Making Uncertified DAG BFT Provably Live with Linear Payload and Quadratic Metadata Communication

Nikita Polyanskii
IOTA Foundation
nikitapolyansky@gmail.com

Sebastian Müller
Aix-Marseille Université
sebastian.muller@univ-amu.fr

Ilya Vorobyev
IOTA Foundation
ilia.vorobev@iota.org

Abstract

Uncertified partially synchronous DAG BFT protocols such as Mysticeti achieve the lowest latency among DAG protocols by omitting per-block quorum certificates, but they lack rigorous liveness proofs: honest parties can desynchronize and never commit a leader again, even under synchrony. To avoid the quadratic communication cost of pushing full blocks, production implementations pull missing ancestors, which results in consensus request amplification, large tail latency, and vulnerability to Byzantine dissemination.

We present Starfish, whose new Push pacemaker gives uncertified DAGs their first rigorous liveness proof, together with safety proofs and latency bounds in the actual message delay δ. Starfish makes Push practical by separating payloads from headers: parties push headers and Reed-Solomon fragments, data-availability certificates form on the DAG itself, and worst-case payload communication becomes linear, at the cost of increasing transaction latency by one round. We also show how multi-signatures and delayed history pushes can be applied to Starfish and Mysticeti, resulting in new protocols with quadratic happy-case metadata communication, the lowest among partially synchronous DAG BFTs.

Without faults, at 100 validators on a synthetic WAN, Starfish sustains ≈ 257 k tx/s at sub-second median latency against Mysticeti's ≈ 190 k. Controlled experiments show that Byzantine selective dissemination can drive honest-transaction median latency under Pull to tens of seconds in both protocols; under Push dissemination, Starfish holds it within about two seconds across the whole fault range, at substantially lower communication cost than Push on Mysticeti. Starfish has replaced Mysticeti on a public blockchain's mainnet. An observational study comparing each validator with itself across the switch finds per-validator consensus requests down 86% and p99 consensus latency down 27%; well-connected validators pay the added round at +16% consensus latency, and bytes sent per sequenced byte rise by 133%.

1 Introduction

Several recent blockchains adopt Byzantine Fault-Tolerant (BFT) consensus based on Directed Acyclic Graphs (DAGs) of blocks to scale throughput by parallelizing block proposals. These protocols operate in the partial synchrony model [12]: among $n = 3f + 1$ parties up to f are Byzantine, and after an unknown Global Stabilization Time (GST) messages are delivered within an unknown bound $\delta \leq \Delta$, where Δ is a known upper bound. Two classes of DAG protocols exist. *Certified DAGs* [1, 9, 31, 33, 34] disseminate blocks by reliable broadcast (RBC) and achieve linear amortized communication with efficient RBCs [7, 10, 32], whereas *uncertified DAGs* [2, 21] rely on best-effort broadcast and thus propose one block per network delay δ instead of $\geq 2\delta$, which gives them the *lowest transaction latency* among DAG protocols.

Despite production adoption (e.g., Mysticeti in Sui), uncertified DAGs have three gaps. First, they lack rigorous liveness proofs: [27] gives an explicit counterexample in which honest Mysticeti parties advance rounds without creating blocks and never commit a leader again. Cordial Miners [21] and the patched Mysticeti of [27] assume synchronization after GST rather than proving it (Section 7). Second, payload is coupled with consensus metadata, so pushing a block with the history a peer may lack costs $O(Mn^2)$ worst-case payload communication per round (M is the total payload per round). Implementations therefore pull: validators broadcast only their own blocks and fetch missing ancestors on demand, one round trip per missing parent, which in production surfaces as request amplification, lagging validators, and tail latency. Third, metadata per round is $O(\kappa n^2)$ (κ is the security parameter), which can result in $O(\kappa n^4)$ communication in the worst case. These gaps raise our central question: can uncertified DAGs be made provably live while staying practical?

Our Contributions. We answer affirmatively:

- **Liveness: the Push pacemaker.** Our *Push pacemaker*, triggering block creation and round advancement, adds new conditions (Table 2). **A2** prevents round jumps from omitting blocks needed to vote for and certify leaders. **C3**, together with immediate history broadcasts, ensures that honest parties enter each round beyond the highest honest round at GST within Δ of one another. The new pacemaker enables the first rigorous liveness proof for uncertified DAGs and the first average latency bounds in the actual message delay δ, tight for Mysticeti (Theorem 11).
- **Practicality: Starfish makes Push affordable.** The liveness guarantee relies on pushing causal history a peer may lack; with full blocks this costs $O(Mn^2)$ worst-case payload communication. Starfish decouples payload from headers: parties push headers and Reed-Solomon fragments, and $2f+1$

Table 1. Partially synchronous DAG BFT protocols: idealized latency and per-round communication (Appendix E). The Mysticeti row uses our Push pacemaker. Push and Lazy-Push denote immediate and delayed history broadcast. CRB denotes consistent reliable broadcast; RS-BRB denotes Reed–Solomon-based Byzantine reliable broadcast; Opt denotes optimistic. Here n is the number of validators, M the average total payload per round, κ the security parameter, δ the actual message delay, and Δ its known upper bound. Communication bounds apply in both happy and worst cases unless distinguished.

Protocol	Broadcast / Pacemaker	Consensus latency	Transaction latency overhead	One failure leader	Communication complexity	Crypto assump.
Bullshark [34]	CRB [9]	7δ	$+1\delta$	$+4\Delta + 4\delta$	happy: $O(\kappa n^3 + Mn)$ worst: $O(\kappa n^4 + Mn^2)$	Hash + PKI + Multi-sig
Sailfish++ [32]	Opt RS-BRB [32]	5δ	$+1\delta$	$+4\Delta + 2\delta$	$O(\kappa n^3 \log n + Mn)$	Hash
Mysticeti [2]	Push (this paper)	4δ	$+0.5\delta$	$+2\Delta + \delta$	$O(\kappa n^4 + Mn^2)$	Hash + PKI
Mysticeti-L	Lazy-Push	4δ	$+0.5\delta$	$+8\Delta + \delta$	happy: $O(\kappa n^2 + Mn)$ worst: $O(\kappa n^3 + Mn^2)$	Hash + PKI + Multi-sig
Starfish	Push	5δ	$+0.5\delta$	$+2\Delta + \delta$	$O(\kappa n^4 + Mn)$	Hash + PKI
Starfish-L	Lazy-Push	6δ	$+0.5\delta$	$+8\Delta + \delta$	happy: $O(\kappa n^2 + Mn)$ worst: $O(\kappa n^3 \log n + Mn)$	Hash + PKI + Multi-sig

payload-availability acknowledgments recorded in headers form a Data Availability Certificate (DAC) on the DAG before a payload is sequenced (Section 5, Figure 2). Payload communication becomes $O(Mn)$, which is order-optimal, at the cost of one additional acknowledgment round. Starfish is the variant deployed in mainnet production.

• **Optional compression: Starfish-L and Mysticeti-L.** We show how multi-signatures make non-leader headers constant-size (Section 6) and how the *Lazy-Push pacemaker* delays history broadcasts (Section 4); together they bring happy-case metadata communication from $O(\kappa n^4)$ to $O(\kappa n^2)$ per round, the lowest in Table 1 and in our evaluation (Section 8.2). Both techniques apply to Starfish and Mysticeti, resulting in Starfish-L and Mysticeti-L. The price is an additional multi-signature assumption, which in practice requires more computational resources, growing steeply with committee size; our BLS implementations appear as Starfish-BLS and Mysticeti-BLS.

• **Unified implementation and controlled evaluation.** We implement Starfish, Mysticeti, their BLS variants, and the certified-DAG baseline Sailfish++ in one Rust testbed with shared infrastructure and matched hardware and workloads (Section 8.1). Switching between Pull and Push in both Starfish and Mysticeti separates what Push buys from what Starfish's coding buys. Under a leader chain bomb attack, in which Byzantine validators release withheld chains only to the leader, Pull drives honest p50 latency to tens of seconds in both protocols; Push keeps measured p50 within 2.1 s, and across the sweep Starfish's headers and fragments cost 1.9–2.3 bytes per honest sequenced byte against 7.2–12.4 for Mysticeti's full blocks (Section 8.4). Without faults, at 100 validators on a synthetic WAN, the three new protocols sustain $\approx$ 257–262 k tx/s against Mysticeti's $\approx$ 190 k, and Mysticeti-BLS keeps Mysticeti's pre-saturation p50 of $\approx$ 0.5 s at a fraction of its communication and several times its CPU (Figure 1).

• **Production deployment and its tradeoffs.** Starfish replaced Mysticeti on a public blockchain's mainnet; we compare each paired validator with itself over three-day windows before and after the switch (Section 8.5). Consensus requests fall by 86% and p99 consensus latency falls by 27%; well-connected validators pay the additional acknowledgment round at +16% typical consensus latency, while the 5 less-connected validators see a median increase of 5.8%; bytes sent per sequenced byte rise by 133% (Table 3). The study is observational, but the switch changed only one consensus-selection parameter at an epoch boundary.

Table 1 compares our protocols with prior certified and uncertified DAG BFTs; see Appendix E in the supplementary material for a detailed discussion.

2 Preliminaries

System Model. We consider a message-passing system $\mathcal{P} := \{p_1, \ldots, p_n\}$ of $n = 3f + 1$ parties. At most f parties are *Byzantine* (or *faulty*) and may deviate arbitrarily from the protocol; the remaining parties are *honest* (or *correct*) and follow it throughout the execution. We use M for the average total payload size per round across all parties.

Under *partial synchrony* [12], message delays are arbitrary before an unknown *Global Stabilization Time* (GST). A message sent between honest parties at time t is delivered by time $\max(t, \text{GST}) + \delta$, where the actual delay bound $\delta \leq \Delta$ is unknown and Δ is known. The *happy case* is a post-GST execution in which all parties are honest.

Cryptographic Assumptions. We assume a computationally bounded adversary with security parameter κ, collision-resistant hashes with κ-bit outputs, and unforgeable digital signatures of size $O(\kappa)$. A public key infrastructure (PKI) associates each party with a public-private key pair; $(m)_p$ denotes m together with party p's signature. For compression, we also use *multi-signatures* [4, 5]: any subset of $2f + 1$ parties can aggregate their signatures on the same message

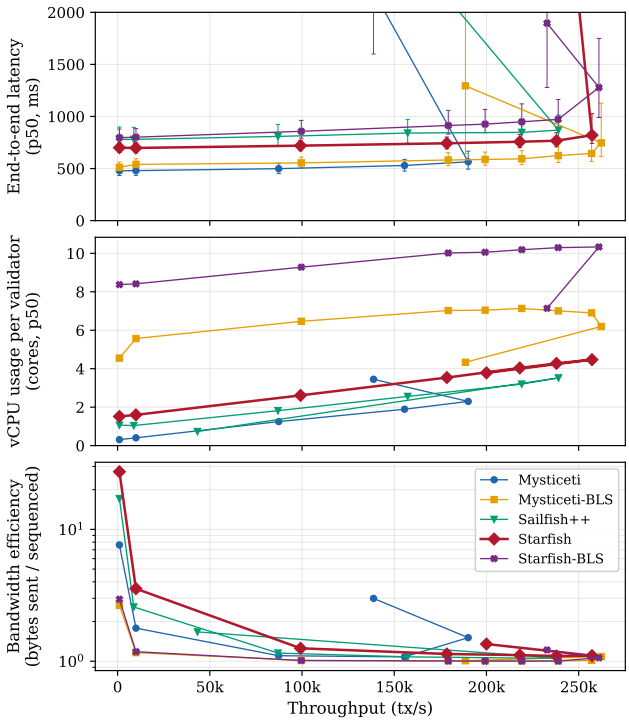


Figure 1. Throughput–latency profile at $n = 100$ validators on a synthetic geo-distributed testbed. From top to bottom: end-to-end transaction latency (p50, with p25/p75 error bars; clipped at 2,000 ms), per-validator vCPU usage (cores, p50), and bandwidth efficiency (bytes-sent / bytes-sequenced, log scale). See Section 8.3 for details.

into a signature of size $O(\kappa)$. Their notation is introduced in Section 6. The "Crypto" column of Table 1 lists which of these primitives each protocol requires.

Broadcast. Certified DAGs use Byzantine reliable broadcast (RBC) [6] for blocks to guarantee data availability and prevent equivocation. RBC satisfies the first three properties below; Byzantine atomic broadcast (BAB) additionally provides total order.

Definition 1 (Byzantine Atomic Broadcast)**.** *Payloads are identified by their sender and round (p, r). Each honest sender broadcasts at most one payload per round. A Byzantine atomic broadcast protocol satisfies:*

- ***Validity.*** *Every payload broadcast by an honest party is eventually delivered by all honest parties.*
- ***Totality.*** *If one honest party delivers a value for a sender-round pair, every honest party eventually delivers a value for that pair.*
- ***Agreement.*** *No two honest parties deliver different values for the same sender-round pair.*
- ***Total order.*** *If an honest party delivers one payload before another, no honest party delivers them in the opposite order.*

Latency. For an honestly created block, consensus latency at an honest party measures the time from block creation until that party sequences its payload. Transaction latency additionally includes the wait for block inclusion after arrival at the submitting party. For analysis, we take the maximum over honest receiving parties. Table 1 separates idealized consensus latency from this inclusion delay; experiments report transaction latency statistics at individual validators.

3 Uncertified DAGs and Commit Rules

We present the DAG model and leader commit rules common to uncertified DAG-based BFT protocols such as Cordial Miners [21] and Mysticeti [2]; Starfish is built on top of it. Unlike certified DAGs, uncertified DAGs do not use RBC, so Byzantine parties may equivocate (create conflicting blocks). The commit rules must therefore ensure that all honest parties reach consistent decisions despite equivocations.

Block Format. Each block of party p_i is signed and has the following format $B = (r, \texttt{ancestors}, \texttt{payload})_{p_i}$, where r is the round number; the ordered list `ancestors` contains at most n hash references[1] to the latest blocks created at or before round $r-1$ by distinct parties such that there are $2f+1$ blocks created at round $r-1$; the payload of the block is `payload` (also denoted as $\texttt{payload}(B)$).

Block DAG and Rounds. Parties concurrently create and multicast signed blocks in rounds starting at 1, with n predefined round-0 blocks known to all parties. Blocks form a directed acyclic graph (DAG) whose vertices are blocks and whose edges point to explicitly referenced ancestors. The pacemaker (Section 4) governs block creation and round advancement.

Leader Block and Scheduler. To order payloads in the blocks of the DAG, parties use the concept of leaders. They agree beforehand on the round-robin leader scheduler: party $p_{(r \bmod n)+1}$ proposes the leader block of round r[2]. Leader blocks have the same[3] structure as other blocks, but the difference is that parties wait for at least a timeout δ_{LT} in round $r+1$ until they receive a leader block of round r.

Slot and Leader Slot. A *slot* is a pair (p_i, r) of a party and a round. A slot is a *leader slot* if p_i is the designated leader of round r, i.e., $i = (r \bmod n) + 1$. Each leader slot is initially in the state UNDECIDED; by interpreting the DAG structure, each party attempts to make a decision TO-COMMIT or TO-SKIP for every leader slot.

[1] A hash reference is the hash of the block, including its signature.
[2] Cordial Miners assigns only one leader per three rounds, while Mysticeti supports multiple leaders in every round. We stick with the option of one leader per round since we prove the latency improvement in this case, compared to Cordial Miners, while we have not observed latency improvements in both theory and practice in the case of the multileader option.
[3] For Starfish, Mysticeti and Cordial Miners, the structure of leader blocks is the same. For Starfish-L and Mysticeti-L, discussed in Section 6, the structure of non-leader and leader blocks differs.

Table 2. New pacemakers Push and Lazy-Push for uncertified DAGs. Modifications, compared to prior works, are highlighted in blue. The leader timeout δ_{LT} is 2Δ in Push and 8Δ in Lazy-Push. The broadcast timeout δ_{BT} is 0 in Push and 3Δ in Lazy-Push.

Advance round $(r-1) \rightarrow r$	**Create block of round** r	**Broadcast history**
A1: local DAG contains $2f+1$ blocks of round $r-1$ AND **A2:** created block of round $r-1$	**C1:** wait for leader and votes: **L1:** local DAG contains leader block of round $r-1$ AND **L2:** local DAG contains $2f+1$ votes for leader of round $r-2$ OR contains *skip pattern* for leader of round $r-2$ OR **C2:** expired timeout δ_{LT} OR **C3:** local DAG contains $2f+1$ round-r blocks from distinct parties	**B1:** created block • own block: now • unknown history: after δ_{BT} OR **B2:** advanced round • unknown history: after δ_{BT}

Patterns for Leader Slot. A block from round $r+1$ votes for the leader block identified by the first reference to the round-r leader slot in its signed `ancestors` list; if no such reference is present, it does not vote for that slot. To make the decision, parties find specific patterns in their local DAGs that allow them to commit the leaders or skip the attempt to find certificates for leader blocks:

- The *certificate pattern* for leader slot (p_i, r): blocks from at least $2f+1$ parties at round $r+1$ vote for the same block B of round r created by the leader p_i. We then say that B is *certified*. Any block from round $r+2$ whose history contains such a pattern is called a *quorum certificate* (QC) for the leader block B.
- The *skip pattern* for leader slot (p_i, r): blocks from at least $2f+1$ parties at round $r+1$ *do not* vote for a block of round r created by leader p_i. Note that there might be no proposal for the leader slot or more than one proposal in the case of equivocations. The slot is skipped by a party if, for any leader block, the party observes $2f+1$ blocks in the next round that do not vote for it.

Visual examples of these patterns are illustrated in Figures S1 and S2 in the appendix.

Commit Rule for Leader Slots. Parties apply *direct decision rules* to all leader slots, then process any remaining UNDECIDED slots using the *indirect decision rule*, proceeding from higher to lower rounds:

- *Direct commit rule:* The party marks a leader slot as TO-COMMIT if the local DAG contains QCs from $2f+1$ distinct parties for that slot. The unique certified leader block from that slot is then called *committed*.
- *Direct skip rule*: The direct decision rule marks a leader slot as TO-SKIP if the local DAG contains a skip pattern for that slot.

For any slot remaining UNDECIDED after direct rules, the party resorts to the *indirect decision rule*. It initially searches for an *anchor*, which is defined as the leader slot with the smallest round number $r' \geq r+3$ that is marked as either UNDECIDED or TO-COMMIT. If the anchor is still marked as UNDECIDED, the party leaves the slot as UNDECIDED.

- *Indirect commit rule*: if the anchor is marked as TO-COMMIT, the party marks the leader slot as TO-COMMIT if the anchor contains a path to a QC for a block from the leader slot. The unique certified leader block from that slot is *committed*.
- *Indirect skip rule*: if the anchor is marked as TO-COMMIT, the party marks the leader slot as TO-SKIP if there is no path from the anchor to a QC for a block from the leader slot.

Examples of direct and indirect decision rules are demonstrated in Figures S1 and S2, and a summary table is provided in Table S2 in the appendix. A pseudocode for this commit rule is depicted in Algorithm S3.

Sequence of Committed Leaders. After applying direct and indirect decisions to all leader slots, each party derives an ordered sequence of committed leader blocks. Specifically, the party iterates over that sequence of leader slots, gets certified leader blocks from slots marked as TO-COMMIT and skips all slots marked as TO-SKIP until it reaches the first slot marked as UNDECIDED.

Linearization of DAG. To sequence payloads in Mysticeti and Cordial Miners, one needs to linearize all the blocks in the DAG. Let $\mathsf{hist}(B)$ denote the *causal history* of block B—the set of all blocks transitively reachable from B via ancestor references. Since the blocks are signed, one can derive a fixed topological ordering sort on blocks in the DAG. Given a sequence of committed leader blocks $C_1, \ldots, C_N$, a total ordering over all blocks in $\mathsf{hist}(C_N)$ is constructed by applying sort to each committed leader's history in sequence, iteratively excluding blocks already ordered from previous leaders (after removing equivocations; see Appendix A.4).

Safety Properties. The safety of uncertified DAG BFTs follows from two facts: *the quorum intersection arguments* (any two sets of $2f+1$ parties share at least one honest party) and *the self-ancestry-based blockchain* (an honest party always references its own block). A quorum of parties participate in every commit decision in the view of any party. Any two quorums share at least one honest party and this honest party prevents a decision from conflicting with a prior one. For completeness, we provide the key properties and their proofs, which were established in [2, 21], in Appendix C.

4 Pacemakers for Uncertified DAGs

In partially synchronous BFT protocols, a *pacemaker* coordinates parties to ensure progress after GST. For uncertified

DAG-based protocols, the pacemaker governs three events: (1) when parties advance rounds, (2) when they create blocks, and (3) when they disseminate consensus-related data to peers.

Existing uncertified DAG protocols lack rigorous liveness proofs. Mysticeti's Lemma 8 [2] claims that after GST all honest parties enter the same round within Δ, but the proof is incomplete [31]—an explicit counterexample demonstrates an infinite trace where no leader ever commits [27]. Cordial Miners' Proposition 38 [21] similarly assumes without proof that all honest parties synchronize after GST. The patched protocol of [27], despite its mechanized proof, also assumes synchronization-after-GST without proof (see Section 7 for details). Round jumps omit blocks needed for leader certificates; a bounded-delay replay contrasts the original and Push pacemakers in Appendix I.1.

We address these gaps by introducing new pacemaker conditions (highlighted in blue in Table 2) that enable formal synchronization proofs. We separate round advancement from block creation. **A2** requires parties to create their blocks before advancing, preventing round jumps from omitting blocks needed to vote for and certify leaders. **C3** lets a lagging party create its block once its local DAG contains $2f + 1$ blocks from distinct parties in that round, without waiting for votes or timeouts. **B2** triggers history broadcasts on round advancement. Under Push, these immediate broadcasts supply the quorums needed by **C3**, ensuring that all honest parties enter any round beyond the highest honest round at GST within Δ of the first honest entrant (Lemma 6). Without **C3**, synchronization and commitment can be delayed by arbitrarily many Δ as the number of missed rounds grows (Appendix D.4). This provides a general fix applicable to both Mysticeti and Cordial Miners.

Party's Local View. Let DAG_i denote party p_i's local DAG. Every block expresses knowledge about the existence of other blocks in its causal history. Therefore, we can define the set of blocks known to party p_j from the point of view of party p_i as follows $\mathsf{known}_i(j) = H_{i,j} \cup S_{i,j}$, where $H_{i,j} = \bigcup \mathsf{hist}(B)$ is the union of histories over all blocks B in DAG_i created by party p_j; $S_{i,j}$ is the set of all blocks that have been sent from p_i to p_j. A potentially unknown history of party p_j from the perspective of p_i is then defined as

$$\mathsf{unknown}_i(j) = \mathsf{DAG}_i \setminus \mathsf{known}_i(j). \tag{1}$$

Causal Availability. A received block is integrated in the local DAG only when its ancestors are already available in the local DAG. This is an important distinction from certified DAGs, where a certificate verifies a block's availability in a quorum of parties and allows the block to be included without such a check.

Cordial Dissemination. Since Byzantine parties could equivocate or send their blocks exclusively to some parties, satisfying the causal availability requirement requires careful rules for dissemination. Each party has to ensure that every honest recipient of its newly proposed block B is aware of $\mathsf{hist}(B)$. For this purpose, parties follow the principle of *cordial dissemination* [30] which states: "Send to others blocks you know and think they need." Thereby, each party tracks which blocks are known or already sent to each peer. When sending its block B, party p_i also broadcasts to peer p_j a potentially unknown history of that block defined as the intersection $\mathsf{hist}(B) \cap \mathsf{unknown}_i(j)$.

We consider two timing variants of pacemakers for cordial dissemination: *Push* sends the unknown history immediately ($\delta_{BT} = 0$) when a block is created or a round is advanced, while *Lazy-Push* sends own blocks immediately while delaying the unknown-history transmission by $\delta_{BT} = 3\Delta$ to reduce metadata communication in the happy case. When a broadcast is triggered, party p_i freezes (snapshots) its local DAG. When the broadcast timeout expires, it sends to each peer p_j the information from this snapshot that is still potentially unknown to p_j according to p_i's current estimate $\mathsf{known}_i(j)$: full blocks in Mysticeti/Mysticeti-L and headers in Starfish/Starfish-L. This lets Lazy-Push tailor the delayed transmission to the sender's up-to-date view of each peer's knowledge and often leaves little or no extra history to send in the happy case. The Push variant ensures faster convergence with leader timeout $\delta_{LT} = 2\Delta$. Under Lazy-Push, the synchronization window grows to 4Δ, and an honest previous-round block may need up to another 4Δ before it can be integrated into all honest local DAGs because its missing history can arrive only via the delayed unknown-history broadcast (3Δ delay plus one network delay). Therefore Lazy-Push uses leader timeout $\delta_{LT} = 8\Delta$.

Pacemaker Conditions. The pacemaker conditions are summarized in Table 2, with our modifications compared to Mysticeti [2] and Cordial Miners [21] highlighted in blue. Both Push and Lazy-Push variants use the same core conditions but differ in timing parameters: Push uses $\delta_{LT} = 2\Delta$ and $\delta_{BT} = 0$, while Lazy-Push uses $\delta_{LT} = 8\Delta$ and $\delta_{BT} = 3\Delta$. The protocol operates as follows (the flow is implemented in ExecuteRound of Algorithm S1):

Round advancement. A party advances from round $r-1$ to round r when two conditions are met: its local DAG contains $2f + 1$ round-$(r-1)$ blocks from distinct parties (**A1**), and it has created its own block in round $r-1$ (**A2**, *new*). The latter prevents parties from "jumping" ahead without contributing blocks that serve as votes and certificates needed for leader commitment.

Block creation. After advancing to round r, a party creates its block when one of three conditions is satisfied. The primary condition (**C1**) requires receiving the leader block from round $r-1$ (**L1**) and observing that the leader of round $r-2$ either received sufficient votes or was skipped (**L2**, *modified to add skip pattern*). When creating a block via **C1**, the party should choose its round-$(r-1)$ references in `ancestors` so that they witness the satisfied **L2** disjunct (votes for the leader of round $r-2$ or a skip pattern), which matters under

equivocation. This waiting ensures honest leaders get committed. If **C1** cannot be satisfied (e.g., due to a faulty leader), a leader timeout of δ_{LT} (**C2**) allows the party to proceed with block creation. The safe jump condition (**C3**, *new*) allows a party to create its block upon receiving $2f+1$ round-r blocks from distinct parties, enabling faster synchronization when lagging behind.

Broadcasting history. A party broadcasts the unknown history $\text{unknown}_i(j)$, defined in (1), to peers when it creates a block (**B1**) or advances to a new round (**B2**, *new*). In the Push variant, this broadcast happens immediately ($\delta_{BT} = 0$), while Lazy-Push sends own blocks immediately and delays the remaining unknown-history transmission by $\delta_{BT} = 3\Delta$ to reduce metadata communication in the happy case. Concretely, when the broadcast is triggered, the sender snapshots its local DAG. When the broadcast timeout expires, it sends to each peer the entries in this snapshot that are not in its current estimate of that peer's known set $\text{known}_i(j)$: full blocks in Mysticeti/Mysticeti-L and headers in Starfish/Starfish-L. Broadcasting on round advancement is essential to prove convergence in party synchronization after GST.

Liveness Properties. The new pacemaker conditions enable rigorous liveness proofs (see Appendix D). Let $r_{\max}$ denote the largest round among honest parties at GST. The key results are:

- **Synchronicity after GST** (Lemmas 6 and 15): For the Push pacemaker, all honest parties enter any round $r > r_{\max}$ within time Δ of each other and create their round-r blocks within time Δ of each other. For Lazy-Push, the synchronization window is 4Δ instead of Δ due to delayed broadcast; moreover, because causal availability may delay integration of the last honest previous-round block by another 4Δ, the leader timeout must be increased to 8Δ.
- **Honest leaders committed** (Lemma 10): Any leader block created by an honest party in round $r \geq r_{\max}$ will be marked TO-COMMIT by the direct decision rule.
- **All leaders decided** (Lemma 12): After GST, any undecided leader block will eventually be decided, i.e., marked as TO-COMMIT or TO-SKIP.

The first property was assumed without proof in prior work [2, 21, 27].

4.1 Latency Guarantees after GST

The above results yield liveness using the known network bound Δ. However, when the actual network delay δ is smaller than Δ, the pacemaker achieves much tighter guarantees in the happy case. For the Push variant, all honest parties enter any round $r > r_{\max}$ within time δ of each other, so synchronization is immediate after GST. For the Lazy-Push variant, the same δ-synchronization is provably achieved a bit later at rounds $r > r_{\max} + n$ because of the delayed unknown history broadcast. We are able to prove that such a synchronization happens when the slowest party becomes the leader in the leader scheduler.

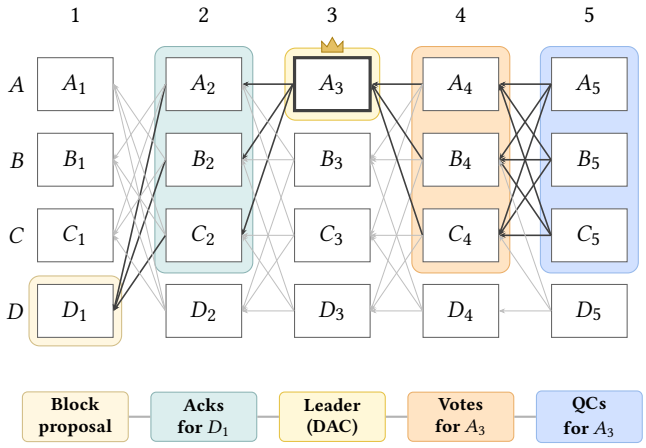


Figure 2. Commit rule in Starfish. Payload in block D1 becomes ready for sequencing when $2f+1$ acknowledgments (A2, B2, C2) are reachable from a committed leader block (A3), which serves as a Data Availability Certificate. Leader A3 is committed when $2f+1$ certificates (A5, B5, C5) each observe $2f+1$ votes (A4, B4, C4). Sequencing requires 5 rounds: proposal, acknowledgments, leader, votes, certificates.

This δ-synchronization property in the happy case enables precise latency analysis. Beyond the all-honest scenario, Byzantine leaders can add an extra delay to the latency. After GST, the time between round r and round $r+1$ blocks created by *slowest* honest parties in these rounds is:

- at most $\delta + \delta_{LT}$ when a leader of round r or round $r-1$ is Byzantine (and creates blocks).
- at most δ when the leader of round r is honest and the round-$(r-1)$ leader is honest or a Byzantine failure (that fails to create a block).

Since $\delta_{LT} = 2\Delta$ for Push and $\delta_{LT} = 8\Delta$ for Lazy-Push, an active Byzantine leader adds at most 2Δ delay for Push or 8Δ delay for Lazy-Push, affecting up to two consecutive rounds (through conditions **C1** in rounds r and $r+1$), while a Byzantine leader failure with an honest leader successor adds only one such delay due to the skip pattern clause in **L2**. Combined with specific commit rules, the above key bounds yield concrete latency bounds summarized in Theorem 11, with detailed proofs in Appendix D.5.

5 Starfish Protocol

This section provides an overview of the Starfish protocol. A detailed description is provided in Appendix A, and the complete pseudocode is presented in Appendix B.

Starfish builds on the uncertified DAG framework described earlier, including the commit rules (Section 3) and the Push pacemaker (Section 4). Starfish introduces two key innovations:

1. *Decoupling payload from metadata*: Block headers contain only a commitment to the encoded payload, allowing

leader block headers to be committed before payload is fully available. Payload is sequenced using Data Availability Certificates (DACs) formed directly on the DAG.

2. *Encoded cordial dissemination*: payload is encoded using Reed-Solomon codes, and parties share encoded fragments rather than full payload.

These reduce the worst-case payload communication costs from $O(Mn^2)$ to $O(Mn)$, while increasing the latency to sequence payload by one additional round.

Block Format. Starfish decouples payload from block headers, unlike the protocols of Section 3. Party p_i's signed header is $B_{\text{header}} = (r, \texttt{ancestors}, \texttt{acks}, \texttt{payloadCommit})_{p_i}$, where r is the round number; `ancestors` contains hash references to parent block headers; `acks` contains hash references to block headers whose payload the party has locally verified against the commitment in that header; and `payloadCommit` is a commitment to the encoded payload. This separation allows leader block headers to be committed based on headers alone, while payload availability is tracked separately via acknowledgments. The full block includes the header and the payload: $B = (B_{\text{header}}, \texttt{payload})$.

DAG Based on Block Headers. Compared to prior works on uncertified DAGs, parties in Starfish keep track of their local DAG by constructing it from block headers (not full blocks).

Reed-Solomon Coding and Reconstruction. In Starfish, the block author encodes a payload of size T into $n = 3f + 1$ fragments using a systematic Reed-Solomon erasure code of dimension $k = f + 1$ over $\mathbb{F}_q$ (where q is the smallest power of 2 with $q \geq n$) [29]; we denote the ith fragment by $\text{frag}_i(B)$. It commits to the fragment set by including a Merkle root over fragment hashes as `payloadCommit` in the block header; fragments are disseminated with Merkle proofs and accepted only if the proof verifies against `payloadCommit`. A party reconstructs the payload from any $k = f + 1$ verified fragments via RS erasure decoding, which can be implemented in $O(T \log n)$ time using Fast Fourier Transform (FFT)-based algorithms [23].

Encoded Fragment Dissemination. Whenever the Push pacemaker triggers cordial dissemination of block headers (Section 4), parties also disseminate payload. The payload of own blocks is disseminated immediately. For blocks B of other parties whose data became available since the last broadcast, party p_i sends $\text{frag}_i(B)$ together with the corresponding Merkle proof and `payloadCommit` of B to each other party. This yields $O(Mn + \kappa n^3 \log n)$ payload communication cost.

Data Availability Certificates. Since block headers are decoupled from payload, the data of a block may not yet be available to all parties when the header is received. To track availability, each block header includes an `acks` field containing hash references to block headers whose payload the party has locally verified against the commitment in that header. We say that a block B is a *Data Availability Certificate (DAC)* for the payload in a block C if blocks from $2f + 1$ distinct parties in $\text{hist}(B)$ acknowledge availability of the payload in C. For example, in Figure 2, block $A3$ is a DAC for $D1$ because $2f + 1$ blocks ($A2$, $B2$, $C2$) in its causal history acknowledge the payload of $D1$. Since at least $f + 1$ of the acknowledging parties are honest and each holds the verified payload, the payload of C remains recoverable even if all Byzantine parties withhold their fragments, provided honest parties retain what they acknowledged (Appendix J).

Sequencing payload. For each block B, let $\text{hist}_{\text{DA}}(B)$ denote the set of all blocks D in $\text{hist}(B)$ such that B is a DAC for the payload of D. Following the commit rule from Section 3, parties obtain a sequence of committed leader blocks $C_1, \ldots, C_N$. Using the fixed topological ordering sort, a total ordering over sequenced payload is constructed by applying sort to $\text{hist}_{\text{DA}}(\cdot)$ of each committed leader in sequence, iteratively excluding payloads already ordered from previous leaders (and removing equivocations; formal definition in Appendix A.5). A commit rule and data availability mechanism of Starfish is depicted in Figure 2. Figure S3 provides a longer walkthrough in the appendix.

6 Improving Metadata Communication

Two techniques address the metadata communication overhead: compressing Starfish block headers with multi-signatures, and disseminating them under the Lazy-Push pacemaker (Section 4). Together they yield Starfish-L; applied to Mysticeti they yield the simpler Mysticeti-L. Pseudocode for both is in Appendix B. Beyond the standard signature $(m)_{p_i}$ of Section 2, we write $\langle m \rangle_{p_i}$ for m together with p_i's partial signature, and $\langle m \rangle_{\text{agg}}$ for the multi-signature that aggregates partial signatures on m from $2f + 1$ distinct parties.

Compressing Ancestors and Leader Certificates. Ancestors play two roles in an uncertified DAG: they carry the votes and certificates that drive the commit rule, and they are traversed to sequence payloads. Compression keeps both: multi-signatures replace the quorum-sized certificate patterns of the first role, while leader headers retain enough references for the second. In Starfish-L non-leader headers and Mysticeti-L non-leader blocks, `ancestors` holds only `self`, a hash reference to the author's own previous header, and optionally $\langle \texttt{leader}_{r-1} \rangle_{p_i}$, a hash reference to the round-$(r-1)$ leader carrying the author's partial signature, included when that leader header is available at creation time (**C1** in Table 2). These partial signatures are the votes: a party that collects $2f + 1$ of them for the round-$(r-2)$ leader, carried in round-$(r-1)$ headers, aggregates them into the leader certificate $\langle \texttt{leader}_{r-2} \rangle_{\text{agg}}$ and publishes it in the optional field `leaderCert` of its own round-r header. Leader headers additionally retain at most n hash references to the latest headers of distinct parties, at least $2f + 1$ of them from round $r - 1$; together with the per-party self-chains, these preserve

both the traversal needed for sequencing and the quorum-intersection argument behind safety (Appendix C.2).

Round Certificates. A compressed header no longer exhibits the $2f+1$ round-$(r-1)$ references that witnessed its author's round advancement, so both protocols attest that advancement with a *threshold clock*: every round-r header carries $\langle r \rangle_{p_i}$, and $2f+1$ such partial signatures aggregate into a constant-size round certificate $\langle r \rangle_{\text{agg}}$, carried as `roundCert` in round-$(r+1)$ headers. Advancement itself is unchanged: a party still needs $2f+1$ round-r headers in its local DAG (**A1**) and its own round-r header (**A2**); the certificate alone is not sufficient.

Reversing Acknowledgments. In Starfish, each block header contains an `acks` field with hash references to block headers whose payload the party has locally verified—on average n references requiring $O(\kappa n)$ per header. Starfish-L reverses the direction of these attestations [9, 16]: "others acknowledged me" replaces "I acknowledge others." A party p_j that verifies the payload of a block B created by p_i returns its partial signature $\langle \texttt{payloadCommit}_B \rangle_{p_j}$ on B's data commitment to p_i, who aggregates $2f+1$ of them into $\langle \texttt{payloadCommit}_B \rangle_{\text{agg}}$—an explicit Data Availability Certificate for B—and publishes it in its next block header. The `acks` field is thus a vector of such certificates for p_i's pending blocks. Under the idealized timing assumptions of Table 1, republishing the aggregate increases consensus latency from 5δ in Starfish to 6δ in Starfish-L. Mysticeti-L keeps payloads in its blocks and carries no acknowledgment field.

Compressed Header Format. Both leader and non-leader block headers in Starfish-L share the format

$$B_{\text{header}} = (\langle r \rangle_{p_i}, \texttt{ancestors}, \texttt{roundCert}, \texttt{leaderCert}, \texttt{acks}, \texttt{payloadCommit})_{p_i},$$

where $\texttt{roundCert} = \langle r-1 \rangle_{\text{agg}}$ (for $r > 0$), $\texttt{leaderCert} = \{\langle \texttt{leader}_{r-2} \rangle_{\text{agg}}?\}$, and ? denotes optional inclusion; as in Section 5, the full block appends the payload, $B = (B_{\text{header}}, \texttt{payload})$. Only `ancestors` differs between leader and non-leader headers: at most two references $\{\texttt{self}, \langle \texttt{leader}_{r-1} \rangle_{p_i}?\}$, hence $O(\kappa)$, for non-leader headers, against at most n references, hence $O(\kappa n)$, for leader headers. Mysticeti-L blocks carry the same fields without `acks` and `payloadCommit`, embedding `payload` directly.

Commit and Sequence Rules. The rules of Section 3 carry over, with signed leader references as votes and round-$(r+2)$ headers carrying `leaderCert` for a round-r leader as its QCs. A direct commit still requires such QCs from $2f+1$ distinct parties; a direct skip still requires $2f+1$ round-$(r+1)$ headers that omit the partial signature; and the indirect rules still take the anchor at the smallest round $r' \geq r+3$, leave the slot undecided while that anchor is undecided, and otherwise test for a path from the anchor to a leader certificate for the slot. For Starfish-L, $\text{hist}_{\text{DA}}(B)$ is obtained by traversing $\text{hist}(B)$ and keeping the blocks whose explicit DACs appear in acknowledgment fields within that history; the payload ordering of Section 5 then applies unchanged. Mysticeti-L sequences blocks exactly as in Section 3.

Recovering Missing Payloads. Starfish-L keeps the same Reed–Solomon encoding and data commitments as Starfish (Section 5), but changes how fragments travel: because Lazy-Push delays unknown-history broadcasts, parties no longer relay other parties' fragments on every broadcast event. Each block author disperses its own block (header and payload) when creating B, so in the happy case every honest party receives B's payload directly and no reconstruction is needed. Before outputting the next payloads in the deterministic sequence derived from committed leaders, a party broadcasts a request—identified by the payload commitment $\texttt{payloadCommit}_B$—for every payload it is missing, and pauses at the first of them until that payload is reconstructed, so that all parties output payloads in the same order. Any party that has stored a verified fragment for B replies with it, together with its Merkle proof when Merkle proofs are used. Since B is sequenced only after an explicit DAC, at least $f+1$ honest parties hold valid fragments, so every honest party eventually collects the $f+1$ fragments that reconstructing $\texttt{payload}(B)$ requires, provided honest parties retain what they acknowledged (Appendix J).

Communication and Costs. In the happy case, Lazy-Push's delayed broadcasts carry no unknown history at all (Corollary 6), so the only metadata a round carries is the parties' own headers: $n-1$ non-leader headers of size $O(\kappa)$ and one leader header of size $O(\kappa n)$, each sent to $n-1$ peers, that is $O(\kappa n^2)$; in Starfish-L the partial signatures returned to authors add a further $O(\kappa n^2)$ per round. Header metadata remains $O(\kappa n^3)$ in the worst case, where parties may have to relay the headers of others. The compression trades communication for computation: each party verifies $O(n)$ partial signatures per round and aggregates them into multi-signatures, at a CPU cost measured empirically in Section 8.2. Theorem 11 states the complete bounds, including payload and fragment proofs.

7 Related Work

DAG consensus builds on a long line of BFT protocols [8, 11, 15, 18, 22, 24]. We focus on DAG designs and their liveness, latency, and communication costs.

Uncertified DAGs and Liveness. Hashgraph [3] pioneered the uncertified DAG approach: blocks are disseminated via Best-Effort Broadcast (BEB), so equivocations may appear in the DAG and data availability must be ensured by the consensus protocol itself; Hashgraph sequences the resulting blocks in a leaderless way, ordering them by their received timestamps. Cordial Miners [21] made the ordering leader-based instead: it partitions the DAG into disjoint three-round waves and assigns a single leader to the first round of each wave, so that committing leader blocks and agreeing on their sequence determines both how the DAG is sliced and how

the blocks within each slice are ordered, which significantly reduces commit latency. Mysticeti [2] pipelines this construction, assigning a leader to every round rather than every third one so that the three-round windows of successive leaders overlap, lowering latency further. Starfish adopts this pipelined commit rule (Section 3). Recent work [27] exhibits an explicit counterexample to the liveness of unmodified Mysticeti and presents a patched protocol with mechanized safety and liveness proofs in the Rocq proof assistant. The liveness proof, however, assumes two synchronization properties after GST (their Appendix B.1, third group). We derive synchronization from the Push pacemaker's rules and use it to prove liveness (Section 4 and lemma 6).

Certified DAGs. Aleph [17], DAG-Rider [20], Tusk [9], Bullshark [34], Shoal [33], Shoal++ [1], and Sailfish [31] disseminate blocks via Byzantine Reliable Broadcast (RBC): every block is certified by a quorum, which guarantees data availability and prevents equivocation but adds rounds and communication. Successive designs reduce commit latency via faster commit rules, interleaved or multiple DAG instances, and no-vote certificates (Appendix F). Sailfish++ [32] proposes an optimistic RBC terminating in two rounds in the happy case and integrates erasure Reed–Solomon coding, achieving the lowest latency among certified DAGs with linear amortized payload communication; it serves as our certified baseline in Section 8. Starfish instead establishes availability through acknowledgments on an uncertified DAG; see [28] for a survey comparing both DAG families.

Hybrid DAGs. BBCA-Chain [25] certifies only leader blocks via RBC and disseminates non-leader blocks via BEB; unlike our L protocols, its non-leader blocks carry only payloads and do not contribute votes in the consensus layer (Appendix F). Concurrent work Angelfish [36] lets parties that do not propose payload send votes via BEB over a certified DAG. This reduces metadata communication only when fewer parties propose payload.

Payload Availability and Coding. Certified DAGs outsource payload availability to broadcast primitives. Narwhal [9] separates dissemination from consensus behind quorum-based availability certificates for transaction batches; Raptr [35] integrates availability certification into consensus by letting parties vote on prefixes of uncertified batches, and Autobahn [16] lets votes acknowledge prefixes of per-party payload lanes. The latter two need only $f+1$ acknowledgments, since one honest party holding the full payload suffices. Starfish instead forms data availability certificates directly on the DAG: a block's payload is sequenced only after $2f+1$ acknowledgments are recorded in later blocks. Since payloads are disseminated as Reed–Solomon fragments and f Byzantine parties may falsely acknowledge, $2f+1$ acknowledgments guarantee $f+1$ honest fragment holders—exactly what reconstruction requires. Erasure and error-correcting codes are well established for reducing RBC communication [7, 10, 14, 20, 26, 32], yet certified protocols confine coding to the broadcast primitive; Starfish integrates Reed–Solomon coding into the DAG itself, encoding history dissemination rather than block certification, and reduces payload communication from the $O(Mn^2)$ of cordial dissemination [21] to $O(Mn)$ per round. See Appendix F for detailed comparisons.

8 Evaluation

8.1 Implementation Overview

Controlled experiments (Q1–Q3) use a Rust consensus testbed[4] implementing Starfish, Mysticeti, their BLS variants, and the certified-DAG baseline Sailfish++ [32]. Starfish-BLS and Mysticeti-BLS implement the compression of Starfish-L and Mysticeti-L using BLS multi-signatures. The testbed is forked from Mysticeti[5], with Sailfish++ consensus logic ported from the authors' implementation[6]. All protocols share networking, storage, serialization, transaction generation, and metrics, with matched hardware and workloads within each comparison. Following prior work [1, 2, 31, 34], the testbed excludes transaction execution and ledger storage.

Production telemetry (Q4) comes from a separate Rust codebase that integrates Mysticeti and Starfish into a public blockchain node. The deployment is on IOTA[7]. Additional implementation details, including payload persistence, recovery, and concurrency, appear in Appendix J.

Block dissemination: Pull vs Push. *Pull* broadcasts authors' blocks and fetches missing ancestors on demand. *Push* includes the history recipients may lack, avoiding a retrieval round trip when references arrive before ancestors (Section 4). For average total payload M per round, forwarding full blocks costs $O(Mn^2)$ across the committee. Starfish instead pushes headers and Reed–Solomon fragments: each validator sends its own payload and one fragment of every other block (Section 5). As an optimization, our Push implementation can use feedback on which authors' headers or fragments may be useful, together with knowledge of the recipient's data, to reduce redundant forwarding (Appendix J).

By default, the testbed uses Push for Starfish and Pull for the other four protocols; production uses the same choices for Starfish and Mysticeti. We evaluate complete consensus protocols, including their commit rules and dissemination strategies. The four subsections ask how latency and resource costs scale with committee size (Q1, Section 8.2), where throughput saturates (Q2, Section 8.3), how a Byzantine leader chain bomb under selective dissemination affects honest latency and leader relay cost, measured under both

[4] https://github.com/iotaledger/starfish.
[5] https://github.com/MystenLabs/mysticeti
[6] https://github.com/qyu100/SFSailfish, branch `OptSFSailfish`.
[7] https://github.com/iotaledger/iota/releases/tag/v1.21.1

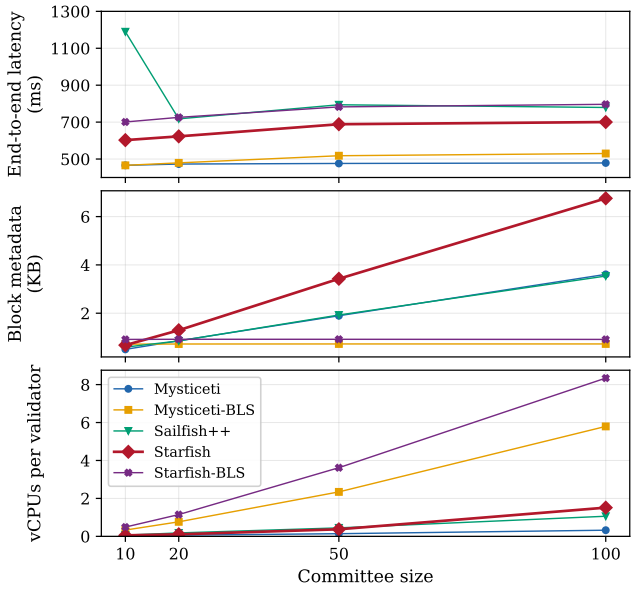


Figure 3. Committee scaling at 1,000 tx/s. From top to bottom: end-to-end transaction latency (p50), block metadata size, and per-validator CPU consumption.

Pull and Push in each of Mysticeti and Starfish (Q3, Section 8.4), and how the mainnet transition from Mysticeti to Starfish changes latency and communication across validators (Q4, Section 8.5).

8.2 Q1: Committee Scaling on Fixed Hardware

Testbed and Simulation Setup. Q1 and Q2 run on the shared testbed: validators on EC2 in a single AWS region under a synthetic wide-area latency overlay drawn from a measured 10-region RTT matrix, a leader timeout of 600 ms and one leader per round for every protocol, 512-byte transactions issued by a client on each validator, and at least 180 s of steady-state operation per run. Appendix I gives the instance type, the region assignment, the RTT matrix (Table S5), and the payload composition. End-to-end latency is the elapsed time from a transaction's arrival at the validator that includes it in a block until another validator sequences it; throughput is sequenced transactions divided by the steady-state duration of the run; we report medians across validators.

Starfish-BLS and Mysticeti-BLS run under Pull rather than the Lazy-Push dissemination their designs prescribe (Section 4). These runs are fault-free with timely leaders, so the leader timeout never expires and its value does not enter the measurements; without missing ancestors Pull issues no requests and Lazy-Push finds no unknown history to send, so every configuration exercises its design's happy-case behavior.

We sweep committee size $n \in \{10, 20, 50, 100\}$ at a fixed total load of 1,000 tx/s and compare Starfish, Starfish-BLS, Mysticeti, Mysticeti-BLS, and Sailfish++ (Figure 3).

Latency. Measured latency broadly follows the round structure of Table 1, with additional implementation and network overhead that the table does not model—signature verification, erasure encoding/decoding, hashing, Merkle proof verification, storage flushes, and scheduling. Mysticeti and Mysticeti-BLS carry payloads inside their blocks and add no round on the data path, staying flat at ≈ 470–530 ms, with an idealized end-to-end latency of 4.5δ (Table 1). Starfish adds one round, since a payload is sequenced only once its availability is attested (≈ 600–700 ms, 5.5δ). Starfish-BLS adds a second, because reversing the acknowledgments makes the author republish the aggregate before anyone can act on it (Section 6), and pays partial-signature verification and aggregation on top (≈ 700–800 ms, 6.5δ). Sailfish++ certifies each block by RBC before it enters the DAG (6δ, above 700 ms). At n=10, Sailfish++ reaches $\approx 1{,}190$ ms because its $\lceil 5n/6 \rceil$ echo quorum leaves only one slack peer in the synthetic WAN (Appendix I). At n=20, 50, and 100, latency falls to ≈ 720–800 ms.

Block header size. Header size is where a protocol keeps its quorum evidence. Aggregate that evidence into a certificate and the header is committee-invariant: Mysticeti-BLS is flat at ≈ 730 B per block, and Starfish-BLS sits at a slightly larger constant for its extra data-availability certificate. Keep the evidence as a list of per-authority references and the header is $O(n)$: a Mysticeti block references between $2f+1$ and n ancestors, so at n=100 its ancestor list alone runs to ≈ 2.5–3.8 KB, three to five times the BLS constant. Starfish carries data-availability acknowledgments on top of that list (Section 6), and its headers grow at roughly twice Mysticeti's rate. Much of that acknowledgment list is redundant—a party acknowledges blocks it also references—and both implementations encode it against the ancestor list rather than repeating it (Appendix J).

vCPUs per validator. The CPU panel is the header panel read from the other side: the $O(n)$ that aggregation removes from the wire reappears as $O(n)$ signature operations per round. Mysticeti-BLS and Starfish-BLS consume ≈ 5.8 and ≈ 8.3 vCPUs per validator at n=100, dominated by partial-signature aggregation and verification, against below ≈ 1.5 for Mysticeti, Starfish, and Sailfish++ across the sweep.

8.3 Q2: Throughput–Latency Profile at $n = 100$

Holding the committee fixed at $n = 100$ on the same testbed, we sweep the offered load through saturation and report end-to-end transaction latency (p50), per-validator vCPU usage, and bandwidth efficiency—bytes sent per byte of sequenced payload—for the same five protocols. Loads are sampled densely near each protocol's saturation point.

Results. For offered loads from 1,000 to 180,000 tx/s, all protocols retain a roughly flat latency profile over more than two orders of magnitude of delivered throughput (Figure 1). Mysticeti and Mysticeti-BLS remain the lowest-latency protocols

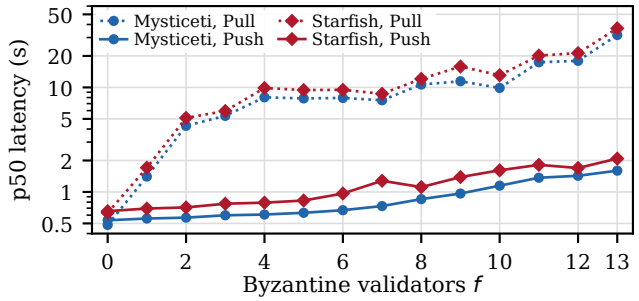


Figure 4. Leader chain bomb at n=40 as the number of attackers f grows to the tolerance bound 40=3 · 13+1. Honest-transaction p50 latency.

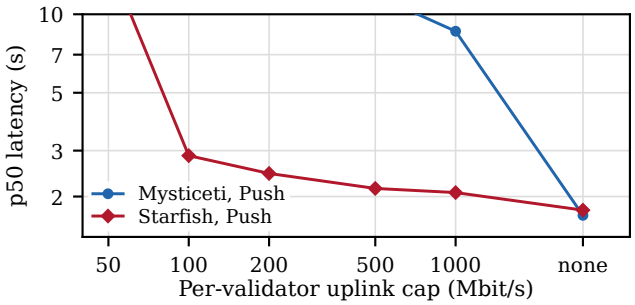


Figure 5. Honest p50 latency versus emulated per-validator uplink cap at n=40 with f=13 attackers. Both protocols run Push; they differ in Starfish's dispersed dissemination and commit rule.

(≈ 480–580 ms), Starfish holds at ≈ 700–740 ms, and Starfish-BLS and Sailfish++ sit in the ≈ 780–910 ms range, matching the ordering predicted by Table 1. Peak sustained throughput separates the designs: Mysticeti-BLS, Starfish-BLS, and Starfish reach ≈ 262 k, ≈ 261 k, and ≈ 257 k tx/s, respectively, while Sailfish++ reaches ≈ 239 k tx/s and Mysticeti saturates earliest at ≈ 190 k tx/s. CPU follows Q1: BLS compression saves metadata at higher CPU cost, whereas Starfish reaches the same throughput ceiling at lower CPU. Bandwidth efficiency separates the protocols only under small load, where metadata dominates a thin payload: the BLS variants are the most efficient at ≈ 2.7–3.0 and Starfish the least at ≈ 27 bytes sent per sequenced byte, and all five converge towards 1 as payload comes to dominate.

8.4 Q3: Leader Chain-Bomb Attack and Relay Cost

Q3 evaluates proactive history relay under *selective dissemination*: its effect on consensus progress and its outbound-bandwidth cost at a sustained payload load. In our *leader chain bomb*, a non-forking variant of Aleph's fork bomb [17], a Byzantine validator withholds every block it creates and, once every n rounds in its own leader round, releases the whole chain, payload included, to the next leader alone. The honest leader references them, so every other honest validator must recover up to n unknown ancestors per attacker before it can vote for or commit the leader block: one request round trip per unknown level under Pull, none under Push.

Setup. We compare Mysticeti and Starfish under both Pull and Push, holding the committee at $n = 3 \cdot 13 + 1 = 40$ and sweeping the number of attackers f from 0 to 13. Push relays full blocks in Mysticeti and headers and fragments in Starfish. Runs last 600 s, three runs per cell with median results reported. The offered load is 10,000 tx/s total, split evenly across validators. These parameters describe the fault sweep of Figure 4; the capped-uplink experiment below keeps the same committee, fault count and offered load. We use Q1's network settings, with all validators running on one host with enough headroom in CPU and memory (Appendix I). Because of its commit rule, Starfish never sequences the attackers' payload, which never reaches f+1 honest holders, whereas Mysticeti eventually does; we exclude it from all latencies and normalize bandwidth by honest sequenced bytes, so every configuration is measured on the same population.

Latency under the leader chain bomb. Without attackers the four configurations commit at 0.48–0.66 s p50 (Figure 4). Attackers separate them immediately, and the gap widens with f: at the tolerance bound Pull reaches 32 s in Mysticeti and 37 s in Starfish, with p99 up to 54 s. Push keeps p50 under about two seconds across the whole sweep, ending at 1.6 and 2.1 s—a factor of 18–20 at f=13—with p99 within 6.5 s. The separation is the recovery path: under Pull every honest validator serializes one request round trip per unknown ancestor level, whereas Push delivers the missing history together with the leader block.

Relay cost. Starfish's coded relay pushes stay within the Pull bandwidth band, while Mysticeti's full-block relay uses 3.5–5.5× as much traffic. We next test how this difference affects latency under capped uplinks.

Leader relay under a capped uplink. Proactive relay only helps if the leader can send it. We cap each validator's outbound link and repeat the attack at n=40 with f=13 (Figure 5), comparing the two Push configurations; Pull is omitted because it already sits at tens of seconds with an uncapped link. Starfish holds 1.8–2.9 s down to 100 Mbit/s and first exceeds ten seconds at 50 Mbit/s, whereas Mysticeti, which relays the same history as full blocks, already shows a sharp rise in p50 latency at 1000 Mbit/s.

8.5 Q4: Production Mainnet Transition

We analyze production telemetry from a public blockchain mainnet with 69 validators during its switch from Mysticeti to Starfish on 2026-04-23, using 07:45 UTC as the comparison reference time. The switch took effect at an epoch boundary through the node's consensus-selection parameter. We compare each validator with itself over three days before and three days after the switch, omitting the ten minutes closest to the switch on each side. Mainnet Mysticeti used

Table 3. Production switch: median over validators of the three-day medians before (Mysticeti) and after (Starfish), and median per-validator change. Green: improvement; orange: cost. Suspensions are per 100 accepted blocks or headers.

Metric	Mysticeti	Starfish	Change (%)
Consensus requests (req/s)	73.8	10.6	−85.5
Suspensions / 100 accepted	21.33	2.06	−89.2
Block creation rate (blocks/s)	17.8	19.1	+7.3
Consensus latency (ms)	222	258	+16.2
Consensus latency p99 (ms)	544	398	−26.5
Bandwidth efficiency	39.8	94.1	+132.7

Pull dissemination (Section 8.1). The study is observational: operator behavior and offered load are uncontrolled. We report the median per-validator percentage change, computed from each validator's three-day medians (Table 3). Appendix H provides the methods and full results. The operator documentation lists the validator hardware profile as 128 GB RAM, a 24-core processor or 48 vCPUs, 4 TB NVMe SSD storage, and a 1 Gbps network uplink;[8] the geographic latency structure of the validator set is characterized in Appendix G.

Consensus requests and suspended blocks. Total outbound consensus requests fall for all paired validators, with the median rate dropping from 73.8 to 10.6 requests/s (−85.5%). This counter includes ancestor fetches and other consensus requests. Ancestor-related suspensions measure dependency stalls: a block or header remains buffered and cannot be integrated into the DAG until its ancestors are locally available. Their rate falls from 21.33 to 2.06 per 100 accepted remote blocks under Mysticeti or headers under Starfish (−89.2%).

DAG progress and block creation. Round advancement and block creation rate increase together, by +6.0% and +7.3%, reaching 19.16 rounds/s and 19.1 own blocks/s, respectively. After the switch to Starfish Push, block creation becomes steadier over time (Figure 6). The gap between the 10th percentile and median block creation rates across validators shrinks by 76%, from 1.24 to 0.30 blocks/s (median over time).

Block creation rate improves for all paired validators. The five lowest-rate proposers, selected using their pre-switch rates, improve from 15.4 to 17.6 own blocks/s (+17.5%). They are located in Hong Kong, Taiwan, Japan, India, and Argentina, outside the network's concentration in Europe and North America.

Consensus latency. Consensus latency measures the time from block creation to payload sequencing. Among the well-connected validators, consensus latency rises from 221 to 258 ms (+16.3%), below the idealized 25% increase from 4δ to 5δ associated with the additional availability round (Table 1). The 5 less-connected validators see a median increase of only 5.8%. Despite this increase in typical latency, the median histogram p99 across all validators reporting latency in both windows falls from 544 to 398 ms (−26.5%). Excluding the last Mysticeti epoch and the first Starfish epoch, hourly consensus latency is also steadier under Starfish: the p50 range narrows from 192–245 to 238–244 ms, p95 from 298–417 to 343–358 ms, and p99 from 476–619 to 393–455 ms. No restarts are recorded in the retained Mysticeti period.

[8] https://docs.iota.org/operator/validator-node/overview

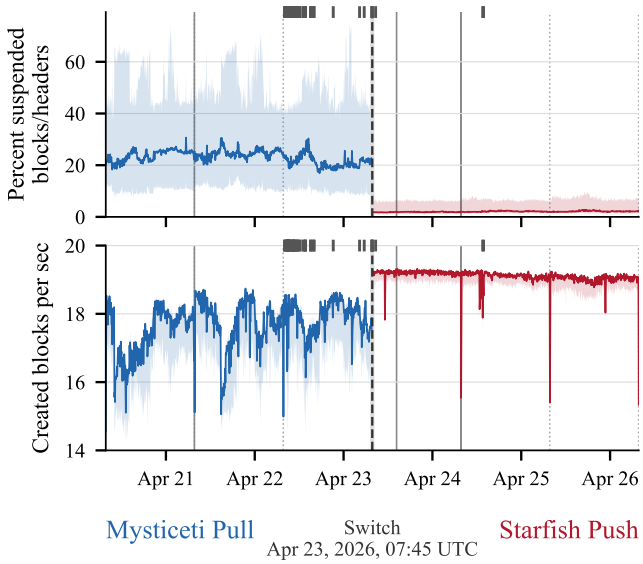


Figure 6. Mainnet transition over ±3 days. Lines show medians across validators; bands span the 10th–90th percentiles. Suspensions are per 100 accepted remote blocks/headers. Faint lines mark epoch changes; ticks mark restarts.

Bandwidth efficiency. Median outbound bandwidth per validator rises from 3.5 to 9.5 MB/s, about 7.6% of the specified 1 Gbps uplink, while consensus output grows from 88.4 to 100.9 kB/s. Bandwidth efficiency worsens from 39.8 to 94.1 bytes sent per sequenced byte (+132.7%).

9 Conclusion

We introduce Push, a new pacemaker that closes the liveness gap in uncertified DAG BFT under partial synchrony. Its conditions jointly prevent skipped block creation, enable catch-up, and disseminate the history needed for synchronization. Starfish makes Push practical with linear payload communication, trading higher ideal-case latency for more robust progress under adverse conditions. Mainnet measurements also show lower variability in block creation and consensus latency. Mysticeti with Pull may be preferable in stable, well-connected networks where low communication overhead takes priority.

We also show how to reduce happy-case metadata communication in uncertified DAG BFTs to quadratic in committee size. In particular, Mysticeti-BLS matches the lowest ideal-case latency among the DAG designs we compare while

substantially reducing communication, at the cost of higher CPU usage.

References

[1] Balaji Arun, Zekun Li, Florian Suri-Payer, Sourav Das, and Alexander Spiegelman. 2025. Shoal++: High Throughput DAG BFT Can Be Fast!. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[2] Kushal Babel, Andrey Chursin, George Danezis, Lefteris Kokoris-Kogias, and Alberto Sonnino. 2025. Mysticeti: Reaching the Limits of Latency with Uncertified DAGs. In *Proc. of the 2025 the Network and Distributed System Security Symposium (NDSS)*.

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

[4] Alexandra Boldyreva. 2003. Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In *International Workshop on Public Key Cryptography*. Springer, 31–46.

[5] Dan Boneh, Ben Lynn, and Hovav Shacham. 2001. Short signatures from the Weil pairing. In *International conference on the theory and application of cryptology and information security*. Springer, 514–532.

[6] Gabriel Bracha and Sam Toueg. 1985. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)* 32, 4 (1985), 824–840.

[7] Christian Cachin and Stefano Tessaro. 2005. Asynchronous verifiable information dispersal. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*. IEEE, 191–201.

[8] Miguel Castro, Barbara Liskov, et al. 1999. Practical byzantine fault tolerance. In *OsDI*, Vol. 99. 173–186.

[9] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In *Proceedings of the 17th European Conference on Computer Systems (EuroSys)*. 34–50.

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

[11] Isaac Doidge, Raghavendra Ramesh, Nibesh Shrestha, and Joshua Tobkin. 2024. Moonshot: Optimizing Block Period and Commit Latency in Chain-Based Rotating Leader BFT. In *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 470–482. https://doi.org/10.1109/DSN58291.2024.00052

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

[13] Shuhong Gao. 2003. A New Algorithm for Decoding Reed-Solomon Codes. In *Communications, Information and Network Security*. The Springer International Series in Engineering and Computer Science, Vol. 712. Springer, 55–68.

[14] Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. 2022. Dumbo-NG: Fast Asynchronous BFT Consensus with Throughput-Oblivious Latency *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 1187–1201.

[15] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. 2022. Jolteon and Ditto: Network-Adaptive Efficient Consensus with Asynchronous Fallback. In *Financial Cryptography and Data Security: 26th International Conference, FC 2022, Grenada, May 2–6, 2022, Revised Selected Papers* (Grenada, Grenada). Springer-Verlag, Berlin, Heidelberg, 296–315.

[16] Neil Giridharan, Florian Suri-Payer, Ittai Abraham, Lorenzo Alvisi, and Natacha Crooks. 2024. Autobahn: Seamless high speed BFT. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 1–23.

[17] Adam Gągol, Damian Leśniak, Damian Straszak, and Michał Świętek. 2019. Aleph: Efficient atomic broadcast in asynchronous networks with byzantine nodes. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies (AFT)*. 214–228.

[18] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. SBFT: A Scalable and Decentralized Trust Infrastructure. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 568–580. https://doi.org/10.1109/DSN.2019.00063

[19] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. 2010. Constant-Size Commitments to Polynomials and Their Applications. In *Advances in Cryptology – ASIACRYPT 2010*. Springer, 177–194.

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

[21] Idit Keidar, Oded Naor, and Ehud Shapiro. 2023. Cordial Miners: A Family of Simple, Efficient and Self-Contained Consensus Protocols for Every Eventuality. In *Proc. of the 37th International Symposium on Distributed Computing (DISC)*.

[22] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2007. Zyzzyva: speculative byzantine fault tolerance *(SOSP '07)*. Association for Computing Machinery, New York, NY, USA, 45–58. https://doi.org/10.1145/1294261.1294267

[23] Sian-Jheng Lin, Tareq Y Al-Naffouri, Yunghsiang S Han, and Wei-Ho Chung. 2016. Novel polynomial basis with fast Fourier transform and its application to Reed–Solomon erasure codes. *IEEE Transactions on Information Theory* 62, 11 (2016), 6284–6299.

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

[25] Dahlia Malkhi, Chrysoula Stathakopoulou, and Maofan Yin. 2024. BBCA-CHAIN: One-Message, Low Latency BFT Consensus on a DAG. In *Proc. of the 2024 Financial Cryptography and Data Security (FC)*.

[26] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The Honey Badger of BFT Protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 31–42. https://doi.org/10.1145/2976749.2978399

[27] Longfei Qiu, Jingqi Xiao, and Zhong Shao. 2026. Mechanized Safety and Liveness Proofs for the Mysticeti Consensus Protocol under the LiDO-DAG Framework. In *Proceedings of the 2026 IEEE Symposium on Security and Privacy (SP)*.

[28] Mayank Raikwar, Nikita Polyanskii, and Sebastian Müller. 2024. SoK: DAG-based Consensus Protocols. In *2024 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 1–18.

[29] Irving S Reed and Gustave Solomon. 1960. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics* 8, 2 (1960), 300–304.

[30] Ehud Shapiro. 2022. Grassroots cryptocurrencies: A foundation for a grassroots digital economy. *arXiv preprint arXiv* 2202 (2022).

[31] Nibesh Shrestha, Aniket Kate, and Kartik Nayak. 2025. Sailfish: Towards Improving Latency of DAG-based BFT. In *Proceedings of the 2025 IEEE Symposium on Security and Privacy (SP)*.

[32] Nibesh Shrestha, Qianyu Yu, Aniket Kate, Giuliano Losa, Kartik Nayak, and Xuechao Wang. 2025. Optimistic, Signature-Free Reliable Broadcast and Its Applications. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. https://doi.org/10.1145/3719027.3765220

[33] Alexander Spiegelman, Balaji Aurn, Rati Gelashvili, and Zekun Li. 2024. Shoal: Improving DAG-BFT Latency And Robustness. In *Proc. of the 2024 Financial Cryptography and Data Security (FC)*.

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

[35] Andrei Tonkikh, Balaji Arun, Zhuolun Xiang, Zekun Li, and Alexander Spiegelman. 2025. Raptr: Prefix Consensus for Robust High-Performance BFT. *arXiv preprint arXiv:2504.18649* (2025).

[36] Qianyu Yu, Giuliano Losa, Nibesh Shrestha, and Xuechao Wang. 2025. Angelfish: Consensus with Optimal Throughput and Latency Across the Leader-DAG Spectrum. *arXiv preprint arXiv:2509.15847* (2025).

A Detailed Description of Starfish Protocol

In this section, we provide a detailed description of Starfish. Pseudocode for the core algorithms for block creation, round advancement, commit rules, and sequencing are in Appendix B.

We describe the protocol for $n = 3f + 1$ parties, where f is the maximum number of Byzantine parties; the remaining $\geq 2f + 1$ parties are honest.

A.1 DAG Construction and Push Pacemaker

Block DAG and rounds. Parties concurrently create blocks in rounds starting with round 1. Each block consists of a block header and data. The n blocks of round 0 are predefined and known to all parties. Each party creates one block per round, references some blocks from previous rounds, and broadcasts it to all parties. All block headers form a directed acyclic graph (DAG), where each block header is a unique vertex and a directed edge from header B to header B' indicates that B references B'. Since there is a one-to-one correspondence between blocks and headers, we use "block" instead of "block header" when this simplification causes no confusion. For example, we say "block B references block B'" when technically referring to their headers.

Every party creates its local DAG based on its own blocks and block headers received from other parties. A received block header is integrated into the local DAG only when all its referenced block headers are already integrated into the local DAG.

Block format. Each block $B = (B_{\text{header}}, \texttt{payload})$ consists of a block header B_{header} and `payload`. Each block header of party p_i has the following format

$$B_{\text{header}} = (r, \texttt{ancestors}, \texttt{acks}, \texttt{payloadCommit})_{p_i}.$$

Here r is the round number. The ordered list `ancestors` contains at most n hash references to the latest blocks from distinct parties created at or before round $r - 1$, with at least $2f + 1$ references from round $r - 1$. Party p_i's own previous block is always referenced. Parties whose latest available blocks have already been referenced by p_i in previous blocks are omitted from this list. The set `acks` contains hash references to block headers whose payload p_i has locally verified against the commitment in that header. Only references not included in the `acks` fields of previous blocks of p_i are included. The field `payloadCommit` is a commitment to the payload. The payload commitment is computed as described in Appendix A.6.

Leader block. Starfish uses a round-robin leader scheduler: in round r, party $p_{(r \bmod n)+1}$ is the designated leader. A block created by that party in round r is called a leader block. Leader blocks have the same structure as other blocks but play a special role in block creation and commit rules. A *leader slot* is a pair (p_i, r), where p_i is the designated leader of round r.

Pacemaker conditions. The pacemaker consists of three parts: round advancement, block creation, and broadcasting. These are implemented in ExecuteRound of Algorithm S1.

Round advancement. A party advances from round $r - 1$ to round r when two conditions are met: its local DAG contains $2f + 1$ round-$(r - 1)$ blocks from distinct parties (**A1**), and it has created its own block in round $r - 1$ (**A2**). The latter prevents parties from "jumping" ahead without contributing blocks that serve as votes and certificates needed for leader commitment.

Block creation. We say that a block B from round $r + 1$ *votes* for a leader block L from round r if L is the first block for leader slot of round r in the list of `ancestors` in block B (Byzantine parties can reference multiple blocks for one slot). After advancing to round r, a party creates its block when at least one of three conditions is satisfied. The primary condition (**C1**) requires receiving the leader block from round $r-1$ (**L1**) and receiving at least $2f+1$ round-$(r-1)$ blocks from distinct parties that vote for the leader of round $r-2$ or $2f+1$ round-$(r-1)$ blocks from distinct parties that do not vote for the leader block of round $r - 2$ (**L2**). When creating a block via **C1**, the party should choose which round-$(r - 1)$ blocks to include in `ancestors` so that they witness the satisfied **L2** disjunct; under equivocation, this requires selecting the appropriate instance from each Byzantine party. The second condition **C2** allows a party to create a block after the timeout of δ_{LT} expires following its advancement to round r. The third condition **C3** allows a party to create its block upon receiving $2f+1$ round-r blocks from distinct parties, enabling faster synchronization when lagging behind.

Broadcasting history. When party p_i creates a new block (**B1**) or advances to the next round (**B2**) it broadcasts the unknown history. Concretely, p_i snapshots its local DAG and, after an optional delay of δ_{BT}, sends to each party j the information from this snapshot that is not in $\text{known}_i(j)$—full blocks in Mysticeti/Mysticeti-L and headers in Starfish/Starfish-L, i.e., what p_i still considers potentially unknown to p_j. When the trigger is **B1**, the creator's own block is sent immediately, and only the remaining unknown history is delayed in Lazy-Push. The set $\text{known}_i(j)$ is formally defined below in (S1).

History of Known Block Headers. We denote $\text{hist}(B)$ as the causal *history* of a block header B, representing the set of all block headers that are transitively reachable from B via its hash references (in `ancestors`). Every party may have a different perception of the current DAG. Let us denote by DAG_i the local DAG of party p_i. Due to network latency, a party does not necessarily know the local DAGs of the other parties. However, every block header expresses knowledge about the existence of other block headers in its causal history. Therefore, we can define the history of block headers known to a party p_j from the point of view of party p_i as follows:

Table S1. Pacemaker Push for Starfish.

Advance round $(r-1) \rightarrow r$	**Create block of round** r	**Broadcast history**
A1: local DAG contains $2f+1$ block headers of round $r-1$ from distinct parties AND **A2:** created block of round $r-1$	**C1:** wait for leader and votes: **L1:** local DAG contains leader block header of round $r-1$ AND **L2:** local DAG contains $2f+1$ block headers from different parties voting for the leader of round $r-2$ OR contains $2f+1$ block headers from different parties that do not vote for the leader of round $r-2$ OR **C2:** expired timeout $\delta_{LT} = 2\Delta$ OR **C3:** received $2f+1$ block headers of round r from different parties	**B1:** created block OR **B2:** advanced round

$$\mathsf{known}_i(j) = H_{i,j} \cup S_{i,j}, \tag{S1}$$

where $H_{i,j} = \bigcup \mathsf{hist}(B)$ is the union over all block headers B in DAG_i created by party p_j and $S_{i,j}$ is all block headers that have already been sent (or pushed) from p_i to p_j. Unknown history is defined as all block headers known to p_i except $\mathsf{known}_i(j)$.

A.2 Identifying DAG Patterns

Slot and leader slot. A *slot* is a pair (p_i, r) of a party and a round. A slot is a *leader slot* if p_i is the designated leader of round r. Each leader slot is initially in the state UNDECIDED; by interpreting the DAG structure, each party attempts to make a decision TO-COMMIT or TO-SKIP for every leader slot.

The TO-COMMIT state allows committing the leader block in a leader slot. The crucial state is UNDECIDED, which forces all subsequent leader slots to wait, mitigating non-deterministic commitments due to network asynchrony without needing a buffer round, unlike prior work [9, 14, 20, 34]. Finally, the TO-SKIP state allows excluding leader slots that do not receive sufficient votes or certificates.

Leader, Voting, and Certifying Rounds. If a leader slot at round r is fixed, we call round r the *leader* round, round $r+1$ the *voting* round, and round $r+2$ the *certifying* round. These rounds are used to make a decision for the leader slot.

Voting for a Leader Block. Each block header created by party p_i at round $r+1$ contains an ordered list `ancestors` with hash references to at least $2f+1$ block headers from distinct parties at round r. Blocks from voting round $r+1$ can vote on a leader from leader round r. A block B from round $r+1$ *votes* for a leader block L from round r if L is the first leader block from round r in the `ancestors` of B.

Patterns for Leader Blocks. Like its predecessors, Starfish operates by interpreting the structure of the DAG and finding patterns that create certificates or skip the attempt to find certificates for leader blocks:

1. The *skip pattern*: blocks from at least $2f+1$ parties at round $r+1$ *do not* vote for a leader block from round r. Note that there might be no proposal for a leader slot or more than one proposal for a leader slot in the case of equivocations. The slot is skipped if, for every block proposal, we observe $2f+1$ blocks from distinct parties in the next round that do not vote for it.
2. The *certificate pattern*: blocks from at least $2f+1$ parties at round $r+1$ *vote* for a leader block B of round r. We then say that B is *certified*. Any block from certifying round $r+2$ that contains in its history such a pattern is called a *quorum certificate* (QC) for the block B.

Quorum Certificates for Leader Blocks. Using these patterns, we derive quorum certificates implicitly for leader blocks by interpreting the DAG structure. Certification ensures that at most one leader block from a given leader slot can be certified. This prevents conflicts caused by equivocation, as conflicting blocks cannot simultaneously meet the $2f+1$ quorum requirement.

Pattern for Data Availability. In Starfish, we decouple block headers from their payload. Unlike Cordial Miners and Mysticeti, we cannot immediately sequence all payloads by *slicing* the DAG using committed leader blocks. Instead, we must check the data availability of payload within slices by inspecting the `acks` fields of blocks.

Similar to the *certificate pattern* above, we introduce

3. The *acknowledgment pattern*: blocks from at least $2f+1$ parties at rounds $> r$ *acknowledge the availability of the payload of a block* B from round r.

We say that B is *certified for data availability*. In contrast to the certificate pattern, we do not require the acknowledgment pattern to appear in one specific round $r+1$ but allow it to appear at any later point after round r.

Data Availability Certificates for Payloads. The *Data Availability Certificate (DAC)* ensures that the payload $\mathsf{payload}(B)$ associated with a block header B is available to a quorum of parties. Any subsequent block that contains

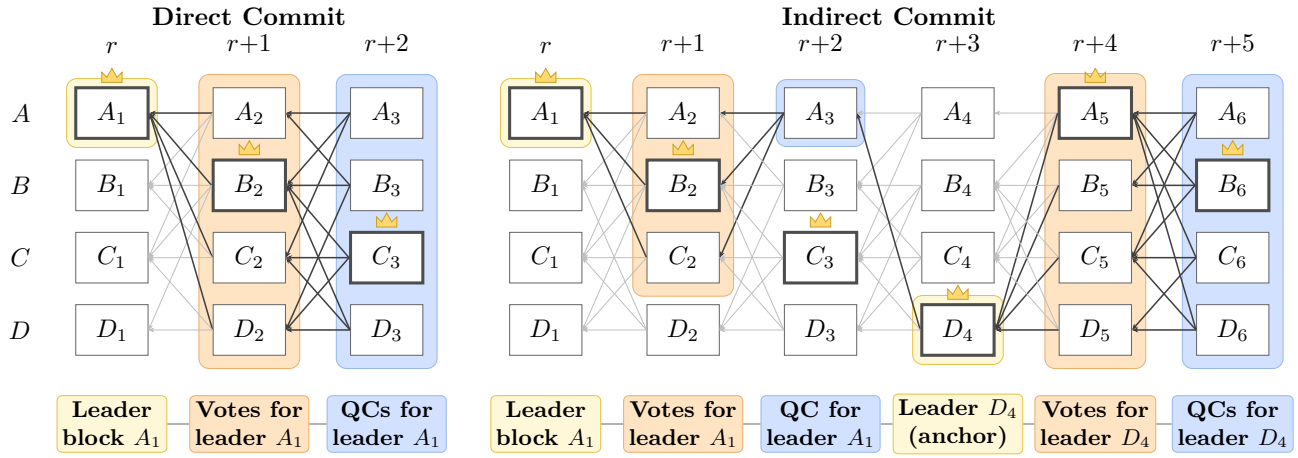


Figure S1. Direct (left) and indirect (right) commit of leader block A_1. Black arrows trace the certification chain: votes from round r+1 to the leader and cert-relaying references from QCs at round r+2 to their supporting voters. Gray arrows are the remaining ancestor references. In the indirect case only A_3 is a QC for A_1; the anchor D_4 at round r+3 reaches A_3 in its history, and D_4 is itself committed via votes at round r+4 and QCs at round r+5, so A_1 is committed indirectly.

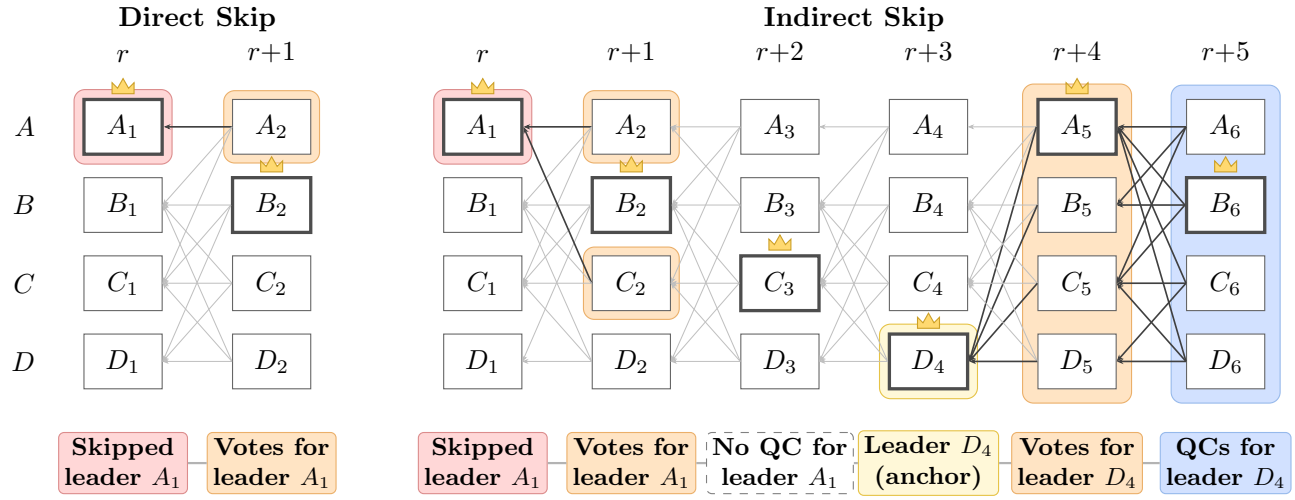


Figure S2. Direct (left) and indirect (right) skip of leader slot $(A, 1)$. Black arrows trace the certification chain (votes plus cert-relaying references); gray arrows are the remaining ancestor references. In the direct case, $2f$+1 blocks at round r+1 do not vote for A_1 (skip pattern). In the indirect case, only two parties vote for A_1 at round r+1, so no block at round r+2 is a QC for A_1; the anchor D_4 at round r+3 has no path to any QC for A_1 in its history, hence A_1 is skipped.

in its causal history an acknowledgment pattern is called a *Data Availability Certificate (DAC)* for the block B.

A.3 Sequence of Committed Leader Block Headers

This section describes the mechanism to find a sequence of committed leader block headers; it is the same as in Mysticeti.

All leader slots are initially in the UNDECIDED state. The goal is to mark all leader slots as either TO-COMMIT or TO-SKIP by detecting certificate or skip patterns or relying on an *indirect decision rule*. Table S2 summarizes the decision rules, which we describe in detail below.

The whole decision rule operates in three steps:

Step 1: Direct Decision Rule. The party applies the following *direct decision rule* to attempt to determine the status of the leader slot.

The party marks a leader slot as TO-COMMIT if it observes QCs from at least $2f + 1$ parties for that slot. In case of equivocation and multiple proposals for one slot, the proposal or

Table S2. Decision rules for a leader slot at round r

Decision	Rounds	Condition
Direct Commit	3	QCs from $2f+1$ distinct parties at round $r+2$
Indirect Commit	≥ 6	Exists a path to a QC at round $r+2$ from the next committed leader in round at least $r+3$
Direct Skip	2	Exists a skip pattern at round $r+1$
Indirect Skip	≥ 6	No path to a QC at round $r+2$ from the next committed leader in round at least $r+3$

block header that received the required quorum of QCs is committed.

The direct decision rule marks a leader slot as TO-SKIP if it observes a *skip pattern* for that slot.

If the direct decision rule fails to mark a slot as either TO-COMMIT or TO-SKIP, the slot remains UNDECIDED and the party resorts to the *indirect decision rule* presented in step 2 below.

Step 2: Indirect Decision Rule. If the direct decision rule leaves a slot undecided, the party applies the indirect decision rule. This rule operates in two stages, initially searching for an *anchor*—the first leader slot with round number ($r' \geq r+3$) marked as either UNDECIDED or TO-COMMIT.

If the anchor is marked as UNDECIDED, the party marks the slot as UNDECIDED. Conversely, if the anchor is marked as TO-COMMIT, the party marks the slot either as TO-COMMIT if the anchor contains a path to a QC for a leader block from the leader slot or as TO-SKIP otherwise.

Step 3: Commit Sequence. After processing all leader slots, the party derives an ordered sequence. Then the party processes this sequence, committing all TO-COMMIT slots and skipping all TO-SKIP slots until reaching the first UNDECIDED slot.

We will prove, in Lemma 12, that, eventually, every leader slot will be decided.

A.4 Linearization of DAG

With a sequence of committed leader block headers established, we now describe how to linearize all headers in the DAG. Let $\mathsf{hist}(B)$ denote the *causal history* of block header B—the set of all block headers transitively reachable from B via ancestor references. Since the block headers are signed, one can derive a fixed topological ordering sort on block headers in the DAG. To exclude equivocations, we keep at most one block header per pair $(B.author, B.round)$: if multiple headers exist for the same party and round, we keep only the first in the sequence below and ignore the rest. Given a sequence of committed leader block headers $C_1, \ldots, C_N$, one constructs a total ordering over all block headers in $\mathsf{hist}(C_N)$ as follows

$$\begin{aligned} &\mathsf{sort}(\mathsf{hist}(C_1)),\\ &\mathsf{sort}(\mathsf{hist}(C_2) \setminus \mathsf{hist}(C_1)), \ldots,\\ &\mathsf{sort}\Big(\mathsf{hist}(C_N) \setminus \bigcup_{i=1}^{N-1} \mathsf{hist}(C_i)\Big). \end{aligned} \tag{S2}$$

A.5 Sequencing payload

We now specify how actual payloads are ordered. For each block B, let $\mathsf{hist}_{\mathsf{DA}}(B)$ denote the set of all blocks D in $\mathsf{hist}(B)$ such that B is a Data Availability Certificate for $\mathsf{payload}(D)$. Given the sequence of committed leader blocks $C_1, \ldots, C_N$ from Appendix A.3 and the topological ordering of block headers in the DAG, we construct a total ordering of sequenced payload in $\mathsf{hist}_{\mathsf{DA}}(C_N)$ as follows

$$\begin{aligned} &\mathsf{sort}(\mathsf{hist}_{\mathsf{DA}}(C_1)), \mathsf{sort}(\mathsf{hist}_{\mathsf{DA}}(C_2) \setminus \mathsf{hist}_{\mathsf{DA}}(C_1)), \ldots,\\ &\qquad \mathsf{sort}\left(\mathsf{hist}_{\mathsf{DA}}(C_N) \setminus \bigcup_{i=1}^{N-1} \mathsf{hist}_{\mathsf{DA}}(C_i)\right). \end{aligned} \tag{S3}$$

This gives a total order of the blocks with availability guarantees. We then remove potential equivocations using the same rule as above: for each party and round, keep only the first block in this sequence. This guarantees that each party contributes at most one block per round to the final ordering, establishing a total ordering of the payloads contained in the remaining blocks.

A.6 Encoded Cordial Dissemination

At the same time as a party broadcasts a history of unknown headers, it also broadcasts some of the encoded payload fragments. In the following, we describe in detail how it works.

Reed-Solomon Codes. An (n, k) Reed-Solomon code over a Galois field $\mathbb{F}_q$ with $n \leq q$ encodes k data symbols into a codeword of n symbols. We use systematic encoding, meaning the first k encoded symbols coincide with the input data. Let $\mathsf{encRS}(m, n, k)$ denote the encoding algorithm that takes a message m of k symbols and outputs n encoded symbols.

Erasure decoding. When the positions of missing or invalid symbols are known (erasures), any k valid symbols suffice to reconstruct the original message. The decoding complexity is $O(n \log n)$ using fast algorithms based on the Fast Fourier Transform [23].

In Starfish, we use a systematic Reed-Solomon code of length $n = 3f+1$ and dimension $k = f+1$ over alphabet $\mathbb{F}_q$ where q is a power of two. For erasure decoding, any $k = f+1$ valid fragments suffice to restore all fragments, and therefore, the original payload.

Encoding and Merkle Tree Construction. We make use of Merkle trees to ensure integrity verification for encoded payload in case of erasure decoding.

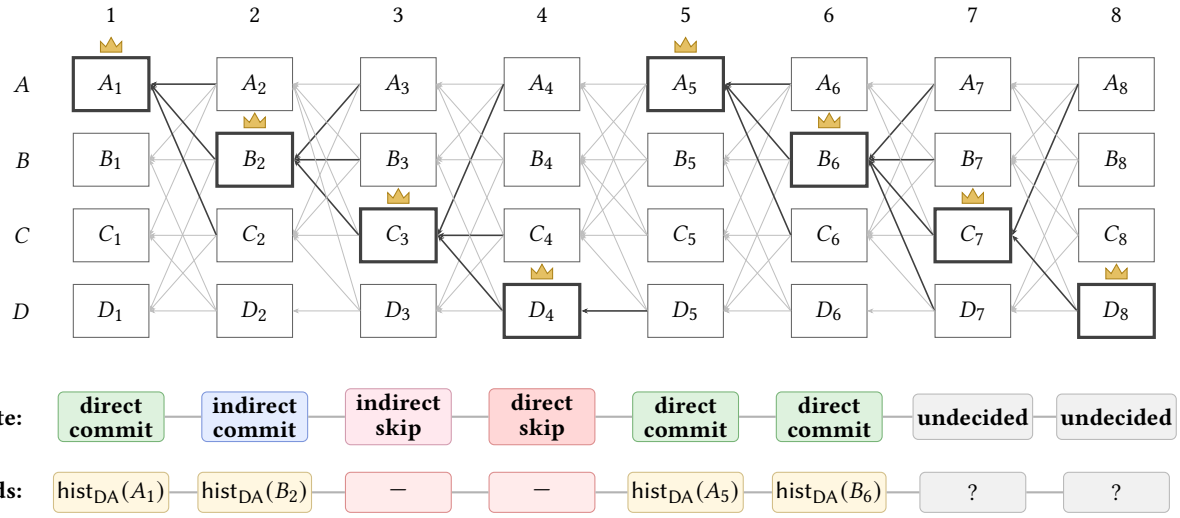


Figure S3. Example DAG with $n = 4$ parties (A–D) over 8 rounds. Crowned blocks are leaders, with rotation $A_1, B_2, C_3, D_4, A_5, B_6, C_7, D_8$. Black arrows denote references to leaders (i.e., votes); gray arrows denote references to non-leader blocks. The *State* row shows the decision reached for each leader slot. The *Payloads* row shows $\text{hist}_{\text{DA}}(L)$ for each committed leader L whose blocks (with data-availability certificates) are appended to the output sequence at that point. Slot A_1 is `direct commit`: round 2 contains 3 votes for A_1 (from A_2, B_2, C_2), and round 3 contains 3 QCs for A_1 (A_3, B_3, C_3 each reach all 3 voters). For slot B_2, we cannot directly decide whether it should be committed or skipped: round 3 contains 3 votes for B_2, so the slot is not directly skipped, but round 4 contains only 1 QC for B_2 (only A_4 reaches all 3 voters), so the slot is not directly committed either. Slot B_2 remains undecided at this stage. For slot C_3, we also cannot directly decide: round 4 contains 3 votes for C_3 (so not directly skipped), but no round-5 block is a QC for C_3 (none reaches all 3 voters), so it is not directly committed either. Slot C_3 remains undecided at this stage. Slot D_4 is `direct skip`: in round 5, A_5, B_5, C_5 all reference only non-leader blocks of round 4, so $2f+1$ blocks fail to vote for D_4 (skip pattern). Slots A_5 and B_6 are `direct commit`: each receives ≥ 3 votes in the next round and ≥ 3 QCs in the round after. The indirect rule now resolves the earlier undecided slots, using as anchor the first committed leader at round $\geq r+3$ for each slot at round r. For B_2 (round 2), the anchor is A_5: its history contains the QC A_4 for B_2, so B_2 is `indirect commit`. For C_3 (round 3), the anchor is B_6 (since A_5 is at round 5, below the $r{+}3 = 6$ threshold): B_6's history contains no QC for C_3 (no round-5 block is a QC), so C_3 is `indirect skip`. Finally, slots C_7 and D_8 remain `undecided`: C_7 has only 2 votes at round 8 (from A_8, D_8), and D_8 has no later round to resolve it.

1. *Encoding the payload:* The payload $\texttt{payload}(B)$ is encoded into $n = 3f+1$ fragments using the Reed-Solomon encoding function with $n = 3f+1$ and $k = f+1$:

$$c = (c_1, c_2, \ldots, c_n) = \text{encRS}(\texttt{payload}(B), n, k).$$

We denote $c_i = \text{frag}_i(B)$ for the i-th fragment of encoded payload $\texttt{payload}(B)$. The first $f{+}1$ fragments, $c_1, c_2, \ldots, c_{f+1}$, are called information fragments. They represent the payload; the other pieces are parity fragments. The original payload can be recovered from any $f+1$ fragments.

More precisely, by setting $k = f+1$, the payload is represented as

$$\begin{pmatrix} x_{1,1} & x_{1,2} & \ldots & x_{1,k} \\ \ldots & \ldots & \ldots & \ldots \\ x_{t,1} & x_{t,2} & \ldots & x_{t,k} \end{pmatrix}$$

where each $x_{i,j} \in \mathbb{F}_q$ and $t = \frac{T}{(f+1)\log_2 q}$ with T being the size (in bits) of $\texttt{payload}(B)$[9]. Every row is treated as an information sequence and encoded into a length-n sequence of elements from $\mathbb{F}_q$ with the Reed-Solomon code

$$\begin{pmatrix} x_{1,1} & x_{1,2} & \ldots & x_{1,n} \\ \ldots & \ldots & \ldots & \ldots \\ x_{t,1} & x_{t,2} & \ldots & x_{t,n} \end{pmatrix}.$$

Each column is a *fragment*. The first k are information fragments representing the payload; the remaining $n-k$ are parity fragments. In other words, k fragments $c_1, \ldots, c_k$ were encoded into n fragments $c_1, \ldots, c_n$ by computing parity fragments $c_{k+1}, \ldots, c_n$.

2. *Creating the Merkle Tree:* After encoding the data, we compute the hash of every fragment:

$$h_i = \text{hash}(c_i) \quad \text{for} \quad i = 1, \ldots, n.$$

These hashes are used to construct a Merkle tree, where the leaf nodes are the hashes of the fragments. The root of the Merkle tree, h, is computed as:

$$h = \text{MerkleRoot}(h_1, h_2, \ldots, h_n).$$

[9] If T is not divisible by $(f+1)\log_2 q$, we round up and use zero padding.

The Merkle root h is stored in the `payloadCommit`field of the block header.

3. *Merkle proofs:* To validate the integrity of an individual fragment c_i, the ith party provides a Merkle proof. A Merkle proof

$$prf_i = \text{MerkleProof}(c_i, i) \tag{S4}$$

is a sequence of sibling hashes from the Merkle tree that allows a party to reconstruct the Merkle root h using only the fragment c_i and its corresponding proof prf_i.

Receiving the Block. Whenever party p_i receives a full block B, it extracts `payload`(B), encodes it, computes the Merkle tree and Merkle root, and validates the correctness of `payloadCommit` in the header. If the commitment is incorrect, then the data is discarded. Otherwise, the party saves the data and acknowledges that it has the data in its future block. Additionally, party p_i saves fragment c_i of encoded data together with its Merkle proof prf_i (see (S4)) and Merkle root h.

Cordial Dissemination of Payload. Whenever the broadcast history event is triggered (see the Push pacemaker description), a party p_i sends to p_j not only the unknown headers, but also some fragments. Party p_i pushes its own fragment c_i of block B to party p_j together with Merkle proof prf_i and Merkle root h for all blocks B such that the payload of B became available to p_i after the previous push from p_i to p_j. In other words, party p_i sends to p_j its own fragments that it has not sent to p_j before.

Data Reconstruction. Whenever party p_i receives a fragment c_j together with its Merkle proof prf_j and data commitment—the Merkle root h—it verifies the correctness of the data commitment with the help of the Merkle proof. If the proof is correct, the party saves the fragment and data commitment.

Once $f + 1$ valid fragments $d_1, \ldots, d_{f+1}$ are collected and placed at their corresponding positions for some data commitment h, the codeword of the RS code is reconstructed using the decoding function:

$$\hat{c} = (\hat{c}_1, \ldots, \hat{c}_n) = \text{decRS}((d_1, d_2, \ldots, d_{f+1}), n, f + 1).$$

After that, the party proceeds with processing of the data in the same way as after the encoding of the data received from the full block. After verifying the reconstructed payload against h, the party stores it. In each subsequent round r, it includes in `acks` the hash references of locally known block headers B with B.`payloadCommit` $= h$ and $B.round < r$ that it has not previously acknowledged. Fragment c_i is saved together with its proof prf_i and h for dissemination to other parties.

B Protocol Pseudocode

This section presents pseudocode for the shared execution skeleton of our uncertified DAG BFT protocols. The pseudocode supports all four protocol variants discussed in the paper: Starfish, Starfish-L, Mysticeti, and Mysticeti-L. Algorithm S1 takes an input parameter `protocolName` (treated as a global constant in the pseudocode) and uses conditional branches to select the few protocol-specific details (pacemaker timeouts δ_{LT} and δ_{BT}, compressed vs. uncompressed vote/certificate checks, payload sequencing, and broadcasting).

Algorithms operate on block headers stored in the local DAG. For readability, we use the terms *block* and *block header* interchangeably whenever there is no ambiguity; when payload matters, we refer explicitly to payload(B) and the Starfish-specific fields. The base uncertified DAG block format is defined in Section 3. Starfish modifies the header format by introducing payload commitments and data-availability attestations (Section 5), and the compressed variants Starfish-L and Mysticeti-L further use partial/multi-signatures, multi-signature-based clocks for round advancement, and an optional leader-certificate field `leaderCert` (Section 6).

For a block (header) B, we use the following field notation (only a subset may be present, depending on `protocolName`): $B.author$ (party index / signer), $B.round$ (round number; in L variants the header carries the signed round value $\langle B.round \rangle_{B.author}$), $B.roundCert$ (round certificate $\langle B.round - 1 \rangle_{\text{agg}}$ in L variants), $B.ancestors$ (ancestor hash references and, in compressed variants, partial signatures), $B.leaderCert$ (optional leader-certificate field in compressed variants), $B.acks$ (header hash references in Starfish; explicit DACs in Starfish-L), and $B.payloadCommit$ (Starfish-style payload commitment).

We also use the following local-DAG notation in the pseudocode. DAG[r] denotes the set of blocks from round r currently stored in the local DAG.

For uncompressed blocks, FirstLeaderReference(B, r) returns the first hash reference in $B.ancestors$, in the order carried in the signed block or header, to a round-r block created by GetPredefinedLeader(r); it returns `null` if there is no such reference.

For Starfish and Starfish-L, Verify(B) validates the signed header; for Mysticeti and Mysticeti-L, it validates the signed full block. VerifyAndStorePayload(B) processes a full Starfish or Starfish-L block as specified in Appendix A.6. It re-encodes the payload and checks that the resulting Merkle root equals $h = B.\texttt{payloadCommit}$. On success, it stores the payload in $PayloadStore[h]$, retains (i, c_i, prf_i) in $FragmentStore[h]$, and returns `true`; on failure, it discards the attached payload and returns `false`. After processing, $B.\texttt{payload}$ denotes $PayloadStore[h]$ if present, and `null` otherwise. Payload processing also applies when the header is already known.

In Starfish and Starfish-L, CreateBlock(r) stores its own payload and its fragment with proof before broadcasting. In Starfish, it includes in $B.acks$ the hashes of locally known headers C with $C.round < r$ and $C.\texttt{payloadCommit} \in PayloadStore$ that p_i has not previously acknowledged. In Starfish-L, it includes previously unpublished explicit DACs for its own blocks from $ReadyAcknowledgments$. It also records its own partial signature on the new block's payload commitment in $PendingSignatures$ and calls TryAggregateSignatures for that commitment.

Algorithm S1 shows how parties execute rounds, create blocks, advance rounds, and trigger cordial dissemination (Push/Lazy-Push). Step conditions are in Table 2. We also provide algorithms for consensus utilities and commit/skip logic in Algorithms S2 and S3, as well as broadcast dissemination details in Algorithm S4. Starfish and Starfish-L specific functions are detailed in Algorithm S5.

Condition **A2** implies that an honest party cannot advance rounds without first creating its own block, so (as long as it makes progress) we expect every honest party to create a block in every round. When receiving blocks/headers, parties enforce causal availability: after signature verification, a received block/header B is integrated into the local DAG only if all block headers referenced in $B.ancestors$ are already present; otherwise B is buffered until missing ancestors arrive (predicate CausalAvailable in Algorithm S4).

Algorithm S1: Uncertified DAG BFT Execution for Party p_i

input : $protocolName \in$ {Starfish, Starfish-L, Mysticeti, Mysticeti-L}
Global variables and constants:
- $protocolName$ // Selected protocol variant (global input)
- $(\delta_{LT}, \delta_{BT}) \leftarrow$ GetPacemakerTimeouts() // Pacemaker timeouts; depend on Push vs. Lazy-Push
- $n \leftarrow 3f + 1$ // Number of parties; f is maximum number of Byzantine parties
- $r_{decided} \leftarrow 0$ // Latest round of block in the sequence of decided leaders
- $r_{highest} \leftarrow 0$ // Highest round of block in local DAG
- $r_{last} \leftarrow 0$ // Round of latest own block
- $RoundQC \leftarrow \emptyset$ // Aggregated round certificates $\langle r \rangle_{\text{agg}}$ (L variants)
- DAG $\leftarrow$ GenesisBlocks() // Local DAG of p_i initialized with predefined blocks of round 0

procedure *Execute*()
- ◇ *Execute the selected protocol variant*
- **for** $r = 1, 2, \ldots$ **do**
  - ExecuteRound(r)

procedure *ExecuteRound*(r)
- ◇ *Execute given round*
- Step 1:
- BroadcastUnknownHistory(δ_{BT}) // Cordial dissemination
- $\tau_{enter}(r) \leftarrow$ LocalTime() // Local time of party p_i
- Step 2:
- **while** *TryCreateBlock*($r, \tau_{enter}(r)$) =false **do**
  - ReceiveMessages() // Updates DAG
  - TryDecide($r_{decided}, r_{highest}$) // Try commit leader blocks and sequence payload
- Step 3:
- $r_{last} \leftarrow r$
- BroadcastUnknownHistory(δ_{BT}) // Cordial dissemination
- Step 4:
- **while** *TryAdvanceToNextRound*(r)=false **do**
  - ReceiveMessages() // Updates DAG
  - TryDecide($r_{decided}, r_{highest}$) // Try commit leader blocks and sequence payload

Algorithm S1: Uncertified DAG BFT Execution for Party p_i (continued)

function *GetPacemakerTimeouts*()
- ◇ *Return* $(\delta_{LT}, \delta_{BT})$
- **if** $protocolName \in$ {*Starfish*, *Mysticeti*} **then**
  - **return** $(2\Delta, 0)$ // Push (Starfish, Mysticeti)
- **return** $(8\Delta, 3\Delta)$ // Lazy-Push (Starfish-L, Mysticeti-L)

function *TryAdvanceToNextRound*(r)
- ◇ *Check conditions* ***A1*** *and* ***A2*** *in Table 2 to enter next round*
- **if** *NumberPartiesInRound*(r) $\geq 2f + 1$ ***and*** $r_{last} = r$ **then**
  - **return** true
- **return** false

function *TryCreateBlock*($r, \tau_{enter}(r)$)
- ◇ *Check conditions* ***C1****,* ***C2*** *or* ***C3*** *in Table 2 to create new block*
- **if** *LeaderConditionsMet*(r) ***or*** *LocalTime*() $\geq \tau_{enter}(r) + \delta_{LT}$ ***or*** *NumberPartiesInRound*(r) $\geq 2f + 1$ **then**
  - CreateBlock(r) // If created by C1, choose round-$(r-1)$ ancestors witnessing L1 and L2; L variants include signed round value $\langle r \rangle_{p_i}$, $B.roundCert = \langle r-1 \rangle_{\text{agg}}$, and may include $B.leaderCert = \langle \text{leader}_{r-2} \rangle_{\text{agg}}$?
  - **return** true
- **return** false

function *LeaderConditionsMet*(r)
- ◇ *Check for given round condition* ***C1*** *in Table 2*
- $leader_{r-1} \leftarrow$ GetPredefinedLeader($r-1$)
- $Leaders \leftarrow \{B \in \text{DAG}[r-1] \text{ s.t. } B.author = leader_{r-1}\}$
- L1 $\leftarrow |Leaders| > 0$
- $leader_{r-2} \leftarrow$ GetPredefinedLeader($r-2$)
- $LeaderProposals \leftarrow \{L \in \text{DAG}[r-2] \text{ s.t. } L.author = leader_{r-2}\}$
- $Votes \leftarrow \text{DAG}[r-1]$
- VoteQuorum $\leftarrow \exists L \in LeaderProposals$ s.t. $|\{B.author : B \in Votes \text{ s.t. IsVote}(B, L) = \text{true}\}| \geq 2f + 1$
- SkipPattern $\leftarrow$ IsDirectlySkippedSlot($r-2$)
- L2 $\leftarrow$ VoteQuorum **or** SkipPattern
- **return** L1 & L2

Algorithm S2: Helper Functions

function *GetPredefinedLeader*(r)
- ◇ *Return index of leader party for round r*
- **return** $(r \bmod n) + 1$

function *NumberPartiesInRound*(r)
- ◇ *Return number of distinct parties with blocks in round r*
- **return** $|\{B.author : B \in \text{DAG}[r]\}|$

function *IsDirectlySkippedSlot*(r)
- ◇ *Return* `true` *if leader slot of round r is directly skipped*
- $leader \leftarrow$ GetPredefinedLeader(r)
- $Votes \leftarrow \text{DAG}[r+1]$ `// All blocks in voting round`
- $LeaderProposals \leftarrow \{B \in \text{DAG}[r] \text{ s.t. } B.author = leader\}$ `// Round-`r` leader's proposed blocks; |`$LeaderProposals$`| > 1 if the leader equivocated`
- **for** $L \in LeaderProposals$ **do**
  - $res \leftarrow |\{B.author : B \in Votes \text{ s.t. IsVote}(B, L) = \texttt{false}\}|$
  - **if** $res < 2f + 1$ **then**
    - **return** `false` `// Found a proposal with fewer than` $2f+1$ `non-voters`
- **if** $|LeaderProposals| = 0$ & *NumberPartiesInRound*$(r+1) < 2f + 1$ **then**
  - **return** `false`
- **return** `true`

function *DirectlyCommittedLeaderBlock*(r)
- ◇ *Return directly committed leader block of round r if any*
- $leader \leftarrow$ GetPredefinedLeader(r)
- $LeaderProposals \leftarrow \{B \in \text{DAG}[r] \text{ s.t. } B.author = leader\}$ `// All proposals for leader slot at round` r
- $Certificates \leftarrow \text{DAG}[r+2]$ `// All blocks in certifying round`
- **for** $L \in LeaderProposals$ **do**
  - $res \leftarrow |\{B.author : B \in Certificates \text{ s.t. IsCert}(B, L) = \texttt{true}\}|$
  - **if** $res \geq 2f + 1$ **then**
    - **return** L
- **return** Undecided(r)

Algorithm S2: Helper Functions (continued)

function *IsVote*(B_{vote}, B_{leader})
- ◇ *Return* `true` *if one block votes for another*
- **if** $protocolName \in \{$*Starfish-L*, *Mysticeti-L*$\}$ **then**
  - VoteCondition $\leftarrow$ "B_{vote} contains $\langle \text{hash}(B_{leader})\rangle_{B_{vote}.author}$ in $B_{vote}.ancestors$"
- **else**
  - VoteCondition $\leftarrow$ "FirstLeaderReference$(B_{vote}, B_{leader}.round) = \text{hash}(B_{leader})$"
- **return** VoteCondition

function *IsCert*(B_{cert}, B_{leader})
- ◇ *Return* `true` *if one block is a certificate for another*
- **if** $protocolName \in \{$*Starfish-L*, *Mysticeti-L*$\}$ **then**
  - CertCondition $\leftarrow$ "$\langle \text{hash}(B_{leader})\rangle_{\text{agg}} \in B_{cert}.leaderCert$"
  - **return** CertCondition
- $Ancestors \leftarrow \{B \in \text{DAG s.t. hash}(B) \in B_{cert}.ancestors\}$
- $res \leftarrow |\{B.author : B \in Ancestors \text{ s.t. IsVote}(B, B_{leader}) = \texttt{true}\}|$
- **return** $res \geq 2f + 1$

function *IsLink*(B_{old}, B_{new})
- ◇ *Return* `true` *if one block is reachable from another*
- **return** $\exists$ sequence of blocks $B_1, \ldots, B_k \in \text{DAG}$, $k \in \mathbb{N}$, s.t.
  $B_1 = B_{old}, B_k = B_{new}$ and
  $\forall j \in [2, k] : \text{hash}(B_{j-1}) \in B_j.ancestors$

function *GetReachableBlocks*(L)
- ◇ *Return all blocks reachable from leader block*
- **return** $\{B \in \text{DAG s.t. IsLink}(B, L) = \texttt{true}\}$

function *IsCertifiedLink*(B_{anchor}, B_{leader})
- ◇ *Return true if block can reach certificate for another block*
- $r \leftarrow B_{leader}.round$
- $Certificates \leftarrow \text{DAG}[r+2]$ `// All blocks in certifying round`
- **return** $\exists B \in Certificates$ s.t. IsCert$(B, B_{leader}) =$ `true` & IsLink$(B, B_{anchor}) =$ `true`

Algorithm S3: Leader Committer and Payload Sequencer

Global variables and constants:

$PayloadSeq \leftarrow [\ ]$ // Global output sequence of payloads
$Sequenced \leftarrow \emptyset$ // Set of $(author, round)$ pairs already accounted in $PayloadSeq$
$Considered \leftarrow \emptyset$ // Headers already considered for payload ordering, identified by header hash
$Ordered \leftarrow \emptyset$ // $(author, round)$ pairs already placed in the Starfish payload order
$PendingPayloads \leftarrow [\ ]$ // Ordered headers whose payloads have not yet been output

procedure TryDecide($r_{decided}, r_{highest}$)
- ◇ *Update sequence of commit/skip decisions for leaders*
- $sequence \leftarrow [\ ]$
- **for** $r \in [r_{highest} - 1$ *down to* $r_{decided} + 1]$ **do**
  - $status \leftarrow$ TryDirectDecide(r)
  - **if** $status$ = Undecided(r) **then**
    - $status \leftarrow$ TryIndirectDecide($r, sequence$)
  - $sequence \leftarrow status || sequence$
- **for** $status \in sequence$ *in ascending round-order* **do**
  - **if** $status$ = Undecided(r) **then**
    - break
  - **if** $status$ = Commit(L) **then**
    - OnCommitLeader(L)
  - $r_{decided} \leftarrow r_{decided} + 1$ // Update last decided round

function TryDirectDecide(r)
- ◇ *Return direct commit/skip decision or undecided state*
- **if** IsDirectlySkippedSlot(r) = true **then**
  - **return** Skip(r)
- $L \leftarrow$ DirectlyCommittedLeaderBlock(r)
- **if** $L \neq$ Undecided(r) **then**
  - **return** Commit(L)
- **return** Undecided(r)

function TryIndirectDecide($r, sequence$)
- ◇ *Return indirect commit/skip decision or undecided state*
- $Anchors \leftarrow \{s \in sequence$ s.t. $s.round \geq r + 3\}$
- **for** $A \in Anchors$ *in ascending round-order* **do**
  - **if** A = Undecided(r) **then**
    - **return** Undecided(r)
  - **if** A = Commit(B_{anchor}) **then**
    - $leader \leftarrow$ GetPredefinedLeader(r)
    - $LeaderProposals \leftarrow \{B \in$ DAG$[r]$ s.t. $B.author = leader\}$ // All proposals for leader slot at round r
    - **for** $L \in LeaderProposals$ **do**
      - **if** IsCertifiedLink(B_{anchor}, L) **then**
        - **return** Commit(L)
    - **return** Skip(r)
- **return** Undecided(r)

Algorithm S3: Leader Committer and Payload Sequencer (continued)

```
procedure OnCommitLeader(L)
    ◇ Dispatch payload sequencing by protocolName
    if protocolName ∈ {Starfish, Starfish-L} then
        SequencePayloadWithDAC(L)
    else
        SequencePayload(L)

procedure TryResumeSequencing()
    ◇ Output the available prefix of pending Starfish or Starfish-L payloads
    if protocolName ∉ {Starfish, Starfish-L} then
        return
    while PendingPayloads ≠ [ ] do
        B ← Head(PendingPayloads)
        AppendPayload(B)
        if (B.author, B.round) ∉ Sequenced then
            return                                                    // Wait for the first missing payload
        PendingPayloads ← Tail(PendingPayloads)

procedure AppendPayload(B)
    ◇ Append payload of block B to global sequence if available
    if (B.author, B.round) ∉ Sequenced then
        if protocolName ∈ {Starfish, Starfish-L} then
            h ← B.payloadCommit
            if h ∈ PayloadStore then
                B.payload ← PayloadStore[h]
            else
                B.payload ← null
            if protocolName = Starfish-L and B.payload = null then
                PullPayload(B)                                        // Request missing payload
        if B.payload ≠ null then
            Sequenced ← Sequenced ∪ {(B.author, B.round)}             // Store (author, round) to ignore equivocations
            PayloadSeq ← PayloadSeq||[B.payload]

procedure SequencePayload(L)
    ◇ Sequence payload from committed leader block
    Reachable ← sort(GetReachableBlocks(L))
    for B ∈ Reachable do
        AppendPayload(B)                                              // Append payload of B if not sequenced already
```

Algorithm S4: Broadcasting for Party p_i

Global variables and constants:

$FragmentStore \leftarrow \emptyset$ // Maps payload commitment h to set of $(party, chunk, proof)$ tuples
$PayloadStore \leftarrow \emptyset$ // Maps payload commitment h to payload
$PendingSignatures \leftarrow \emptyset$ // Partial payload signatures indexed by commitment (Starfish-L)
$ReadyAcknowledgments \leftarrow \emptyset$ // Aggregated explicit DACs (Starfish-L)
$RoundShares \leftarrow \emptyset$ // Maps round r to set of $(party, \langle r \rangle_{party})$ tuples (L variants)
$\forall j : Known[j] \leftarrow$ GenesisBlocks() // Genesis blocks are globally known
$Suspended \leftarrow \emptyset$ // Buffer of verified blocks/headers whose ancestors are not yet in DAG

// Message tag type: $tag \in \{$Block, Header, Fragment, PayloadRequest, PSign$\}$.

procedure SendMessage$(j, messages)$
  ◇ *Send messages to party p_j and update Known for blocks/headers*
  for $(tag, msg) \in messages$ **do**
    Send$(p_j, (tag, msg))$
    if $tag \in \{$Block, Header$\}$ **then**
      $Known[j] \leftarrow Known[j] \cup msg$ // Update Known only for blocks/headers

procedure BroadcastUnknownHistory(δ_{BT})
  ◇ *Broadcast unknown history to peers*
  $DAG_{snap} \leftarrow$ DAG // Freeze local DAG at the moment this procedure is called
  for $j \in \{1, \ldots, n\} \setminus \{i\}$ **do**
    $Own \leftarrow \{B \in DAG_{snap} \setminus Known[j]$ s.t. $B.author = i\}$
    SendMessage$(j, [(\text{Block}, Own)])$ // Send own blocks immediately (for all protocols).
  Wait(δ_{BT}) // Caller continues in parallel.

  for $j \in \{1, \ldots, n\} \setminus \{i\}$ **do**
    $Unknown \leftarrow DAG_{snap} \setminus Known[j]$ // Unknown history from the frozen snapshot; see (1)
    if $protocolName \in \{$Starfish, Starfish-L$\}$ **then**
      $msgs \leftarrow [(\text{Header}, Unknown)]$
    else
      $msgs \leftarrow [(\text{Block}, Unknown)]$
    if $protocolName =$ Starfish **then**
      $NewFragments \leftarrow$ GetNewFragments(j) // Own fragments not previously sent to party p_j
      $msgs \leftarrow msgs || [(\text{Fragment}, f)$ for each $f \in NewFragments]$
    SendMessage$(j, msgs)$ // Send blocks/headers and fragments

Algorithm S4: Broadcasting for Party p_i (continued)

procedure *ReceiveMessages*()
- ◇ *Receive headers/blocks/fragments and update local state*
- $(tag, sender, msg) \leftarrow$ ReceiveNetworkMessage()
- **if** $tag \in$ {Header, Block} **then**
  - **for** $B \in msg$ **do**
    - **if** *Verify*(B) = true **then**
      - **if** $protocolName \in$ {*Starfish-L*, *Mysticeti-L*} **then**
        - $RoundShares[B.round] \leftarrow RoundShares[B.round] \cup \{(B.author, \langle B.round \rangle_{B.author})\}$
        - **if** $B.round > 0$ **then**
          - $RoundQC[B.round - 1] \leftarrow B.roundCert$ // Learn $\langle B.round - 1 \rangle_{agg}$ from a future-round block
        - TryAggregateRoundCert($B.round$)
      - $payloadVerified \leftarrow$ false
      - **if** tag = Block ***and*** $protocolName \in$ {*Starfish*, *Starfish-L*} **then**
        - $payloadVerified \leftarrow$ VerifyAndStorePayload(B)
      - AddToDAG(B) // Integrate subject to causal availability; buffer otherwise
      - **if** $payloadVerified$ **then**
        - **if** $protocolName =$ *Starfish-L* **then**
          - SendMessage($B.author$, [(PSign, $\langle B$.payloadCommit$\rangle_{p_i}$)]) // Acknowledge verified, retained payload
        - TryResumeSequencing()
- **if** tag = Fragment **then**
  - $(j, c_j, prf_j, h) \leftarrow msg$ // Fragment: (party id, chunk, proof, payload commitment)
  - // Verify the proof at fragment position j.
  - **if** *VerifyFragment*(h, j, c_j, prf_j) = true **then**
    - $FragmentStore[h] \leftarrow FragmentStore[h] \cup \{(j, c_j, prf_j)\}$
    - TryReconstructPayload(h) // Attempt reconstruction if $f + 1$ fragments available
- **if** tag = PayloadRequest **then**
  - OnPayloadRequest($msg, sender$)
- **if** tag = PSign ***and*** $protocolName =$ *Starfish-L* **then**
  - $(h, \langle h \rangle_{sender}) \leftarrow msg$ // Partial signature on payload commitment
  - $PendingSignatures[h] \leftarrow PendingSignatures[h] \cup \{(sender, \langle h \rangle_{sender})\}$
  - TryAggregateSignatures(h) // Attempt aggregation if $2f + 1$ signatures available

procedure *TryAggregateRoundCert*(r)
- ◇ *Aggregate partial round signatures into a round certificate*
- **if** $r \in RoundQC$ **then**
  - **return** // Already aggregated/learned
- $Signatures \leftarrow RoundShares[r]$
- **if** $|Signatures| \geq 2f + 1$ **then**
  - $RoundQC[r] \leftarrow$ AggregateThresholdSig($Signatures$)

function *CausalAvailable*(B)
- ◇ *Return* true *iff all ancestors referenced by B are already in DAG*
- **return** $\forall h \in B.ancestors : \exists B' \in$ DAG s.t. hash$(B') = h$

procedure *AddToDAG*(B)
- ◇ *Causal-availability integration of received blocks/headers*
- $Suspended \leftarrow Suspended \cup \{B\}$
- **while** $\exists B' \in Suspended$ *s.t.* *CausalAvailable*(B') = true **do**
  - $B' \leftarrow$ any $B' \in Suspended$ s.t. CausalAvailable(B') = true
  - $Suspended \leftarrow Suspended \setminus \{B'\}$
  - DAG $\leftarrow$ DAG $\cup \{B'\}$
  - $r_{highest} \leftarrow \max(r_{highest}, B'.round)$
  - $Known[B'.author] \leftarrow Known[B'.author] \cup$ GetReachableBlocks(B') // Update blocks reachable from B'

Algorithm S5: Starfish/Starfish-L Specific Functions

procedure *TryReconstructPayload*(h)
- ◇ *Reconstruct payload if sufficient fragments available*
- **if** $h \in PayloadStore$ **then**
  - **return** // Already reconstructed
- $Fragments \leftarrow FragmentStore[h]$
- **if** $|\{j : (j, *, *) \in Fragments\}| \geq f + 1$ **then**
  - $payload \leftarrow$ DecodeRS($Fragments$) // Reed-Solomon decode from $f + 1$ fragments
  - $(c_1, \ldots, c_n) \leftarrow$ EncodeRS($payload, n, f + 1$)
  - **if** *MerkleRoot*(*hash*(c_1), …, *hash*(c_n)) $\neq h$ **then**
    - **return**
  - $PayloadStore[h] \leftarrow payload$
  - $FragmentStore[h] \leftarrow FragmentStore[h] \cup \{(i, c_i, \text{MerkleProof}(c_i, i))\}$
  - TryResumeSequencing() // May unblock sequencing prefix

procedure *SequencePayloadWithDAC*(L)
- ◇ *Extend the payload order with blocks certified for availability by L*
- $Eligible \leftarrow \{B \in$ GetReachableBlocks(L) s.t. IsDataAvailCert(L, B)$\}$
- $New \leftarrow Eligible \setminus Considered$
- $Considered \leftarrow Considered \cup Eligible$
- **for** $B \in sort(New)$ **do**
  - **if** $(B.author, B.round) \notin Ordered$ **then**
    - $Ordered \leftarrow Ordered \cup \{(B.author, B.round)\}$
    - $PendingPayloads \leftarrow PendingPayloads || [B]$
    - **if** $protocolName =$ *Starfish-L* **then**
      - PullPayload(B) // Prefetch every newly ordered missing payload
- TryResumeSequencing()

function *IsAck*(B_{ack}, B)
- ◇ *Return* `true` *if block acks a tx data of past block*
- **return** hash(B) $\in B_{ack}.acks$

Algorithm S5: Starfish/Starfish-L Specific Functions (continued)

function *IsDataAvailCert*(B_{DAC}, B_{check})
- ◇ *Return* `true` *if one block is DAC for another*
- $Reachable \leftarrow$ GetReachableBlocks(B_{DAC})
- **if** $protocolName =$ *Starfish* **then**
  - **return** $|\{B.author : B \in Reachable$ s.t. IsAck(B, B_{check}) = `true`$\}| \geq 2f + 1$ // Quorum of acknowledgments
- **else**
  - **return** $\exists B \in Reachable$ s.t. $\langle B_{check}.\texttt{payloadCommit}\rangle_{\text{agg}} \in B.\texttt{acks}$ // Starfish-L: explicit DAC proof

procedure *PullPayload*(B)
- ◇ *Request payload reconstruction if not available*
- $h \leftarrow B.\texttt{payloadCommit}$
- **if** $h \in PayloadStore$ **then**
  - $B.\texttt{payload} \leftarrow PayloadStore[h]$ // Payload already reconstructed in background
  - **return**
- **if** $B.\texttt{payload} \neq \texttt{null}$ **then**
  - **return** // Payload directly available in block
- SendMessage($*$, [(`PayloadRequest`, h)]) // Broadcast request; ReceiveMessages handles fragments

procedure *OnPayloadRequest*(h, j)
- ◇ *Reply with a locally stored fragment for h*
- **if** $\exists (c_i, prf_i)$ *s.t.* $(i, c_i, prf_i) \in FragmentStore[h]$ **then**
  - SendMessage(j, [(`Fragment`, (i, c_i, prf_i, h))])

procedure *TryAggregateSignatures*(h)
- ◇ *Aggregate partial signatures into DAC proof*
- **if** $\exists \sigma \in ReadyAcknowledgments$ *s.t.* $\sigma = \langle h \rangle_{agg}$ **then**
  - **return** // Already aggregated
- $Signatures \leftarrow PendingSignatures[h]$
- **if** $|Signatures| \geq 2f + 1$ **then**
  - $\langle h \rangle_{\text{agg}} \leftarrow$ AggregateThresholdSig($Signatures$) // BLS threshold aggregation
  - $ReadyAcknowledgments \leftarrow ReadyAcknowledgments \cup \{\langle h \rangle_{\text{agg}}\}$
  - $PendingSignatures[h] \leftarrow \emptyset$ // Clear pending signatures to save memory

C Safety Analysis

Starfish was largely inspired by prior uncertified DAG-based BFTs such as Cordial Miners and Mysticeti. This section contains proofs for totality, agreement, and total order, three of the four BAB properties defined in Section 2. These proofs are based on similar considerations as in the prior work.

C.1 Safety of Starfish

Lemma 1. *If at a round k, $2f+1$ blocks from distinct parties certify a leader block B from round $k-2$, then all leader blocks at future rounds $> k$ will have a path to a certificate for B from round k.*

Proof. Consider any leader block B' at round $r > k$. The block B' has `ancestors` containing at least $2f+1$ blocks from round $r-1$. By quorum intersection with the $2f+1$ certifiers at round k, there exists an honest party p who created both a block in B'.`ancestors` and a certifying block at round k. Since parties reference their own previous blocks via `self`, B' has a path through p's block chain to the certificate for B. □

The proof for the following lemma is different from the one provided in [2] since we use a more general condition for direct skipping of a leader slot, see Algorithm S2.

Lemma 2. *If an honest party commits a leader block of round r, then no honest party decides to directly skip the leader slot of round r.*

Proof. If an honest party p directly or indirectly commits a leader block L of round r, then there exists a set W_L of $2f+1$ parties voting in round $r+1$ for L. The set W_L includes $f+1$ honest parties. For the sake of contradiction, assume another party p' directly skipped the leader slot of round r. At the time of skipping, p' had a local DAG such that for each (if any) leader block L' of round r, there were $2f+1$ non-voters for L'. We consider two cases:
1. At least one leader block L' exists in the local DAG of p'. Among the $2f+1$ parties that do not vote for L', at least one is an honest party from W_L. Therefore, $L' \neq L$. Moreover, the local DAG of p' must also contain L because of this honest non-voter. This means there must be $2f+1$ non-voters for L, which contradicts quorum intersection.
2. No leader block of round r exists in the local DAG of p', but the local DAG contains blocks from $2f+1$ parties in voting round $r+1$. However, one of these $2f+1$ parties is an honest party from W_L, implying L is in the local DAG. This contradicts the assumption of case 2. □

Lemma 3. *If an honest party directly commits a leader block of round r, then no honest party will decide to skip the leader slot of round r.*

Proof. For the sake of contradiction, assume that an honest party p directly commits a leader block B of round r, and an honest party p' skips the leader slot of round r. By Lemma 2, p' cannot skip directly. Therefore, p' must skip indirectly.

Let B' be the anchor block when p' decided to indirectly skip the leader slot of round r. Then B' is from round $r' > r+2$. Since party p directly committed B, there exist $2f+1$ blocks from different parties certifying B in round $r+2$. By Lemma 1, there is a path from B' to a certificate for B, meaning B should not have been skipped. □

Lemma 4. *At most one leader block from a given leader slot can be certified.*

Proof. For the sake of contradiction, assume two block proposals for a slot each gather a quorum of votes. The intersection of these quorums must contain at least one honest party who voted for both blocks in the same slot, which is a contradiction. □

Corollary 1. *No two honest parties commit distinct leader blocks for the same slot.*

Lemma 5. *All honest parties have a consistent state for each leader slot, i.e., if two parties have decided the state of the slot, then both either commit the same block or skip the slot.*

Proof. Assume that at some moment parties p_1 and p_2 have inconsistent states. Pick a slot from the latest round r where one party commits the leader block and the other skips it. Without loss of generality, assume p_1 commits the leader block and p_2 skips it. By Lemmas 2 and 3, both parties made their decisions indirectly. Let r_1 and r_2 be the rounds from which p_1 and p_2 made their decisions, respectively. Then p_1 skips all slots in rounds $[r+3, r_1)$ and commits in round r_1, while p_2 skips all slots in rounds $[r+3, r_2)$ and commits in round r_2. Since r was assumed to be the last round with different decisions for p_1 and p_2, we have $r_1 = r_2$. The decision to indirectly commit or skip in round r depends only on the causal history of a committed leader block in round $r_1 = r_2$. By Corollary 1, this committed leader block must be the same for both p_1 and p_2. This leads to a contradiction. □

Corollary 2. *All honest parties have a consistent sequence of committed leader blocks; i.e., the sequence for one party is a prefix of another's, or vice versa.*

The following theorem applies to Starfish: parties sequence payload using committed leader blocks and identify which blocks in the causal past serve as DACs.

Theorem 1 (Totality, Agreement, and Total Order)**.** *If one honest party delivers the payload of block B, then all honest parties will deliver the payload of block B. Moreover, all honest parties deliver payloads in the same order.*

Proof. Sequencing block B's payload after committing leader block L depends on whether L's causal history contains acknowledgments from $2f+1$ parties. This history is the same for all parties. By Corollary 2, honest parties have prefix-consistent sequences of committed leader blocks, so they

deliver payloads in a consistent order. Thus, all honest parties deliver payloads in the same order.

We note that each party delivers at most one block per round from each party, since we remove equivocating blocks; see Appendix A.5. □

C.2 Safety of Starfish-L

The safety properties established in the previous subsection extend directly to the compressed block format of Section 6. The key observation is that Lemma 1 applies only to leader blocks, and in the compressed format, leader blocks retain full `ancestors` arrays containing at most n hash references to the latest blocks from distinct parties (with at least $2f + 1$ from round $r - 1$), while each non-leader header references its author's previous header, preserving that author's block chain. Thus the quorum intersection argument in the proof of Lemma 1 applies unchanged.

The remaining safety lemmas do not depend on the structure of non-leader `ancestors` fields: Lemmas 2 and 4 use quorum intersection on votes, Lemma 5 follows from the previous lemmas, and Theorem 1 follows from the consistent leader sequence.

D Liveness Analysis

In this section, we first prove that Push and Lazy-Push pacemakers ensure liveness. This completes the proof that all uncertified DAG BFTs with these pacemakers (Mysticeti, Cordial Miners, Starfish, Mysticeti-L and Starfish-L) are BABs. After that we analyze the latency after GST.

We distinguish two liveness properties. First, block header liveness ensures that any block header issued by an honest party will be reachable from some committed leader after GST and included in a linearized block header sequence. This proof applies to Mysticeti and Cordial Miners with the Push pacemaker[10]. Since Mysticeti and Cordial Miners sequence payloads simultaneously with headers, block header liveness is equivalent to payload liveness. Thus, proving block header liveness for Push pacemaker proves liveness for modified Mysticeti and Cordial Miners.

In Starfish, payload sequencing is separated from header sequencing. Thus, we additionally prove payload liveness in Theorem 3, which follows from header liveness and the guarantees provided by DACs.

D.1 Block-Header Liveness for the Push Pacemaker

Let $r_{\max}$ be the largest round among honest parties at the moment of GST. The first property we prove concerns synchronization of honest parties after GST, which was not demonstrated in prior work on uncertified DAGs.

Lemma 6 (Synchronicity after GST)**.** *All honest parties enter any round $r > r_{\max}$ within time Δ of each other. Moreover, all honest parties create their round-r blocks within time Δ of each other.*

Proof. Fix any round r higher than any round achieved by honest parties at the moment of GST. Let p_1 be the first honest party that enters round r, and let t_1 be this entry time. Condition **B2** implies that, upon entering round r, party p_1 broadcasts to every other party the blocks it knows and believes they may be missing (unknown history).

Consider any other honest party p. By time $t_1 + \Delta$, party p receives this broadcast. In particular, it receives enough blocks from every round $r' < r$ to satisfy the round-advancement condition **A1** for these rounds (since p_1 has already advanced through them and thus has a quorum of blocks for each such round). Moreover, the catch-up condition **C3** allows p to create any missing blocks without waiting for timeouts: whenever p is in some round $r' < r$ and has received $2f + 1$ blocks of that round, it can immediately create its own round-r' block and then advance. Since local computation is negligible compared to message delay, by time $t_1 + \Delta$ every honest party has caught up through rounds $< r$ and entered round r. Therefore, all honest parties enter round r within the time interval $[t_1, t_1 + \Delta]$.

For block creation in round r, let p_2 be the first honest party that creates a round-r block at time t_2 (possibly $p_2 = p_1$). This creation cannot be triggered by **C3**, since at time t_2 there are fewer than $2f + 1$ round-r blocks from honest parties and Byzantine parties can contribute at most f. If p_2 creates its block due to the timeout condition **C2**, then all honest parties create their round-r blocks within $[t_2, t_2 + \Delta]$ because their round-entry times (and hence timeout expirations) differ by at most Δ. Otherwise, p_2 creates its block due to **C1**. In this case, **B1** requires p_2 to broadcast its block together with unknown history; thus by time $t_2 + \Delta$ every other honest party has received the information needed to satisfy **C1** and can create its round-r block. Hence, all honest parties create their round-r blocks within a time interval of length Δ. □

Lemma 7 (Timely delivery)**.** *For any round $r > r_{\max}$, all blocks of honest parties from round $r - 1$ are integrated into every honest local DAG before the round-r timeout of any honest party expires.*

Proof. Fix any round $r > r_{\max}$. Let t_{GST} be the time of GST, and let t_{last} be the creation time of the last honest round-$(r - 1)$ block. In the Push pacemaker, when an honest party creates a block it immediately sends both the block and any unknown history needed for causal availability. Hence all round-$(r - 1)$ blocks from honest parties are integrated into every honest local DAG by time $t_{integrate} = \max(t_{GST}, t_{last}) + \Delta$.

Let t_p be the time an honest party p enters round r. Since no honest party has reached round r by time GST (by definition of $r_{\max}$), we have $t_p \geq t_{GST}$. Moreover, by Lemma 6, all honest parties enter round r within a window of length

[10] For Cordial Miners, leader slots do not exist in every round, so conditions **L1** and **L2** are considered satisfied if $r - 1$ and $r - 2$ are not leader rounds, respectively

Δ. Since every honest party must create its own round-$(r-1)$ block before entering round r (**A2**), we have $t_{last} \leq \max_{u \text{ honest}} t_u \leq \min_{u \text{ honest}} t_u + \Delta \leq t_p + \Delta$, and thus $t_p \geq t_{last} - \Delta$.

Combining $t_p \geq t_{GST}$ and $t_p \geq t_{last} - \Delta$ with $\delta_{LT} = 2\Delta$, we get

$$t_p + \delta_{LT} \geq \max(t_{GST}, t_{last}) + \Delta = t_{integrate},$$

so all honest round-$(r-1)$ blocks are integrated into p's local DAG before its round-r timeout expires. □

Lemma 8 (Consistent leader reference)**.** *For any round $r > r_{\max}$, if all honest parties receive the leader block of round $r-1$ before their round-r timeout expires, then all round-r blocks created by honest parties reference this leader block.*

Proof. If the round-r block creation for honest party p is triggered by condition **C1**, then p's block references the leader block of round $r-1$. If the block creation is triggered by the timeout condition **C2**, p has already received the leader block before timeout and will reference it.

We now consider the case when condition **C3** triggers block creation for party p. Consider the $f+1$ honest parties that create blocks first in round r, denoted set W. Their block creation cannot be triggered by **C3** (insufficient blocks), so they were triggered by **C1** or **C2** and referenced the leader block by the above arguments. For party p triggered by **C3** (receiving $2f+1$ round-r blocks), at least one block B is from W and references the round-$r-1$ leader. Party p receives the round-$(r-1)$ leader block with B and will reference it. □

Lemmas 7 and 8 imply the following corollary.

Corollary 3. *Suppose an honest party occupies the leader slot of round $r-1$ such that $r > r_{\max}$. Then all round-r blocks of honest parties reference the leader block of honest party from round $r-1$.*

Using similar arguments as in the proof of Lemma 8, one can prove the following statements about voting blocks.

Lemma 9. *Suppose that for some round r, each honest party has received before its r-round timeout expires $2f+1$ blocks voting for a leader block of round $r-2$. Then all round-r blocks created by honest parties reference the $2f+1$ blocks voting for the leader block of round $r-2$.*

Proof. If the round-r block creation for some honest party p was triggered by the condition **C1** then the block references the $2f+1$ blocks voting for the leader block of round $r-2$ by definition. If it was triggered by the timeout condition **C2** then party p has, by assumption, already received $2f+1$ blocks voting for the leader block of round $r-2$ and will reference them.

It remains to explain the case when condition **C3** triggered the block creation. In this case, we consider $f+1$ honest parties, which are the fastest to create a block in round r. The block creation couldn't have been triggered by condition **C3** since there are not enough round-r blocks for it. Therefore, they have referenced $2f+1$ blocks voting for the leader block of round $r-2$ by the above arguments. Denote the set of these $f+1$ honest parties by W. The block creation for party p was triggered by **C3**, i.e., p received $2f+1$ round-r blocks. There is at least one block B from W among them, and this block references $2f+1$ blocks voting for the leader block of round $r-2$. The party p received them together with B and will reference them in its block. □

Corollary 3 and Lemmas 7 and 9 imply the following corollary:

Corollary 4. *Suppose an honest party occupies the leader slot of round $r-2$ such that $r > r_{\max}+1$. Then all round-r blocks of honest parties will be certificates for the leader block of the honest party from round $r-2$.*

Lemma 10 (Honest leaders committed)**.** *Any leader block created by an honest party in round $r \geq r_{\max}$ will be marked* TO-COMMIT *by the direct decision rule.*

Proof. Consider a leader block created by an honest party p in round $r \geq r_{\max}$. By Corollary 4 all honest parties will certify this leader block in round $r+2$. These $\geq 2f+1$ certificates will be obtained by every party, and every honest party will mark the leader block of round r TO-COMMIT by the direct decision rule[11]. □

The next two lemmas are analogues of Lemmas 11 and 12 in Mysticeti [2] with minor adjustments. We provide the proofs for completeness. They apply only to Starfish and Mysticeti: Cordial Miners does not use pipelined leaders, so Lemma 11 is not needed, and Lemma 12 is immediate.

Lemma 11. *The round-robin leader schedule ensures that in any $n+2$ consecutive rounds, there are three consecutive rounds in which their leaders are honest.*

Proof. Consider the cyclic order of leaders induced by the round-robin schedule. In every cycle of n rounds, each party is the leader exactly once, so there are at most f Byzantine leaders in the cycle.

Suppose for contradiction that in the cyclic order there are no three consecutive rounds whose leaders are honest (including the wrap-around triples that cross the cycle boundary). Then, after every Byzantine leader there can be at most two consecutive honest leaders before the next Byzantine leader. Counting around the cycle gives $n \leq 3f$, contradicting $f < \frac{n}{3}$. Thus, the cyclic order contains three consecutive honest leaders.

Any window of $n+2$ consecutive rounds contains a full cycle of n leaders plus the first two leaders of the next cycle, and therefore contains every length-3 consecutive segment of the cyclic order (including the wrap-around ones). Hence, it contains three consecutive rounds with honest leaders. □

[11] Cordial Miners has only this rule

Lemma 12 (All leaders decided)**.** *After GST, any undecided leader block will eventually be decided, i.e., marked as* TO-COMMIT *or* TO-SKIP*.*

Proof. Consider an arbitrary undecided leader block from round r. By Lemma 11 there will be 3 consecutive rounds k, $k+1$, $k+2$ with honest leaders in the corresponding leader slots for some k such that $\max(r, r_{\max}) \le k \le \max(r, r_{\max}) + n - 1$. The leader blocks in these 3 rounds will be marked TO-COMMIT by Lemma 10. Let us iterate over all undecided leader blocks before round k in the order of decreasing rounds and apply the indirect decision rule. For each undecided block, the corresponding anchor will be marked TO-COMMIT, so by the indirect decision rule the undecided block will be marked TO-COMMIT or TO-SKIP. □

Lemmas 10 and 12 imply the liveness of the consensus.

Theorem 2. *Any leader block created by an honest party in round $r \ge r_{\max}$ will be committed. Any block header created by an honest party appears in a subdag of some leader and is eventually sequenced.*

D.2 Liveness of Starfish

Lemma 13. *The payload of any block created by an honest party will eventually get a data availability certificate that will be committed.*

Proof. A block B created by an honest party at t_1 will be received by every honest party by time $\max(t_{GST}, t_1) + \Delta$. Each honest party will acknowledge the data availability of the payload in its next block. Let t_2 be the time when the last honest party includes the acknowledgment. Every leader block, created after $t_2 + \Delta$ by an honest party will be a DAC for the payload of B. By Lemmas 10 and 12, such a leader block will eventually be decided and committed; thereby, the payload of block B will be sequenced at latest after committing such a leader block. □

Lemma 14. *If the payload of a block has a data availability certificate in the view of one honest party, then the payload of that block will eventually be available to every honest party.*

Proof. Let L be a block that serves as a data availability certificate (DAC) for a block B in the view of some honest party. By definition of a DAC in Starfish, the causal history of L contains acknowledgments for B from at least $2f+1$ parties. Among these acknowledgers, at least $f+1$ are honest. Each honest acknowledger has locally verified the availability of payload(B) and therefore can compute a valid encoded fragment of payload(B).

When an honest party acknowledges payload(B) in one of its blocks, it triggers a broadcast history event (condition **B1** or **B2**) and, as part of encoded cordial dissemination (Section 5), sends its own fragment to each peer to which it has not previously sent that fragment. After GST, all such fragment messages between honest parties are delivered within time Δ. Therefore, every honest party eventually receives at least $f+1$ valid fragments of payload(B) from honest parties and can reconstruct payload(B) using Reed-Solomon erasure decoding with parameter $k = f+1$. □

Theorem 3 (Validity)**.** *The payload of every block created by an honest party will eventually be sequenced.*

Proof. Consider a block B created by an honest party p. By Lemma 13, this block will get a DAC L that is decided and committed. When attempting to sequence the payload for this block, one needs to check data availability for all blocks that got DACs in the sequence of committed leader blocks ending at L. By Lemma 14, the payloads of all such blocks will eventually be available to every honest party. □

D.3 Liveness of Lazy-Push Variants

The liveness properties established above extend to the compressed Lazy-Push variants of Section 6. Recall that Starfish-L and Mysticeti-L use the Lazy-Push pacemaker with leader timeout $\delta_{LT} = 8\Delta$ and delayed broadcast timeout $\delta_{BT} = 3\Delta$, see Table 2. The extra 3Δ delay in unknown-history dissemination widens the post-GST synchronization window from Δ to 4Δ. In addition, under causal availability an honest round-$(r-1)$ block may need up to another 4Δ after its creation to become integrated into every honest local DAG: its own block is sent immediately, but missing ancestors may arrive only in the delayed unknown-history broadcast. Therefore the timely-delivery argument for Lazy-Push requires $\delta_{LT} = 8\Delta$.

These modifications affect the synchronization bounds after GST.

Lemma 15 (Synchronicity after GST for Lazy-Push)**.** *All honest parties enter any round $r > r_{\max}$ within a time interval of length 4Δ and create their blocks in round r within a time interval of length 4Δ.*

Proof. Fix any round $r > r_{\max}$. Let p_1 be the first honest party that enters round r, and let t_1 be its round-r entry time.

Round entry. Condition **B2** implies that, when p_1 enters round r, it schedules the unknown-history broadcast for time $t_1 + 3\Delta$; hence, by time $t_1 + 4\Delta$ every honest party has received this broadcast. Since p_1 has already advanced through rounds $< r$, its local DAG contains a quorum of blocks for each round $r' < r$, in particular $2f+1$ blocks from round $r-1$. The unknown-history broadcast supplies every honest party with the blocks it is missing. Using the catch-up condition **C3** (unchanged in Lazy-Push), a lagging party can create any missing blocks for rounds $< r$ without waiting for timeouts: whenever it has received $2f+1$ blocks of its current round, it can create its own block for that round and then advance. Since local computation is negligible compared to message delays, every honest party enters round r by time $t_1 + 4\Delta$. Therefore, all honest parties enter round r within a time interval of length 4Δ.

Block creation. Let p_2 be the first honest party that creates a round-r block, and let t_2 be this creation time. This creation cannot be triggered by **C3**, since Byzantine parties can contribute at most f round-r blocks before any honest round-r block is created. If p_2 creates its round-r block due to the timeout condition **C2** (with $\delta_{LT} = 8\Delta$), then all honest parties create their round-r blocks within a time interval of length 4Δ because their round-entry times (and hence timeout expirations) differ by at most 4Δ. Otherwise, p_2 creates its round-r block due to **C1**. In Lazy-Push, when **B1** is triggered, the creator sends its own block immediately, but delays the transmission of any remaining unknown history by 3Δ. Thus, by time $t_2 + 4\Delta$ every honest party has received p_2's block and any additional prerequisite blocks it was missing to satisfy **C1**, and (by the previous part) has entered round r; hence it can create its own round-r block by then. Therefore, all honest parties create their round-r blocks within a time interval of length 4Δ. □

Corollary 5 (Timely delivery for Lazy-Push)**.** *Lemma 7 holds for Lazy-Push variants with leader timeout $\delta_{LT} = 8\Delta$ (per **C2**).*

Proof. Fix any round $r > r_{\max}$. Let t_{GST} be the time of GST, and let t_{last} be the creation time of the last honest round-$(r-1)$ block. Consider any honest round-$(r-1)$ block B created at time $t_B \le t_{last}$ by an honest party q. When q creates B, **B1** makes q send the block B itself to every peer immediately and schedule a delayed unknown-history transmission for time $t_B + 3\Delta$. When that timeout expires, for each peer j, party q sends the blocks or headers from $\mathsf{hist}(B)$ that are not in its current estimate $\mathsf{known}_q(j)$ of j's known set. If some ancestor $A \in \mathsf{hist}(B)$ is not sent to j at that time, then $A \in \mathsf{known}_q(j) = H_{q,j} \cup S_{q,j}$: if $A \in H_{q,j}$, then j already knows A from its own history, and if $A \in S_{q,j}$, then q has already sent A to j, so after GST it arrives within one network delay. Therefore, by time $\max(t_{GST}, t_B) + 4\Delta$ every honest party has received all blocks needed to satisfy causal availability for B and integrate it into the local DAG. Since $t_B \le t_{last}$, all honest round-$(r-1)$ blocks are integrated into every honest local DAG by time

$$t_{integrate} = \max(t_{GST}, t_{last}) + 4\Delta.$$

Let t_p be the time an honest party p enters round r. Since no honest party has reached round r by time GST (by definition of $r_{\max}$), we have $t_p \ge t_{GST}$. Moreover, by Lemma 15, all honest parties enter round r within a window of length 4Δ. Since every honest party must create its own round-$(r-1)$ block before entering round r (**A2**), we have $t_{last} \le \max_{u \text{ honest}} t_u \le \min_{u \text{ honest}} t_u + 4\Delta \le t_p + 4\Delta$, and thus $t_p \ge t_{last} - 4\Delta$.

Combining $t_p \ge t_{GST}$ and $t_p \ge t_{last} - 4\Delta$ with $\delta_{LT} = 8\Delta$, we get

$$t_p + \delta_{LT} \ge \max(t_{GST}, t_{last}) + 4\Delta = t_{integrate},$$

so all honest round-$(r-1)$ blocks are integrated into p's local DAG before its round-r timeout expires. □

The remaining liveness lemmas about leader-slot decisions (Lemmas 8–12 and Theorem 2) rely on the relevant leader/voting blocks being integrated into honest local DAGs before timeouts and thus extend to both Lazy-Push variants by Corollary 5. Mysticeti-L sequences payloads with headers as in Mysticeti. The payload liveness statements (Lemmas 13–14 and Theorem 3) extend to Starfish-L as well: in Starfish-L a data availability certificate is carried explicitly as an aggregated multi-signature (Section 6), which may add rounds to the *latency* of sequencing but does not affect eventual liveness after GST.

D.4 Role of C3

Proposition 4 (No uniform recovery bound without C3)**.** *Remove only **C3** from Push, retaining **A2** and timeout 2Δ. For every constant $C > 0$, an execution with $n = 10$ and $f = 3$ exists in which no leader commits during the first $C\Delta$ after GST, and honest entry times into round $r_{\max} + 1$ differ by more than $C\Delta$.*

Proof. Before GST. Partition honest parties into five fast parties F and two lagging parties L. Let F occupy leader positions $0, \dots, 4$ modulo 10. Before GST, keep F synchronized and withhold all peer messages from L, which have created only round-1 blocks. In each round, Byzantine parties create one block each, using seven previous-round blocks that exclude the previous leader and prefer Byzantine parents. The eight active parties suffice for these quorums; timeouts permit an arbitrarily long prefix. For a fast leader, the five fast parties vote and the three Byzantine parties do not. No non-genesis leader has a certificate in this prefix. Choose an integer $k > C + 3$, set $R = 10k + 3$, and place GST just after F enters round $R + 1$.

Waits after GST. After GST, deliver new honest messages with delay Δ and release the withheld prefix at GST + Δ; Byzantine parties do not relay newly created blocks of L sooner. Let t_s be the earliest creation time of a round-s block by a party in L. For each earlier round $j = 1, 11, \dots, 10k - 9$ whose leader is in F, a voting quorum requires both lagging parties' round-$(j+1)$ blocks. At most five non-voters exist, so the skip condition cannot hold. Creating a round-$(j+2)$ block via **C1** therefore requires receiving a newly created round-$(j+1)$ vote from the other lagging party, which takes at least Δ. By **A2**, a lagging party's round-$(j+2)$ timer starts only after it creates its own round-$(j+1)$ block, so proceeding via **C2** requires waiting at least 2Δ. Thus

$$t_{j+2} \ge t_{j+1} + \Delta, \qquad t_{10k-7} \ge \text{GST} + k\Delta.$$

The inequalities accumulate because **A2** makes creation times monotone.

Resulting delay. Meanwhile F enters round $R+2$ within 3Δ of GST, using its timers and the Byzantine-assisted quorums. Honest entry times into round $R+2 = r_{\max}+1$ therefore differ by more than $C\Delta$. Until GST+$C\Delta$, leaders through round $R-2$ have at most the two lagging parties as certificate authors,

while later leaders still lack the lagging parties' votes; no commit quorum exists. With **C3**, the same quorum history lets parties in L create their missing blocks immediately on receipt, under our negligible local computation assumption. □

The selected leader slots of parties in F remain undecided during this delay, so pruning an already decided prefix does not remove these dependencies. The proposition concerns Push with only **C3** removed; parties still create their blocks sequentially via **C1** or **C2**. It does not establish an infinite stall. The jump handler of [27, Alg. 4] additionally creates undecided intermediate blocks immediately; the proposition does not apply to that complete repaired protocol.

D.5 Latency of Starfish after GST

The following latency analysis applies to Starfish. Now we discuss the situation when the actual transmission latency after GST is upper bounded by $\delta < \Delta$, where the upper limit Δ is known to the parties, and δ is unknown. At first we note that in this case Lemma 6 holds with upper limit δ instead of Δ. We rewrite it with the corresponding change.

Lemma 16. *All honest parties enter any round $r > r_{\max}$ within a time interval of length δ and create their blocks in round r within a time interval of length δ.*

Proof. The proof of Lemma 6 uses only that after GST messages between honest parties are delivered within the post-GST delay bound. Replacing Δ by the (unknown) actual bound δ yields the claim. □

Lemma 17. *Assume that the leaders of rounds $r-1$, r, $r+1$ are honest, and $r > r_{\max}$. Then round-r leader block will be marked TO-COMMIT by all honest parties not later than 4δ after the moment of the creation of this leader block.*

In case when all parties are honest and $r > r_{\max} + 2$ the round-r leader block will be committed not later than after 4δ.

Proof. Suppose a leader block was created by an honest party at the moment t in round r. By Lemma 16 all other honest parties will create their round-r blocks not later than at $t+\delta$. At the moment $t+2\delta$ all round-r blocks created by honest parties will be obtained by everyone. Every honest party will be able to enter round $r+1$ at the moment $t+2\delta$. Every honest party will create its block in round $r+1$ at the moment $t+2\delta$ or earlier. Indeed, at the moment $t+2\delta$ every honest party will have received the round-r leader block and all other round-r blocks from honest parties. Since the leader of round $r-1$ is honest by Corollary 3 every round-r block from honest party will reference the leader of the round $r-1$. Hence, the condition **C1** will be satisfied at $t+2\delta$, and every honest party will create its block in round $r+1$ at the moment $t+2\delta$ or earlier.

All blocks from round $r+1$ created by honest parties will be obtained by everyone at $t+3\delta$ or earlier. So, at the moment $t+3\delta$ or earlier all honest parties will enter the round $r+2$. Since the leader of the round $r+1$ is honest, the leader block of round $r+1$ will be received at $t+3\delta$ or earlier. All honest blocks from round $r+1$ will reference the leader block from round r by Corollary 3. So, at the moment $t+3\delta$ the condition **C1** in round $r+2$ will be satisfied for all honest parties, so they create their blocks at $t+3\delta$ or earlier. By Corollary 4 these blocks will certify the leader of the round r.

Every honest party will receive at least $2f+1$ certificates at the moment $t+4\delta$ or earlier and mark the round-r leader block TO-COMMIT.

The leader block can be marked TO-COMMIT but not committed if some of the previous leader blocks are UNDECIDED. If all parties are honest, then from Lemma 10, we conclude that the leaders of round $r-2$, $r-1$, r will be marked TO-COMMIT. Similarly to Lemma 12 all previous leaders will be decided after that, and round-r leader will be committed. □

Lemmas 18 to 21 serve as useful tools for estimating latencies in various scenarios.

Lemma 18. *After GST, the time interval between the creation of blocks by an honest party p in round r and $r+1$ is*
1. *at most 2δ if the leaders of rounds r and $r-1$ are honest.*
2. *at most $2\delta + 2\Delta$ otherwise.*

Proof. If a block in round r was created at the moment t, then all other honest parties will also create their blocks not later than at $t+\delta$ by Lemma 16. Then, by time $t+2\delta$, party p receives enough round-r blocks from honest parties to advance to round $r+1$ (in particular, at least $2f+1$ such blocks). If condition **C1** is not satisfied earlier, the round-$(r+1)$ timeout expires at time $t+2\delta+2\Delta$, and p creates its round-$(r+1)$ block.

If the leaders of round r and $r-1$ are honest, then condition **C1** is satisfied at the moment $t+2\delta$ by Corollary 3, and the block will be created at moment $t+2\delta$ or earlier. □

Lemma 19. *After GST, the time that an honest party p spends in round r is*
1. *at most 3δ if the leaders of rounds $r-1$ and $r-2$ are honest.*
2. *at most $2\delta + 2\Delta$ otherwise.*

Proof. If an honest party p enters round r at moment t, then all other honest parties broadcast their blocks from round $r-1$ and enter round r not later than at $t+\delta$ by Lemma 16. The party p and all other parties obtain all blocks from honest parties not later than at $t+2\delta$. If leaders of round $r-1$ and $r-2$ are honest, then condition **C1** is satisfied, and all honest parties, including p, create their blocks at $t+2\delta$ or earlier. All parties receive round-r blocks from honest parties not later than at $t+3\delta$, and can advance to the next round.

If at least one of the leaders of rounds $r-1$ and $r-2$ is Byzantine, then the party p may need to wait until its timeout expires. Anyway, at the moment $t+2\Delta$ or earlier, the party p creates and broadcasts its round-r block. Applying Lemma 16, other parties will do the same at latest at $t+2\Delta+\delta$.

Therefore, at moment $t + 2\Delta + 2\delta$ or earlier, all honest parties receive round-r blocks from all honest parties and advance to round $r + 1$.

□

The following statement is key for establishing latency bounds over long executions (e.g., average and worst-case end-to-end latencies with 0 or f Byzantine parties). It bounds the pace at which the last honest block advances in consecutive rounds after GST: with honest leaders the DAG progresses by (at least) one round per time δ, while Byzantine leaders may introduce additional waiting up to the timeout.

Lemma 20. *Let $t_{last,r}$ denote the moment when the last honestly created block[12] of round r was created. Then after GST, $t_{last,r+1} - t_{last,r}$ is upper bounded by*
1. at most δ if the leaders of rounds r and $r-1$ are honest.
2. at most $\delta + 2\Delta$ if the leader of round r or $r-1$ is Byzantine.
3. at most δ if the leader of round $r-1$ is Byzantine and the leader of round r is honest, and the leader of round $r-1$ fails to create its block in round $r-1$.

Proof. If the last round-r block from an honest party was created at the moment t then at the moment $t + \delta$ all honest parties will obtain all round-r blocks from honest parties. It is enough to enter round $r+1$. If leaders of rounds r and $r-1$ are honest then the condition **C1** is satisfied, and they can create and broadcast their blocks from round $r + 1$.

If at least one of the leaders of round r and $r - 1$ is Byzantine, then they may need to wait their timeouts. In that case all honest parties create their blocks from round $r + 1$ not later than at $t + \delta + 2\Delta$.

If the leader of round $r-1$ is Byzantine and *failed* to create its block and leader of round r is honest, then round-r blocks from honest parties create a skip pattern for the leader round $r - 1$ and the condition **C1** is satisfied at $t + \delta$. □

Lemma 21. *If all parties are honest, then after GST, the time interval between the creation of two (consecutive) leaders' blocks is at most 2δ.*

Proof. If the leader block of round r was created at the moment t, then at the moment $t + \delta$ all other blocks from round r are created, and at moment $t+2\delta$ they are received by every party. Then the leader block for round $r + 1$ will be created not later than at $t+2\delta$ since the condition **C1** is satisfied. □

Theorem 5 (Worst-case latency, all honest parties)**.** *Suppose all parties are honest. After GST, the worst consensus latency is at most 7δ.*

Proof. We divide the latency into two subintervals I1 and I2. I1 is the time interval from the creation of a block B until the creation of the last round-r block such that all parties at or before round r acknowledged the availability of the payload of block B. The interval I2 is from the end of I1 until a leader block serving as a DAC is committed by every honest party.

Let $t_{first,r}$ denote the moment when the first round-r block is created, equivalently the creation time of the fastest honest party's round-r block (since all parties are assumed to be honest in this statement). Choose r to be a minimal round such that $t_{first,r} \geq t + \delta$, where t is the moment of creation of block B. Then all round-r blocks acknowledge block B.

Now we are ready to upper bound the duration of I1. Using Lemma 20, Lemma 16, and the fact that r is minimal such that $t_{first,r} \geq t + \delta$, we obtain

$$t_{last,r} - t \leq (t_{last,r} - t_{last,r-1}) + (t_{last,r-1} - t_{first,r-1}) + (t_{first,r-1} - t) < 3\delta.$$

Regarding I2, round-$(r + 1)$ leader block L is guaranteed to be a DAC for the corresponding payload, since all parties acknowledged its availability at or before round r. All honestly created blocks at round $r + 3$ will certify L. Thus, applying Lemma 20, I2 is bounded by the progress of slowest blocks in the DAG from round r to round $r + 3$ and the delivery of round-$(r + 3)$ blocks, i.e. 4δ.

So, the worst consensus latency is at most 7δ. □

Theorem 6 (Worst-case latency, f Byzantine parties)**.** *If $f < \frac{n}{3}$ parties are Byzantine, then after GST, the worst-case consensus latency of any block created by an honest party is upper bounded by $4f\Delta + n\delta + O(\Delta)$. If Byzantine parties fail to create their blocks in their leader rounds, then the latency can be estimated as $2f\Delta + n\delta + O(\Delta)$*

Proof. Let an honest party p create block B at time t. We divide the interval until its payload is sequenced by every honest party into three subintervals I1, I2, and I3. I1 is the time until the local DAG of every honest party contains a quorum of acknowledgments for the payload of B. I2 is the time from the end of I1 until there are three consecutive leaders from honest parties. Finally, I3 is the time until the third leader in this sequence is committed, ensuring that the payload is sequenced.

First estimate the duration of I1. At the moment t the party p creates the block B. At moment $\leq t + \delta$ all honest parties receive the block B. Applying Lemma 18, at moment $\leq t+3\delta+2\Delta$ all honest parties create blocks with acknowledgement for B. At the moment $\leq t + 4\delta + 2\Delta$ all honest parties have in their local DAGs a quorum of acknowledgements for the payload of block B. Any honest leader committed after this moment will sequence the block B.

Second we bound the duration of I2. By Lemma 11 there will be 3 consecutive honest leaders in $n + O(1)$ rounds, which will be marked TO-COMMIT by Lemma 10. After this all previous leaders that were UNDECIDED will be marked TO-COMMIT or TO-SKIP. The time interval until three consecutive honest leaders is $n\delta + 4f\Delta + O(\Delta)$ by Lemma 20. Indeed, two honest leaders in the scheduler allow the slowest honest party to create blocks with delay between them at most δ;

[12] Block created by a slowest honest party

one Byzantine leader among a window of two rounds can increase this to $\delta + 2\Delta$; if Byzantine leader does not create its block it can affect the latency in only one round, resulting in $\delta + 2\Delta$.

Finally it remains to bound the duration of I3. It takes up to $O(\Delta)$ to commit one leader among the three consecutive honest leaders.

Therefore, the total time to sequence the payload can be bounded by $n\delta + 4f\Delta + O(\Delta)$ in the worst case if f Byzantine nodes are active and $n\delta + 2f\Delta + O(\Delta)$ if they fail to produce their blocks when they are leaders. □

Theorem 7 (Worst-case latency, 1 Byzantine party)**.** *Suppose the number of parties is at least 6. If one party is Byzantine, then after GST the worst-case consensus latency of a block created by an honest party is at most* $12\delta + 4\Delta$*. If the Byzantine party fails to create its block, the latency is bounded by* $12\delta + 2\Delta$*.*

Proof. Let B be a round-r block created by an honest party p at time t_1. By time $t_2 = t_1 + \delta$, all honest parties have received B and added it to their local DAGs.

All acknowledgments for B created by honest parties appear in rounds $\geq r + 1$. Let $\hat{r}$ be the smallest round such that every honest party has created a block acknowledging B in round $\hat{r}$ or earlier. In particular, $\hat{r} \geq r + 1$.

Let t_3 be the creation time of the last round-$\hat{r}$ block created by an honest party. Consider an honest party $\hat{p}$ that acknowledges B in round $\hat{r}$. At time t_2, party $\hat{p}$ has already created its round-$(\hat{r} - 1)$ block; by Lemma 16, all honest parties create their round-$(\hat{r} - 1)$ blocks by time $t_2 + \delta$. By Lemma 20, we have

$$t_3 \leq \begin{cases} t_2 + 2\delta, & \text{if the leaders of rounds } \hat{r} - 1 \\ & \text{and } \hat{r} - 2 \text{ are honest,} \\ t_2 + 2\delta + 2\Delta, & \text{otherwise.} \end{cases}$$

By time $t_4 = t_3 + \delta$, every honest party has received from every honest party a block acknowledging B. Hence, any leader block created by an honest party and committed after time t_4 will serve as a DAC for B.

Let $r' \geq \hat{r}$ be the round of the last leader block created by an honest party before time t_4. By Lemma 16, the slowest honest party creates its round-r' block by some time $t_5 \leq t_4 + \delta$.

We now distinguish cases based on whether the Byzantine party is the scheduled leader in rounds $r' - 1, r', r' + 1, r' + 2$.
1. The Byzantine party is not the leader in rounds $r' - 1, r', r' + 1, r' + 2$. Then the round-$(r' + 3)$ blocks certify the (honest) leader block of round $r' + 1$. Moreover, by the argument of Lemma 12, once three consecutive honest leader blocks are marked TO-COMMIT, the payload of B is sequenced. By Lemma 20, the slowest honest party creates its round-$(r' + 3)$ block by time $t_5 + 3\delta$, and after an additional δ these blocks are delivered to all honest parties. Altogether, the consensus latency of B is at most $9\delta + 2\Delta$.
2. The Byzantine party is the leader in round $r' - 1$. In the worst case, B is sequenced only once the honest leader blocks of rounds $r', r'+1, r'+2$ are all marked TO-COMMIT, which happens after round-$(r' + 4)$ blocks are delivered. By Lemma 20, the slowest honest party creates its round-$(r' + 4)$ block by time $t_5 + 4\delta + 2\Delta$, and after an additional δ these blocks are delivered. Thus, the consensus latency is at most $10\delta + 4\Delta$.
3. The Byzantine party is the leader in round r'. Since $n \geq 6$ and $r' \geq \hat{r}$, the Byzantine party cannot also be the leader in rounds $\hat{r} - 1$ or $\hat{r} - 2$; hence $t_3 \leq t_2 + 2\delta$ and therefore $t_5 - t_1 \leq 5\delta$. In the worst case, B is sequenced once the honest leader blocks of rounds $r' + 1, r' + 2, r' + 3$ are marked TO-COMMIT, which happens after round-$(r' + 5)$ blocks are delivered. By Lemma 20, the slowest honest party creates its round-$(r' + 5)$ block by time $t_5 + 5\delta + 4\Delta$, and after an additional δ these blocks are delivered. Thus, the consensus latency is at most $11\delta + 4\Delta$.
4. The Byzantine party is the leader in round $r' + 1$. As in the previous case, we have $t_5 - t_1 \leq 5\delta$. In the worst case, B is sequenced once the honest leader blocks of rounds $r' + 2, r' + 3, r' + 4$ are marked TO-COMMIT, which happens after round-$(r' + 6)$ blocks are delivered. By Lemma 20, the slowest honest party creates its round-$(r' + 6)$ block by time $t_5 + 6\delta + 4\Delta$, and after an additional δ these blocks are delivered. Thus, the consensus latency is at most $12\delta + 4\Delta$.
5. The Byzantine party is the leader in round $r' + 2$. As in the previous cases, we have $t_5 - t_1 \leq 5\delta$. In the worst case, B is sequenced once the honest leader blocks of rounds $r' - 1, r', r' + 1$ are marked TO-COMMIT, which happens after round-$(r' + 3)$ blocks are delivered. By Lemma 20, the slowest honest party creates its round-$(r' + 3)$ block by time $t_5 + 3\delta + 2\Delta$, and after an additional δ these blocks are delivered. Thus, the consensus latency is at most $9\delta + 2\Delta$.

Taking the maximum over the cases gives a worst-case consensus latency of at most $12\delta + 4\Delta$.

If the Byzantine party crashes and does not create its leader blocks, then the timeout penalty of size 2Δ can be incurred at most once in the above reasoning (rather than twice), improving the worst-case bound to $12\delta + 2\Delta$. □

D.6 Latency of Mysticeti after GST

We bound worst-case and average consensus latency for Mysticeti with the Push pacemaker in the happy case.

Theorem 8 (Worst-case latency, all honest parties)**.** *Suppose all parties are honest. After GST, the worst-case consensus latency for Mysticeti is at most* 6δ*.*

Proof. All the lemmas from the previous subsection are still true for Mysticeti with Push pacemaker. Here, we bound the time until a block is sequenced.

Let an honest party p create block B at time t. We divide its consensus latency into two subintervals I1 and I2. For the round r chosen below, I1 ends when the last round-r block is

created; the leader block L of this round has B in its history. I2 ends when every party commits L.

Let $t_{first,r}$ be the time when the first block in round r was created, $t_{last,r}$ is the time of creation of the last block in the round. Choose a minimal r such that $t_{first,r} \geq t+\delta$. Since r is minimal we have $t_{first,r-1} < t+\delta$. The leader block L of round r references the block B or another block created by party p after B. In any case, B belongs to the history of leader block L. The last block in round r is created at $t_{last,r} \leq t_{last,r-1} + \delta$ by Lemma 20. By Lemma 16 $t_{last,r-1} \leq t_{first,r-1} + \delta \leq t + 2\delta$. Thus, I1 is at most 3δ.

Every honest party votes for the leader block L in round $r+1$ and certifies it in round $r+2$. By Lemma 20 $t_{last,r+2} - t_{last,r} \leq 2\delta$. After additional δ all certifying blocks from round $r+2$ are delivered to all parties and leader L is committed. Thus, I2 is at most 3δ.

To sum up, the worst-case consensus latency is at most 6δ.

□

Theorem 9 (Average latency, all honest parties)**.** *Suppose all parties are honest. After GST, the average consensus latency for Mysticeti is at most* $5\frac{1}{3}\delta$.

Proof. We define intervals I1 and I2 in the same way as in the proof of Theorem 8. The length of $I2$ is upper bounded in the same way by 3δ.

In Theorem 8 we proved that the length of I1 is upper bounded by 3δ for all blocks. We strengthen the bound for the average case by proving that for at least $2f+1$ blocks from every round the length of I1 is at most 2δ.

Indeed, the leader block L from round r references at least $2f+1$ blocks from round $r-1$. Let t be the creation time of one such block. As before, let $t_{first,r}$ be the time when the first block in round r was created, and let $t_{last,r}$ be the creation time of the last block in the round. Then

$$t_{last,r} - t = (t_{last,r} - t_{last,r-1}) + (t_{last,r-1} - t).$$

Applying Lemma 20 and Lemma 16, we get $t_{last,r} - t \leq 2\delta$.

So, for $2f+1$ blocks in every round the length of I1 is at most 2δ, and for the rest $(n-(2f+1))$ it is at most 3δ. Since $2f+1 > \frac{2}{3}n$, the average length of I1 is at most $(2\cdot\frac{2}{3}+3\cdot\frac{1}{3})\delta = 2\frac{1}{3}\delta$.

So, the total average consensus latency is at most $5\frac{1}{3}\delta$.

□

Tightness of the average happy-case bound for Mysticeti. The bound of Theorem 9 is essentially tight even when all parties are honest, but the adversary can control the network delay up to δ. We now show that there exists an all-honest post-GST execution that attains this value up to an arbitrarily small $\varepsilon > 0$.

We sketch such an execution. For every round r we split parties into a *fast* set Q_r of size $2f-1$, a *slow* set S_r of size f, and a *middle* set M_r of size 2. The division is not fixed, and a party can move between sets from round r to round $r+1$. Parties from the set Q_r create their round-r block at $t_{first,r} = (\delta-2\varepsilon)r$, parties from the set S_r at $t_{last,r} = (\delta-2\varepsilon)(r+1)+\varepsilon$, and parties from the set M_r create their blocks at $t_{mid,r} = (\delta-2\varepsilon)(r+1)-\varepsilon$. The leader of round r will always be in the set Q_r. Following the protocol's round-robin schedule, the leader of round r is the party $p \in \{1,2,\dots,n\}$ such that $p \equiv r+1 \pmod n$. We interpret all party indices in the set definitions below modulo n, using representatives in $\{1,2,\dots,n\}$. Define

$$\begin{aligned} Q_r &= \{r+1, r, \dots, r-2f+3\}, \\ M_r &= \{r+2, r+3\}, \\ S_r &= \{1,2,\dots,n\} \setminus (Q_r \cup M_r). \end{aligned}$$

Then the leader of round r belongs to Q_r and the leader of round $r+1$ belongs to M_r.

We schedule delivery delays for round-r blocks to be exactly $\delta-2\varepsilon$ in all cases except the following.

1. Delay for round-r block delivery from parties from M_r to parties from S_{r+1} is δ.
2. Delay for round-r block delivery from parties from M_r to parties from Q_{r+1} is ε.
3. Delivery from every party to itself is always zero.

In this case, any party from Q_{r+1} receives $2f+1$ blocks at $t_{first,r+1}$ from parties in $Q_r \cup M_r$ and advances to round $r+1$. One of these blocks is the leader block from round r, so condition **L1** is satisfied. All these $2f+1$ blocks vote for the leader of round $r-1$ by Corollary 3, so condition **L2** is also satisfied. Therefore, **C1** is triggered and all parties in Q_{r+1} create their round-$(r+1)$ blocks at $t_{first,r+1}$.

Parties from S_{r+1} receive all round-r blocks at $t_{last,r+1}$ and can then create their round-$(r+1)$ blocks. Before that moment, they may have only their own round-r block and blocks from parties in Q_r, i.e., at most $2f$ blocks.

Parties from M_{r+1} receive round-r blocks from Q_r and M_r at $t_{mid,r+1}$, which is enough to create a block for round $r+1$. Before that moment, a party $p \in M_{r+1}$ may have round-r blocks only from Q_r and itself, i.e., at most $2f$ blocks.

So, the described construction is possible. Now we analyze the average consensus latency. The leader block of round r receives references from all round-$(r+1)$ blocks and is certified by every block from round $r+2$. However, to commit this leader block L, a party p must receive at least $2f+1$ certificates for L. If p belongs to S_{r+3}, this happens at $t_{last,r+3}$, so the latency from the creation of L until its commitment by all parties is $3(\delta-2\varepsilon)+\delta-\varepsilon = 4\delta-7\varepsilon$.

Now we estimate the latency from the creation of a round-r block B until the creation of the first leader block L such that B is in the history of L. For the leader of round r (party $r+1$), this latency is 0. Every other block from Q_r is referenced by the leader L_{r+1} of round $r+1$, so for them the latency is $\delta-2\varepsilon$. Round-r blocks of parties from M_r are also referenced by the leader of round $r+1$, but they are created later, and their latency is ε. Round-r blocks for parties from S_r are not

referenced by the leader L_{r+1}, because they are created later and $t_{last,r} > t_{first,r+1}$. They are also not referenced by the leader L_{r+2} created by party $r+3$, because they are delivered to party $r+3$ at

$$t_{last,r} + \delta - 2\varepsilon = (\delta - 2\varepsilon)(r+2) + \varepsilon > t_{first,r+2}.$$

So the only way the round-r block B created by a party $p \in S_r$ can appear in the history of L_{r+2} is if some round-$(r+1)$ block B' created by a party $p' \in Q_{r+1} \cup M_{r+1}$ references B. Since block B is delivered to every party except p later than the leader L_{r+2} is created, this is only possible if $p' = p$, which implies $p = r+4$. Thus, for the block B created by party $r+4$ the latency is at most $t_{first,r+2} - t_{last,r} = \delta - 3\varepsilon$. For all other blocks created in round r by parties in $S_r \setminus \{r+4\}$, the latency is at least $t_{first,r+3} - t_{last,r} = 2\delta - 5\varepsilon$.

To sum up, the average consensus latency for this scenario is at least

$$\begin{aligned} 4\delta - 7\varepsilon + \frac{1}{3f+1}\Big(&0 + (2f-2)(\delta - 2\varepsilon) + 2\varepsilon \\ &+ (\delta - 3\varepsilon) + (f-1)(2\delta - 5\varepsilon)\Big) \\ &= 4\delta - 7\varepsilon + \frac{4f-3}{3f+1}\delta - \frac{9f-8}{3f+1}\varepsilon \\ &= 5\frac{1}{3}\delta + O(\varepsilon + \delta/f). \end{aligned}$$

D.7 Happy-Case Latency for Lazy-Push Variants

We now show that the happy-case timing lemmas from the previous subsection extend to the Lazy-Push variants, Mysticeti-L and Starfish-L. These lemmas depend only on the shared Lazy-Push pacemaker (own blocks broadcast immediately, unknown history delayed by 3Δ) under the happy-case assumptions (all parties honest, transmission latency at most $\delta < \Delta$). They do not use Starfish's DAC mechanism, erasure coding, or payload sequencing.

The key observation is that **B1** makes a party broadcast its own block immediately—only the unknown history broadcast is delayed by 3Δ. When all parties are honest, a few rounds after GST, every party receives all n blocks from each round via the immediate own-block broadcasts. After synchronization, delayed unknown-history broadcasts therefore send no additional data, as shown in Corollary 6.

Lemma 22. *Suppose all parties are honest and actual message latency is at most $\delta < \Delta$. Then after an initial synchronization phase of at most n rounds following $r_{\max} + 1$, all honest parties create their round-r blocks within a time interval of length δ for each subsequent round r.*

Proof. Let $X_r = t_{last,r} - t_{first,r}$ denote the block creation spread in round r, where $t_{first,r}$ and $t_{last,r}$ are the minimum and maximum block creation times.

Initial spread. By Lemma 15, all parties create round-$(r_{\max}+1)$ blocks within 4Δ, so $X_{r_{\max}+1} \le 4\Delta$.

Recurrence relation. We show $X_{r+1} \le \max(X_r, \delta)$. In the happy case ($\delta < \Delta$ and all parties honest), parties create blocks via **C1**, which requires:
(1) Creating own round-r block (**A2**),
(2) Receiving $2f+1$ round-r blocks (**A1**),
(3) Receiving the round-r leader block (**L1**),
(4) Receiving $2f+1$ votes for the leader of round $r-1$ (**L2**).
Let $T_{(2f+1)}$ be the time when the $(2f+1)$-th round-r block is created, and T_L be the creation time of the round-r leader's block. For any party p, let $d_p^Q, d_p^L \in [0, \delta]$ denote the (actual) message delays until p receives (respectively) the $(2f+1)$-th prerequisite round-r block and the round-r leader block. Note that in the happy case all round-r blocks reference the leader block of round $r-1$. Then the creation time $T_p(r+1)$ satisfies:

$$T_p(r+1) = \max\big(T_p(r),\ T_{(2f+1)} + d_p^Q,\ T_L + d_p^L\big).$$

Let $T_{\max} = \max(T_{(2f+1)}, T_L)$. Note that $t_{first,r} \le T_{\max} \le t_{last,r}$. The earliest valid creation time is bounded by information availability:

$$t_{first,r+1} \ge T_{\max}.$$

Also, since $d_p^Q, d_p^L \le \delta$, we have $T_p(r+1) \le \max(T_p(r), T_{\max} + \delta)$ for every p, and thus

$$t_{last,r+1} \le \max(t_{last,r}, T_{\max} + \delta).$$

If $t_{last,r} \ge T_{\max} + \delta$, then $t_{last,r+1} \le t_{last,r}$, while **A2** implies $t_{last,r+1} \ge t_{last,r}$, hence $t_{last,r+1} = t_{last,r}$ and

$$X_{r+1} \le t_{last,r} - T_{\max} \le t_{last,r} - t_{first,r} = X_r.$$

If $t_{last,r} < T_{\max} + \delta$, then $t_{last,r+1} \le T_{\max} + \delta$, implying

$$X_{r+1} \le (T_{\max} + \delta) - T_{\max} = \delta.$$

Thus, $X_{r+1} \le \max(X_r, \delta)$.

Convergence. We calculate the spread reduction over n rounds. If $X_r \le \delta$, then $X_{r+1} \le \delta$. Suppose $X_r > \delta$. If $X_{r+1} > \delta$, the recurrence above implies $t_{last,r} \ge T_{\max} + \delta$. Let p_{slow} be a party with $T_{p_{slow}}(r) = t_{last,r}$. We claim that as long as $X_{\ell+1} > \delta$, the same p_{slow} remains the last to create its round-$(\ell+1)$ block. Indeed, if $X_{\ell+1} > \delta$ then

$$t_{last,\ell} \ge T_{\max,\ell} + \delta$$

(where $T_{\max,\ell}$ denotes the quantity $T_{\max}$ for round ℓ), so any party u with $T_u(\ell) \le T_{\max,\ell} + \delta$ satisfies

$$T_u(\ell+1) \le T_{\max,\ell} + \delta \le t_{last,\ell} = T_{p_{slow}}(\ell) \le T_{p_{slow}}(\ell+1),$$

while parties with $T_u(\ell) > T_{\max,\ell} + \delta$ have $T_u(\ell+1) = T_u(\ell)$ and cannot overtake p_{slow}. Consider the round-robin leader schedule. Within n rounds, the party p_{slow} will be the leader in some round k. In that round, the leader block is created by p_{slow} at time $T_L = T_{p_{slow}}(k) = t_{last,k}$, hence $T_{\max,k} \ge T_L = t_{last,k}$. Then

$$t_{first,k+1} \ge T_{\max,k} \ge t_{last,k}.$$

Since $t_{last,k+1} \le \max(t_{last,k}, T_{\max,k} + \delta) = T_{\max,k} + \delta$, we have

$$X_{k+1} \le (T_{\max,k} + \delta) - T_{\max,k} = \delta.$$

Thus, the spread collapses to δ once the slowest party serves as leader.

□

Lemma 23. *In the happy case, for rounds $r > r_{\max} + n + 1$ (after the synchronization phase of Lemma 22), Lemma 18 holds for Lazy-Push variants: the time interval between the creation of blocks by an honest party in rounds r and $r+1$ is at most 2δ.*

Proof. Let p be any honest party that creates its round-r block at time t. We show p creates its round-$(r+1)$ block by time $t + 2\delta$.

By Lemma 22, all round-r blocks are created within an interval of length δ. Let T_{first} and T_{last} denote the first and last creation times. Since $t \in [T_{first}, T_{last}]$ and $T_{last} - T_{first} \leq \delta$, we have $T_{last} \leq T_{first} + \delta \leq t + \delta$.

All n round-r blocks are created by time $T_{last} \leq t+\delta$. Since message latency is at most δ, all blocks are delivered to p by time $T_{last} + \delta \leq t + 2\delta$.

For p to create its round-$(r+1)$ block via **C1**, it needs: (1) the round-r leader block (**L1**), and (2) $2f+1$ votes for the round-$(r-1)$ leader (**L2**). By time $t+2\delta$, p has received all $n \geq 2f+1$ round-r blocks, including the leader's block, satisfying **L1**. For **L2**: since both leaders are honest, by Corollary 3, all round-r blocks vote for the honest round-$(r-1)$ leader. Thus p observes $n \geq 2f+1$ votes for the round-$(r-1)$ leader, satisfying **L2**. With both **L1** and **L2** satisfied, **C1** holds and p creates its round-$(r+1)$ block by time $t + 2\delta$. □

Lemma 24. *In the happy case, Lemma 20 holds for Lazy-Push variants: $t_{last,r+1} - t_{last,r} \leq \delta$.*

Proof. All n round-r blocks are created by time $t_{last,r}$. Since message latency is at most δ, all round-r blocks are delivered to all parties by time $t_{last,r} + \delta$.

At time $t_{last,r} + \delta$, every party has received all $n \geq 2f+1$ round-r blocks (satisfying **A1**), its own round-r block (**A2**), and the round-r leader block (**L1**). For happy case condition **L2** is satisfied by Corollary 3. With both **L1** and **L2** satisfied, **C1** holds.

Therefore, all parties can create their round-$(r+1)$ blocks by time $t_{last,r} + \delta$, yielding $t_{last,r+1} \leq t_{last,r} + \delta$. □

Corollary 6. *In the happy case, the delayed unknown history broadcast in **B1** and **B2** sends no additional data, and the Lazy-Push variants achieve $O(\kappa n^2 + Mn)$ communication complexity per round.*

Proof. When party p creates round-r block B at time t, it broadcasts B immediately via **B1**. By Lemma 22 all other parties create their round-r blocks by the moment $t + \delta$, and party p receives all these blocks by $t + 2\delta$. By Corollary 3 all those received blocks reference the leader block from round $r-1$. If B is not a leader block, its history includes only its own chain and the round-$(r-1)$ leader block with its history. Since party p has received blocks referencing the round-$(r-1)$ leader block from all other parties by $t + 2\delta$, i.e., before the timer for broadcasting unknown history fires at $t + 3\Delta$, there is no need to broadcast anything.

Now consider the case where B is a leader block. Again, by the moment $t + \delta$ all parties create their round-r blocks. By Lemma 24 all parties create their block for round $r+1$ by $t + 2\delta$. By Corollary 3 all these blocks reference the block B. Party p receives from each party a block that references B by time $t + 3\delta$, i.e. before the timeout 3Δ for broadcasting history expires. So, there is no need to broadcast unknown history to anyone. Hence, only own blocks are exchanged via immediate broadcast: $n-1$ parties each send their $O(\kappa)$-sized non-leader block to $n-1$ peers, and one leader sends its $O(\kappa n)$-sized block to $n-1$ peers, yielding $O(\kappa n^2)$ metadata communication per round. The payload component is $O(Mn)$ in the happy case, so the total communication is $O(\kappa n^2 + Mn)$. □

The next corollary justifies the Mysticeti-L entries in the all-honest worst-case and average latency bounds of Theorem 11.

Corollary 7 (Happy-case latency of Mysticeti-L)**.** *Suppose all parties are honest and actual message latency is at most $\delta < \Delta$. After the synchronization phase of Lemma 22, Mysticeti-L has worst-case consensus latency at most 6δ and average consensus latency at most $5\frac{1}{3}\delta$.*

Proof. The proofs of Theorems 8 and 9 use three pacemaker-level facts: (i) honest blocks in each round are created within a δ-wide window, (ii) $t_{last,r+1} - t_{last,r} \leq \delta$, and (iii) the leader block of round r references at least $2f+1$ blocks from round $r-1$ (used in the average-case strengthening of Theorem 9). They also use Mysticeti's leader-based commit rule (propose at round r, vote at r+1, certify at r+2). Lemmas 22 and 24 establish (i) and (ii) for Lazy-Push variants after the synchronization phase. Property (iii) holds for Mysticeti-L by construction: per Section 6, Mysticeti-L leader blocks retain at most n hash references in `ancestors` with at least $2f+1$ from round $r-1$. Mysticeti-L preserves Mysticeti's commit rule, so the same I1/I2 calculations apply unchanged. □

Theorem 10 (Worst-case latency in the happy case (Starfish-L))**.** *Suppose all parties are honest and actual message latency is at most $\delta < \Delta$. After the synchronization phase of Lemma 22, the worst-case consensus latency of Starfish-L is at most 10δ.*

Proof. Let party p create block B at time t. Every party receives and verifies its payload by time $t + \delta$ and returns a partial signature on its payload commitment to p. Thus, p can aggregate an explicit DAC by time $t + 2\delta$. By Lemma 23, the next header H created by p publishes this DAC at some time $s \leq t + 4\delta$.

We now apply the header-ordering argument of Corollary 7, since Starfish-L uses the same compressed DAG and leader commit rule. Let r be the smallest round with $t_{first,r} >$

$s + \delta$. By Lemmas 22 and 24,

$$t_{last,r} \leq t_{last,r-1} + \delta \leq t_{first,r-1} + 2\delta \leq s + 3\delta.$$

All parties have received H and its causal history by time $s + \delta$. Therefore, the round-r leader block L contains H in its history, directly or through a self-chain. This history also contains B and its explicit DAC.

Every round-$(r + 2)$ header certifies L. These headers are received by every party by time $t_{last,r} + 3\delta \leq s + 6\delta$, so every party commits L by that time. All payloads in this and earlier committed leader histories were broadcast by the creation of L and are received by time $t_{last,r} + \delta$. Thus, payload recovery adds no delay to this bound, and every party sequences payload(B) by time $s + 6\delta \leq t + 10\delta$. □

D.8 Main Theorem

The latency analysis above, combined with the safety proofs in Appendix C, establishes the following main result.

Theorem 11. *Starfish, Mysticeti (with Push Pacemaker), Starfish-L, and Mysticeti-L are Byzantine Atomic Broadcast protocols in the partial synchrony model. Let M denote the average total payload per round and κ the security parameter (hash/signature size).*

Communication complexity per round:

- Communication complexity:
  - Mysticeti: $O(\kappa n^4 + Mn^2)$.
  - Mysticeti-L: $O(\kappa n^2 + Mn)$ *(happy case),* $O(\kappa n^3 + Mn^2)$ *(worst case).*
  - Starfish: $O(\kappa n^4 + Mn)$.
  - Starfish-L: $O(\kappa n^2 + Mn)$ *(happy case),* $O(\kappa n^3 \log n + Mn)$ *(worst case).*

In the worst case, for $\kappa = O(1)$, Starfish achieves $O(n)$ amortized communication complexity when $M = \Omega(n^3)$, and Starfish-L achieves $O(n)$ when $M = \Omega(n^2 \log n)$. In the happy case, Mysticeti-L and Starfish-L achieve $O(n)$ when $M = \Omega(n)$.

For the Lazy-Push variants (Mysticeti-L and Starfish-L), the all-honest latency bounds below hold after the synchronization phase of Lemma 22.

Worst-case consensus latency in the happy case:

- Mysticeti: $\leq 6\delta$.
- Mysticeti-L: $\leq 6\delta$.
- Starfish: $\leq 7\delta$.
- Starfish-L: $\leq 10\delta$.

Worst-case consensus latency (Byzantine parties):

- Starfish with 1 Byzantine party (requires $n \geq 6$): $\leq 12\delta + 4\Delta$.
- Starfish with f Byzantine parties: $\leq n\delta + 4f\Delta + O(\Delta)$.

Average consensus latency (all honest parties):

- Mysticeti: $\leq 5\frac{1}{3}\delta$.
- Mysticeti-L: $\leq 5\frac{1}{3}\delta$.

E Discussion of Table 1

The average consensus and end-to-end latencies are computed under the following assumptions:

- All parties are honest[13].
- All messages between distinct parties have transmission delay δ.
- Every uncompressed block and every compressed leader block of round r references all blocks from round $r - 1$.
- All parties are perfectly synchronized, i.e., they enter the round r at the same time.
- Individual transactions arrive at parties continuously at a uniform rate. They are buffered, and all buffered transactions are included in the next created block.

These assumptions intentionally describe an idealized happy-case execution; the latencies shown in Table 1 are the average latencies in this timing model. In deployed executions, several factors can increase latency: (1) blocks from the next round do not necessarily reference all blocks from the previous round, causing blocks with transactions to wait for the next leader and increasing synchronization delays; (2) parties enter rounds at different times due to clock skew and network asynchrony; (3) message delivery times vary within the bound δ rather than arriving uniformly.

For example, Table 1 gives an average consensus latency of 4δ for Mysticeti in this idealized model. Under the more general happy-case assumptions of Appendix D.6, we prove an upper bound of $5\frac{1}{3}\delta$ that is essentially tight. For Starfish, the idealized value is 5δ, while the worst-case consensus latency in the happy case is at most 7δ. We use idealized latencies in Table 1 because comparable general upper bounds are not available for all protocols.

All new protocols proposed in the paper deliver better tradeoffs in terms of latency and worst/happy-case communication costs in some regimes. For large enough $M = \Omega(\kappa n^3)$, Starfish achieves in the worst case linear amortized communication complexity with the same consensus latency of 5δ as certified DAG BFT Sailfish++. The latter, however, requires slightly larger $M = \Omega(\kappa n^3 \log n)$ and a larger overhead of 1δ for the end-to-end latency vs 0.5δ in uncertified DAGs, since transactions in certified DAGs wait for a longer time on average until being included as payload in blocks. Mysticeti-L and Starfish-L achieve quadratic $O(\kappa n^2)$ happy-case metadata communication costs, while all other DAG-based partially synchronous BFTs require at least $O(\kappa n^3)$. Starfish-L has linear payload costs of $O(Mn)$ for both happy and worst-case at the cost of higher latency 6δ, while Mysticeti-L has $O(Mn)$ payload costs in the happy case with (lowest for DAGs) latency 4δ and $O(Mn^2)$ payload costs in the worst case. The "One failure leader" column shows Starfish and Mysticeti's resilience: due to the Push pacemaker they have only $+2\Delta + \delta$ additional delay when a Byzantine party fails to create its

[13] For the "One failure leader" column, we consider one faulty party.

leader block during its turn, compared to $+4\Delta + 2\delta$ or higher in other DAG-based protocols.

Recall that we include in this comparison certified DAG-based protocols such as Bullshark [34], which uses Consistent Broadcast Primitive [9], and Sailfish++ [32] with its Reliable Broadcast Primitive based on Reed-Solomon coding (with erasure decoding and Merkleization for verifying data integrity) [7], and uncertified DAG-based protocols such as Mysticeti [2] and Starfish with our Push pacemaker, Starfish-L and Mysticeti-L with our Lazy-Push pacemaker.

We present results for one leader per round, though some table protocols support multiple leaders. For example, Mysticeti can support n leaders per round, which could improve latency under the above assumptions. However, outside this idealized model, we found no quantitative evidence that multiple leaders improve latency in Byzantine scenarios. The referenced production Mysticeti release [14] uses one leader per round.

The faulty leader is assumed to be the leader in a round right after the round of a block with payload.

We start the analysis with Mysticeti with pipelined leaders and a single leader in each round. The block latency for a non-leader block is 4δ since it takes one round to be referenced by the next leader and 3 rounds to commit the leader. Therefore, the average consensus latency is $\frac{3\delta+4\delta(n-1)}{n} = 4\delta - \delta/n$. The overhead of 0.5δ is added to the end-to-end latency since a transaction is waiting on average this time to be included by the party. For one leader failure, we added $2\Delta + \delta$ because we apply our Push pacemaker in which the condition **L2** includes the skipped pattern. Specifically, let a block B be from round $r-1$ and created at time t and the faulty leader be from round r. Blocks of rounds r will be created at time $t + \delta$, while blocks of round $r + 1$ will wait for a leader timeout $\delta_{LT} = 2\Delta$ (due to the absence of the leader block at round r) and will be created at time $t + 2\delta + 2\Delta$. The following blocks, voting and certifying the leader of round $r+1$ will be created without delay, thanks to the skipped pattern. This leads to the overhead of $2\Delta + \delta$ due to the additional round and the waiting time of the faulty leader. Due to the Push pacemaker, every party will send to every other party all other blocks leading to the communication complexity of $O(\kappa n^4 + Mn^2)$ in both happy and worst cases. For the Mysticeti-L, the same idealized calculation applies except that the Lazy-Push pacemaker uses the higher leader timeout $\delta_{LT} = 8\Delta$, so the one-failure overhead becomes $8\Delta + \delta$; the communication-complexity discussion remains unchanged. In the happy case, no party will send any other blocks to other parties (except own), so the total communication complexity is just $O(\kappa n^2 + Mn)$ because non-leader metadata is constant-size. In the worst case of f Byzantine leaders, the parties could be forced to forward blocks of others, leading to n times larger communication complexity.

[14] https://github.com/MystenLabs/sui/releases, release: MAINNET-V1.65.2

We now analyze Starfish latency. Unlike Mysticeti, a DAC must be formed for payload of block B in a committed leader block before sequencing B's payload. This requires one more round, resulting in 5δ consensus latency and 5.5δ end-to-end latency. The leader faulty case can be analyzed similarly due to the same Push pacemaker, leading to the additional overhead of $2\Delta + \delta$. Encoded fragment dissemination costs $O(Mn + \kappa n^3 \log n)$, including Merkle proofs. For Starfish-L, we delay the unknown history broadcast and use pulling for collecting encoded fragments, leading to the improved communication complexity of $O(\kappa n^2 + Mn)$ in the happy case.

The values for Bullshark and Sailfish++ are taken from the papers [31, 32, 36], where a similar analysis under the same assumption was performed and leader and non-leader block latencies are computed. For Bullshark the leader block is committed in time 4δ in the happy case if one uses CRB from [9], which allows parties to advance rounds at pace 2δ. Since leaders are assigned every two rounds, non-leader blocks have additional latency of 2δ and 4δ, making an average consensus latency of 7δ. The end-to-end latency overhead is δ—the average time a transaction waits until it is included in a block. A faulty leader could increase the latency by $4\Delta + 2\delta$. In the happy case, parties do not need to forward blocks of others, leading to the complexity of $O(\kappa n^3 + Mn)$, while in the worst case this can be increased by a factor of at least f, leading to $O(\kappa n^4 + Mn^2)$. Sailfish++ has a faster commit rule with more frequent leader blocks, and its optimistic RBC lets parties advance rounds every 2δ, giving an average consensus latency of 5δ. We omit its predecessor Sailfish [31] from Table 1, since Sailfish++ improves on it in every column but communication: the slower RBC of Sailfish advances rounds every 4δ for an average consensus latency of 9δ, at $O(\kappa n^3 + Mn)$ against the $O(\kappa n^3 \log n + Mn)$ of Sailfish++.

F Extended Related Work

This section expands Section 7, which states only what separates Starfish from each line of prior work; the paragraphs below follow the same order and record how the predecessors themselves differ.

Uncertified DAG Details. All uncertified DAG-based protocols inherit their mechanism for resolving equivocations from Hashgraph [3]: parties always have to reference their own last blocks, and they sequence only blocks certified by $2f + 1$ parties. Here, a block B *certifies* another block L in its causal past if B can reach blocks from $2f + 1$ parties such that each block V from these $2f + 1$ *votes* for L, i.e., L is reachable from V and V cannot reach any equivocation of the party that created L. Beyond the single-leader pipelining described in Section 7, Mysticeti [2] also proposes using multiple leaders in every round; this version does not guarantee any latency improvement in the case of Byzantine parties

and may instead lead to a larger latency due to the undecided state of Byzantine leader blocks.

For payload dissemination, Cordial Miners proposed the "cordial dissemination" where parties "push" possibly unknown history of the DAG to other parties. This approach leads to $O(\kappa n^4 + Mn^2)$ communication complexity. In this approach each party becomes responsible for dissemination of blocks in the causal history of its own blocks. Therefore, it requires using signatures for identification of which blocks were created by which authors, leading to a PKI requirement for running the protocol. We further propose an alternative Lazy-Push pacemaker which achieves $O(\kappa n^3 + Mn)$ communication complexity for Mysticeti/Cordial Miners in the happy case; by applying a compressed format to non-leader blocks, the communication cost is reduced to $O(\kappa n^2 + Mn)$, as in Mysticeti-L and Starfish-L.

Certified DAG Details. Bullshark [34] uses disjoint waves of two rounds and assigns one leader to each wave; by local interpretation of edges in the DAG, parties commit the leader block and sequence all blocks in its causal past. Shoal [33] reduces the latency of non-leader blocks by interleaving two instances of Bullshark. Sailfish [31] and Shoal++ [1] similarly optimize their commit rules for leader blocks, using first messages instead of certified blocks to count votes, and support multiple leaders in every round. Sailfish additionally proposes a no-vote certificate—an analogue of the skip pattern in the uncertified DAG approach—to skip more quickly the leader blocks of parties that fail to create them, improving latency in the case of such misbehavior. Shoal++, in turn, supports multiple instances of DAGs that improve end-to-end transaction latency. We note, however, that neither the multi-leader versions of these protocols nor the multi-DAG version of Shoal++ have proven performance improvements in the case of Byzantine nodes.

Hybrid DAG Details. In BBCA-Chain [25], the split between certified leader blocks and best-effort non-leader blocks carries a dissemination cost: when Byzantine parties "selectively" send their blocks only to the leader, additional latency can be incurred to download the missing blocks, and the leader becomes responsible for propagating a linear number of blocks.

Payload Dissemination Details. Certified DAGs outsource the data-availability problem to an additional RBC primitive, as blocks need to be certified before entering the consensus protocol. In addition, implementations of many certified DAG-based protocols use Narwhal [9], which separates transaction dissemination from consensus using quorum-based availability certificates. Blocks in certified DAGs, in this case, become lightweight since they contain only digests of certified transaction batches. However, this separate layer increases the resulting end-to-end latency due to additional network trips for forming certificates for transaction batches. To improve the latency, Raptr [35] introduces a hybrid approach that integrates transaction certification into the consensus, allowing nodes to vote on a prefix of (uncertified) transaction batches anchored by the proposed leader block. Autobahn [16] similarly separates payload dissemination by allowing each party to create its own transaction "lane", a chain of transaction batches, and other parties' votes can soon acknowledge the availability of a prefix of the lane.

Data Availability Details. Using a new block structure, Starfish decouples payload dissemination from the consensus task. In Starfish, the payload from blocks is included in the final ordering only after creating data availability certificates ($2f + 1$ acknowledgments) directly on the DAG. This transaction availability concept is different from Narwhal, Raptr, and Autobahn. Compared to Starfish, in Narwhal, parties vote on blocks that include already "well-disseminated" certified transaction batches, which increases the end-to-end latency. Both Autobahn and Raptr can operate with uncertified transaction batches. To sequence transactions, they require Proofs-of-Availability consisting of $f + 1$ acknowledgments, as this guarantees that at least one honest party has payload locally available. Starfish requires $2f + 1$ acknowledgments primarily because Reed-Solomon coding is used: f Byzantine parties could lie, and $k = f + 1$ encoded fragments are required for reconstruction (making k smaller increases code redundancy and bandwidth requirements). PoAs in Starfish rely on the block's signatures. Autobahn leaves the task of constructing PoAs to the data dissemination layer. Both Starfish and Raptr integrate constructing PoAs into the consensus layer, but Starfish allows more granular transaction commitment.

Communication Complexity and Coded Broadcasts Details. The use of erasure and error-correcting Reed-Solomon codes in constructing Reliable Broadcast primitives is well established [7, 10, 32]. Notably, Honeybadger BFT [26] applies erasure coding [7] to Bracha's RBC [6], reducing its communication complexity to achieve linear amortized communication complexity. Similarly, DAG-Rider [20] and Dumbo-NG [14] leverage erasure codes to optimize RBC communication costs. At a high level, these primitives disperse a payload of size T into n coded fragments and reconstruct it from any $k = \Theta(n)$ valid fragments; with FFT-based Reed–Solomon implementations, erasure decoding can be done in $O(T \log n)$ time [23], while fragment integrity is typically ensured via Merkle proofs of size $O(\kappa \log n)$. Sailfish++ employs an RBC for block dissemination with erasure RS decoding and Merkle proofs. Certified DAG-based BFT protocols may also rely on the RBC from [10], which does not use Merkle proofs for verifying data integrity and relies on error-decoding algorithms of Reed-Solomon codes. This reduces Merkle-proof overhead at the cost of higher computational complexity. However, the currently known error-correcting decoding (with unknown corrupted positions) algorithms incur higher computational

overhead than erasure decoding, e.g. $O(T \log^2 n)$ [13]. Moreover, RBC from [10] may attempt reconstruction f times in the worst case, leading to total $O(Tn \log^2 n)$ computational overhead.

In summary, certified DAG-based BFT protocols treat erasure or error coding as a separate optimization within the RBC protocol and do not integrate it directly into the DAG structure as we do in Starfish. Finally, we note that one can reduce the proof overhead in payload dissemination by replacing Merkle proofs with constant-size KZG opening proofs [19] (at the cost of assuming an SRS and the discrete logarithm (DL) and t-Strong Diffie–Hellman (t-SDH) assumptions underlying KZG).

G Mainnet Validator Geography

The located validator set has a geographically structured latency profile (Figure S4). The raw measurements are directional: for each peer-to-host stream, they record how long consensus data from one validator takes to be processed by the receiving validator. For the geographic view we aggregate each unordered operator pair into one edge by averaging the two directional means over the 11h50m Mysticeti window. We show only the Mysticeti regime because the geographic deployment and transit paths are not protocol-specific, while the metric itself still includes receiver-side processing and protocol-dependent message sizes. Absolute pairwise stream-processing latencies can therefore shift slightly under Starfish, but the figure still characterizes the mainnet deployment by showing which validator pairs are close enough to form the low-latency bulk and which pairs populate the long-latency tail. Using validator coordinates to identify regional buckets, intra-EU and intra-US pairs form the low-latency bulk, with p50 pairwise stream latency of 39 ms; trans-Atlantic links reach p90 of 134 ms, while a thin trans-Pacific tail extends to p99 of 371 ms. This context motivates reading the transition through the distribution of operational metrics, not only through medians: high-latency validator pairs are the setting in which missing-history recovery, request amplification, and lagging proposal acceptance matter most.

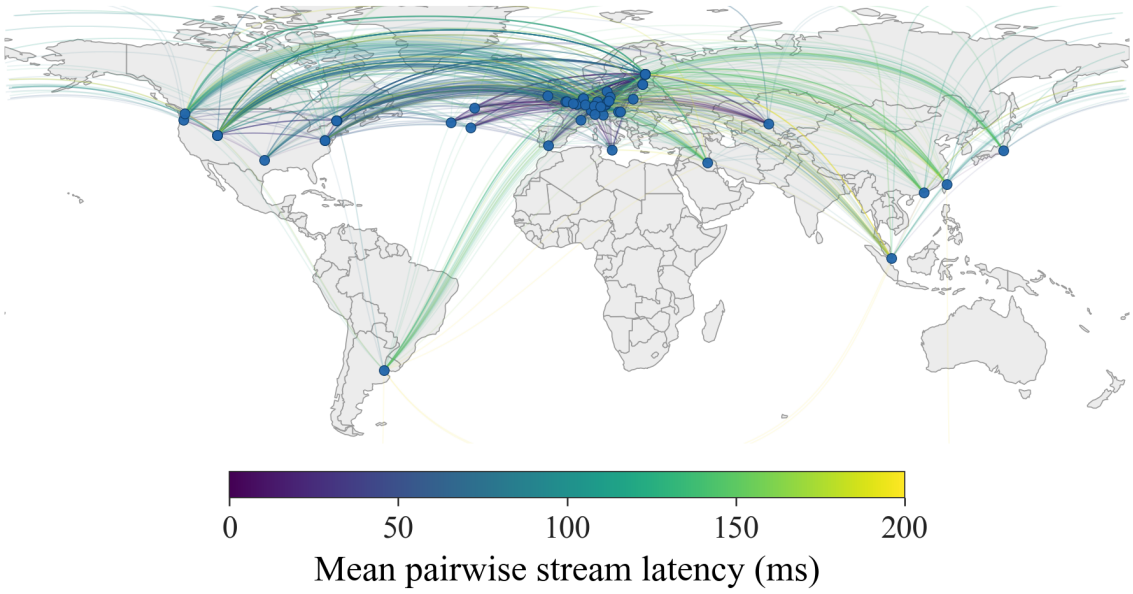


Figure S4. Geographic latency structure of the production mainnet validator set before the protocol switch. The figure overlays symmetrized inter-validator stream-processing latency on the locations of active validators that publish geographic coordinates. It describes the deployed mainnet network rather than pure network RTT or a Mysticeti–Starfish protocol comparison. The color scale is capped at 200 ms, so higher-latency edges are saturated yellow.

H Mainnet Study: Methodology and Per-Validator Results

This appendix records what the numbers in Section 8.5 measure.

Windows and pairing. The primary comparison uses three days before and after 2026-04-23 07:45 UTC, omitting the ten minutes closest to this reference time on each side. The telemetry records the protocol change at 07:42 UTC. We retain the fixed cohort of validators paired in the immediate twelve-hour comparison for progress and request measurements. Fewer validators reported requests immediately after the switch; reporting recovered in the next epoch, on April 24 at 07:43 UTC. A fixed cohort keeps these reporting changes from changing which validators contribute to the paired comparison. Latency uses validators with sufficient observations in both original windows. Each metric requires at least 120 observed 30-second samples on each side; missing observations are never interpolated. The well-connected group has pre-switch three-day median consensus latency below 300 ms. The five lowest-rate proposers are selected separately using their pre-switch three-day block creation rates.

Metrics and coverage. The exports evaluate per-validator Prometheus series every 30 seconds. Five-minute counter rates measure consensus requests, accepted own blocks or headers, and suspensions per 100 accepted remote blocks or headers. Round advancement is the five-minute slope of the threshold-clock round gauge; windows crossing backward jumps, epoch changes, or observed restarts are excluded. Request coverage is 99.4% before and 100.0% after the switch. Round-slope coverage is 99.2% and 99.7%, respectively. The latency histogram series have 58.6% and 66.6% coverage over the fixed latency cohort.

Bandwidth and load normalization. Bandwidth sums request and response byte-counter rates. Total bandwidth includes both directions; outbound traffic comprises requests sent and responses served. Bandwidth efficiency uses only outbound traffic. We use ten-minute rates and retain a validator's sum only when every contributing counter series has an observation in both the first and last minute of the window. This avoids understated rates from windows whose retained samples cover only a short interval. After this check and the minimum-sample requirement, bandwidth efficiency uses paired validators, with outbound coverage of 34.2% before and 33.8% after the switch. Consensus output is the two-minute rate of serialized transaction bytes delivered to the consensus handler, including protocol messages and entries later removed as duplicates. Its coverage is 51.1% before and 58.3% after the switch for these validators. Bandwidth efficiency divides each validator's window-median outbound byte rate by its window-median consensus output byte rate; lower is better. The table reports the median paired change.

Medians and tails. Consensus latency measures elapsed time from block creation until its payload is sequenced. Typical latency is each validator's three-day median of five-minute histogram means, followed by the median across validators. For tails, we compute per-validator p99 values

from the histogram buckets over five-minute windows and summarize them by their three-day median. These histogram quantiles measure individual latency observations rather than percentiles of the rolling means. The histogram has 50 ms buckets between 150 and 500 ms. Table S3 reports 95% percentile bootstrap intervals for the median paired change, using 2,000 resamples of validators and a deterministic seed for each measure.

For temporal variability, we compute per-validator p50, p95, and p99 from one-hour histograms and take the median across a fixed set of validators with estimates in every hour. We report the minimum and maximum of these hourly medians, excluding hours overlapping the last Mysticeti epoch or the first Starfish epoch.

Window sensitivity. Table S4 compares the primary ±72 h window with ±12, ±24, and ±120 h windows, holding the cohorts and the pre-switch group assignments fixed. The progress improvements persist across these windows. Typical consensus latency among well-connected validators changes by about one percentage point across window widths. Latency among the five less-connected validators is more sensitive to the window because one validator has substantially higher latency in the later days; these observations remain in the primary comparison. The increase in bytes sent per sequenced byte is also larger over the wider windows.

Table S3. Three-day paired mainnet comparison with 95% bootstrap intervals. Suspensions are per 100 accepted remote blocks or headers; bandwidth efficiency is bytes sent per sequenced byte.

Metric	**Myst.**	**Starfish**	**Change (%)** [95% CI]
Consensus requests (req/s)	73.8	10.6	−85.5 [-86.0, -84.7]
Suspensions / 100 accepted	21.33	2.06	−89.2 [-91.6, -86.5]
Round advancement (rounds/s)	18.08	19.16	+5.95 [+5.95, +5.96]
Block creation rate (blocks/s)	17.8	19.1	+7.3 [+6.7, +7.7]
Consensus latency (ms)	222	258	+16.2 [+15.3, +16.7]
Consensus latency p99 (ms)	544	398	−26.5 [-26.8, -25.9]
Bandwidth efficiency	39.8	94.1	+132.7 [+112.2, +155.8]
Five lowest-rate proposers (blocks/s)	15.4	17.6	+17.5 [+11.8, +43.3]
Consensus latency: less-connected (ms)	353	362	+5.8 [+0.4, +168.6]
Consensus output (kB/s)	88.4	100.9	+14.2 [+14.2, +14.3]
Outbound bandwidth (MB/s)	3.5	9.5	+165.7 [+142.6, +192.0]
Total bandwidth (MB/s)	7.1	19.5	+177.0 [+163.3, +181.8]

Table S4. Median per-validator change in percent for four window widths around the switch. The primary comparison uses ±72 h.

Metric	±12 h	±24 h	±72 h	±120 h
Requests	−87.5	−87.7	−85.5	−85.9
Suspensions / 100 accepted	−90.7	−90.6	−89.2	−89.6
Round advancement	+4.9	+5.6	+6.0	+5.5
Block creation rate	+5.9	+6.7	+7.3	+6.5
Five lowest-rate proposers	+17.1	+15.9	+17.5	+16.8
Consensus latency (well-connected)	+17.2	+16.4	+16.3	+16.9
Consensus latency (less-connected)	+2.5	−1.0	+5.8	+6.4
Consensus latency p99 (all)	−24.3	−25.8	−26.5	−26.0
Total bandwidth	+119.9	+121.3	+177.0	+163.2
Outbound bandwidth	+112.6	+116.2	+165.7	+157.9
Consensus output	+8.4	+8.6	+14.2	+13.4
Bandwidth efficiency	+96.2	+99.1	+132.7	+127.4

I Additional Evaluation Details

Controlled testbed setup. For every controlled simulation in Sections 8.2 and 8.3 we use a single shared testbed configured through our consensus orchestrator (Section 8.1). Validators are deployed as Amazon EC2 c5d.4xlarge instances (16 vCPUs, 32 GiB RAM, up to 10 Gbps network, 400 GB local NVMe SSD) in a single AWS region (eu-west-1). To mimic a globally distributed network on top of this single-region deployment, we apply a synthetic wide-area latency overlay between every pair of validators based on a measured 10-region RTT matrix (Table S5); each validator is assigned round-robin to one of US East (USE1), US West (USW1), Canada Central (CAC1), EU West (EUW1), EU South (EUS2), EU North (EUN1), South America East (SAE1), Asia Pacific South (APS1), Asia Pacific Southeast (APSE2), and Asia Pacific Northeast (APNE1). Each entry of the RTT matrix is the average of 10 ping measurements between virtual machines in the corresponding regions, rounded up; the one-way delay applied between two validators equals half the RTT between their assigned virtual regions.

Each honest validator runs a local client that issues a continuous stream of 512-byte transactions; the aggregate cluster load is split equally across all validators. Each transaction consists of an 8-byte submission timestamp followed by 504 random bytes, for a total of 512 bytes, and is essentially incompressible by both the storage engine and network-layer compression. End-to-end latency is measured as the elapsed time from when a transaction arrives at the validator that includes it in a block until the transaction is sequenced and ready for execution by another validator; we report the median across validators of each validator's p50, and likewise p99 where stated. Throughput is defined as the number of sequenced transactions divided by the steady-state duration of the run. We use a leader timeout of 600 ms per round for all protocols; one leader is enabled per round throughout. Each run lasts at least 180 s of steady-state operation.

Table S5. Round-trip latencies between regions (milliseconds)

	USE1	USW1	CAC1	EUW1	EUS2	EUN1	SAE1	APS1	APSE2	APNE1
USE1	1	65	14	68	104	110	112	201	198	146
USW1	65	1	78	127	163	172	175	226	137	108
CAC1	14	78	1	67	106	103	122	189	196	142
EUW1	68	127	67	1	29	38	176	125	254	199
EUS2	104	163	106	29	1	50	215	143	281	238
EUN1	110	172	103	38	50	1	220	148	268	245
SAE1	112	175	122	176	215	220	1	299	309	254
APS1	201	226	189	125	143	148	299	1	150	140
APSE2	198	137	196	254	281	268	309	150	1	101
APNE1	146	108	142	199	238	245	254	140	101	1

Leader chain-bomb setup. The Q3 runs of Section 8.4 reuse the transaction generator, the metric definitions, the 10-region RTT matrix (Table S5), and the same fixed 600 ms leader timeout; unlike the fault-free sweeps, the timeout does fire here, in the rounds an attacker leads. The deployment differs: all validators run as tasks of a single process on one host rather than on separate EC2 instances, and transactions of 512 bytes are split evenly over all validators. The fault sweep of Figure 4 runs on a 192-core host at 10,000 tx/s for 600 s, three runs per cell with median results reported, with outbound links uncapped. The uplink sweep of Figure 5 uses the same harness on a 192-core host at 10,000 tx/s, where an emulated FIFO uplink serializes every message at the cap before applying its propagation delay and the downlink stays unlimited. The fault-sweep runs have no drain phase, so honest transactions still in flight when a run ends count as uncommitted; the uplink sweep adds a drain phase after the load stops. The emulator omits TCP dynamics.

Small-committee Sailfish++ latency. The $n = 10$ Sailfish++ latency spike in Section 8.2 comes from the optimistic-RBC fast path: each block must collect $\lceil 5n/6 \rceil$ echo messages, which at $n = 10$ is 9 of 10 peers (only one slack slot), so the round latency is dominated by the slowest cross-region peer in the committee. As n grows, regions are duplicated in the round-robin assignment, the $\lceil 5n/6 \rceil$ quorum no longer needs to include the most distant peer, and Sailfish++ converges back to its asymptotic profile.

Load-sweep latency, CPU, and bandwidth. In the unsaturated regime of Q2, latency stays close to each protocol's idealized δ-multiple from Table 1. Mysticeti and Mysticeti-BLS sit at $\approx 480{-}580$ ms across offered loads from 1,000 to 180,000 tx/s, consistent with the 4.5δ row. Starfish holds at $\approx 700{-}740$ ms (5.5δ), Starfish-BLS at $\approx 800{-}910$ ms (6.5δ, with additional partial-signature aggregation cost), and Sailfish++ at $\approx 780{-}840$ ms (6δ). All five curves remain flat as delivered throughput grows by more than two orders of magnitude.

CPU usage scales roughly linearly with sustained throughput within each protocol: Starfish moves from 1.5 to 4.5 vCPUs from 1 k to 260 k tx/s, Mysticeti from 0.3 to 2.3, and Sailfish++ from 1.0 to 3.5. The BLS variants run at a higher floor: Starfish-BLS reaches 10 vCPUs at 240 k tx/s and Mysticeti-BLS ≈ 7 vCPUs at the same throughput, reflecting the partial-signature aggregation cost identified in Q1. Bandwidth efficiency converges towards 1 as load grows for every protocol: at high load, payload bytes dominate the metadata overhead. Starfish, the least efficient at 1,000 tx/s, reaches ≈ 1.10 at 260 k tx/s. The Sailfish++ penalty from its 3-phase RBC remains visible across the unsaturated range but is bounded.

Saturation behavior in Q2. Mysticeti hits its throughput ceiling earliest. Its uncompressed ancestor lists incur more metadata per block than the BLS variants at $n = 100$ (Figure 3), while Pull can add recovery overhead: as load approaches saturation, blocks may arrive faster than validators can fetch missing parents, and each missing-parent request consumes a full round trip of bandwidth and CPU. This suggests that request amplification and resource pressure contribute to Mysticeti's lower throughput ceiling. Starfish's Push dissemination can deliver missing history without waiting for an explicit request. The BLS variants use a constant number of ancestors per non-leader block, sharply reducing the metadata that must be pulled. Sailfish++ creates blocks at a lower rate because it is a certified DAG protocol requiring at least two communication rounds to certify each block, which naturally limits the rate of missing-parent requests. Beyond the throughput ceiling, delivered throughput drops and latency rises sharply within a single offered-load step (e.g., Starfish 257 k tx/s @ 820 ms $\rightarrow$ 200 k tx/s @ ≈ 10 s; Sailfish++ 239 k tx/s @ 869 ms $\rightarrow \approx 43$ k tx/s @ >5 s).

I.1 Replaying the Liveness Counterexample

We replay 300 rounds of the Mysticeti counterexample of [27, §3.2, Fig. 5] in deterministic virtual time, with seven honest and three Byzantine equal-stake parties (quorum seven). The original round-jumping rule [27, Alg. 3] and our Push pacemaker use the same delivery rule and broadcast full local history on block creation; Push also broadcasts on round

Table S6. Counterexample replay over 300 rounds. Own-chain gaps count missing own blocks across all seven honest parties; commits are the minimum number of committed leaders per honest party.

Pacemaker	Own-chain gaps	Min. commits
Original round-jumping	387	0
Push pacemaker	0	292

entry. Controllers and transport are modeled; signed, empty-payload Mysticeti blocks pass through our actual verifier, causal block manager, and leader commit logic.

At round r, five rotating honest parties $S(r)$ receive previous-round supporters. The sets include each honest round leader and satisfy $|S(r) \cup S(r+1)| \leq 6$. Byzantine parties create supporting and non-supporting blocks, neither forming a certificate. New members of $S(r+1)$ then receive a round-r quorum containing Byzantine non-supporters: the original rule skips intervening rounds, whereas Push fills them. Starting from common genesis, the two delivery phases occur at ticks $10r$ and $10r+1$, with $\Delta = 100$ ticks and leader timeout 200 ticks. Both delivered and pending honest broadcasts, including forwarded Byzantine blocks, remain within Δ throughout; no timer expires.

Maximum observed honest-message delay is 51 ticks in both runs. The original rule produces at most six certificate authors per leader and commits none; Push commits at least 292 leaders per honest party with no own-chain gaps (Table S6). Honest commit logs are consistent, contiguous prefixes from round 1, and a normal-delivery control produces identical DAGs under both rules. Some final-round messages remain in flight. Disabling C3 leaves this replay's DAGs unchanged; C3 supplies the immediate catch-up used in Lemma 6. This finite replay does not measure production performance; general liveness is proved in Appendix D.

To isolate C3, a second schedule retains A2 and all other rules. Before GST, five honest parties advance while two receive no peer messages; Byzantine parties supply non-supporting blocks. After GST, all honest messages arrive within Δ, and timers expire normally. From identical states with maximum honest round 24 (respectively 46) at GST, disabling C3 yields zero commits over the next 20Δ (40Δ), while enabling C3 yields at least 18 (37) per honest party. Without C3, entry times for the first new round span 19Δ (37Δ); with C3, they coincide. Continuing these schedules eventually restores commits without C3. Proposition 4 gives the general construction: recovery delay can grow without a constant bound in Δ as the pre-GST backlog grows, without an infinite stall.

J Additional Implementation Details

Networking and serialization. Every node maintains a persistent TCP connection to every other node, forming a full mesh of $n(n-1)/2$ bidirectional links. We use tokio[15] for asynchronous networking with no RPC framework, and bincode[16] for length-prefixed message serialization. LZ4 compression is applied to network messages. Each transaction consists of an 8-byte submission timestamp followed by 504 random bytes, for a total of 512 bytes, making it effectively incompressible. Block dissemination is configurable per protocol: Mysticeti, Mysticeti-BLS, Sailfish++, and Starfish-BLS default to *Pull* (authors broadcast their own blocks; recipients request missing ancestors on demand); Starfish defaults to *Push*, in which a node proactively sends its own block plus the slice of its causal past that the recipient is known to need. Which slice that is comes from hints exchanged in prior communication; the hint mechanism reduces the communication cost of both erasure-coded fragments and redundant additional ancestors.

Dissemination cost and indirect routes. The per-round bounds of Section 8.1 set the capacity of each mode. At $n = 64$ on a 1 Gb/s uplink with 512-byte transactions, pushing full blocks caps payload throughput near 4k tx/s, whereas pushing one's own payload plus fragments raises the ceiling to roughly 60k tx/s; both are back-of-envelope bounds, not measurements. What Push buys in geo-distributed networks is a *triangle-inequality effect*: a block authored by validator A may reach validator C faster through an intermediate relay B (i.e., $d(A,B) + d(B,C) < d(A,C)$) than over the direct link. Under Pull, C may first learn of A's block from B's reference to it, but must then wait for A's own copy over the slower direct link or spend a request round trip to B; under Push, B forwards A's block together with its own, so C can use both on arrival.

Cryptography. Block headers are signed with Ed25519 (64-byte signatures) via ed25519-consensus[17] and digested with blake3[18] ($\kappa = 256$ bits). A block reference is a 38-byte triple of round number (4 bytes), author index (2 bytes), and digest (32 bytes). The two BLS variants—Mysticeti-BLS and Starfish-BLS—additionally rely on BLS12-381 via the `blst` library, with 48-byte aggregate signatures and 96-byte public keys.

Acknowledgment encoding. A Starfish block header carries data-availability acknowledgments alongside its ancestor references, and the two sets overlap: a party acknowledges payloads of blocks that it often also references. Both codebases compress that overlap with indices rather than storing a second list of up to n 38-byte references, so duplicated references are not transmitted twice; how much this saves depends on how far the two sets coincide. The testbed marks the suffix the acknowledgment list shares with the ancestor list by a single index and stores only the acknowledged references falling outside it, while production keeps

[15] https://tokio.rs
[16] https://docs.rs/bincode/
[17] https://docs.rs/ed25519-consensus/
[18] https://docs.rs/blake3/

a combined reference vector delimited by two boundary indices.

Erasure coding and Merkle commitments. Starfish and Starfish-BLS encode block payloads with a systematic Reed–Solomon erasure code via the reed-solomon-simd[19] crate, which works over $\mathbb{F}_{2^{16}}$ and uses FFT-based decoding ($O(T \log n)$ for a payload of size T); the crate exploits SIMD instructions when fragments exceed two bytes and falls back to scalar code otherwise. The block creator commits to the fragment set by computing a Merkle tree over the BLAKE3 hashes of the fragments using the rs_merkle[20] crate; the resulting Merkle root is embedded in the block header and each disseminated fragment carries a Merkle proof of length $O(\kappa \log n)$ that receivers verify against the Merkle root before accepting the fragment.

Payload persistence and recovery. In both codebases, a validator acknowledges a payload only after verifying it and successfully writing it to RocksDB[21]. The testbed also persists fragments; production caches them in memory and regenerates evicted fragments from the retained payloads on disk. Production retains full payloads for the current epoch. Recovery across epoch boundaries uses the blockchain's checkpoint state synchronization, external to Starfish, which any of the at least $f + 1$ honest validators in the epoch-advancing quorum can serve. Storage is provisioned for the bound set by the block-size limit and the epoch length; exhaustion counts as a fault.

The testbed's default storage backend persists block metadata, transaction payloads, and commit metadata in separate RocksDB column families, configured with a 2 GB write buffer, pipelined writes, LZ4 upper-level compression, and Zstd dictionary compression on the bottommost level. The storage backend is tuned for maximum write throughput on NVMe storage. A commit syncer fetches committed history for nodes that fall more than 500 rounds behind.

Ordering and concurrency. In both codebases, one core thread integrates verified blocks into the local DAG, controls block creation and round advancement, applies the commit rule, and sequences payloads. It creates its own block before advancing, persists it before broadcasting, and buffers received blocks until their ancestors are in the local DAG. Signature and payload verification and network I/O run concurrently on other tasks. In the testbed's Tokio runtime, each node runs one asynchronous task per peer connection plus dedicated tasks for the consensus core, the shard-reconstruction worker pool, the leader timeout, and missing-block recovery. Per-connection workers handle fragment encoding, hashing, Merkle-tree construction, and signature verification, keeping this work off the core thread.

[19] https://crates.io/crates/reed-solomon-simd
[20] https://docs.rs/rs_merkle/
[21] https://rocksdb.org/

AI assistance. OpenAI Codex and Anthropic Claude assisted with manuscript editing, technical consistency checks, LaTeX formatting, and preparation of the Q3 experiment script and the Q4 analysis scripts. The authors take responsibility for the final manuscript and its claims and results.