procedure order_vertices()
12:
                      return v
                                                                                                                                                         27:
                                                                                                                                                                        while ¬leaderStack.isEmpty() do
13:
               return 丄
                                                                                                                                                                               v \leftarrow leaderStack.pop()
                                                                                                                                                         28:
14: procedure get_leader_vertex(r)
                                                                                                                                                         29:
                                                                                                                                                                               verticesToDeliver \leftarrow \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \notin \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \notin \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \notin \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \notin \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \notin \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \notin \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \notin \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \notin \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \notin \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \notin \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \notin \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \notin \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \notin \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land v' \in \{v'
15:
               return get_vertex(L_r, r)
                                                                                                                                                             deliveredVertices}
                                                                                                                                                         30:
                                                                                                                                                                               for every v' \in verticesToDeliver in some deterministic order
16: procedure a\_bcast_i(b, r)
                                                                                                                                                             do
17:
               blocksToPropose.enqueue(b)
                                                                                                                                                                                      output a_deliver_i(v'.block, v'.round, v'.source)
                                                                                                                                                         31:
                                                                                                                                                         32:
                                                                                                                                                                                      deliveredVertices \leftarrow deliveredVertices \cup \{v'\}
```

Figure 1: Basic data structures for Angelfish. The utility functions are adapted from [39,41].

vertices. Each non-leader vertex v contains two types of outgoing edges: strong edges and weak edges. In addition to strong edges and weak edges, a leader vertex also contains a leader edge. The strong edges of a vertex v in round r connect to vertices from round r-1, while the weak edges link to at most n-f vertices from earlier rounds < r-1 for which

no other path from v exists. A path is a sequence of strong, weak, or leader edges that connects two vertices. For a round r leader vertex, a leader edge references a leader vertex from some round < r - 1. A leader path is a path from a leader vertex  $v_k$  to another leader vertex  $v_\ell$  such that all intermediate vertices along the path are leader vertices, and each edge in

```
Local variables:
       round \leftarrow 1; buffer \leftarrow \{\}
33: upon r_deliver_i(v, r, p) do
                                                                                                     77: procedure try_add_to_dag(v)
          if v.source = p \land v.round = r \land (v.source = L_r \land is\_valid(v)) then
34:
                                                                                                               if \forall v' \in v.srongEdges \cup v.weakEdges \cup v.leaderEdges : v' \in
35:
               if v.propose = true then
                                                                                                        \bigcup_{k\geq 1} \textit{DAG}_i[k] then
36:
                    VertexProposers[r] \leftarrow VertexProposers[r] \cup \{p\}
                                                                                                     79:
                                                                                                                    DAG_i[v.round] \leftarrow DAG_i[v.round] \cup \{v\}
37:
               if \neg try\_add\_to\_dag(v) then
                                                                                                     80:
                                                                                                                    buffer \leftarrow buffer \setminus \{v\}
38:
                   buffer \leftarrow buffer \cup \{v\}
                                                                                                     81:
                                                                                                                    return true
39.
               else
                                                                                                     82:
                                                                                                                return false
40:
                   for v' \in buffer : v'.round > r do
                                                                                                     83: procedure create_new_vertex(r)
41:
                        try_add_to_dag(v')
                                                                                                     84:
                                                                                                               v.round \leftarrow r
42:
                                                                                                     85:
                                                                                                               v.source \leftarrow P_i
     upon receiving \langle vt, r \rangle_p do
                                                                                                     86:
                                                                                                               v.block \leftarrow blocksToPropose.dequeue()
43:
          if vt.round = r then
                                                                                                     87:
                                                                                                               v.propose \leftarrow \text{true if } P_i \text{ intends to propose vertex in round } r+1 \text{ else}
44.
               votes[r] \leftarrow votes[r] \cup \{vt\}
45:
               if vt.propose = true then
                                                                                                        false
                                                                                                     88.
                   VertexProposers[r] \leftarrow VertexProposers[r] \cup \{p\}
46:
                                                                                                                    upon |DAG_i[r-1]| \ge \max(|VertexProposers[r-2]| - f, 0) do
47:
                   sigPropose[r] \leftarrow sigPropose[r] \cup \{vt.sig\}
                                                                                                     89:
48:
                                                                                                     90:
                                                                                                                         v.strongEdges \leftarrow DAG_i[r-1]
49:
                   sigNoPropose[r] \leftarrow sigNoPropose[r] \cup \{vt.sig\}
                                                                                                     91:
                                                                                                                    if \langle \text{timeout}, r-1 \rangle_i is sent then
                                                                                                     92:
50:
                                                                                                                         v.strongEdges \leftarrow v.strongEdges \setminus \{v' : v'.source = L_{r-1}\}
          if \exists v' \in DAG_i[r-1] : v'.source = L_{r-1} \land vt.strongEdge = L_{r-1} then
51:
               sigCommit[r] \leftarrow vt.sig
                                                                                                     93:
                                                                                                               if P_i = L_r \land \not\exists v' \in DAG_i[r-1] : v'.source = L_{r-1} then
                                                                                                     94:
                                                                                                                    for r' = r down to 1 do
52:
     upon receiving \mathcal{PVC}_r^p do
                                                                                                     95:
                                                                                                                         if \exists v'' \in v.strongEdges s.t. v''.source = L_{r'-1} then
53:
          sigPropose[r] \leftarrow sigPropose[r] \cup \{ \mathcal{PVC}_{p,r}.sig \}
                                                                                                     96:
                                                                                                                             v.tc \leftarrow v.tc \cup \{TC_{r'-1}\}
54:
          VertexProposers[r] \leftarrow VertexProposers[r] \cup \{p\}
                                                                                                     97.
55: upon receiving \mathcal{NPVC}_r^p do
                                                                                                     98:
                                                                                                                              v^* \leftarrow \text{get\_leader\_vertex}(r')
          \textit{sigNoPropose}[r] \leftarrow \textit{sigNoPropose}[r] \cup \{\mathcal{NPVC}_r^p.sig\}
56:
                                                                                                     99.
                                                                                                                              v.leaderEdge \leftarrow v^*
                                                                                                     100:
                                                                                                                               break
57: upon receiving CVC_{p,r} do
                                                                                                     101:
                                                                                                                 set_weak_edges(v, r)
58:
          sigCommit[r] \leftarrow sigCommit[r] \cup \{CVC_{p,r}.sig\}
                                                                                                     102:
                                                                                                                 return v
59: upon |sigPropose[r]| + |sigNoPropose[r]| + |DAG_i[r]| > n - f \land (\exists v' \in a)
                                                                                                     103: procedure create_new_vote(r)
  DAG_i[r]: v'.source = L_r \vee TC_r is received for round = r) do
                                                                                                     104
                                                                                                                 vt.round \leftarrow r
60:
          if |DAG_i[r]| < n - f then
                                                                                                     105:
                                                                                                                 vt.source \leftarrow P_i
61:
               \mathcal{PVC}_r^i \leftarrow \text{AggregateBLS}(sigPropose[r])
                                                                                                     106:
                                                                                                                 vt.propose \leftarrow true if P_i intends to propose vertex in round r+1
               \mathcal{NPVC}_r^i \leftarrow \text{AggregateBLS}(sigNoPropose[r])
                                                                                                        else false
62:
                                                                                                      107:
               multicast \mathcal{PVC}_r^i and \mathcal{NPVC}_r^i
                                                                                                                 if \langle \text{timeout}, r-1 \rangle_i is sent then
63:
                                                                                                     108:
                                                                                                                      vt.strongEdge \leftarrow \bot
          if P_i = L_{r+1} \land \nexists v' \in DAG_i[r] : v'.source = L_r then
64:
                                                                                                     109:
               wait until receiving \mathcal{T}\mathcal{C}_r
65:
                                                                                                     110:
                                                                                                                      if r > 1 \land \exists v' \in DAG_i[r-1] : v'.source = L_{r-1} then
66:
               for r' = r - 1 down to 1 do
                                                                                                     111:
                                                                                                                          vt.strongEdge \leftarrow v'
                   if \exists v'' \in DAG_i[r'] : v''.source = L'_r then
67:
                                                                                                     112.
                                                                                                                 return \langle vt, r \rangle_i
68:
                        break
69:
                                                                                                     113: procedure advance_round(r)
70:
                        wait until receiving \mathcal{T}\mathcal{C}_{r'}
                                                                                                     114:
                                                                                                                 round \leftarrow r; start\ timer
71:
          advance\_round(r+1)
                                                                                                     115:
                                                                                                                 if P_i = L_r \vee P_i intends to propose vertex in round r then
                                                                                                     116:
                                                                                                                      broadcast_vertex(round)
72: upon timeout do
                                                                                                     117:
73:
          if \exists v' \in DAG_i[round] : v'.source = L_{round} then
                                                                                                     118:
                                                                                                                      multicast_vote(round)
74:
               multicast \langle timeout, round \rangle_i
                                                                                                     119: upon init do
75: upon receiving \mathcal{TC}_r such that r > round do
                                                                                                     120:
                                                                                                                 advance_round(1)
          multicast TC_r
```

Figure 2: Angelfish: DAG construction protocol for party  $P_i$ 

the path is either a leader edge or a strong edge. Each vertex also contains v.tc, which stores a set of timeout certificates (each  $\mathcal{TC}$  consisting of a quorum of timeout messages in a round). Different from Bullshark [41] and Sailfish [39], Angelfish introduces a new field v.propose, which indicates whether the party intends to propose a vertex in the next round. This information helps determine how many vertices a party should wait for in the next round.

In addition to vertex v, Angelfish introduces a separate vote structure vt. A vote includes a strong edge only when

it supports the leader vertex from the previous round; otherwise, it carries no reference. Each vote also contains a field *vt.propose*, analogous to *v.propose* in vertices, which similarly indicates whether the party intends to propose a vertex in the next round.

**DAG** construction protocol. The DAG construction protocol is outlined in Figure 2. In each round r, a leader proposes a vertex and a non-leader party  $P_i$  proposes a vertex v or a vote message vt, depending on whether it has transactions to pro-

pose and whether doing so is consistent with the *propose* flags (i.e., v.propose and vt.propose) in round r-1. To propose a vertex or a vote in round r, each party  $P_i$  waits to deliver a round r-1 vertices and receive b votes for round r-1, such that  $a+b \ge n-f$ , and includes the round r-1 leader vertex, until a timeout occurs in round r-1 (see Line 59). Here, votes are counted jointly from two sources: (i) received vote messages, and (ii) signatures extracted from valid vote certificates. We distinguish two types of vote certificates according to the vote's propose flag: a propose vote certificate  $(\mathcal{PVC})$  aggregates vote messages with propose = true, while a no-propose vote certificate ( $\mathcal{NPVC}$ ) aggregates vote messages with propose = false. After satisfying the round-entry condition, if votes were used to enter the round,  $P_i$  aggregates signatures from the corresponding vote messages and vote certificates to construct  $\mathcal{PVC}_r^i$  and  $\mathcal{NPVC}_r^i$ , and then broadcasts them.

After  $P_i$  successfully advance to round r by delivering round r-1 leader vertex along with at least n-f vertices and votes, if  $P_i$  intends to propose a vote message, it immediately broadcasts  $\langle vt, r \rangle_i$ . If it intends to propose a round r vertex, it first computes the number of parties that are expected to propose in round r-1, based on the *propose* flag included in round r-2 vertices. Once the number of delivered round r-1 vertices reaches this expected count minus f, it proceeds to propose a round r vertex (see Line 89). The "-f" margin is designed to prevent faulty parties from equivocating by assigning conflicting *propose* flags to different vertices. Notably, referencing the round r-1 leader vertex functions as a "vote" for it. These votes are subsequently used to commit the leader vertex. Therefore, waiting for the leader vertex until a timeout ensures that honest parties contribute votes toward it, helping to commit the leader vertex with minimal latency when the leader is honest (after GST).

If an honest party  $P_i$  does not receive the round r-1 leader vertex when its timer expires, it multicasts  $\langle \texttt{timeout}, r-1 \rangle$  to all parties (see Line 72). Upon receiving n-f round r-1 timeout messages (denoted by  $\mathcal{T}\mathcal{C}_{r-1}$ ),  $P_i$  can enter round r if it has delivered a round r-1 vertices and received b votes with the total number a+b being at least n-f (see Line 59). It also multicasts  $\mathcal{T}\mathcal{C}_{r-1}$  so that all the other parties will see  $\mathcal{T}\mathcal{C}_{r-1}$ . In Angelfish, we require that if a party sends a round r-1 timeout message, the round r vertex or vote it sends will not include a strong edge to round r-1 leader vertex, which serves as a "no-vote" for the round r-1 leader vertex.

We place an additional constraint on the leader vertex. A round r leader vertex needs to either (1) have a strong edge to the round r-1 leader vertex, or (2) include a leader edge referencing a leader vertex from some earlier round r' < r-1, and contain timeout certificate  $\mathcal{TC}_{r''}$  for each round r'' such that  $r'+1 \leq r'' \leq r-1$ . Each  $\mathcal{TC}_{r''}$  serves as an evidence that a quorum of parties did not "vote" for the round r'' leader vertex. Hence, the round r'' leader vertex cannot be committed and it is safe to lack a leader path to the round r'' leader vertex.

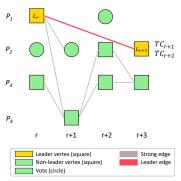


Figure 3: This represents the set of local vertices and votes observed by  $P_2$ .  $L_{r+3}$  did not deliver round r+1 and round r+2 leader vertices; therefore,  $L_{r+3}$ 's vertex must contain  $\mathcal{TC}_{r+1}$  and  $\mathcal{TC}_{r+2}$ .

An illustration is provided in Figure 3.

Upon delivering a round r vertex v, if v is a leader vertex, each party  $P_i$  verifies its validity by invoking is\_valid(v). The function is valid(v) returns true if v contains a leader edge referencing a round r' leader vertex, and includes timeout certificate  $\mathcal{TC}_{r''}$  for each round r'' such that  $r'+1 \leq$  $r'' \le r - 1$ . Only valid vertices v are added to  $DAG_i[r]$  via try\_add\_to\_dag(v). For non-leader vertices, this validity check is not required; they are directly passed to try\_add\_to\_dag(v), which succeeds once  $P_i$  has delivered all vertices that establish a path from v in  $DAG_i$ . If try\_add\_to\_dag(v) fails, the vertex is placed in a *buffer* for later reattempts. Additionally, when try\_add\_to\_dag(v) succeeds, any buffered vertices are retried for inclusion in DAGi. In parallel, upon receiving a vote message  $\langle vt, r \rangle_*$ , the party  $P_i$  classifies its signature into different arrays depending on (i) whether the propose flag is set to true, and (ii) whether the message references a leader vertex. Upon receiving a vote certificate,  $P_i$  first verifies the signatures and then stores them in the corresponding array. Once  $P_i$  enters round r using votes, it aggregates the signatures from the propose and no-propose arrays to form  $\mathcal{PVC}_r^l$ and  $\mathcal{NPVC}_r^i$ , which it then broadcasts so that other parties can also receive these signatures. Moreover, since vertices are broadcasted via RBC, once a vertex is delivered by an honest party, all other parties will eventually deliver it as well. Therefore, once an honest party has collected n-f votes and vertices, all other parties will eventually receive enough votes and vertices to enter the next round.

**Jumping rounds.** In addition to advancing rounds sequentially, our protocol allows honest parties in round r' < r to "jump" directly to round r+1. This occurs when they either observe at least f+1 round r vertices and votes, including the round r leader vertex, or receive a  $\mathcal{TC}_r$  (see Section 3.2 in Figure 4). We require only f+1 vertices and votes, rather than n-f, because this already ensures that at least one honest party has advanced to round r+1. If the lagging party is  $L_{r+1}$ , then in addition to collecting f+1 round r vertices and votes, it must also wait until it has delivered either the round

```
121: upon |sigPropose[r]| + |sigNoPropose[r]| + |DAG_i[r]| \ge f + 1 \land
  (\exists v' \in DAG_i[r] : v'.source = L_r \lor TC_r \text{ is received for } r > round) do
122:
           if P_i = L_{r+1} \land \nexists v' \in DAG_i[r] : v'.source = L_r then
                wait until receiving TC_r
123:
124:
                for r' = r - 1 down to 1 do
                    if \exists v'' \in DAG_i[r'] : v''.source = L'_r then
125:
126:
                        break
127:
                         wait until receiving TC_{r'}
128:
129:
           advance\_round(r+1)
130: procedure create_new_vertex(r)
131:
           v.round \leftarrow r
           v.source \leftarrow P.
132:
           v.block \leftarrow blocksToPropose.dequeue()
133:
134:
           v.propose \leftarrow \text{true if } P_i \text{ intends to propose vertex in round } r+1
  else false
135:
           if r > 1 then
                v.strongEdges \leftarrow DAG_i[r-1]
136:
               if \langle \text{timeout}, r-1 \rangle_i is sent then
137:
                    v.strongEdges \leftarrow v.strongEdges \setminus \{v' : v'.source = \}
138:
  L_{r-1}
139:
           if P_i = L_r \land \not\exists v' \in DAG_i[r-1] : v'.source = L_{r-1} then
140:
               for r' = r down to 1 do
                    if \exists v'' \in v.strongEdges s.t. v''.source = L_{r'-1} then
141:
142:
                        v.tc \leftarrow v.tc \cup \{TC_{r'-1}\}
143:
144.
                        v^* \leftarrow \text{get\_leader\_vertex}(r')
145:
                        v.leaderEdge \leftarrow v^*
146:
                        break
           set_weak_edges(v, r)
147:
148:
           return v
```

Figure 4: Angelfish: Jumping rounds

r leader vertex or some earlier leader vertex (say round r'), together with the corresponding T Cs for rounds between r' and r+1, before proposing the round r+1 leader vertex. When jumping from round r' to round r+1, parties neither propose vertices nor send votes for the skipped rounds. In particular, they do not need to wait for the condition of observing the number of vertex proposals minus f before proposing a round r+1 vertex. This design allows slow parties to catch up more efficiently.

Angelfish has a better jumping mechanism compared to Sailfish [39], since Sailfish requires delivering the leader vertex (or receiving a  $\mathcal{TC}$ ) together with n-f vertices to jump rounds, whereas Angelfish requires only the leader vertex (or  $\mathcal{TC}$ ) plus f+1 vertices and votes. This makes slow parties in Angelfish more likely to catch up than in Sailfish. Besides, when a slow party enters a new round without downloading transactions, it can send a vote message instead of proposing an empty block, thereby reducing communication overhead.

Committing and ordering the DAG. In our protocol, only leader vertices are committed, while non-leader vertices are ordered in a deterministic manner as part of the causal history of a leader vertex. This ordering occurs when the leader vertex is (directly or indirectly) committed, as defined in the order\_vertices function.

The commit rule is presented in Figure 5. An honest party

```
Local variables:
       committedRound \leftarrow 0
149: upon receiving a set W of first messages for round r+1 ver-
  tices s.t. \forall v' \in \mathcal{W} : \exists v \in v'.strongEdges s.t. v.source = L_r \land |\mathcal{W}| +
  |sigCommit[r]| \ge n - f do
           try\_commit(r, W, sigCommit[r])
151: procedure try_commit(r, W, S)
152:
           v \leftarrow \text{get leader } \text{vertex}(r)
153:
           if committedRound < r then
154:
                commit_leader(v)
155
                if |\{v' \in \mathcal{W} \mid strong\_edge(v', v)\}| < n - f then
156:
                    \mathcal{CVC}_r^i \leftarrow \text{AggregateBLS}(\mathcal{S})
157:
                    multicast \mathcal{CVC}_r^i
158:
      procedure commit_leader(v)
           leaderStack.push(v)
159:
160:
           r \leftarrow v.round - 1
           v' \leftarrow v
161:
162:
           while r > committedRound do
163:
                v_s \leftarrow \text{get\_leader\_vertex}(r)
164.
                if leader_path(v', v_s) then
                    leaderStack.push(v_s)
165:
166:
                    v' \leftarrow v_s
167:
                r \leftarrow r - 1
168
           committedRound \leftarrow v.round
169:
           order_vertices()
```

Figure 5: Angelfish: The commit rule for party  $P_i$ 

 $P_i$  commits a round r leader vertex once it observes n-f vertices and votes in total, including both (i) first RBC messages from round r+1 vertices that reference the round r leader vertex, and (ii) votes, including both vote messages that support  $v_{\ell}$  and the commit vote certificate  $\mathcal{CVC}_{r+1}$ , which is a collection of signatures from vote messages supporting  $v_{\ell}$ .  $P_i$ does not need to wait for the delivery of round r+1 vertices. 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 same value that will ultimately be delivered. Among the n-f round r+1 vertices and votes, at least f+1 originate from honest parties and will eventually be delivered, ensuring that the delivered value matches the first received value (from the first RBC message). Furthermore, since at least f+1 honest parties send round r+1 vertices and votes with a strong edge to the round r leader vertex, it is impossible to form a  $\mathcal{TC}_r$ . Consequently, the round r' > r leader vertex (if it exists) will necessarily have a leader path to the round r leader vertex, ensuring the safety of the commit. Similar to propose vote certificates, if commitment is achieved using votes, then  $P_i$  aggregates signatures from the relevant vote messages and commit vote certificates to generate its own commit vote certificate, and then broadcasts it to help other parties commit.

When directly committing a round r vertex  $v_k$  in round r,  $P_i$  traces back through earlier rounds to *indirectly* commit leader vertices  $v_m$  if a leader path exists from  $v_k$  to  $v_m$ . This process continues until  $P_i$  encounters a round r' < r where it has already directly committed a leader vertex. Our protocol ensures that once any honest party directly commits a round

r leader vertex  $v_k$ , all leader vertices of subsequent rounds r' > r will have a leader path to  $v_k$ . This guarantees that  $v_k$  will eventually be committed by all honest parties.

Remark on timeout parameter  $\tau$ . The timeout parameter  $\tau$  must be set sufficiently long to guarantee that, upon entering round r, an honest party has enough time to deliver both the round r leader vertex—if broadcast by an honest leader—and at least n-f round r vertices and votes before its timer expires. The precise value of  $\tau$  depends on the underlying RBC primitive used to disseminate vertices, as well as the specific conditions under which an honest party  $P_i$  transitions to round r. When using Bracha's RBC [7], our protocol requires  $\tau = 2\Delta$  if  $P_i$  enters round r after delivering the leader vertex from round r-1, and  $\tau=5\Delta$  otherwise. We prove in Claim 7 that these timeout bounds are sufficient.

**Intuition behind designing propose flag.** We design the *propose* flag to require parties to wait for the observed number of vertex proposals minus f. This ensures that more vertices are referenced as strong edges. Without this condition, faulty parties could accelerate the progression of rounds by prematurely sending vote messages. As a result more vertices are included as weak edges, which increases their commit latency by at least one RBC delay.

In practice, an honest party needs to predict the *propose* flag based on the received transactions. If the prediction is incorrect, two outcomes are possible. When the party sets the *propose* flag to true but proposes an empty block without transactions, the overall throughput is slightly reduced. On the other hand, when the party sets the *propose* flag to false but still proposes a block, occasional mispredictions have little impact. However, if many parties behave this way, some blocks will only be referenced as weak edges, which in turn increases the overall latency.

We present detailed security analysis in Section C.

#### 4 Overview of Multi-leader Angelfish

In Angelfish, the latency to commit the leader vertex is shorter than that for non-leader vertices. To improve the latency for multiple vertices, we extend Angelfish to support multiple leaders per round. In the best-case scenario, when all these leaders are honest, the respective leader vertices can be committed with a latency of one RBC plus  $1\delta$ .

In Multi-leader Angelfish, multiple leaders are chosen randomly in each round. Each must propose a vertex. One is designated as the main leader, the others as secondary leaders. The main leader's responsibility is to ensure that its vertex has a leader path to all prior leader vertices and provide "proofs" for any missing ones. A vote message may reference multiple leaders if it supports them. For a round r main leader vertex  $L_r$  that does not reference the round r-1 main leader vertex, Multi-leader Angelfish requires it to reference (i) a main leader vertex from some earlier round r' < r-1 via a

leader edge, (ii) a set of  $\mathcal{T}\mathcal{C}$ s for rounds r'+1 to r-1 (parties send timeout messages only for main leaders), and (iii) an  $\mathcal{NVC}$  (no-vote certificate) for some leader vertex  $\ell$  in round r' or r'+1. An  $\mathcal{NVC}^{\ell}_{r'}$  indicates that  $L_r$  has leader edges to all leaders before  $\ell$  in round r'. An illustration is given in Figure 6.

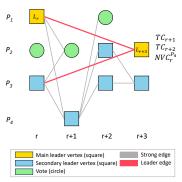


Figure 6: This represents the set of local vertices and votes observed by  $P_2$ . In round r, the secondary leaders are  $P_3$  and  $P_4$ .  $L_{r+3}$  did not deliver  $P_4$ 's round r leader vertex, nor the round r+1 and round r+2 main leader vertices; therefore,  $L_{r+3}$ 's vertex must contain  $\mathcal{T}C_{r+1}$ ,  $\mathcal{T}C_{r+2}$ , and  $\mathcal{N}\mathcal{V}C_r^{P_4}$ .

For a party  $P_i$  that is not the main leader, it enters round r upon either delivering the round r-1 leader vertex or observing  $TC_{r-1}$ , and additionally having delivered a vertices and received b vote messages with  $a+b \le n-f$ . Each vote message must be re-broadcast to ensure propagation. After entering round r,  $P_i$  broadcasts a no-vote message for any missing round r-1 leaders, enabling any main leader in round r or higher to form a no-vote certificate.

A round r leader vertex is directly committed when it is supported by n-f round r+1 vertices (first message of RBC) and votes, and all earlier leaders in the same round's leader list have been committed. Once a main leader vertex is directly committed, earlier leaders connected by leader paths are indirectly committed as well, ensuring that any leader vertices missed in previous rounds are eventually confirmed by all honest parties.

We present details of protocol in Section B and security analysis in Section D.

## 5 Empirical Evaluation

We implement Angelfish and evaluate its performance by comparing it on a real-world testbed against the state-of-the-art DAG-based protocol Sailfish [39] and a state-of-the-art protocol with asynchronous data-dissemination, Jolteon [15]. Unfortunately, no implementation of a state-of-the-art leader-based protocol that use AVID (e.g., DispersedSimplex [36]) is available. Instead, we obtain a baseline lower bound of 220ms on the latency achievable by such protocols as follows: we use Hydrangea [38], a latency-optimal (3 $\delta$ ), leader-based

protocol, which we configure to use no mempool and empty blocks. In addition, we evaluate the performance of Multileader Angelfish as well as Angelfish under failure scenarios.

Our implementation modifies the core consensus logic of the open-source Sailfish codebase [37] to create Angelfish [49] and Multi-leader Angelfish [50].

Experimental setup. We conducted our evaluations on the Google Cloud Platform (GCP), deploying nodes evenly across five distinct regions: us-east1-b (South California), us-west1-a (Oregon), europe-west1-b (Belgium), europe-north1-b (Finland) and asia-northeast1-a (Japan). We employed e2-standard-16 instances [34], each featuring 16vCPUs, 64GB of memory, and up to 16Gbps network bandwidth. All nodes ran on Ubuntu 22.04, and round-trip latencies between GCP regions range from roughly 35 milliseconds to 250 milliseconds (details appear in Table 2). Sailfish and Angelfish use BLS signatures and multi-signatures.

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

	Destination*					
Source	us-e1	us-w1	eu-w1	eu-n1	as-n1	
us-east1-b	0.70	63.95	93.05	115.44	157.61	
us-west1-a	65.03	0.69	135.15	157.31	89.59	
europe-west1-b	93.06	135.18	0.55	31.25	223.37	
europe-north1-b	115.21	156.29	31.22	0.77	248.25	
asia-northeast1-a	167.57	89.60	228.51	249.32	0.67	

<sup>\*</sup> Region names are abbreviated versions of the source regions.

In our evaluations, each party generates a configurable number of transactions (512 random bytes each) for inclusion in its vertex, with a vertex containing up to 10,000 transactions (i.e., 5 MB). Each experiment runs for 180 seconds with 50 nodes. Timeouts are set to 0.5 seconds. Latency is measured as the average time between the creation of a transaction and its commit by all non-faulty nodes. Throughput is measured by the number of committed transactions per second.

**Methodology.** In our evaluations, we gradually increased the number of transactions per vertex. As shown in Figure 7, throughput increases accompanied by a slight increase in latency up to an inflection point quickly leading to saturation.

Performance comparison under fault-free case. We compare the performance of Sailfish, Jolteon and Angelfish under fault-free scenarios. In Angelfish, the parameter propose\_rate denotes the fraction of nodes that propose vertices in each round, while the other nodes only vote. Figure 7 presents the latency of Sailfish and Angelfish as throughput increases from left to right, and Angelfish is evaluated under varying propose\_rate values of 1, 0.8 and 0.4. Figure 7b provides a zoomed-in view of Figure 7a for throughput below 130 KTps.

In the implementation, we fix the propose\_rate for each round and randomly select vertex proposers according to this rate. If the leader is not among the selected proposers, we

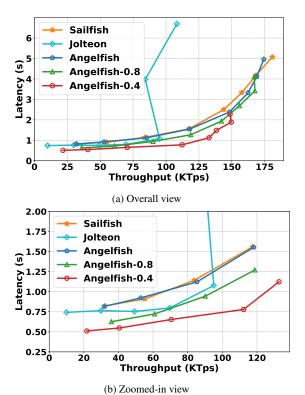


Figure 7: Throughput vs. latency with 50 nodes

additionally require the leader to propose a vertex. As a result, each round has either 50× propose\_rate or 50× propose\_rate +1 vertex proposers. Both Angelfish and Sailfish employ a 2-round RBC and adopt an optimistic signature verification strategy: nodes aggregate signatures without verifying each one individually and verify only the final aggregate.

When propose\_rate equals 1, Angelfish and Sailfish perform nearly indistinguishably; this is expected since the two protocols behave identically in this scenario. When the propose\_rate is 0.8 or 0.4, Angelfish achieves lower latency than Sailfish, since a higher propose\_rate requires nodes to wait for proposals from more regions and reduces the number of early one-round votes, both of which increase latency.

In the case of propose\_rate= 0.8, 40 (41) nodes propose vertices in each round. In Angelfish, a party must deliver at least the number of proposers it observes minus the fault tolerance, i.e., 40(41) - 16 = 24(25), in order to enter the next round. As a result, each party waits for between 24 and 40 vertices, corresponding to proposals from the nearest 3–5 regions. If the threshold of n - f is not reached by the time leader vertex is delivered, they gather vote messages to advance to the next round. In the case of propose\_rate = 0.4, the minimum number of vertices a node must wait for is  $50 \times 0.4 - 16 = 4$ , which means nodes only need to wait for proposals from the nearest 1 to 5 regions. Additionally, more nodes use vote messages than with propose\_rate = 0.8. Thus, the latency of propose\_rate = 0.4 is lower than that of propose\_rate = 0.8.

Angelfish, with propose\_rate values of 0.8 and 0.4, outperforms Jolteon under low-throughput scenarios. The latency bottleneck of Sailfish arises from the requirement that parties must wait for the delivery of n-f vertices in addition to the leader vertex before entering a new round, whereas Jolteon's leader proposes transactions batches with available availability certificates without waiting. Moreover, Sailfish must deliver all the references of a block in order to commit it, while in Jolteon each block has only a single ancestor.

Finally, with propose\_rate=0.4, Angelfish processes roughly 20 KTps at 0.5s latency, which is only about twice the lower-bound latency we measure when Hydrangea orders empty blocks. This suggest that the latency of Angelfish will closely match the best leader-based protocols.

Comparing BEB of vote messages to RBC of empty vertices. Since Angelfish allows a party to skip proposing a vertex when it has no transaction, while Sailfish does not, we modified Sailfish to match the same setting [52]. Specifically, we interpret the propose\_rate in Sailfish as the fraction of nodes that propose vertices containing transactions, while the remaining nodes propose empty vertices. As shown in Figure 8, vote messages have a large advantage especially at low propose\_rate: in this situation, Angelfish reaches peak throughput almost double that of Sailfish, and its latency is at least 20% better at equal.

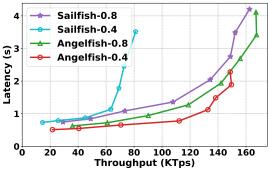


Figure 8: Using empty blocks in Sailfish VS Angelfish. Throughput vs. latency at various propose\_rate with 50 nodes.

**Performance comparison under failures.** Figure 9 presents the performance of Angelfish [48] and Sailfish [51] with 16 crash failures among 50 nodes. We assign a failed leader every three rounds, and the 16 crashed nodes are uniformly distributed across five regions. The propose\_rate is set to 1.

In Sailfish, once the timer expires, after receiving a timeout certificate and entering a new round, each party sends a no-vote message to the leader. The new leader must then collect a quorum of no-vote messages before it proposes a vertex. In contrast, Angelfish requires only the timeout certificate to proceed. This optimization eliminates one message delay compared to Sailfish, which explains the latency gap observed in the figure. As the crash frequency increases, this gap will be more pronounced.

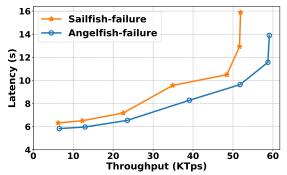


Figure 9: Throughput vs. latency at 50 nodes with 16 crash failures

**Performance comparison of Multi-leader Angelfish.** We evaluated Multi-leader Angelfish [50] in failure-free settings with 10 and 20 leaders per round. The latency is presented in Figure 10. In the figure, MLAF-leader\_num=*x* denotes configurations with *x* leaders, while MLAF-leader\_num=10-propose\_rate=0.4 indicates the case with 10 leaders where 60% of parties abstain from proposing vertices.

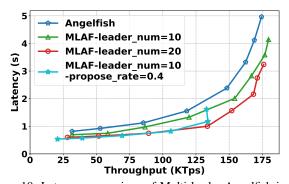


Figure 10: Latency comparison of Multi-leader Angelfish in the absence of failures

As shown in the figure, consensus latency improves as the number of leader vertices increases. Moreover, with propose\_rate=0.4, Multi-leader Angelfish is now only 36% slower than the lower bound obtained with Hydrangea proposing empty blocks.

#### 6 Related Work

In the vein of PBFT [11], traditional leader-based protocols [17,25,32,45,53] use a rotating leader that is responsible for broadcasting data and driving consensus. This achieves low latency but throughput is limited by the leader's bandwidth. To achieve higher throughput, newer protocols try to make use of the bandwidth of all parties to disseminate data.

Some protocols like RCC [18], MIR-BFT [43], or ISS [44] spread the load by operating parallel protocol instances that each use a different leader and combine their outputs.

Protocols like Destiny [4], Jolteon [15], or Autobahn [16] reliably store data using an asynchronous data-dissemination

layer and separately perform consensus on data references. This achieves high throughput but adds data-storage latency to consensus latency. Autobahn is able to linearize entire causal histories at once after network blips.

In DAG-based protocols, parties create data blocks in parallel and organize them in a DAG whose structure is then interpreted to arrive at a total ordering. HashGraph [6] pioneered this approach with an unstructured, asynchronous DAG. Aleph [14] first introduced a layered DAG built in rounds. DAG-Rider [22] further optimizes performance and resilience and is quantum safe, and Narwhal [12] showed that the approach achieves high performance in practice.

Bullshark [41] introduces leader vertices (also called anchors) and timeouts in the DAG-building process in exchange for better latency in the common case, and it is the first partially-synchronous DAG-based protocol with embedded consensus. Shoal [40] runs multiple instances of Bullshark in a pipeline to further reduce latency. Shoal++ [3] make further improvements upon Bullshark by using multiple leader vertices per round and intertwines the outputs of parallel protocol instances. Sailfish [39] first achieves a leader vertex every round and 3 message delays to commit leader vertices.

Uncertified DAG-based protocols such as Cordial Miners [24], BBCA-Chain [31], or Mysticeti [5] use best-effort broadcast to disseminate DAG vertices instead of reliable broadcast. This avoids the latency of reliable broadcast but results in brittle performance [3].

Finally, protocols like HoneyBadger [33], DispersedLedger [46], and DispersedSimplex [36] use asynchronous verifiable information dispersal (AVID) [10] based on erasure codes to evenly use all system bandwidth. In particular, DispersedSimplex retains the latency-optimized structure of partially-synchronous protocols like PBFT while solving the leader-bottleneck problem using AVID. Erasure-coded AVID however introduces processing, communication, and data-expansion costs.

While all the protocols above achieve dramatically higher throughput than traditional leader-based protocols, the latter still reign over latency. In concurrent work, Morpheus [30] and Grassroots Consensus [23] attempt to solve this problem by dynamically switching between a low-throughput mode that behave like a traditional leader-based protocol and a high-throughput mode that uses decoupled data-dissemination. In contrast, Angelfish is able to adapt continuously along this spectrum to achieve the best tradeoff at any point.

## References

[1] Suiscan – sui blockchain explorer. https://suiscan.xyz/mainnet/txs/tx-blocks. Online; accessed 04-Sep-2025.

- [2] Michael Anoprenko, Andrei Tonkikh, Alexander Spiegelman, and Petr Kuznetsov. Dags for the masses. *arXiv preprint arXiv:2506.13998*, 2025.
- [3] Balaji Arun, Zekun Li, Florian Suri-Payer, Sourav Das, and Alexander Spiegelman. Shoal++: High throughput {DAG}{BFT} can be fast and robust! In 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25), pages 813–826, 2025.
- [4] Balaji Arun and Binoy Ravindran. Scalable byzantine fault tolerance via partial decentralization. *Proc. VLDB Endow.*, 15(9):1739–1752, May 2022.
- [5] Kushal Babel, Andrey Chursin, George Danezis, Lefteris Kokoris-Kogias, and Alberto Sonnino. Mysticeti: Low-latency dag consensus with fast commit path. In Network and Distributed System Security Symposium (NDSS), 2025.
- [6] Leemon Baird. The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. Swirlds Tech Reports SWIRLDS-TR-2016-01, Tech. Rep, 34:9–11, 2016.
- [7] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.
- [8] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus, November 2019.
- [9] Christian Cachin and Stefano Tessaro. Asynchronous verifiable information dispersal. In 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05), pages 191–201. IEEE, 2005.
- [10] Christian Cachin and Stefano Tessaro. Asynchronous verifiable information dispersal. In 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05), pages 191–201. IEEE, 2005.
- [11] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, number 1999 in 99, pages 173–186, 1999.
- [12] 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.
- [13] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.
- [14] Adam Gkagol, Damian Leśniak, Damian Straszak, and Michał Świketek. 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.
- [15] 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.
- [16] Neil Giridharan, Florian Suri-Payer, Ittai Abraham, Lorenzo Alvisi, and Natacha Crooks. Autobahn: Seamless high speed bft. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 1–23, 2024.
- [17] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: A scalable and decentralized trust infrastructure. In 2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN), pages 568–580. IEEE, 2019.
- [18] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. RCC: Resilient Concurrent Consensus for High-Throughput Secure Transaction Processing. In 2021 IEEE 37th International Conference on Data Engineering (ICDE), pages 1392–1403, April 2021.
- [19] Amores-Sesar Ignacio, Grøndal Viktor, Holmgård Adam, and Ottendal Mads. Dag it off: Latency prefers no common coins. *arXiv preprint arXiv:2508.14716*, 2025.
- [20] Kazuharu Itakura. A public-key cryptosystem suitable for digital multisignature. *NEC research and development*, 71:1–8, 1983.
- [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, PODC'21, pages 165–175, New York, NY, USA, July 2021. Association for Computing Machinery.
- [22] 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.
- [23] Idit Keidar, Andrew Lewis-Pye, and Ehud Shapiro. Grassroots consensus. *arXiv preprint arXiv:2505.19216*, 2025.

- [24] Idit Keidar, Oded Naor, Ouri Poupko, and Ehud Shapiro. Cordial miners: Fast and efficient consensus for every eventuality. In 37th International Symposium on Distributed Computing (DISC 2023). Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2023.
- [25] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of twenty*first ACM SIGOPS symposium on Operating systems principles, pages 45–58, 2007.
- [26] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.
- [27] Leslie Lamport. The Part-time Parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.
- [28] Leslie Lamport. Lower Bounds for Asynchronous Consensus. In *Future Directions in Distributed Computing*, Lecture Notes in Computer Science, pages 22–23. Springer, Berlin, Heidelberg, 2003.
- [29] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.
- [30] Andrew Lewis-Pye and Ehud Shapiro. Morpheus consensus: Excelling on trails and autobahns. *arXiv* preprint *arXiv*:2502.08465, 2025.
- [31] 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.
- [32] J.-P. Martin and L. Alvisi. Fast Byzantine Consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, July 2006.
- [33] 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.
- [34] [n.d.]. Google cloud platform general-purpose machine family for compute engine. Online; accessed 26-July-2025.
- [35] Nikita Polyanskii, Sebastian Mueller, and Ilya Vorobyev. Starfish: A high throughput bft protocol on uncertified dag with linear amortized communication complexity. *Cryptology ePrint Archive*, 2025.

- [36] Victor Shoup. Sing a song of simplex. In 38th International Symposium on Distributed Computing (DISC 2024), pages 37–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024.
- [37] Nibesh Shrestha. Sailfish github repository. https://github.com/nibeshrestha/sailfish. [Online; accessed 02-March-2025].
- [38] Nibesh Shrestha, Aniket Kate, and Kartik Nayak. Hydrangea: Optimistic Two-Round Partial Synchrony, 2025.
- [39] Nibesh Shrestha, Rohan Shrothrium, Aniket Kate, and Kartik Nayak. Sailfish: Towards improving the latency of dag-based bft. In 2025 IEEE Symposium on Security and Privacy (SP). IEEE, 2025.
- [40] Alexander Spiegelman, Balaji Aurn, Rati Gelashvili, and Zekun Li. Shoal: Improving dag-bft latency and robustness. In *International Conference on Financial Cryp*tography and Data Security, 2024.
- [41] 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.
- [42] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: The partially synchronous version. *arXiv preprint arXiv:2209.05633*, 2022.
- [43] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. Mir-bft: High-throughput bft for blockchains. *arXiv preprint arXiv:1906.05552*, 92, 2019.
- [44] Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. State machine replication scalability made simple. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, pages 17–33, New York, NY, USA, March 2022. Association for Computing Machinery.
- [45] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. Basil: Breaking up BFT with ACID (transactions). In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 1–17, New York, NY, USA, October 2021. Association for Computing Machinery.
- [46] Lei Yang, Seo Jin Park, Mohammad Alizadeh, Sreeram Kannan, and David Tse. {DispersedLedger}:{High-Throughput} byzantine consensus on variable bandwidth networks. In 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22), pages 493–512, 2022.

- [47] 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.
- [48] Qianyu Yu. Angelfish-failure github repository. https://github.com/qyu100/sf-dag/tree/angelfish-failure.
- [49] Qianyu Yu. Angelfish github repository. https://github.com/qyu100/sf-dag/tree/angelfish.
- [50] Qianyu Yu. Multi-leader angelfish github repository. https://github.com/qyu100/sf-dag/tree/multi-leader-angelfish.
- [51] Qianyu Yu. Sailfish-failure github repository. https://github.com/qyu100/sf-dag/tree/sailfish-failure.
- [52] Qianyu Yu. Sailfish-with-propose-rate github repository. https://github.com/qyu100/sf-dag/tree/safilsih-bls-multisigs-add-propose-rate.
- [53] Qianyu Yu, Giuliano Losa, and Xuechao Wang. Tetrabft: Reducing latency of unauthenticated, responsive bft consensus. In *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing*, pages 257–267, 2024.

## **A** Extended Preliminaries

**Definition A.1** (Byzantine reliable broadcast [7]). In a Byzantine reliable broadcast (RBC), a designated sender  $P_k$  may invoke  $r\_bcast_k(m)$  to propagate an input m. Each party  $P_i$  may then output (we also say commit) the message m via  $r\_deliver_i(m, P_k)$  where  $P_k$  is the designated sender. The reliable broadcast primitive satisfies the following properties:

- **Validity.** If the designated sender  $P_k$  is honest and calls  $r\_bcast_k(m)$  then every honest party eventually outputs  $r\_deliver(m, P_k)$ .
- **Agreement.** If an honest party  $P_i$  outputs r\_deliver $_i(m, P_k)$ , then every other honest party  $P_j$  eventually outputs r\_deliver $_i(m, P_k)$ .
- Integrity. Each honest party P<sub>i</sub> outputs r\_deliver<sub>i</sub> at most once regardless of m.

# **B** Multi-leader Angelfish In-Depth

**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

```
Local variables:
                                                                                                                                                > The struct of a vertex in the DAG
       struct vertex v:
               v.round - the round of v in the DAG
               v.source - the party that broadcasts v
               v.block - a block of transactions
               v.propose - if the party will propose a vertex in the next round
               v.strongEdges - a set of vertices in v.round - 1 that represent strong edges
               v.weakEdges - a set of vertices in rounds < v.round - 1 that represent weak edges
               v.leaderEdges - leader vertices in round < v.round - 1 that represent leader edge (for leader vertices)
               v.tc - a set of timeout certificates for round \leq v.round - 1 (if any)
               v.nvc - a no-vote certificate for round \leq v.round - 1 (if any)
       struct vote vt:
                                                                                                                                                                 > The struct of a vote
               vt.round - the round of vt
               vt.source - the party that broadcasts vt
               vt.propose - if the party will propose a vertex in the next round
               vt.strongEdges - leader vertices in v.round-1 that the party votes for; otherwise, \bot
       no-vote certificate \mathcal{N}\mathcal{V}\mathcal{C} - An array of sets, each containing a quorum of no-vote messages
                                                                                                 199: procedure create_new_vertex(r)
                                                                                                 200:
                                                                                                            v.round \leftarrow r
                                                                                                 201:
                                                                                                            v.source \leftarrow P_i
                                                                                                 202:
                                                                                                            v.block \leftarrow blocksToPropose.dequeue()
170: upon receiving \langle vt, r \rangle_p do
                                                                                                 203:
                                                                                                            v.propose \leftarrow true if P_i intends to propose vertex in round r+1
171:
           if vt.round = r then
                                                                                                    else false
               votes[r] \leftarrow votes[r] \cup \{vt\}
172:
                                                                                                 204:
                                                                                                            if r > 1 then
173:
               if vt.propose = true then
                                                                                                                 upon |DAG_i[r-1]| \ge \max(|VertexProposers[r-2]| - f, 0) do
                                                                                                 205:
174:
                    VertexProposers[r] \leftarrow VertexProposers[r] \cup \{p\}
                                                                                                 206:
                                                                                                                     v.strongEdges \leftarrow DAG_i[r-1]
175:
               multicast \langle vt, r \rangle_p
                                                                                                 207:
                                                                                                                 if \langle \text{timeout}, r-1 \rangle_i is sent then
176: upon |votes[r]| + |DAG_i[r]| > n - f \land (\exists v' \in DAG_i[r] : v'.source = L_r \lor drown
                                                                                                 208:
                                                                                                                     v.strongEdges \leftarrow v.strongEdges \setminus \{v' : v'.source = L_{r-1}\}
  TC_r is received for r \ge round) do
                                                                                                 209:
                                                                                                            if P_i = L_r \land \exists v' \in DAG_i[r-1] : v'.source = L_{r-1} then
177:
           rnd \leftarrow r
                               > round of the last main leader vertex delivered
                                                                                                 210:
                                                                                                                 for r' = r down to 1 do
178:
           if P_i = L_{r+1} \land \nexists v' \in DAG_i[r] : v'.source = L_r then
                                                                                                                     if \not\exists v'' \in v.strongEdges s.t. v''.source = L_{r'-1} then
                                                                                                 211:
               wait until receiving TC_r
179:
                                                                                                 212:
                                                                                                                         v.tc \leftarrow v.tc \cup \{TC_{r'-1}\}
180:
               for r' = r - 1 down to 1 do
                                                                                                 213:
                                                                                                                     else
                    if \exists v'' \in DAG_i[r'] : v''.source = L'_r then
181:
                                                                                                                          for \ell \in \mathcal{ML}_{r'-1} do
                                                                                                 214:
182:
                        rnd \leftarrow r
                                                                                                                              if \nexists v^* \in DAG_i[r'-1] : v^*.source = \ell then
                                                                                                 215:
183:
                        break
                                                                                                                                   v.nvc \leftarrow \mathcal{NVC}_{r'-1}^{\ell}
                                                                                                 216:
184:
                    else
                                                                                                 217:
                                                                                                                                   break
185:
                        wait until receiving \mathcal{TC}_{r'}
                                                                                                 218:
                                                                                                                              else
186:
           advance\_round(r+1, rnd)
                                                                                                 219:
                                                                                                                                   v.leaderEdges \leftarrow v.leaderEdges \cup \{v^*\}
187: procedure advance_round(r, rnd)
                                                                                                 220:
                                                                                                                          break
188:
           round \leftarrow r; start\ timer
                                                                                                 221:
                                                                                                             set_weak_edges(v, r)
           for \ell \in \mathcal{ML}_{r-1} \wedge \mathbf{do}
189.
                                                                                                 222:
                                                                                                             return v
190:
               if \exists v' \in DAG_i[r-1] : v'.source = \ell then
                                                                                                 223: procedure create_new_vote(r)
191:
                    multicast \langle no-vote, \ell, r-1 \rangle_i
                                                                                                 224:
                                                                                                             vt.round \leftarrow r
192:
           if P_i = I_{in} then
                                                                                                 225:
                                                                                                             vt.source \leftarrow P_i
               wait until \exists v' \in DAG_i[rnd-1] : v'.source = \ell \ \forall \ell \in \mathcal{ML}[:x]
193:
                                                                                                 226:
                                                                                                             vt.propose \leftarrow true if P_i intends to propose vertex in round r + 1
  or \mathcal{N}\mathcal{VC}^{\ell}_{rnd-1} is received, where \ell'=\mathcal{ML}[x+1]
                                                                                                    else false
194:
               broadcast_vertex(round)
                                                                                                 227:
                                                                                                            if \langle \text{timeout}, r-1 \rangle_i is sent then
195:
           else if P_i intends to propose vertex in round r then
                                                                                                 228:
                                                                                                                 vt.strongEdges \leftarrow \bot
196:
               broadcast_vertex(round)
                                                                                                 229:
                                                                                                            else
197:
           else
                                                                                                 230:
                                                                                                                 if r > 1 \land \exists v' \in DAG_i[r-1] : v'.source = L_{r-1} then
198:
               multicast_vote(round)
                                                                                                 231:
                                                                                                                     for \ell \in \mathcal{M} \mathcal{L}_{r-1} do
                                                                                                                          if \exists v'' \in DAG_i[r-1] : v''.source = \ell then
                                                                                                 232:
                                                                                                 233:
                                                                                                                              vt.strongEdges \leftarrow vt.strongEdges \cup \{v''\}
                                                                                                 234:
                                                                                                            return \langle vt, r \rangle_i
```

Figure 11: Multi-leader Angelfish: DAG construction

the main leader is referred to as the main leader vertex, and the vertices proposed by the secondary leaders are termed secondary leader vertices. Inspired by Multi-leader Sailfish protocol [39], we introduce a no-vote certificate, formed by collecting a quorum of no-vote messages. In the Multi-leader Angelfish, the main leader's responsibility is to ensure

that the main leader vertex has a leader path to all leader vertices from the previous rounds and to collect a no-vote certificate for any missing leader vertices.

To determine the multiple leaders in a given round, we define a deterministic function,  $get_nultiple_leaders(r)$ , 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 Angelfish, 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, and  $\mathcal{ML}_r[1]$  is the main leader. 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.

**DAG construction protocol.** The basic data structures differ slightly from Angelfish, with the changes highlighted in blue in Figure 11. A vertex may contain leader edges referencing vertices from more than just the main leader. Moreover, a vertex collects an  $\mathcal{NVC}$  if some leader vertices are not delivered. A vote message also votes not only for the main leader but also for the secondary leaders. To accommodate multiple leaders within a round, the protocol's round advancement rules are modified as shown in Figure 11.

Recall that in Angelfish, 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  $\langle \texttt{timeout}, r \rangle$  message. Upon receiving either the round r leader vertex or  $\mathcal{TC}_r$  (along with n-f round r vertices and votes)  $P_i$  advances to round r+1.

In Multi-leader Angelfish,  $P_i$  sends  $\langle \texttt{timeout}, r \rangle$  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. Each party  $P_i$  wait for the round r main leader vertex or  $\mathcal{TC}_r$  (along with n-f round r vertices and votes) to advance to round r+1. Otherwise, waiting for  $\mathcal{TC}$  to enter a new round could allow a single faulty leader to slow down the protocol. Here the votes refer to vote messages only. When receiving a vote message, each party broadcasts the vote message to ensure all the other parties also receive it. However, the main leader vertex must still provide proof for any undelivered leader vertices, which motivates the introduction of no-vote messages.

After entering round r+1,  $P_i$  sends  $\langle no\text{-vote}, \ell, r \rangle$  for every  $\ell \in \mathcal{ML}_r$  from which it did not receive the round r leader vertex. Unlike Multi-leader Sailfish [39], no-vote messages are broadcast rather than sent only to the round r+1 leader. This choice is motivated by the structure of leader paths: in Angelfish, each leader has a leader path to some previous round leader; in Multi-leader Angelfish, the main leader has leader paths to the previous-round leaders (or collects no-vote certificates for any missing ones). Thus, no-vote messages should be received by all subsequent leaders.

In Multi-leader Sailfish [39], they have the constraint on the main leader vertex as follows: The round r+1 main leader vertex  $v_k$  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  $\langle \text{no-vote}, \ell, r \rangle$  (referred to as  $\mathcal{NVC}_r^{\ell}$ ), where  $\ell = \mathcal{ML}_r[x+1]$ . In our protocol, this requirement is modified as follows. Suppose  $L_{r'}$  is the last main leader that

 $L_{r+1}$  has delivered. If  $L_{r+1}$  has not delivered round r main leader vertex, then the round r+1 main leader vertex  $v_k$  must contain  $\mathcal{T}_{\mathcal{C}_{r''}}$  for all rounds r'' such that r' < r'' < r. Furthermore, the round r+1 main leader vertex  $v_k$  must establish leader paths to all leader vertices corresponding to leaders in  $\mathcal{ML}_{r'}[:x]$  and contain  $\mathcal{NVC}_{r'}^{\ell}$ , where  $\ell = \mathcal{ML}_{r'}[x+1]$  and r' < r (see Section B). An illustration is provided in Figure 6. If the main leader vertex has leader 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 is\_valid() function is also appropriately updated to ensure that these constraints are met. The constraint for other round r+1 vertices remain unchanged. For round r+1 vote messages, we require that they vote for all leaders and have strong edges to all delivered leader vertices if the round r main leader is delivered (see Section B). Since if round r main leader is not delivered, the strong edges to secondary leaders are not used in the commit rule, and therefore are not required.

```
Local variables:
                      committedRound \leftarrow 0
235: procedure try_commit(r, S)
236:
                                  CLS \leftarrow []
237:
                                  for \ell \in \mathcal{ML}_r do
238:
                                               v \leftarrow \text{get\_vertex}(\ell, r)
                                               if |collect| \ge n - f \land committedRound < r then
239:
240:
                                                            CLS \leftarrow CLS \parallel v
241:
                                               else break
242:
                                  commit_leaders(CLS)
243:
                    procedure commit_leaders(cls)
244:
                                  leaderStack.push(cls)
245:
                                  v' \leftarrow cls[0]
246:
                                  r \leftarrow v'.round - 1
247:
                                  while r > committedRound do
                                               \mathcal{CMV} \leftarrow []
248:
249:
                                               for \ell \in \mathcal{ML}_r do
250:
                                                            v \leftarrow \text{get\_vertex}(\ell, r)
251:
                                                            if leader_path(v', v) then
252:
                                                                           \mathcal{CMV} \leftarrow \mathcal{CMV} \parallel v
253:
                                                            else break
254:
                                               if \mathcal{CMV} \neq [] then
255:
                                                            v' \leftarrow \mathcal{CMV}[1]
                                                                                                                                                  \triangleright main leader vertex for round r
256:
                                               leaderStack.push(\mathcal{CMV})
257:
                                               r \leftarrow r - 1
258:
                                  committedRound \leftarrow cls[0].round
259:
                                  order_vertices()
260: procedure order_vertices()
261:
                                  while ¬leaderStack.isEmpty() do
                                                \mathcal{CMV} \leftarrow leaderStack.pop()
262:
263:
                                               for v \in \mathcal{CMV} do
                                                                                                                                                            \triangleright iterate over \mathcal{CMV} in order
                                                            verticesToDeliver \leftarrow \{v' \in \bigcup_{r>0} DAG_i[r] \mid path(v,v') \land AG_i[r] 
               v' \notin deliveredVertices
265:
                                                            for every v' \in verticesToDeliver in some deterministic
               order do
266:
                                                                          output a_deliver<sub>i</sub>(v'.block,v'.round,v'.source)
267:
                                                                         deliveredVertices \leftarrow deliveredVertices \cup \{v'\}
```

Figure 12: Multi-leader Angelfish: commit rule

**Committing and ordering the DAG.** Similar to Angelfish, 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 in Figure 12. The commit rule is adapted from Multi-leader Sailfish [39], and since the underlying intuition remains the same, the underlying intuition is not repeated here for brevity.

In our protocol, an honest party  $P_i$  directly commits a round r leader vertex  $v_k$  corresponding to  $\mathcal{ML}_r[x]$  when it observes "first messages" (of the RBC) for round r+1 vertices and votes with a total number being n-f with strong edges 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 n-f round r+1 vertices and votes with strong paths to the leader vertices corresponding to the leaders in  $\mathcal{ML}_r[x+1:]$ . The commit rule is presented in try commit() function (see Section B).

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 leader 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  $\mathcal{ML}_{r'}[1]$  until it reaches a round r'' < r in which it previously directly committed a leader vertex (see Section B). 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 leader 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.

## C Security Analysis

We say that a leader vertex  $v_i$  is committed directly by party  $P_i$  if  $P_i$  invokes commit\_leader( $v_i$ ). Similarly, we say that a leader vertex  $v_j$  is committed indirectly if it is added to leaderStack in Line 165. 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 the use of reliable broadcast to propagate a vertex v, along with the requirement to wait for its causal history of vertices 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  $v \in DAG_i[r]$  and  $v' \in DAG_j[r]$  such that 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 round r leader vertex  $v_k$ , then for every round r' leader vertex  $v_\ell$  such that r' > r, there exists a leader path from  $v_\ell$  to  $v_k$ .

*Proof.* If an honest party directly commits a round r leader vertex  $v_k$ , it must have observed n-f "support" signals in total, including both (i) first RBC messages from round r+1vertices that reference the round r leader vertex, and (ii) votes (vote messages  $\langle vt, r+1 \rangle_*$  that support it and signatures from  $\mathcal{CVC}_{r+1}$ ). By quorum intersection, there cannot be n-fvertices and votes that do not have a strong edge to  $v_k$ . Since if an honest party  $P_i$  has sent a  $\langle timeout, r \rangle_i$ , it will not propose a round r+1 vertex or vote that has strong edge to it, then  $\mathcal{TC}_r$  will not form. Thus, a leader edge from a higher round > r leader vertex to a low round < r leader vertex will not exist. In our protocol, every leader vertex must have a leader edge or strong edge referencing a leader vertex from some previous round, thereby forming a continuous leader path. Since  $v_k$  cannot be skipped, any leader vertex  $v_\ell$  must have a leader 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  $v_\ell$  in round  $r' \ge r$ , then  $P_j$  (directly or indirectly) commits  $v_k$  in round r.

*Proof.* If r' = r, by Fact 1,  $v_k = v_\ell$  and  $P_j$  directly commits  $v_k$ . When r' > r, by Claim 1,  $v_\ell$  has a leader path to  $v_k$ . By the code of commit\_leader, after directly committing a leader vertex in round r',  $P_j$  tries to commit a leader vertex if there exists a leader path between the two leader vertices from a smaller round until it meets the last round r'' < r' that directly committed a leader vertex. If r'' < r < r',  $P_j$  must indirectly commit  $v_k$  in round r. If r < r'', by inductive argument and Claim 1,  $P_j$  must indirectly commit  $v_k$  after 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  $v_\ell$  and  $v_\ell'$  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  $P_j$  commit 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 without loss of generality that  $r_i \le r_j < r_i' \le r_j'$ . By Claim 2, both  $P_i$  and  $P_j$  will (directly or indirectly) commit the same leader vertices in round  $\min(r_i', r_j')$ . Assume without loss of generality that  $\min(r_i', r_j') = r_i'$ . By Fact 1, both  $DAG_i$  and  $DAG_j$  will contain  $v_k'$  and all leader vertices have a path from  $v_k'$  in  $DAG_i$ . According to the commit\_leader procedure,

after directly or indirectly committing the leader vertex  $v'_k$ , each party will attempt to indirectly commit leader vertices from earlier rounds by leader path until reaching a round in which they have previously directly committed a leader vertex. Consequently, both  $P_i$  and  $P_j$  will indirectly commit the leader vertices in the leader path from rounds  $\min(r'_i, r'_j)$  down to  $\max(r_i, r_j)$ . Furthermore, due to the fixed logic of the commit\_leader code, both parties will commit the same leader vertices between rounds  $\min(r'_i, r'_j)$  and  $\max(r_i, r_j)$  in the same order.

By inductively applying Claim 3 for every pair of honest parties, we get the following:

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

**Lemma 1** (Total order). If an honest party  $P_i$  outputs  $a\_deliver_i$   $(b,r,P_k)$  before  $a\_deliver_i$   $(b',r',P_\ell)$ , then no honest party  $P_j$  outputs  $a\_deliver_j$   $(b',r',P_\ell)$  before  $a\_deliver_j$   $(b,r,P_k)$ .

*Proof.* By Corollary 1, all honest parties commit the same leader vertices in the same order. According to the logic of order\_vertices, parties process committed leader vertices in that order and a\_deliver all vertices in their causal history based on a predefined rule. By Fact 1, every honest party has an identical causal history in their DAG for each committed leader. This establishes the lemma. □

**Lemma 2** (Agreement). If an honest party  $P_i$  outputs  $a\_deliver_i$  ( $v_i.block, v_i.round, v_i.source$ ), then every other honest party  $P_j$  eventually outputs  $a\_deliver_j$  ( $v_i.block, v_i.round, v_i.source$ ).

*Proof.* Since  $P_i$  outputs a\_deliver $_i(v_i.block, v_i.round, v_i.source)$ , it follows from the order\_vertices logic there exists a leader vertex  $v_k$  committed by  $P_i$  such that  $v_i$  is in the causal history of some leader vertex  $v_k$ . When  $P_i$  eventually commits leader vertex  $v_k$ , by Lemma 1,  $P_j$  also commits  $v_k$ . By Fact 1, the casual histories of  $v_k$  in  $DAG_i$  and  $DAG_j$  are the same. Thus, when  $P_j$  orders the casual histories of  $v_k$ , it outputs  $a\_deliver_j(v_i.block, v_i.round, v_i.source)$ .

**Lemma 3** (Integrity). For every round  $r \in \mathbb{N}$  and party  $P_k \in \mathcal{P}$ , an honest party  $P_i$  outputs a\_deliver<sub>i</sub> $(b, r, P_k)$  at most once regardless of b.

*Proof.* An honest party  $P_i$  outputs a\_deliver $_i(v.block, v.round, v. source)$  only when vertex v is in  $DAG_i$ . Note that v is added with to  $P_i$ 's DAG upon the reliable broadcast r\_deliver $_i(v.block, v.round, v. source)$  event. Therefore, the proof follows from the Integrity property of reliable broadcast.

**Validity.** We rely on GST to prove validity. Additionally, we use RBC protocol of Bracha [7] to establish validity. Bracha's

RBC protocol operates in 3 communication steps and ensures the RBC properties at all times. After GST, it offers 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, then all honest parties will deliver M by time  $t + 3\Delta$ .

**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+1. If  $P_i$  enters round r+1 at time t, then all honest parties will enter round r or higher by  $\max(t_g, t) + 2\Delta$ .

*Proof.* Observe that by time t,  $P_i$  must have delivered round r-1 vertices and received r-1 votes (vote messages and vote certificates), with their total count being n-f. Since when an honest party enters a round by votes, it will send vote certificates, then all honest parties must have received the vote certificates by  $\max(t_g, t) + \Delta$ . By Property 2, all honest parties must have delivered round r-1 vertices by  $\max(t_g, t) + 2\Delta$ .

We then prove the claim by considering  $P_i$  entering round by delivering round r leader vertex (say  $v_k$ ) and by receiving  $\mathcal{T}C_{r-1}$ . 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  $\mathcal{T}C_{r-1}$  which arrives all honest parties by  $\max(t_g,t)+\Delta$ . For  $L_r$ , if it enters round r, it also has to deliver round r' leader vertex and receive  $\mathcal{T}C$ s of rounds between r' and r. Similarly, by  $\max(t_g,t)+2\Delta$  it will deliver round r' leader vertex and by  $\max(t_g,t)+\Delta$  it will receive these  $\mathcal{T}C$ s, since for each round, either leader vertex is delivered or  $\mathcal{T}C$  exists or both. 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. All honest parties keep entering higher rounds.

*Proof.* Suppose all honest parties are in round r or higher. Let party  $P_i$  be in round r. If there exists an honest party  $P_j$  in a round r' > r at any time, then by Claim 4, all honest parties will eventually enter round r' or higher. Otherwise, all honest parties remain in round r. Note that upon entering round r, all honest parties first compute the number of parties that will propose a vertex in round r-1, say c, based on the round r-2 vertices and votes. Then they wait for  $\max(c-f,0)$  round r-1 vertices. After that, all honest parties r\_bcast a round r vertex, or broadcast vote certificates so each party will deliver round r vertices and receive votes with a total number of n-f.

If no honest party has delivered the round r leader vertex, then by the timeout rule, all honest parties will multicast  $\langle \text{timeout}, r \rangle$  and receive  $\mathcal{TC}_r$ . In addition, if  $L_{r+1}$  delivers a round r' < r leader vertex, it will also receive  $\mathcal{TC}_{r'+1}, ... \mathcal{TC}_r$ , so that there is a leader edge to round r' leader vertex. Thus,

all honest parties will move to round r+1. On the other hand, if at least one honest party has delivered the round r leader vertex, then by Fact 1, all honest parties will eventually deliver this round r leader vertex as well. With round r vertices and the round r leader vertex delivered and votes received with a total number of vertices and votes being n-f, all honest parties will proceed to round r+1.

**Claim 6.** If an honest party enters round r then at least f + 1 honest parties must have already entered round r - 1.

*Proof.* If an honest party  $P_i$  enters round r, it must do so either directly from round r-1 or by jumping rounds from some round < r-1. Either of the case, there must be at least an honest party has delivered round r-1 vertices and received  $\langle vt, r-1 \rangle_*$  or vote certificates with a total number of n-f. At least f+1 of these vertices and votes are sent by honest parties while they are in round r-1. Thus, f+1 honest parties must have already entered r-1.

**Claim 7.** If the first honest party to enter round r does so after GST and  $L_r$  is honest, then there will be at least n-f round r+1 vertices and votes that have strong edges to the round r leader vertex.

*Proof.* We prove the claim by considering the case where all honest parties set their timeout parameter  $\tau = 5\Delta$ . Let t be the time when the first honest party (say  $P_i$ ) entered round r. Observe that no honest party sends  $\langle \text{timeout}, r \rangle$  before  $t + 5\Delta$  due to its round timer expiring. Thus,  $\mathcal{TC}_r$  does not exist before  $t + 5\Delta$ . Additionally, by Claim 6, 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  $t + 5\Delta$  and no honest party enters a higher round by receiving a timeout certificate before  $t + 5\Delta$ .

Since  $P_i$  entered round r at time t, by Claim 4, all honest parties will enter round r or higher by  $t+2\Delta$ . If any honest party enters a round higher than r+1 before  $t+5\Delta$ , there exist at least n-f round r+1 vertices and votes. This is because for an honest party to enter round r', there must exist at least one honest party has delivered round r'-1 vertices and received  $\langle vt, r'-1\rangle_*$  with a total number of n-f. By transitive argument, there must exist n-f round r+1 vertices and votes. Moreover, no honest party sends a  $\langle \text{timeout}, r \rangle$  message before  $t+5\Delta$ . Since an honest party does not send a round r+1 vertex or  $\langle vt, r+1\rangle_*$  without a strong edge to round r leader vertex (say  $v_k$ ) without receiving  $\mathcal{T}C_r$ , this implies n-f round r+1 vertices and votes must have a strong edge to  $v_k$ .

Also, note that if an honest party enters round r+1 before  $t+5\Delta$ , there must be an honest party have delivered n-f round r vertices and received  $\langle vt,r\rangle_*$  with a total number of n-f, along with vertex  $v_k$  (since  $\mathcal{TC}_r$  do not exist before  $t+5\Delta$ ). Thus, its round r+1 vertex or vote must have a strong edge to  $v_k$ .

Now consider the case where no honest party has entered a round higher than r before time  $t+5\Delta$ . By Claim 4, all honest parties enter round r by  $t+2\Delta$ . Upon entering round r, each honest party invokes r\_bcast on its round r vertex or multicast its vote. By Property 1, round r vertices from all honest parties are delivered by  $t+5\Delta$ . Thus, all honest parties in round r will deliver round r vertices and receives  $\langle vt,r\rangle_*$  with a total number being n-f along with  $v_k$  and send round r+1 vertices or votes with a strong edge to  $v_k$ .

By the commit rule and Claim 7, we get the following:

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

**Lemma 4** (Validity). If an honest party  $P_i$  calls  $a\_bcast_i(b,r)$  then every honest party eventually outputs a deliver<sub>i</sub> $(b,r,P_i)$ .

*Proof.* When an honest party  $P_i$  calls a\_bcast\_i(b, r), it pushes b into the blocksToPropose queue. By Claim 5,  $P_i$  continuously progresses to higher rounds, creating new vertices in each of these rounds. Consequently,  $P_i$  will eventually create a vertex  $v_i$  with block b in some round r and reliably broadcast it. By the Validity property of reliable broadcast, all honest parties will eventually deliver  $v_i$ . According to Fact 1, all honest parties will add  $v_i$  to their DAGs. Following the create\_new\_vertex procedure, once  $v_i$  is added to  $DAG_j[r]$ , every subsequent vertex created by  $P_j$  will contain a path to  $v_i$ . By Corollary 2, the leader vertex proposed by an honest leader is directly committed by all honest parties after GST. By the code of order\_vertices, each party  $P_j$  will eventually invoke a\_deliver\_j( $b, r, P_i$ ). By Lemma 2, all honest parties will eventually invoke a deliver( $b, r, P_i$ ).

# D Multi-leader Angelfish Security Analysis

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 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'.

**Claim 8.** If an honest party  $P_i$  directly commits a leader vertex  $v_k$  in round r, then for every main leader vertex  $v_\ell$  in round r' such that r' > r, there exists a leader path from  $v_\ell$  to  $v_k$ .

*Proof.* If  $v_k$  is a main leader vertex, the proof is the same as Claim 1. We now consider the case where  $v_k$  is not a main leader vertex.

If an honest party directly commits a round r leader vertex  $v_k$ , it must have observed n - f messages in total, including

both (i) first RBC messages from round r+1 vertices that reference the round r leader vertex, and (ii) vote messages  $\langle vt, r+1 \rangle_*$  that support it. By quorum intersection, it is impossible for n-f such vertices and votes to lack a strong edge to  $v_k$ ; equivalently, n-f no-vote messages for  $v_k$  cannot exist. By the commit rule, if  $v_k$  is committed, then all leaders in the round r leader list that precede  $v_k$  are also committed (see Section B). Similarly, no-vote messages for these vertices  $\mathcal H$  do not exist. Thus, a leader edge from a higher round > r main leader vertex to a low round < r leader vertex will not exist. In our protocol, every main leader vertex must have a leader edge or a strong edge referencing a main leader vertices from some previous round, thereby forming a continuous leader path. Since  $\mathcal H$  cannot be skipped, any main leader vertex  $v_\ell$  must have a leader path to  $v_k$ .

The indirect commit rule of a main leader vertex in Multi-leader Angelfish is identical to the indirect commit rule of the leader vertex in Angelfish. Thus, the proof of the following claim (Claim 9) remains identical to Claim 2 except Claim 8 needs to be invoked (instead of Claim 1).

**Claim 9.** 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  $v_\ell$  in round  $r' \ge r$ , then  $P_j$  (directly or indirectly) commits  $v_k$  in round r.

**Claim 10.** 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  $v_\ell$  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 leader 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  $v_\ell$  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 leader paths from  $v_\ell$  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 leader 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 8, 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 11.** 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  $\min(r'_i, r'_i)$  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  $r_i \le r_j < r_i'$ . 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 9, party  $P_j$  commits at least  $\mathcal{ML}_{r_i'}[1]$  in round  $r_i'$ . Otherwise, by Claim 10, 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 leader paths from  $\mathcal{ML}_{r_i'}[1]$ . And, this process is repeated by indirectly committing leader vertices of earlier round with leader paths from  $\mathcal{ML}_{r_i'}[1]$  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 11 between any two pairs of honest parties we obtain the following corollary.

**Corollary 3.** Honest parties commit the same leaders in the same order.

The proof of the following total order lemma (Lemma 5) remains identical to Lemma 1 except Corollary 3 needs to be invoked (instead of Corollary 1).

**Lemma 5** (Total order). *Multi-leader Sailfish satisfies Total order.* 

**Agreement.** The agreement proof remains identical to Lemma 2 except Lemma 5 needs to be invoked (instead of Lemma 1).

**Integrity.** The integrity proof remains identical to Lemma 3.

**Validity.** We again rely on GST to prove validity and utilize the RBC protocol from Bracha RBC [7].

**Claim 12.** 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 by time t,  $P_i$  must have delivered round r-1 vertices and received r-1 votes, with their total count

being n-f. Since when an honest party receives a vote, it will broadcast the vote, then all honest parties must have received these votes by  $\max(t_g,t)+\Delta$ . By Property 2, all honest parties must have delivered round r-1 vertices by  $\max(t_g,t)+2\Delta$ . We then prove part (i) of the claim by considering  $P_i$  entering round by delivering round r main leader vertex (say  $v_k$ ) and by receiving  $\mathcal{TC}_{r-1}$ . 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  $\max(t_g,t)+\Delta$ .

Having delivered  $v_k$  or received  $\mathcal{T}_{r-1}$  (along with n-fround r-1 vertices and votes), an honest party  $P_i$  broadcasts (no-vote,  $P_k$ , r-1) for all  $P_k \in \mathcal{ML}_{r-1}$  if  $P_i$  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 broadcast (no-vote,  $P_k$ , r-1). Thus,  $L_r$  will receive  $\mathcal{N}\mathcal{V}\mathcal{C}_{r-1}^{P_k}$  by time max $(t_g,t)+3\Delta$ . For  $L_r$ , if it does not deliver round r-1 main leader vertex, it must instead deliver the main leader vertex of some earlier round r' < r, together with the TCs of all rounds between r' and r, as well as the corresponding  $\mathcal{NVC}$ . By Property 2,  $L_r$  delivers the round r' vertices by  $\max(t_g, t) + 2\Delta$ . It receives  $\mathcal{TC}_{r'}$  and the corresponding  $\mathcal{NVC}$  by  $\max(t_g,t) + \Delta$ . 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  $\mathcal{M} \mathcal{L}_{r'}[:x]$  and  $\mathcal{NVC}_{r'}^p$  where r' < r and  $p = \mathcal{ML}_{r'}[x+1], L_r$  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 13. 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 12, all honest parties will enter round r' or higher. Otherwise, all honest parties are in round r. Note that upon entering round r, all honest parties first compute the number of parties that will propose a vertex in round r-1, say c, based on the round r-2 vertices and votes. Then they wait for  $\max(c-f,0)$  round r-1 vertices. After that, all honest parties r\_bcast a round r vertex, or multicast a vote message, so each party will deliver round r vertices and receive  $\langle vt,r\rangle_*$  with a total number of n-f. 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  $\langle \texttt{timeout}, r \rangle$  and subsequently receive  $\mathcal{TC}_r$ . Having delivered  $v_k$  or received  $\mathcal{TC}_r$ , an honest party  $P_i$  sends  $\langle \texttt{no-vote}, P_k, r \rangle$  for all  $P_k \in \mathcal{ML}_r$  if  $P_i$  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  $\mathcal{T}_{C_r}$ , then all honest parties will multicast  $\langle \text{no-vote}, P_k, r \rangle$ . In addition, if  $L_{r+1}$  delivers a round r' < r leader vertex but does not deliver leader vertices from round r' + 1 to round r, it will also receive  $\mathcal{T}_{C_{r'+1}}, ... \mathcal{T}_{C_r}$ , so that there is a leader edge to round r' leader vertex. Moreover, since  $\langle \text{no-vote}, P_k, r' \rangle$  for each  $P_k \in \mathcal{M}_{L_{r'}}$  is multicast,  $L_{r+1}$  will also receive  $\mathcal{N}_r \mathcal{V}_{r'+1}^{P_k}$ . If all the leaders in round r' are delivered,  $L_{r+1}$  will also receive  $\mathcal{N}_r \mathcal{V}_{r'+1}^{P_k}$  since  $\langle \text{no-vote}, P_k, r' + 1 \rangle$  for each  $P_k \in \mathcal{M}_r \mathcal{L}_{r'+1}$  is multicast. By Property 2, if at least one honest party delivered the leader vertex corresponding to  $P_k$ ,  $L_{r+1}$  will deliver the leader vertex corresponding to  $P_k$ . Thus, one of the following cases occurs:

- If  $L_{r+1}$  delivers round r main leader vertex, then  $L_{r+1}$  either delivers a leader vertex corresponding to  $P_k$  in round r or receives  $\mathcal{NVC}_r^{P_k}$  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  $p = \mathcal{ML}_r[x+1]$ , it will advance to round r+1.
- If  $L_{r+1}$  does not deliver round r main leader vertex, then it will deliver a leader vertex corresponding to  $P_k$  in round r', together with  $\mathcal{T} C_{r'+1}, ... \mathcal{T} C_r$ .
  - If not all the leaders in round r' are delivered, then  $L_{r+1}$  receives  $\mathcal{NVC}_{r'}^{P_k}$  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.
  - If all the leaders in round r' are delivered, then  $L_{r+1}$  receives  $\mathcal{NVC}_{r'+1}^{P_k}$  for all  $P_k \in \mathcal{ML}_{r'+1}$ . Since  $L_{r+1}$  waits for  $\mathcal{NVC}_{r'+1}^p$  where  $p = \mathcal{ML}_{r'+1}[x+1]$ ,  $L_{r+1}$  will advance to round r+1.

The proof of the following claim (Claim 14) remains identical to Claim 7 except Claim 12 needs to be invoked (instead of Claim 4).

**Claim 14.** If the first honest party to enter round r does so after GST and  $L_r$  is honest, then there exists at least n - f round r + 1 vertices and votes with strong edges to round r main leader vertex.

By the commit rule and Claim 14, the following corollary follows.

**Corollary 4.** 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 6) remains identical to Lemma 4 except Corollary 4 needs to be invoked (instead of Corollary 2).

Lemma 6 (Validity). Multi-leader Sailfish satisfies Validity.

As demonstrated in Claim 14, a round r main leader vertex (proposed by an honest leader) is always committed by round r+1 (after GST).