

Barnacle: Adaptive Multi-Leader Scheduling for DAG-Based Consensus

Zeno de Angeli¹, Alexandru Ianov Vitanov^{2*}, Philipp Jovanovic^{3,1}, Lefteris Kokoris-Kogias³, Alberto Sonnino^{3,1}, Pasindu Tennage⁴, and Igor Zablotchi³

¹ University College London

² Columbia University

³ Mysten Labs

⁴ Digital Asset

Abstract. In DAG-based consensus, all validators propose blocks concurrently, and designated leader blocks drive transaction commit. Having multiple leader slots per round cuts queuing latency, yet production deployments run a single leader because of head-of-line blocking: a slow leader stalls the pipeline for at least one leader timeout, and for several waves when its slot must wait for the fallback indirect decision rule. This risk grows with the leader count. We introduce **Barnacle**, an add-on that adapts the leader count at run time. Every interval, it measures on the agreed committed DAG the fraction of slots decided as commit by the direct rule, and drives the leader count with additive increase, multiplicative decrease. The measurement requires no extra messages and no cryptography, and is deterministic. **Barnacle** is generic over DAG protocols; we instantiate it on four protocols spanning the Byzantine ($3f + 1$, $5f + 1$), crash-only ($2c + 1$), and mixed ($5f + 3c + 1$) fault models, with proven safety and liveness. Results show **Barnacle** matches the best static leader count in every regime: in a healthy network its latency is 6–13% lower than a single leader’s, and under degradation it matches a single leader while remaining 35–56% below a static high count. We are currently collaborating with the Sui team to integrate **Barnacle** into the Sui blockchain.

1 Introduction

The demand for high-throughput, low-latency blockchains has driven a surge of research on DAG-based Byzantine fault-tolerant consensus [12,6,22,1,2,19], which now runs in several production networks [26,10]. In a DAG-based consensus protocol, every validator proposes a block in every round, referencing previous-round blocks to build a directed acyclic graph. Rather than relying on a single proposer per round to sequence transactions, the protocol deterministically designates certain blocks as *leaders*; committing a leader block commits all transactions in its causal history (the DAG vertices reachable by backwards references). For operators of these deployments, end-to-end latency is the primary

* Work done while at University College London.

metric to optimize. Recent protocols explore multi-leader configurations to cut latency further: Shoal [21], Shoal++ [1], Mysticeti [2], Sailfish [18], Sailfish++ [19], Blue Bottle [30], Orcaella [15], and Nemo-Nemo [14] all support electing several leaders per round.

The appeal of multi-leaders stems from the need to reduce queuing latency, the delay between a validator receiving a transaction and its inclusion in a leader block. In a DAG, transactions proposed in non-leader blocks commit only once a subsequent leader block references them. The effect is concrete. In Sui’s deployment of Mysticeti [2], validators propose a block roughly every 75 ms and the end-to-end latency is about 400 ms. A transaction that reaches the validator leading the next round waits, in expectation, half a block interval (about 35 ms) before it is included in a leader block; a transaction that reaches any non-leading validator additionally waits a full round (75 ms) for its block to be referenced by the next leader, a roughly 20% increase in end-to-end latency. With L leaders among n validators in a round, a transaction lands on a leader with probability L/n , so the expected queuing latency shrinks as L grows.

Despite this theoretical advantage, to our knowledge, production deployments currently run a single leader per round [25]. Even protocols that introduce the multi-leader feature remain cautious; the original Mysticeti paper, for instance, recommends deploying with only two leaders or a small constant [2].

The reason for this conservative operational choice is head-of-line blocking, which pits two key latency factors against each other: every additional leader slot cuts queuing latency, but also adds another leader that can stall the pipeline. The mechanics are as follows. Each round consists of one or more designated leader *slots* (positions for leader blocks). Commits proceed in slot order, and a slot is decided across a *wave* (a fixed number of consecutive rounds where references and votes accumulate). Validators advance a round after assembling a quorum of its blocks, but wait up to a fixed leader timeout for the round’s leader blocks before proposing without them. A slow leader therefore blocks at the head of the pipeline, with a cost that depends on how far its block traveled in time. In round-level blocking, the leader block reaches nobody in time; every validator waits out the shared leader timeout, and the slot is then decided as skip by the fast, optimistic *direct decision rule*. In commit-level blocking, the leader block reaches only part of the committee; neither the commit nor the skip quorum exists, so the slot stays undecided, and because leaders commit in slot order, every later slot waits until the fallback *indirect decision rule* resolves it, at least a wave later. Either way, the more slots a round has, the more rounds contain a slow leader and the more blocking occurs; and either way the slot is not decided as commit by the direct rule. Any static choice of leader count is wrong in some regime: a high count degrades performance during network instability, while a low count forgoes the latency benefit in healthy conditions (Figure 1).

To resolve this tension, we introduce **Barnacle**, a mechanism to dynamically adapt the number of leaders per round to observed network conditions. It re-evaluates the leader count every **Interval** rounds (typically a few seconds).

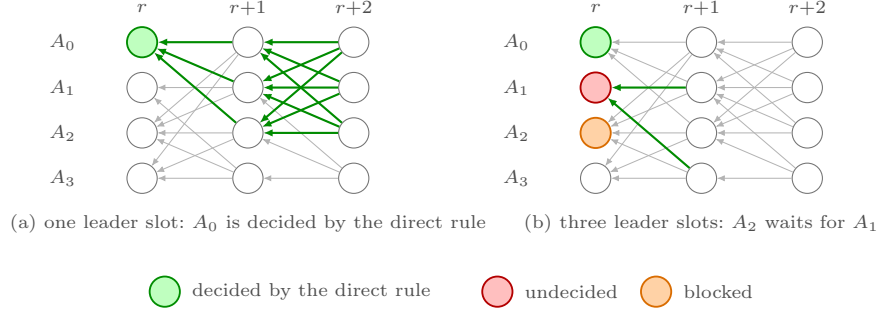


Fig. 1. Head-of-line blocking in a DAG. Each column is a round; each row is a validator; nodes are proposed blocks and arrows are references (votes). The same DAG ($n=4$; validator A_1 has poor network conditions so its round- r block promptly reaches only part of the committee: the commit-level case) is interpreted with one leader slot per round (a) and with three (b). In (a), leader A_0 's block gathers enough references (green) and is decided by the direct decision rule at the end of the wave ($r+2$). In (b), leader A_1 's block gathers too few references for the direct rule and will be decided later by the indirect decision rule; leader A_2 's block gathers enough references but, because leaders commit in round-then-slot order, it waits until A_1 's slot is decided.

Measuring network health in a BFT system is itself delicate: validators exchanging latency reports or voting on perceived health would add protocol messages and rest on unverifiable local observations, leaving honest validators without a common value to act on. The fundamental insight behind **Barnacle** is that a DAG-based protocol already exposes the information required to measure network health. In a DAG, every validator proposes a block in every round, so a committed window of the DAG contains every leader block and every vote of that period, and this window is agreed upon by all honest validators. DAG protocols decide each leader with two rules: a fast rule for ideal conditions (the direct decision rule) and a slower fallback that resolves a leader by recursing through later leaders (the indirect decision rule). The recently committed DAG therefore lets each validator count how many leaders of the recent past were decided under ideal conditions, i.e. by the direct rule, at no cost. **Barnacle** uses this count as an indicator of network health for the near future, on the assumption that conditions persist for the next few seconds. The measurement requires no extra messages and no cryptography; it is deterministic and computed on agreed state, so every honest validator obtains the same value.

We design **Barnacle** as a generic add-on over a minimal interface exposed by the DAG protocol. The control loop requires only the leader schedule, the committed prefix, and the base protocol's direct-commit predicate. Because this interface is abstract, **Barnacle** is agnostic to the fault model. We instantiate the interface on four base protocols: Mysticeti ($n = 3f + 1$, Byzantine), Blue Bottle ($n = 5f + 1$, Byzantine with a two-round commit), Nemo-Nemo ($n = 2c + 1$, crash-fault tolerant), and Orcaella ($n = 5f + 3c + 1$, mixed: f Byzantine plus c crashed). The mechanism never inspects quorum sizes. While we focus on uncertified DAGs, the same measurement applies to certified DAGs (Section 4).

Contributions. We make the following contributions:

- We introduce the first mechanism that adapts the leader count of a DAG-based consensus protocol at run time, namely deciding how many validators lead where prior dynamic schedulers [5,28,21] choose *who* leads, with no additional protocol messages or cryptographic tools (Sections 3 and 4).
- We prove that safety and liveness hold for any deterministic update rule, by hand (Section 5) and machine-checked in Lean 4 (Appendix C); the formulation is a generic add-on over an abstract base protocol under explicit assumptions.
- We show that one implementation runs unchanged across the $3f + 1$, $5f + 1$, crash ($2c + 1$), and mixed ($5f + 3c + 1$) fault models, instantiated on Mysticeti, Blue Bottle, Nemo-Nemo, and Orcaella (Section 6).
- We show that **Barnacle** matches the best static leader count in every regime (Section 7): in a healthy network its latency is 6–13% lower than with a single leader per round, and under degradation it matches a single leader while remaining 35–56% below a static high count.

Furthermore, we are collaborating with the Sui team to integrate **Barnacle** into the Sui blockchain.

2 Background and Model

This section fixes the model and the requirements **Barnacle** must meet (Section 2.1), then recalls the structure of DAG-based consensus protocols that **Barnacle** builds on (Section 2.2).

2.1 System model and problem

Fault model. We consider a system of n validators where a static, computationally bounded adversary controls a subset of them: up to f validators that deviate arbitrarily from the protocol (Byzantine) and up to c that may only halt (crashed). We consider three fault models; which one applies is fixed by the base protocol **Barnacle** runs on: the Byzantine model admits only Byzantine faults (Mysticeti [2] with $n = 3f + 1$, Blue Bottle [30] with $n = 5f + 1$); the crash model admits only crashes (Nemo-Nemo [14] with $n = 2c + 1$); and the mixed model admits both (Orcaella [15] with $n = 5f + 3c + 1$). In all models the remaining validators are honest and follow the protocol. **Barnacle** never inspects n , f , c , or which model applies; it inherits them from the base protocol.

Network. Validators communicate over a partially synchronous network [7]. Before a Global Stabilization Time (GST), the adversary controls message timing and delivery, though messages between honest validators are guaranteed to eventually arrive. After GST, the network becomes synchronous: any message sent at time x is guaranteed to arrive by time $\max\{x, \text{GST}\} + \Delta$, for a known bound Δ . In this model, after GST, a base protocol whose leader timeout is at least Δ decides every honest leader’s slot by the direct decision rule (detailed in Section 2.2); only slots led by crashed or Byzantine validators can remain undecided, at most $f+c$ per round. This is the model in which the base protocols

prove Safety and Liveness (Definition 1), and in which we prove that **Barnacle** preserves them.

Deployments. In practice, validators advance to the next round upon receiving a quorum of blocks, and wait for the round’s leader block(s) only for a short, fixed time after that quorum: 200 ms in Sui’s Mysticeti [25]. Waiting for a pessimistic Δ every round would result in high latency, so the timeout is tuned to the common case. An honest leader whose block arrives after the timeout, because of congestion or temporary network delays is referenced by too few blocks and its slot is not decided as commit by the direct rule (Section 2.2), as if the leader were faulty. Hence the symptom **Barnacle** acts on, a slot not decided as commit by the direct rule, arises from faulty leaders in the model but from temporary network degradations in deployments; **Barnacle** only ever observes the symptom.

Problem. **Barnacle** enhances a base protocol that solves Atomic Broadcast.

Definition 1 (Atomic Broadcast). *Atomic Broadcast satisfies the following properties:*

- **Agreement:** *If one honest validator commits a block, all honest validators eventually commit it.*
- **Integrity:** *Each honest validator commits a block at most once, and only if it was proposed by its author (a requirement that is trivial under crash faults).*
- **Validity:** *If an honest validator proposes a block, all honest validators eventually commit it.*
- **Total Order:** *If an honest validator commits a block b before block b' , no honest validator commits b' before b .*

Barnacle must meet two requirements. (R1) Correctness: the base protocol composed with **Barnacle** still satisfies every property of the definition. (R2) Performance, stated informally: (a) in healthy conditions **Barnacle** introduces no latency penalty over the best static leader count; (b) when slots stall, its latency tracks the best static leader count for the prevailing conditions, and it recovers when conditions do. R1 is established formally, by hand-written proofs (Section 5) and a Lean formalization (Appendix C); R2 is a performance goal outside the formal model and is validated empirically in Section 7.

2.2 DAG-based consensus

DAG-based protocols operate in rounds. Every validator proposes one block per round, referencing a quorum of previous-round blocks. Validators build a local directed acyclic graph (DAG) from delivered blocks; a block is added only once its whole causal history is present.

Leader slots and decision rules. Each round has one or more slots in a fixed order. The leader of slot i of round r is a deterministic, public function of (r, i) . The leader’s round- r block is the leader block of the slot. Every validator proposes every round regardless of how many slots there are; a non-leader’s block is an ordinary block.

Table 1. Base protocols and the parameters of their decision rules: wave length; commit and skip thresholds, the numbers of validators whose references let the direct rule decide a slot as commit or as skip; and link size, the number of certified paths the indirect rule requires.

Protocol	Fault model	Wave length	Commit	Skip	Links
Mysticeti [2]	$n=3f+1$, Byzantine	3	$n-f$	$n-f$	1
Blue Bottle [30]	$n=5f+1$, Byzantine	2	$n-f$	$n-f$	$n-3f$
Nemo-Nemo [14]	$n=2c+1$, crash	2	$n-c$	$n^\dagger$	1
Orcaella [15]	$n=5f+3c+1$, mixed	2	$n-f-c$	$n-f-c$	$n-3f-2c$

[†]Nemo-Nemo’s direct rule never decides a skip; a threshold of n can only trigger when every validator’s block is present, so in practice skips are decided by the indirect rule.

A wave is a fixed number of consecutive rounds, two or three in the protocols we consider (comprising propose, vote, and optionally certify rounds). The *direct decision rule* decides a slot from the references inside its wave alone: a slot is decided as commit if enough validators reference the leader block in the pattern the protocol prescribes, and as skip if enough validators fail to. Otherwise, it remains undecided, and the *indirect decision rule* later takes over. This rule first deterministically designates a future leader as *anchor*; then decides a slot as commit if the anchor’s causal history links to the leader block through enough certified paths, and as skip otherwise. Decided leaders are then ordered in round-then-slot order and their causal histories are linearized deterministically [12] into a committed prefix.

Certified and uncertified DAGs. We describe **Barnacle** over uncertified DAGs, which use best-effort broadcast without explicit certificates: references serve as votes, and a quorum of matching votes in the next round plays the role of a certificate. We instantiate **Barnacle** on four base protocols chosen to span fault models: Mysticeti [2] ($n = 3f + 1$, Byzantine, three-round waves), Blue Bottle [30] ($n = 5f + 1$, Byzantine, two-round waves), Nemo-Nemo [14] ($n = 2c + 1$, crash faults only, two-round waves), and Orcaella [15] ($n = 5f + 3c + 1$, mixed: f Byzantine plus c crashed, two-round waves). Table 1 summarizes the parameters of their decision rules. Nothing in **Barnacle** depends on these values: quorum sizes live exclusively inside the base protocol’s decision rules (Section 4). In certified DAGs [6,22,21], certificates are explicit objects and their direct rule counts references to the leader in the next round; the same measurement applies because every validator’s certified block still exists in every round, though this is not evaluated here.

3 Overview

Barnacle is a control loop on the leader count. The loop runs once every **Interval** rounds (e.g. 50, a few seconds). Its trigger is a *pivot*: the first leader a validator commits more than **Interval** rounds after the previous pivot. On committing a pivot, the validator takes the pivot’s causal history over the last **Interval** rounds as the window, counts the slots that the direct decision rule decided as commit, and divides by the number expected under the current leader count: the *direct decision rate*. If the rate is at least **threshold** (e.g. 0.96) the leader count

```

1: Interval, maxLeaders, threshold ▷ E.g., 50 rounds, 5 leaders, 0.96
2: leadersPerRound  $\leftarrow$  1, backoff  $\leftarrow$  0, lastRound  $\leftarrow$  0 ▷ State

3: procedure ONCOMMIT( $b_{leader}$ ) ▷ Called after committing  $b_{leader}$ 
4:    $r \leftarrow b_{leader}.round$ 
5:   if  $r \leq \text{lastRound} + \text{Interval}$  then return
6:    $W \leftarrow$  causal history of  $b_{leader}$  in rounds  $[r - \text{Interval}, r]$ 
7:    $observed \leftarrow$  slots of  $W$  decided as commit by the direct rule
8:    $expected \leftarrow (\text{Interval} - \text{waveLength} + 1) \times \text{leadersPerRound}$ 
9:   if  $observed \geq \text{threshold} \times expected$  then ▷ Healthy: additive increase
10:     $\text{leadersPerRound} \leftarrow \min(\text{leadersPerRound} + 1, \text{maxLeaders})$ 
11:    backoff  $\leftarrow$  0
12:   else ▷ Stalled slots: multiplicative decrease
13:     $\text{leadersPerRound} \leftarrow \max(\text{leadersPerRound} - 2^{\text{backoff}}, 1)$ 
14:    backoff  $\leftarrow$  backoff + 1
15:   lastRound  $\leftarrow$   $r$  ▷ The new count applies to rounds  $> r$ 

```

Algorithm 1: Barnacle, abstract form (runs at every party on top of the base protocol)

grows by one, up to **maxLeaders** (e.g. 5); otherwise it shrinks by 1, 2, 4, ... on consecutive unhealthy windows, never below one: additive increase, multiplicative decrease, as in TCP congestion control. The new count applies to the rounds after the pivot, with no transition phase. Algorithm 1 states the rule.

Why it works. Every validator proposes a block in every round, so the window contains every leader block and reference needed. Because the window is the pivot’s causal history, all honest validators that commit the pivot hold the same window, compute the same leader count, and agree on every configuration change (Section 5). The leader schedule is an interpretation of the DAG, not part of it, so the base protocol’s safety applies within each configuration and the only delicate point is the switch, which pinning changes to committed pivot rounds handles. The test is one integer comparison (e.g. $100 \times observed \geq 96 \times expected$), with no messages and no cryptography. This agreement argument nowhere uses the shape of the AIMD rule: safety and liveness hold for any deterministic update rule computed on the window (Section 5); AIMD is our choice, justified in Section 4.2.

Why the direct decision rate. A slot not decided as commit by the direct rule is exactly one that caused head-of-line blocking: either it timed out into a skip, or it stays undecided, waits at least a wave for the indirect decision rule, and holds every later slot with it. The direct decision rate therefore measures blocking in the window, whichever form it took. Measuring under the current leader count only keeps the rule to one comparison per interval, and the multiplicative decrease makes it react within a few intervals when the direct decision rate collapses.

Base protocol interface. Barnacle is defined against any DAG-based consensus protocol that provides the following:

- (A1) *Rounds and slots.* The execution is structured in sequential rounds $r = 1, 2, \dots$; in round r , every validator proposes one block; slot i of round r , for i below the current leader count, is led by a deterministic function of (r, i) and the current configuration.

- (A2) *Causal completeness*. A validator adds a newly received block to its local DAG only once the block’s entire causal history is available (this claim is formalized in Section 5).
- (A3) *Direct decision predicate*. There exists a predicate that, given a DAG, decides a slot (r, i) using only blocks of rounds $[r, r + \text{waveLength})$, representing the base protocol’s direct decision rule alongside the indirect decision rule’s certified-link test. All quorum sizes, and hence the fault model ($n = 3f+1$ or $5f+1$ under Byzantine faults, $2c+1$ under crash faults, $5f+3c+1$ under mixed faults), reside within this predicate; **Barnacle** never inspects them.
- (A4) *Base safety and liveness*. For any fixed configuration, honest validators commit the exact same sequence of leaders (safety), and after Global Stabilization Time, they commit new leaders infinitely often (liveness). For the leader rotation used in this paper, the liveness half is proved rather than assumed (Appendix C).

4 The Barnacle Protocol

Section 4.1 shows how **Barnacle** plugs into the base protocol so that no round ever mixes two leader counts (Algorithm 2) and how the base protocols instantiate the interface; Section 4.2 shows how the window is measured and the count updated so that the result is identical at every honest validator (Algorithm 3). The base protocol’s own decision rules (Algorithms 4 and 5) are in Appendix A.

4.1 Integration with the base protocol

A *configuration* is a leader count together with the round at which it takes effect; the configuration in force at round r is the one with the largest start round at most r , and $\ell(r)$ denotes its leader count (Algorithm 2). A validator maintains its current leader count in the variable `leadersPerRound`.

Decision and update procedures. The TRYDECIDE procedure (Algorithm 2) is the base protocol’s unchanged mechanism, except that the number of slots it evaluates per round is the leader count of the configuration in force. For each round and each slot, the validator instantiates a DECIDER (Algorithms 4 and 5, Appendix A) with the wave offset, the slot index, and the local DAG DAG, and applies the direct decision rule followed by the indirect decision rule.

The TRYCOMMIT procedure (Algorithm 2) walks the decision sequence produced by TRYDECIDE in order up to the first undecided slot, appending any committed leaders to the committed prefix. When it commits a leader whose round exceeds `lastRound + Interval`, this leader becomes the pivot where the next leader count is decided. The validator calls UPDATELEADERS (Algorithm 3) and returns immediately. The new configuration starts at the round after the pivot’s: on the next invocation, the remaining slots of the pivot’s round are decided under the old leader count, so every slot of rounds up to and including the pivot’s is decided under the configuration in force at its round, and no round ever mixes two counts. Triggering the update when the round is more than

```

1: waveLength ▷ Rounds per wave, Table 1
2: leadersPerRound ▷ Current leader count;  $\ell(r)$  is the count in force at round  $r$  (Section 4.1)
3: Interval ▷ Rounds between updates, e.g., 50
4: lastRound  $\leftarrow 0$  ▷ Round of the last update

5: procedure TRYCOMMIT( $s_{\text{committed}}, r_{\text{highest}}$ ) ▷  $s_{\text{committed}}$ : the last committed slot
6:    $S \leftarrow \text{TRYDECIDE}(s_{\text{committed}}, r_{\text{highest}})$ 
7:    $S_{\text{commit}} \leftarrow []$  ▷ Holds committed leaders
8:   for  $s \in S$  do
9:     if  $s = \perp$  then break
10:    if  $s = \text{Commit}(b_{\text{leader}})$  then
11:       $S_{\text{commit}} \leftarrow S_{\text{commit}} \parallel b_{\text{leader}}$ 
12:      if  $b_{\text{leader}}.\text{round} > \text{lastRound} + \text{Interval}$  then
13:         $\text{leadersPerRound} \leftarrow \text{UPDATELEADERS}(b_{\text{leader}})$  ▷ For the next round
14:        return  $S_{\text{commit}}$  ▷ The pivot round's remaining slots keep the old count
15:   return  $S_{\text{commit}}$ 

16: procedure TRYDECIDE( $s_{\text{committed}}, r_{\text{highest}}$ )
17:    $S \leftarrow []$  ▷ Holds decisions
18:   for  $r \leftarrow r_{\text{highest}}$  down to  $s_{\text{committed}}.\text{round}$  do
19:     for  $l \leftarrow \ell(r) - 1$  down to 0 do
20:       if  $(r, l) \leq (s_{\text{committed}}.\text{round}, s_{\text{committed}}.\text{index})$  then break
21:        $D \leftarrow \text{DECIDER}(r \bmod \text{waveLength}, l, \text{DAG})$  ▷ Instance of Algorithms 4 and 5
22:        $w \leftarrow D.\text{WAVENUMBER}(r)$ 
23:        $s \leftarrow D.\text{TRYDIRECTDECIDE}(w)$ 
24:       if  $s = \perp$  then  $s \leftarrow D.\text{TRYINDIRECTDECIDE}(w, S)$ 
25:        $S \leftarrow s \parallel S$ 
26:   return  $S$ 

```

Algorithm 2: Protocol Main Function

Interval rounds after the last update, rather than strictly at a multiple of **Interval**, makes the logic robust to the pivot round's slot being skipped.

Base protocols. The only protocol-specific code is the decider (Algorithm 5), which is parameterized by **waveLength**, **commitThreshold**, **skipThreshold**, and **linkSize** (Table 1, Appendix A). Mysticeti [2] handles Byzantine validators ($n = 3f + 1$) using three-round waves, thresholds of $n - f$, and one certified link. Blue Bottle [30] handles Byzantine validators ($n = 5f + 1$) using two-round waves, so the vote block is itself the certificate; it uses thresholds of $n - f = 4f + 1$, and **linkSize** = $n - 3f = 2f + 1$ certified links for the indirect decision rule, which is what keeps a commit by the indirect rule consistent with a skip by the direct rule once the vote is the certificate. Nemo-Nemo [14] handles crashed validators ($n = 2c + 1$) with two-round waves, a commit threshold of $c + 1$, a skip threshold of n (so the direct rule never decides a skip in practice), and one link. Orcaella [15] handles f Byzantine plus c crashed validators ($n = 5f + 3c + 1$) with two-round waves, so the vote block is again the certificate; it uses thresholds of $n - f - c = 4f + 2c + 1$ and **linkSize** = $n - 3f - 2c = 2f + c + 1$ certified links, generalizing Blue Bottle's consistency argument to the mixed fault model,

```

1: Interval, lastRound ▷ Defined in Algorithm 2
2: maxLeaders ▷ Upper bound on the leader count, e.g., 5
3: threshold ▷ Direct decision rate considered healthy, e.g., 0.96
4: backoff  $\leftarrow 0$  ▷ Consecutive decreases

5: procedure UPDATELEADERS( $b_{leader}$ )
6:    $W \leftarrow \text{GETSUBDAG}(b_{leader})$ 
7:    $observed \leftarrow \text{COUNTDIRECTCOMMITTS}(W, \text{leadersPerRound})$ 
8:    $expected \leftarrow \text{EXPECTEDCOMMITTS}(\text{leadersPerRound})$ 
9:   if  $observed \geq \text{threshold} \times expected$  then ▷ Healthy: additive increase
10:     $\text{leadersPerRound} \leftarrow \min(\text{leadersPerRound} + 1, \text{maxLeaders})$ 
11:     $\text{backoff} \leftarrow 0$ 
12:   else ▷ Stalled slots: multiplicative decrease
13:     $\text{leadersPerRound} \leftarrow \max(\text{leadersPerRound} - 2^{\text{backoff}}, 1)$ 
14:     $\text{backoff} \leftarrow \text{backoff} + 1$ 
15:    $\text{lastRound} \leftarrow b_{leader}.\text{round}$ 
16:   return  $\text{leadersPerRound}$ 

17: procedure GETSUBDAG( $b_{leader}$ )
18:    $r \leftarrow b_{leader}.\text{round}$ 
19:   return  $\{b \in \text{DAG}[r'] : r - \text{Interval} \leq r' \leq r \wedge \text{ISLINK}(b, b_{leader})\}$ 

20: procedure EXPECTEDCOMMITTS( $k$ ) ▷ Decidable slots of the window
21:   return  $(\text{Interval} - \text{waveLength} + 1) \times k$ 

22: procedure COUNTDIRECTCOMMITTS( $W, k$ ) ▷ Slots  $\in W$  decided by direct rule,  $k$  leaders
23:    $count \leftarrow 0$ 
24:   for  $r \in \{b.\text{round} : b \in W\}$  do
25:     for  $l \leftarrow 0$  to  $k - 1$  do
26:        $D \leftarrow \text{DECIDER}(r \bmod \text{waveLength}, l, W)$  ▷ Algorithms 4 and 5 reading  $W$ 
27:        $w \leftarrow D.\text{WAVELENGTH}(r)$ 
28:       for  $b \in D.\text{LEADERBLOCK}(w)$  do ▷ Leader block(s) of the slot
29:         if  $D.\text{SUPPORTEDLEADER}(w, b)$  then  $count \leftarrow count + 1$ 
30:   return  $count$ 

```

Algorithm 3: Update Leader Algorithm

and setting $c = 0$ recovers Blue Bottle exactly. Barnacle’s code (Algorithms 2 and 3) is identical in all four.

In certified DAGs, such as Narwhal and Bullshark [6,22], or Shoal [21], certificates are explicit objects and the direct decision rule counts references to the leader in the next round. Every validator’s certified block still exists in every round, so the window measurement applies unchanged; we do not evaluate this.

4.2 Updating the leader count

Upon committing a pivot, UPDATELEADERS (Algorithm 3) proceeds in three steps: it extracts the window from the pivot’s causal history, measures how many

of the window’s slots the base protocol decided as commit by the direct rule, and updates the leader count from the resulting direct decision rate.

Step 1: the window. The procedure GETSUBDAG (Algorithm 3) defines the window W as the pivot’s causal history restricted to the last `Interval` rounds. Because it is a function of the pivot only, the window is identical at every honest validator that commits it. It contains every block of those rounds that the pivot references; in particular, every leader block and every reference the direct decision rule reads, because every validator proposes a block in every round.

Step 2: the measurement. For each round of W and each slot below the current leader count, COUNTDIRECTCOMMITTS (Algorithm 3) instantiates a DECIDER whose DAG is W rather than the local DAG DAG. It takes the slot’s leader block, or blocks, via LEADERBLOCK (Algorithm 4), and counts those for which SUPPORTEDLEADER (Algorithm 5) holds, i.e., the slot is decided as commit by the direct rule within W . Evaluating on W rather than the local DAG is what keeps the count identical across validators. The EXPECTEDCOMMITTS procedure (Algorithm 3) computes the slots of the window that can be decided within it as $(\text{Interval} - \text{waveLength} + 1)$ rounds times the leader count. The window spans $\text{Interval} + 1$ rounds, of which the last `waveLength` are excluded rather than counted as failures: the `waveLength` – 1 rounds below the pivot’s have their certify round above the pivot, outside W , and the pivot’s own round contributes a single block to W (the pivot itself, since no other block of that round can be its ancestor), so its slots can never reach the commit threshold within W .

Step 3: the update. The UPDATELEADERS procedure calculates the direct decision rate as the ratio of observed to expected decisions. If this rate is at least `threshold`, the leader count increases by one, capped at `maxLeaders`, and `backoff` is reset to 0. Otherwise, the leader count decreases by 2^{backoff} , floored at 1, and `backoff` is incremented. Additive increase probes for more parallelism cautiously, one leader per interval, so a wrong step costs at most one interval of one stalled slot. Multiplicative decrease sheds parallelism within a few intervals when slots stall, because a single stalled slot already blocks every slot behind it. The cost of the one-interval lag before the decrease is evaluated in Section 7. In the implementation, the threshold test is executed as an integer comparison.

5 Correctness

This section establishes that Barnacle preserves the correctness of the base protocol, relying solely on the interface A1–A4 from Section 3; nothing below is specific to a base protocol or to the AIMD rule. We provide the statements and proof sketches; Appendix B contains the full proofs, and every statement is machine-checked in Lean (Appendix C). We open source our Lean formalization.⁵

Notation. A *configuration* C is a pair consisting of a start round and a leader count $\ell(C)$; the configuration in force at round r is the one with the largest start

⁵ <https://github.com/gdanezis/lean-dag> (commit 24d727f)

round at most r (Section 4.1). Every validator starts in $C_0 = (0, 1)$. A validator *closes* C_k when it commits the pivot of C_k , which is the first committed slot whose round exceeds the start round of C_k by more than **Interval**. The pivot's round becomes the start round of C_{k+1} . Because a validator's DAG is finite at any given time, its run closes only finitely many configurations. Accordingly, every statement below compares validators based on the configurations that *both* have reached, and liveness asserts that runs reach every height, never that a completed infinite sequence exists. After Global Stabilization Time (GST), messages between honest validators arrive within a known bound Δ .

Fact 1. *Let b be a block held by an honest validator. (i) The validator's DAG contains the entire causal history of b . (ii) Every honest validator eventually holds b . (iii) Any two honest validators that hold b hold the same causal history of b .*

Proof sketch. (i) is A2; (ii) is the delivery guarantee of the broadcast layer; (iii) follows by induction on the causal history, using (i) and (ii). $\square$

Lemma 1. *Any two honest validators that commit the same pivot compute the same window.*

Proof sketch. The window is a function of the pivot's causal history and **Interval** alone, and the histories coincide by Fact 1(iii); GETSUBDAG is deterministic. $\square$

Proposition 1. *For any deterministic update rule, such as the AIMD rule of Algorithm 3, any two honest validators agree on every configuration both have reached, its start round and its leader count, and on the verdict of every slot of every configuration both have closed.*

Proof sketch. Induction over configurations. Both validators start in C_0 . Within C_k , every verdict up to and including the pivot's round is a derivation against the one fixed schedule of C_k (Section 4.1), so the two validators' verdicts — in particular the pivot itself — are equal by the safety half of A4. The next configuration is one deterministic function of one agreed pivot and its agreed window (Lemma 1), so C_{k+1} is equal at both. A validator that has not reached the pivot has closed a strict prefix and agrees on it, and closes the gap once its view converges (Fact 1(ii)). No synchrony assumption and no base-protocol fact beyond A4-safety is used. $\square$

Theorem 1. *Barnacle composed with any base protocol satisfying A1–A4 satisfies the safety of Definition 1: the committed sequences of any two honest validators agree as far as both reach and grow only by appending (Agreement, Total Order), and no block is committed twice (Integrity).*

Proof sketch. A validator's committed sequence is determined by its verdicts, which agree configuration by configuration (Proposition 1); hence the sequences agree on the common range and extend by prefixes. For Integrity, the commit

boundary is a slot, and a commit verdict names a block of its slot’s round and leader, so a block occupies exactly one slot. $\square$

Remark 1. With the constant update rule, **Barnacle** is the base protocol: the same schedule, the same verdicts, the same ledger. The AIMD rule additionally keeps the leader count within $[1, \text{maxLeaders}]$, and its threshold test is one integer comparison.

Lemma 2. *After GST, if every honest validator has closed configuration C_k , then some honest validator closes C_{k+1} within bounded time, and every honest validator does within a further Δ .*

Proof sketch. The liveness half of A4 supplies committed slots past the threshold round of C_{k+1} ’s pivot; the least such slot is the pivot. The lemma consumes base *safety* as well: the commit that liveness supplies and the verdict that the schedule update reads are identified by agreement. Delivery within Δ then propagates the pivot’s history to every honest validator (Fact 1(ii)), and determinism closes C_{k+1} everywhere. $\square$

Theorem 2. *After GST, every honest validator’s run reaches every height: for every k it closes configuration C_k , and committed slots keep appearing.*

Proof sketch. Within a configuration, by the per-configuration clause of A4-liveness; across configurations, by Lemma 2 applied repeatedly. The leader count may oscillate, but configuration start rounds strictly increase. In the finite-horizon form of the statement (Appendix C), every slot lying at least a wave plus the commit gap below the horizon is decided; the margin is necessary, not an artifact: no protocol can decide a slot at the last round under the horizon, since no decision anchor’s wave fits above it. $\square$

Remark 2. For the leader schedule used in this paper, the per-configuration liveness clause of A4 is proved rather than assumed, for all four base protocols at every leader count (Appendix C). Because the classic argument — a wave of consecutive honest-led slots — has no instance once several leaders share a round, a new argument replaces it: a descent through the first slot of each round and a pigeonhole on the rotation, yielding a commit gap of $n + \text{waveLength} - 1$ rounds.

What Barnacle does not guarantee. The chosen leader count is a heuristic, not proved optimal. Byzantine validators can bias one window’s measurement and make the count suboptimal for one interval, but the count stays within $[1, \text{maxLeaders}]$ and never falls below the single-leader baseline (Section 7).

6 Implementation

Barnacle is a scheduler module of ~ 270 LOC that adds no messages, no cryptography, and no persistent state beyond the current configuration, and runs unchanged on all four base protocols. We implement it in Rust by forking the Mysticeti codebase [20], inheriting its networking (`tokio` [27] over TCP), signatures (`ed25519-consensus` [29]), and write-ahead-log crash recovery. Our simulator replaces the networking stack behind the same interface: a discrete-event controller emulates the `tokio` runtime and the TCP links, so every other component, consensus logic included, runs unmodified in simulation and in deployment. We are open-sourcing our implementation, along with the simulator and scripts⁶.

The scheduler module implements Algorithm 3: it extracts the window, counts the slots decided as commit by the direct rule within the window using the base protocol’s own wave length and commit threshold, and applies the AIMD update. An additional ~ 500 LOC integrate the scheduler into the commit path and the protocol configuration. In the commit path (Algorithm 2), the committer truncates the decision sequence at the pivot, invokes the scheduler, and rebuilds its decider instances for the new leader count. The protocol configuration gains three parameters: an adaptive scheduling toggle, the maximum leader count, and the interval. This code runs unchanged across all protocol variants, as the scheduler reads the wave length and thresholds from the protocol description and never inspects the fault model.

Simulation layer. We inherit this simulation layer from the base codebase, and extend it (~ 600 LOC) to model time-varying, heterogeneous networks, where every link has an independent one-way latency range in each direction. A run is scripted as a sequence of phases. Each phase can designate specific validators as slow, degrading every link that touches them, or optionally only their outbound links to a chosen subset of validators, beyond the leader timeout. In the configuration of Section 7 a slow validator’s blocks reach every other validator late, so its leader slots miss the direct rule after the leader timeout; degrading only part of its outbound links instead reproduces the partial-quorum situation of Figure 1. The simulator records, per validator and sampling window, the committed latency, the leader count in force, and the leader-slot decisions by type, to support the plots and numbers of Section 7.

7 Evaluation

This section benchmarks Barnacle on the base protocols of Section 6. Our evaluation has a single goal: quantify what adapting the leader count buys over the best *static* leader count in each regime. Our evaluation makes the following claims:

- **C1:** In a healthy network, Barnacle matches the latency of the best static leader count.

⁶ <https://github.com/zenodea/hammerheadv2/tree/feat/leader-scheduler>
(commit `bb99146`)

- **C2:** When slow leaders stop slots from being decided as commit by the direct rule, **Barnacle** reduces the leader count until that stops happening and matches the latency of the best static leader count for the degraded regime, well below that of a high static count.
- **C3:** **Barnacle** quickly adapts in both directions: it reduces the leader count when the network degrades and increases it when it heals; the transition adds little to the degraded-phase latency.
- **C4:** Claims **C1** to **C3** hold unchanged across base protocols with different fault models, wave lengths, and decision quorums, with identical scheduler code and parameters.

Benchmarking and testing BFT protocols under actual Byzantine behavior is an open problem [3]; the state of the art establishes worst-case guarantees through formal proofs, which we give in Section 5 for both safety and liveness.

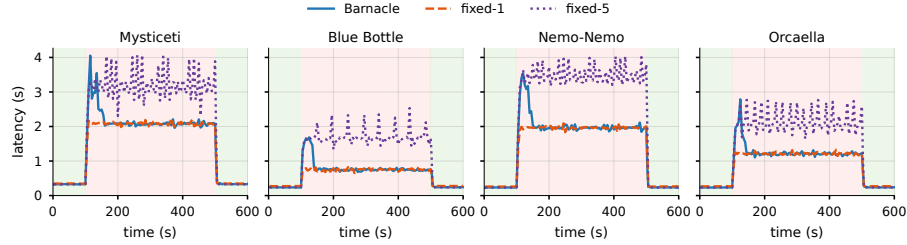
7.1 Experimental setup

Simulator and scenario. We rely on simulation because degradations must be identical across base protocols and leader configurations, switch on and off at scripted times, and remain reproducible. None of this is possible in a cloud deployment, where slowness cannot be injected on demand and never repeats identically. We simulate a committee of $n=10$ validators in a full mesh where the one-way link latency is uniform in $[50, 100]$ ms. The execution consists of three phases: a healthy phase of 100 s, a degraded phase of 400 s, and a healthy recovery phase of 100 s. During the degraded phase, a set of *slow* validators has every link that touches it, in both directions, raised to a latency uniform in $[1000, 1300]$ ms, beyond the 1 s leader timeout; all other links are unchanged. The slow set is as large as the base protocol tolerates while the remaining validators still form a quorum: three validators for Mysticeti ($f=3$) and Nemo-Nemo ($c=3$), one for Blue Bottle, whose $4n/5+1$ quorum tolerates a single fault at $n=10$, and two for Orcaella ($f=1, c=1$), whose $n-f-c$ quorum of eight tolerates exactly two faults at $n=10$. A slow validator falls rounds behind, so when it is a leader its block reaches nobody in time: every other validator waits out the leader timeout before proposing without it, and the slot is then skipped: directly for Mysticeti, Blue Bottle, and Orcaella (round-level blocking), or through the indirect decision rule for Nemo-Nemo, whose direct-skip quorum is unanimity, in which case the slot holds every later slot until an anchor resolves it (commit-level blocking, Figure 1). Either way the slot is not decided as commit by the direct rule and lowers the direct decision rate, and with more leader slots per round, more rounds contain a slow leader: three rounds in ten with one leader, seven in ten with five. The **Barnacle** scheduler operates with **Interval** = 25 rounds (about 2 s while healthy and 10–20 s while degraded, since rounds slow down), **maxLeaders** = 5, and **threshold** = 0.96.

Configurations and metrics. We compare three configurations of each base protocol: **Barnacle**, with the scheduler enabled, and two static baselines that never change their leader count, **FIXED-1** with a single leader per round and

Table 2. Mean end-to-end latency (ms) per phase; the degraded phase is its steady window (200–500s). *Barnacle* rows give the difference to *FIXED-1* and to *FIXED-5*.

Protocol	Configuration	Healthy	Degraded (steady)	Recovery
Mysticeti	FIXED-1	348	2,084	424
	FIXED-5	326	3,183	478
	Barnacle	328 (-6%, +1%)	2,077 (-0%, -35%)	408 (-4%, -15%)
Blue Bottle	FIXED-1	274	753	301
	FIXED-5	236	1,693	305
	Barnacle	238 (-13%, +1%)	745 (-1%, -56%)	270 (-10%, -11%)
Nemo-Nemo	FIXED-1	259	1,964	344
	FIXED-5	232	3,484	396
	Barnacle	234 (-10%, +1%)	1,965 (+0%, -44%)	308 (-10%, -22%)
Orcaella	FIXED-1	268	1,212	311
	FIXED-5	233	2,130	331
	Barnacle	235 (-12%, +1%)	1,209 (-0%, -43%)	284 (-9%, -14%)

**Fig. 2.** End-to-end latency over time under *Barnacle* and static leader counts (one and five), for Mysticeti, Blue Bottle, Nemo-Nemo, and Orcaella. The network goes healthy $\rightarrow$ degraded $\rightarrow$ healthy.

FIXED-5 with five, the two extremes of the range the scheduler may choose from. *Barnacle* runs start with a leader count of five. Our primary metric is end-to-end latency from client submission to commit, measured per validator over sampling windows of 3s while healthy and 5s while degraded. We report the mean over validators, from one run per configuration with a common seed so that the three configurations of a protocol see the same network draws. The per-phase means are summarized in Table 2; the “steady” degraded window is defined as [200, 500] s, after every *Barnacle* run has reached one leader.

7.2 Healthy network

In the first and last 100s of each panel in Figure 2, the three configurations are indistinguishable except that *FIXED-1* is slower. The *Barnacle* configuration equals *FIXED-5* within 1% and beats *FIXED-1* by 6% (Mysticeti, 328 vs 348 ms), 13% (Blue Bottle, 238 vs 274 ms), 10% (Nemo-Nemo, 234 vs 259 ms), and 12% (Orcaella, 235 vs 268 ms). The gap to *FIXED-1* is the queuing-latency saving of running five leaders instead of one (Section 1). *Barnacle* retains this saving in full: every healthy window has a direct decision rate of 1, so the scheduler never lowers the leader count below five. This confirms Claim C1.

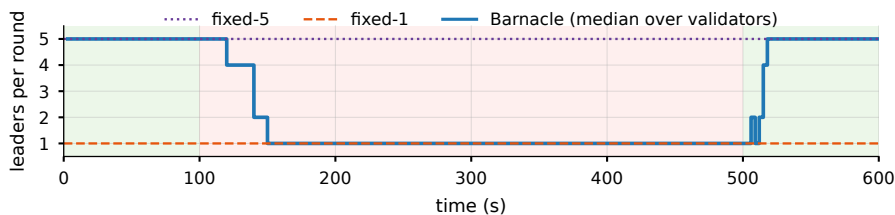


Fig. 3. Leader count chosen by Barnacle over time (Mysticeti): back-off within one interval of the onset, multiplicative decrease $5 \rightarrow 4 \rightarrow 2 \rightarrow 1$, additive recovery.

7.3 Degraded network and recovery

At $t=100$ s latency rises for every configuration, and by an amount that follows the leader count: a round containing a slow leader costs every validator the leader timeout, so the mean round stretches from ~ 90 ms to ~ 380 ms under FIXED-1 and ~ 730 ms under FIXED-5, and commit latency grows with it. As shown in Figure 3 (Mysticeti), Barnacle starts backing off within 15 s of the onset, about one interval at the slower round rate, and then follows the $-1, -2, -4$ sequence of the multiplicative decrease, stepping $5 \rightarrow 4 \rightarrow 2 \rightarrow 1$ and reaching one leader about 45 s into the phase, after which it remains flat. Latency (Figure 2, Table 2) follows the leader count: in the steady window, the Mysticeti Barnacle configuration achieves 2.08 s, equal to FIXED-1 and 35% below the 3.18 s of FIXED-5; Blue Bottle reaches 0.75 s vs 0.75 s (-1%) for FIXED-1 and 1.69 s (-56%) for FIXED-5; Nemo-Nemo 1.97 s vs 1.96 s (0%) and 3.48 s (-44%); and Orcaella 1.21 s vs 1.21 s (0%) and 2.13 s (-43%). Over the whole degraded phase, descent included, the Barnacle configuration is 5% (Mysticeti), 9% (Blue Bottle), 6% (Nemo-Nemo), and 6% (Orcaella) above FIXED-1, which bounds the cost of the transient. No configuration stalls: every 5 s window of every validator commits transactions throughout the degraded phase. During recovery at $t=500$ s, latency returns to the healthy level for all configurations within seconds. The leader count climbs $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$ one step per interval, arriving back at five about 15 s after healing. The recovery numbers in Table 2 are means over the 100 s after healing and thus cover two parts: the drain of the backlog, shared by all configurations, and a healthy remainder in which Barnacle, back at five leaders, regains the multi-leader advantage while FIXED-1 keeps its single leader. This is why Barnacle leads FIXED-1 in this column too. This confirms Claim C2 and Claim C3.

7.4 Generality across fault models

The behavior shown in Figures 2 and 3 remains the same across Mysticeti [2,20], Blue Bottle [30], Nemo-Nemo [14], and Orcaella [15], employing identical scheduler code and parameters. Only the absolute latencies differ, set by each base protocol’s wave length and quorums: the price of a slow leader slot ranges from a directly skipped slot (Mysticeti, Blue Bottle, Orcaella) to an indirectly decided one (Nemo-Nemo), and the penalty of FIXED-5 over FIXED-1 in the steady window ranges from 53% (Mysticeti) to 125% (Blue Bottle). This confirms Claim C4.

We do not evaluate the sensitivity to `Interval` and `threshold`; the threshold of 0.96 tolerates about four non-direct slots per window under five leaders and none under a single leader, and no per-protocol tuning was needed.

8 Related Work

All prior multi-leader DAG protocols fix the leader count at configuration time; to our knowledge, **Barnacle** is the first to adapt it at run time, and the first to compose with any of them as an add-on.

Multi-leader DAG protocols (static leader count). The single-leader DAG lineage includes DAG-Rider [12], Narwhal and Tusk [6], Bullshark [22], and Cordial Miners [13]. To cut latency, recent designs support several leaders per round. Shoal [21] and Shoal++ [1] commit several anchors per round on certified DAGs. Sailfish and Sailfish++ [18,19] guarantee a leader every round and specify a multi-leader variant. Mysticeti [2] operates on uncertified DAGs and recommends two leaders. Blue Bottle [30] targets $n = 5f+1$ with two-round waves, Nemo-Nemo [14] brings the design to the crash-fault model, Orcaella [15] to a mixed model tolerating f Byzantine and c crashed validators ($n = 5f+3c+1$), and Mahi-Mahi [11] is an asynchronous protocol with a configurable number of leaders per round. We instantiate **Barnacle** on four of these protocols.

Leader selection: who leads. Reputation-based election protocols determine who leads. Carousel [5] applies this to chained protocols. Hammerhead [28] implements reputation for DAG protocols and is deployed in Sui. Shoal [21] adds reputation to certified DAGs, and Liu et al. [16] formalize reputation-based election as a protocol-independent abstraction with generic correctness guarantees, as we do for the leader count. These belong to the same technique family as ours: they derive the schedule deterministically from the committed history, requiring no messages. However, they use a different knob: they exclude unreliable validators from the leader set, whereas **Barnacle** sizes the leader set. The approaches are orthogonal and composable: reputation removes unreliable validators from the leader set, and **Barnacle** then sizes it.

Multi-BFT and multi-proposer protocols. Parallel consensus systems such as Mir-BFT [23], ISS [24], RCC [9], and Autobahn [8] multiplex parallel consensus instances or proposers into one log. The straggler problem of the global ordering, where a slow instance delays the log, is head-of-line blocking in another guise. Ladon [17] attacks this by ordering blocks dynamically. **Barnacle** instead adapts the degree of parallelism; applying it to these protocols is future work.

Adaptive BFT. BFTBrain [31] switches between whole protocols at run time with reinforcement learning, using features that include the fraction of requests committed on the fast path, and coordinates learning through consensus. **Barnacle** adapts a single parameter with a closed-form deterministic rule that needs no learning and no coordination beyond the committed DAG. The rule itself is additive increase, multiplicative decrease, the congestion-control law analyzed by Chiu and Jain [4], which **Barnacle** applies to regulate the leader count dynamically.

9 Conclusion

Because of head-of-line blocking, no static leader count is right in every regime. We introduced **Barnacle**, which adapts the leader count with an AIMD control loop on the direct decision rate measured on the committed DAG. The mechanism is local, deterministic, and message-free, and preserves the base protocol’s safety and liveness. Our evaluation (Section 7) demonstrates that **Barnacle** matches the best static leader count in every regime, reducing latency over a single leader when healthy and avoiding the severe degradation of high leader counts during instability. **Barnacle** is currently being integrated into the Sui blockchain. Because correctness holds for any deterministic update rule, the rule can be swapped without touching the proofs (Section 5); future work includes replaying the window under counterfactual leader counts to jump directly to the best one, and extending the approach to multi-proposer protocols outside DAGs.

Acknowledgements

This work is partially funded by Mysten Labs. We thank George Danezis for insightful discussions and for reviewing the Lean 4 formalization of **Barnacle** (Appendix C), and Mingwei Tian and Arun Koshy for their feedback on the early design of **Barnacle**.

References

1. Arun, B., Li, Z., Suri-Payer, F., Das, S., Spiegelman, A.: Shoal++: High throughput DAG BFT can be fast and robust! In: USENIX NSDI (2025)
2. Babel, K., Chursin, A., Danezis, G., Kichidis, A., Kokoris-Kogias, L., Koshy, A., Sonnino, A., Tian, M.: Mysticeti: Reaching the latency limits with uncertified DAGs. In: NDSS (2025)
3. Bano, S., Sonnino, A., Chursin, A., Perelman, D., Li, Z., Ching, A., Malkhi, D.: Twins: BFT systems made robust. In: International Conference on Principles of Distributed Systems (OPODIS) (2021)
4. Chiu, D.M., Jain, R.: Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN Systems* **17**(1), 1–14 (1989)
5. Danezis, G., Kokoris-Kogias, L., Sonnino, A., Spiegelman, A.: Be aware of your leaders. In: Financial Cryptography and Data Security (FC) (2022), <https://sonnino.com/papers/carousel.pdf>
6. Danezis, G., Kokoris-Kogias, L., Sonnino, A., Spiegelman, A.: Narwhal and Tusk: A DAG-based mempool and efficient BFT consensus. In: EuroSys (2022)
7. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)* **35**(2), 288–323 (1988)
8. Giridharan, N., Suri-Payer, F., Abraham, I., Alvisi, L., Crooks, N.: Autobahn: Seamless high speed BFT. In: ACM Symposium on Operating Systems Principles (SOSP) (2024)

9. Gupta, S., Hellings, J., Sadoghi, M.: RCC: Resilient concurrent consensus for high-throughput secure transaction processing. In: IEEE International Conference on Data Engineering (ICDE) (2021)
10. IOTA Foundation: IOTA. <https://www.iota.org> (2026)
11. Jovanovic, P., Kokoris-Kogias, L., Kumara, B., Sonnino, A., Tennage, P., Zabolotchi, I.: Mahi-Mahi: Low-latency asynchronous BFT DAG-based consensus (2024)
12. Keidar, I., Kokoris-Kogias, E., Naor, O., Spiegelman, A.: All you need is DAG. In: ACM PODC (2021)
13. Keidar, I., Naor, O., Shapiro, E.: Cordial Miners: Fast and efficient consensus for every eventuality. In: DISC (2023)
14. Kerur, R., Tennage, P., Jovanovic, P., Malkhi, D., Sonnino, A., Zabolotchi, I.: Finding Nemo-Nemo: CFT DAG-based consensus in the WAN (2026)
15. Kokoris-Kogias, L., Sonnino, A.: Orcaella: Hybrid fault tolerance with client-selectable finality latency (2026)
16. Liu, X., Zhang, Z., Li, Z., Sun, J., Liu, J., Jiang, P.: Reputation-based leader election under partial synchrony: Towards a protocol-independent abstraction with enhanced guarantees (2025)
17. Lyu, H., Xie, S., Niu, J., Feng, C., Zhang, Y., Beschastnikh, I.: Ladon: High-performance multi-BFT consensus via dynamic global ordering. In: ACM/IFIP International Middleware Conference (2024)
18. Shrestha, N., Shrothrium, R., Kate, A., Nayak, K.: Sailfish: Towards improving the latency of DAG-based BFT. In: IEEE S&P (2025)
19. Shrestha, N., Yu, Q., Kate, A., Losa, G., Nayak, K., Wang, X.: Optimistic, signature-free reliable broadcast and its applications. In: ACM CCS (2025)
20. Sonnino, A.: Mysticeti: Uncertified DAG-based consensus reference implementation. <https://github.com/asonnino/mysticeti> (2024)
21. Spiegelman, A., Arun, B., Gelashvili, R., Li, Z.: Shoal: Improving DAG-BFT latency and robustness (2023), <https://arxiv.org/abs/2306.03058>
22. Spiegelman, A., Giridharan, N., Sonnino, A., Kokoris-Kogias, L.: Bullshark: DAG BFT protocols made practical. In: ACM CCS (2022)
23. Stathakopoulou, C., David, T., Pavlovic, M., Vukolić, M.: Mir-BFT: High-throughput robust BFT for decentralized networks (2019)
24. Stathakopoulou, C., Pavlovic, M., Vukolić, M.: State machine replication scalability made simple. In: European Conference on Computer Systems (EuroSys) (2022)
25. Sui contributors: Sui: reference implementation. <https://github.com/MystenLabs/sui> (2026), protocol configuration `mysticeti.num.leaders.per.round`
26. Sui Foundation: Sui. <https://sui.io> (2026)
27. The Tokio Team: Tokio. <https://tokio.rs> (2024)
28. Tsimos, G., Kichidis, A., Sonnino, A., Kokoris-Kogias, L.: HammerHead: Score-based dynamic leader selection. In: IEEE International Conference on Distributed Computing Systems (ICDCS) (2024), <https://sonnino.com/papers/hammerhead.pdf>
29. de Valence, H.: Ed25519 for consensus-critical contexts. <https://crates.io/crates/ed25519-consensus> (2024)
30. Vander Vos, P., Sonnino, A., Tsimos, G., Jovanovic, P., Kokoris-Kogias, L.: Blue-Bottle: Fast and robust blockchains through subsystem specialization (2025)
31. Wu, C., Qin, H., Amiri, M.J., Loo, B.T., Malkhi, D., Marcus, R.: BFTBrain: Adaptive BFT consensus with reinforcement learning. In: USENIX Symposium on Networked Systems Design and Implementation (NSDI) (2025)

A Base-Protocol Pseudocode

This appendix specifies the base-protocol decision rules that the algorithms of Section 4 invoke. A single generic decider covers all four base protocols: Algorithm 5 contains the decision rules, parameterized by the three thresholds `commitThreshold`, `skipThreshold`, and `linkSize`; Algorithm 4 contains the decider instance (wave geometry and DAG predicates) and is identical for every protocol; Table 1 gives the per-protocol values. The pseudocode follows Mysticeti’s published algorithms [2], generalized where the four protocols differ.

A.1 One decider for four protocols

Six changes over the pseudocode published in the original papers [2,30,14,15] make it generic: (i) The vote round is the round after the propose round, since no protocol decides in fewer than two rounds; for two-round waves the vote and certify rounds coincide. (ii) Quorum sizes are parameters rather than hard-coded $2f+1$. (iii) ISVOTE is a parent lookup: a round- r leader block can only be a direct parent of a round- $(r+1)$ block, and taking the first matching parent makes a block vote for at most one block of an equivocating leader. (iv) ISCERT returns ISVOTE when the vote and certify rounds coincide: the vote block is itself the certificate. (v) The skip test is per slot, counting blocks that vote for no block of the slot, not per equivocating leader block. (vi) ISCERTIFIEDLINK counts certified links and compares against `linkSize`.

A.2 The decider instance

An instance (Algorithm 4) is parameterized by a wave offset, a slot index, and the DAG it reads (the local DAG in Algorithm 2, the window in Algorithm 3). Pipelining uses one instance per offset, making every round the propose round of some wave, a strategy that follows Mysticeti [2]. LEADERBLOCK returns the slot’s leader block, or several under equivocation. ISVOTE and ISLINK are the two primitive relations: direct parenthood of the leader block, and reachability via references.

A.3 The decision rules

In Algorithm 5, by the direct decision rule, a slot is decided as skip if `skipThreshold` blocks of the vote round vote for no block of the slot, as commit if `commitThreshold` blocks of the certify round certify one of its leader blocks, and stays undecided otherwise. By the indirect decision rule, the anchor is the first later slot, beyond the wave, that is not decided as skip. While the anchor is undecided, the slot stays undecided. Once the anchor is decided as commit, the slot is decided as commit if at least `linkSize` certify-round blocks are simultaneously certificates for its leader block and ancestors of the anchor, and as skip otherwise.

```

1: waveOffset = i                                ▷ First Decider parameter (i): pipelining offset
2: leaderOffset = l                             ▷ Second Decider parameter (l): slot in the round
3: DAG                                           ▷ Third Decider parameter: the DAG the rules read
4: waveLength                                   ▷ 3 (Mysticeti); 2 (Blue Bottle, Nemo-Nemo)

5: procedure WAVELENGTH(r)
6:   return (r - waveOffset) / waveLength

7: procedure PROPOSEROUND(w)
8:   return (w × waveLength) + waveOffset

9: procedure VOTEROUND(w)
10:  return PROPOSEROUND(w) + 1                    ▷ Two rounds is the minimum

11: procedure CERTIFYROUND(w)
12:  return PROPOSEROUND(w) + (waveLength - 1)    ▷ = VOTEROUND(w) if waveLength = 2

13: procedure GETDECISIONBLOCKS(w)
14:  return DAG[CERTIFYROUND(w)]

15: procedure LEADERBLOCK(w)                      ▷ Several blocks upon equivocation
16:  r_propose ← PROPOSEROUND(w)
17:  l ← GETLEADER(r_propose + leaderOffset)      ▷ Deterministic
18:  return {b ∈ DAG[r_propose] : b.author = l}

19: procedure ISVOTE(b_vote, b_leader)             ▷ b_vote is in round b_leader.round + 1
20:  P ← [b ∈ b_vote.parents : (b.author, b.round) = (b_leader.author, b_leader.round)]
21:  return P ≠ [] ∧ P[0] = b_leader              ▷ One vote per equivocating leader

22: procedure ISLINK(b_old, b_new)
23:  return ∃ a sequence of k ∈ ℕ blocks b1, ..., bk s.t.
        b1 = b_old ∧ bk = b_new ∧ ∀ j ∈ [2, k] : bj-1 ∈ bj.parents

```

Algorithm 4: Decider Instance: Rounds and DAG Helpers (identical for all base protocols)

For Mysticeti [2] (waveLength=3), a certificate is a block carrying `commitThreshold` votes; all thresholds are $n-f$; one certified link suffices because after a direct skip no certificate can exist. For Blue Bottle [30] (waveLength=2), the vote is the certificate, and `linkSize` = $n-3f$ keeps a commit by the indirect decision rule consistent with a skip by the direct decision rule. For Nemo-Nemo [14] (crash faults), thresholds are $c+1$; the skip threshold n never triggers in practice, so skips are decided by the indirect decision rule; equivocation does not arise. For Orcaella [15] (waveLength=2, mixed faults), the vote is likewise the certificate, with thresholds of $n-f-c$ and `linkSize` = $n-3f-2c$ certified links keeping a commit by the indirect decision rule consistent with a skip by the direct decision rule; setting $c=0$ recovers Blue Bottle exactly.

B Full Proofs

This appendix proves every statement of Section 5, in the same order and under the same names; each statement is restated before its proof. Each proof names the assumption of Section 3 at the step that consumes it. The Lean development (Appendix C) machine-checks every statement; the proofs here are the human-readable arguments, and they follow the machine-checked ones step for step.

Fact 1 (DAG facts, restated). *Let b be a block held by an honest validator.*
(i) The validator's DAG contains the entire causal history of b . (ii) Every honest

```

1: commitThreshold  $\triangleright$  Direct rule, commit:  $n - f$  (Mysticeti, Blue Bottle);  $n - c$  (Nemo-Nemo);  $n - f - c$  (Orcaella)
2: skipThreshold  $\triangleright$  Direct rule, skip:  $n - f$  (Mysticeti, Blue Bottle);  $n$  (Nemo-Nemo);  $n - f - c$  (Orcaella)
3: linkSize  $\triangleright$  Indirect rule, commit: 1 (Mysticeti, Nemo-Nemo);  $n - 3f$  (Blue Bottle);  $n - 3f - 2c$  (Orcaella)

4: procedure ISCERT( $b_{cert}, b_{leader}$ )
5:    $w \leftarrow \text{WAVENUMBER}(b_{leader}.round)$ 
6:   if CERTIFYROUND( $w$ ) = VOTEROUND( $w$ ) then  $\triangleright$  Two-round waves: vote = certificate
7:     return ISVOTE( $b_{cert}, b_{leader}$ )
8:    $V \leftarrow \{b \in b_{cert}.parents : \text{ISVOTE}(b, b_{leader})\}$ 
9:   return  $|V| \geq \text{commitThreshold}$ 

10: procedure SKIPPEDLEADER( $w$ )  $\triangleright$  Blocks voting for no block of the slot
11:    $r_{vote} \leftarrow \text{VOTEROUND}(w)$ 
12:    $B_{vote} \leftarrow \{b \in \text{DAG}[r_{vote}] : \forall b_{leader} \in \text{LEADERBLOCK}(w), \neg \text{ISVOTE}(b, b_{leader})\}$ 
13:   return  $|B_{vote}| \geq \text{skipThreshold}$ 

14: procedure SUPPORTEDLEADER( $w, b_{leader}$ )
15:    $B_{decision} \leftarrow \text{GETDECISIONBLOCKS}(w)$ 
16:   return  $|\{b \in B_{decision} : \text{ISCERT}(b, b_{leader})\}| \geq \text{commitThreshold}$ 

17: procedure ISCERTIFIEDLINK( $b_{anchor}, b_{leader}$ )
18:    $w \leftarrow \text{WAVENUMBER}(b_{leader}.round)$ 
19:    $B_{decision} \leftarrow \text{GETDECISIONBLOCKS}(w)$ 
20:   return  $|\{b \in B_{decision} : \text{ISCERT}(b, b_{leader}) \wedge \text{ISLINK}(b, b_{anchor})\}| \geq \text{linkSize}$ 

21: procedure TRYDIRECTDECIDE( $w$ )
22:   if SKIPPEDLEADER( $w$ ) then return Skip( $w$ )
23:   for  $b_{leader} \in \text{LEADERBLOCK}(w)$  do  $\triangleright$  Loop over equivocations
24:     if SUPPORTEDLEADER( $w, b_{leader}$ ) then return Commit( $b_{leader}$ )
25:   return  $\perp$ 

26: procedure TRYINDIRECTDECIDE( $w, S$ )
27:    $r_{certify} \leftarrow \text{CERTIFYROUND}(w)$ 
28:    $s_{anchor} \leftarrow \text{first } s \in S \text{ s.t. } r_{certify} < s.round \wedge s \neq \text{Skip}$ 
29:   if  $s_{anchor} = \text{Commit}(b_{anchor})$  then
30:     if  $\exists b_{leader} \in \text{LEADERBLOCK}(w) : \text{ISCERTIFIEDLINK}(b_{anchor}, b_{leader})$  then
31:       return Commit( $b_{leader}$ )
32:     else return Skip( $w$ )
33:   return  $\perp$   $\triangleright$  The anchor is undecided or not found

```

Algorithm 5: Decision Rules (the only part that varies across base protocols, Table 1)

validator eventually holds b . (iii) Any two honest validators that hold b hold the same causal history of b .

Proof. (i) A2 states that a validator adds a block to its DAG only once the block's entire causal history is available; by induction over the order of additions, every block in the DAG has its causal history in the DAG. (ii) The broadcast layer guarantees that a block added by one honest validator is eventually received by every honest validator, which then adds it, its causal history being available by the same guarantee applied recursively. (iii) A block names its references as part of its content, so the causal history of b — the set of blocks reachable from b by following references — is determined by b alone; by (i), each of the two validators holds exactly this set. $\square$

Lemma 1 (Window Agreement, restated). *Any two honest validators that commit the same pivot compute the same window.*

Proof. GETSUBDAG (Algorithm 3) defines the window as the restriction of the pivot's causal history to the **Interval** rounds up to the pivot's round; the round

of a block is part of its content. A validator that commits the pivot holds its entire causal history, by Fact 1(i), so the window is fully contained in its DAG; by Fact 1(iii), the two validators hold the same causal history; and GETSUBDAG is deterministic, so the two restrictions are the same window. $\square$

Proposition 1 (Leader-Count Agreement, restated). *For any deterministic update rule — the AIMD rule of Algorithm 3 is one instance — any two honest validators agree on every configuration both have reached, its start round and its leader count, and on the verdict of every slot of every configuration both have closed.*

Proof. By induction over k . For the base case, both validators initialize $C_0 = (0, 1)$ with back-off 0 (Algorithms 2 and 3). For the inductive step, assume both validators have reached C_k and agree on its start round, leader count, and back-off; we show that they agree on every verdict of C_k 's rounds, and, if both close C_k , on the pivot and on C_{k+1} .

Verdicts agree. By the update semantics (Section 4.1), every slot of the rounds from C_k 's start round up to and including its pivot's round is decided against the fixed schedule of C_k — $\ell(C_k)$ slots per round, slot (r, l) led by $\text{GETLEADER}(r + l)$. This includes the derivations that the indirect decision rule performs: an indirect decision for a slot of those rounds reads decision anchors at higher rounds, and on the invocation that produces the verdict, those higher slots are evaluated under the same count, since TRYCOMMIT switches the count only after returning (Algorithm 2). Every verdict of C_k at either validator is therefore a verdict of one fixed schedule, and the safety half of A4 makes the two validators' verdicts equal wherever both have decided.

Pivots agree. Suppose both validators close C_k , with pivots a_1 and a_2 , each the least committed slot whose round exceeds C_k 's start round by more than **Interval**. If $a_1 \neq a_2$, assume without loss of generality $a_1 < a_2$ in slot order. Slot a_1 is committed at the first validator, so by verdict agreement its verdict at the second validator is the same commit; but a_1 lies past the threshold and below a_2 , contradicting the minimality of a_2 . Hence $a_1 = a_2$, and by verdict agreement both validators commit the same pivot block.

The next configuration agrees. C_{k+1} 's start round is the pivot's round, agreed. Its leader count and back-off are the output of the update rule, a deterministic function of the agreed leader count and back-off of C_k , the agreed pivot block, and its window, agreed by Lemma 1. Hence C_{k+1} is equal at both validators.

A validator that has not closed C_k agrees, by the induction so far, on the strict prefix of configurations it has reached, which is all the statement claims. No step above uses synchrony, and no base-protocol fact beyond the safety half of A4 is consumed. $\square$

Theorem 1 (Safety, restated). *Barnacle composed with any base protocol satisfying A1–A4 satisfies the safety of Definition 1: the committed sequences of any two honest validators agree as far as both reach and grow only by appending (Agreement, Total Order), and no block is committed twice (Integrity).*

Proof. A validator's committed sequence up to its K -th closed configuration is the concatenation, in configuration order, of the per-configuration ledgers: for each C_k , the committed blocks of C_k 's rounds in slot order, as read from the verdicts. By Proposition 1, two validators agree on every configuration and every verdict of the configurations both have closed, so their per-configuration ledgers are equal up to the lower of the two heights, and the concatenations agree as far as both reach. A validator's own sequence at a lower height is by construction a prefix of its sequence at a higher one: later configurations only append. This gives Agreement and Total Order.

For Integrity, suppose a block L appears twice. A commit verdict names a block of its slot's round and leader (the decision rules only ever commit a leader block of the slot under decision), so both occurrences of L lie at slots with L 's round and L 's author as leader. Within one configuration, the leaders of one round are distinct validators — the rotation assigns $\text{GETLEADER}(r), \dots, \text{GETLEADER}(r + \ell - 1)$, which are distinct because $\text{maxLeaders} \leq n$ — so the two slots are one slot, and a slot carries one verdict. Across configurations, the rounds of distinct configurations are disjoint, and L 's round lies in exactly one of them. $\square$

Remark 1 (Conservativity, restated). *With the constant update rule, Barnacle is the base protocol: the same schedule, the same verdicts, the same ledger. The AIMD rule additionally keeps the leader count within $[1, \text{maxLeaders}]$, and its threshold test is one integer comparison.*

Proof. Under the constant rule, every update returns the leader count and back-off it was given, so by induction from $C_0 = (0, 1)$ every configuration has one leader and back-off 0. With one leader per round, slot numbers coincide with round numbers and the schedule is the base protocol's own; every verdict is then a derivation of the base protocol's decision rules on the unmodified DAG, and the committed sequence is the base protocol's ledger. The bounds of the AIMD rule are read off Algorithm 3: the increase is capped at maxLeaders and the decrease is floored at 1, and the threshold test multiplies both sides into integers. $\square$

Lemma 2 (Configuration Progress, restated). *After GST, if every honest validator has closed configuration C_k , then some honest validator closes C_{k+1} within bounded time, and every honest validator does within a further Δ .*

Proof. Closing C_{k+1} means committing its pivot: the least committed slot whose round exceeds C_{k+1} 's start round by more than Interval . Fix an honest validator v . After GST, the liveness half of A4, applied to the fixed schedule of C_{k+1} , guarantees two things within bounded time: every slot of C_{k+1} 's rounds up to any fixed horizon is decided (with the wave-plus-commit-gap margin below the horizon), and some slot at a round within the commit gap c of the threshold round is committed. Let κ_0 be such a committed slot; the pivot exists and lies at or below κ_0 in slot order.

Two identifications, and this is where the lemma consumes base *safety*: the commit that liveness supplies is a verdict on the network’s joint view, while the pivot that the update reads is v ’s own verdict; the safety half of A4 makes them the same verdict. Likewise, every slot between C_{k+1} ’s start and κ_0 is decided, and TRYCOMMIT’s walk (Algorithm 2), which stops at the first undecided slot, therefore reaches the pivot and commits it. Validator v then invokes the update rule and closes C_{k+1} .

Finally, within a further Δ , every honest validator holds the pivot and its causal history (Fact 1(ii), with delivery bounded by Δ after GST), decides the same slots by the safety half of A4, and closes C_{k+1} by determinism of the update rule (Proposition 1). $\square$

Theorem 2 (Liveness, restated). *After GST, every honest validator’s run reaches every height: for every k it closes configuration C_k , and committed slots keep appearing.*

Proof. By induction on k : height 0 holds initially, and Lemma 2 extends every height by one within bounded time after GST. Each closing commits at least the pivot, so committed slots keep appearing. Each pivot’s round exceeds the previous start round by more than **Interval**, so configuration start rounds strictly increase even though the leader count oscillates. The per-configuration clause of A4-liveness that Lemma 2 invokes carries a margin: a slot is guaranteed decided only when a commit gap and a wave still fit below the horizon of the DAG. The margin is necessary, not an artifact of the proof: a slot whose direct decision fails — a Byzantine leader can always cause this — is decided only through a committed slot at least a full wave above it, whose own wave must also fit under the horizon; no protocol decides such a slot at the last rounds under the horizon. $\square$

Remark 2 (Liveness of the rotation, restated). *For the leader schedule used in this paper, the per-configuration liveness clause of A4 is proved rather than assumed, for all four base protocols at every leader count (Appendix C), with a commit gap of $n + \text{waveLength} - 1$ rounds.*

Proof. Each of the four base protocols supplies two descent facts after GST, from its own theorems: a slot led by a validator of its reliable set is decided as commit by the direct rule within its wave, whatever the schedule; and a slot is decided as soon as some committed slot lies at least a full wave above it with no eligible slot in between (the indirect decision rule). The reliable set misses at most f validators for Mysticeti and Blue Bottle, at most $f + c$ for Orcaella, whose mixed bound enters only here, and at most $n - \lceil (n + 1)/2 \rceil$ for Nemo-Nemo, whose crash bound enters only here, through the reliable set being a majority.

The classic liveness argument asks for a wave of consecutive honest-led *slots*, and under the rotation GETLEADER($r + l$) it has no instance once several leaders share a round: at $n = 4$ and two leaders per round, three consecutive rounds

already name every validator. The proof instead descends through round *heads*. Under the schedule with m leaders, the head of round ρ — its first slot — is slot $m\rho$, and it is led by $\text{GETLEADER}(\rho)$ *whatever m is*; this is what lets one argument serve every leader count. If the heads of **waveLength** consecutive rounds are all led by reliable validators, the last of them is decided as commit by the direct rule, and every slot within a wave below the first is decided: the committed head lies exactly a wave above it, and no slot strictly between is a full wave above it, so the indirect decision rule applies with nothing in between.

It remains to find such heads: a pigeonhole on the rotation. Round-robin visits the n validators cyclically, so the heads of **waveLength** consecutive rounds are **waveLength** consecutive validators in the rotation order. If every window of **waveLength** consecutive rounds in a cycle contained a head led outside the reliable set, then mapping each of the n window starts to such a head would inject the n starts into the **waveLength** positions times the at most s excluded validators, giving $n \leq \text{waveLength} \cdot s$; the committee bounds ($n \geq 3f + 1$ with **waveLength** = 3 for Mysticeti, $n \geq 5f + 1$ with **waveLength** = 2 for Blue Bottle, $n \geq 5f + 3c + 1$ with **waveLength** = 2 and slack $f + c$ for Orcaella, and a majority with **waveLength** = 2 for Nemo-Nemo) exclude this. Hence within any $n + \text{waveLength} - 1$ consecutive rounds some **waveLength** consecutive heads are reliable-led, which bounds the commit gap by $n + \text{waveLength} - 1$. $\square$

C Lean Formalization

Every statement of Section 5 is machine-checked in Lean 4. The theorems are stated once over an abstract base protocol and an arbitrary deterministic update rule, then instantiated at Mysticeti, Blue Bottle, Nemo-Nemo, and Orcaella. The formalization contains no **sorry** and relies on no axioms beyond Lean’s three standard ones (propositional extensionality, quotient soundness, and choice). The development spans $\sim 3,900$ lines, partitioned so a human reader audits only definitions and theorem statements ($\sim 1,900$ lines). The proofs ($\sim 2,000$ lines) are AI-generated, checked by the kernel and need no reading. A further $\sim 3,200$ lines of executable witnesses evaluate every definition on concrete DAGs before any theorem uses it, guarding against vacuous statements. These witnesses include a run whose leader count rises at a pivot, two validators’ runs identified by the agreement theorem, a DAG on which the liveness theorems produce verdicts end to end, and a mixed-fault DAG that one admissible indirect threshold commits and another skips.

C.1 The Base-Protocol Interface

The base protocol is formalized as a structure providing a universe of blocks with views, a direct decision predicate reading one wave, and a decision relation parametric in the leader schedule. Its laws are the assumptions A1–A4 of Section 3 made precise — A1 is data rather than a law, the laws capture A2, A3, and the safety half of A4, and the liveness half is the clause stated further down — and

Table 3. Paper statements (Section 5) and their machine-checked counterparts. Each Lean result is a **Statement** file (audited) with a separate kernel-checked proof.

Paper statement	Lean result	Content
Lemma (Window Agreement)	Window	A view holding the pivot holds its whole causal history; two views compute one window.
Prop. (Leader-Count Agreement)	Agreement	Two runs, from any views to any heights, agree on every configuration (start round, count, back-off) and every verdict of their common ranges; for any update rule.
Theorem (Safety)	Ledger	The committed sequence is agreed as far as both runs reach, grows by prefixes, and holds no block twice.
Remark (Conservativity)	Conservativity	Under the constant rule every verdict is a verdict of the base protocol at one leader.
AIMD rule (Algorithm 3)	Aimd	The count stays in $[1, \text{maxLeaders}]$; the healthy and unhealthy steps; the integer threshold test.
Lemma (Configuration Progress)	Progress	A run past the synchrony round extends by one configuration; consumes the agreement law.
Theorem (Liveness)	Progress	Runs of every height exist under the horizon that height needs.
Remark (Rotation liveness)	Heads	Round-robin is live at every leader count, from the descent laws and a pigeonhole, with gap $n + \text{waveLength} - 1$.
Base protocols (A1–A4)	Mysticeti, Odontoceti, Nemo, Orcaella	Each protocol satisfies the laws and the descent laws; Blue Bottle appears as Odontoceti, Nemo-Nemo’s safety needs no fault bound.

they are exactly what Section 5 consumes. The fault model lives only in each protocol’s instantiation, never in the interface.

A base protocol exposes views V of a universe of blocks, each carrying a round, an author, and references (A1); a predicate $\text{DirectCommit}_V(L, r)$; and a relation $\text{Decided}_S(V, s, v)$, for S a leader schedule assigning a leader to every slot (A1), s a slot, and v a verdict (a block or skip), subject to:

- *Causal completeness (A2)*. A view contains every block that its blocks reference.
- *Agreement (A4, safety)*. For a fixed schedule S : $\text{Decided}_S(V_1, s, v_1)$ and $\text{Decided}_S(V_2, s, v_2)$ imply $v_1 = v_2$.
- *Direct decisions are decisions (A3)*. If L is a candidate of slot s (right round, right author) and $\text{DirectCommit}_V(L, \text{round}(s))$, then $\text{Decided}_S(V, s, L)$.
- *Decisions are about candidates*. $\text{Decided}_S(V, s, L)$ implies L is a candidate of s .

C.2 Runs and Liveness

The Barnacle mechanism relies on a dynamic schedule. The schedule of a configuration with m leaders assigns slot (r, l) to $\text{GETLEADER}(r + l)$. The leader of a slot does not depend on m ; only which slots exist does. The window is the pivot’s causal history. The observed count is over slots: a slot counts when it has a candidate decided as commit by the direct rule within the window. Two such candidates of one slot are one block, by agreement. The threshold is the integer pair (96, 100) and the test an integer comparison. An update rule is any function mapping (count, backoff, DAG, pivot) to (count’, backoff’). The AIMD rule of Algorithm 3 and the constant rule are two instances.

Runs are formalized as prefixes. A run closed to height K carries configurations $0, \dots, K$ defining a start round, leader count, and back-off. Each configuration is closed by a pivot, which is the least committed slot past the interval. Every

verdict of a configuration's rounds is derived against that configuration's fixed schedule, formalizing the semantics of Section 4.1. The next configuration is the update rule applied to the pivot. A finite DAG closes finitely many configurations, so there is no total run. Every theorem compares runs of any two heights on the configurations both reach, matching the prefix formulation of Section 5.

Liveness avoids physical clocks by treating synchrony structurally. Each protocol defines when a DAG is good from a round to a horizon N , meaning a reliable quorum is synchronized and populating the rounds. Liveness is a clause on a schedule S with commit gap c , captured by the following property:

On a good DAG, (a) every slot lying at least c rounds plus a wave under the horizon is decided, and (b) within any c consecutive rounds in that region some slot is committed.

The margin under the horizon is the one Section 5 shows necessary.

C.3 Results

Table 3 maps each statement of Section 5 to its machine-checked counterpart. Mysticeti satisfies the interface laws and the descent laws at slack f with commit gap $n + 2$. Blue Bottle satisfies them at slack f with gap $n + 1$. Nemo-Nemo's safety consumes no fault bound at all. Its liveness uses the crash bound only for a synchronized majority of live validators to exist, with gap $n + 1$. Orcaella satisfies the laws at slack $f + c$ with gap $n + 1$; its mixed bound enters only through the reliable set.