Mysticeti: Reaching the Latency Limits with Uncertified DAGs

Kushal Babel*†, Andrey Chursin‡, George Danezis‡§, Anastasios Kichidis‡, Lefteris Kokoris-Kogias‡¶, Arun Koshy‡, Alberto Sonnino‡§, Mingwei Tian‡

*Cornell Tech, †IC3, ‡Mysten Labs, §University College London (UCL), ¶IST Austria

***Abstract*—We introduce Mysticeti-C, the first DAG-based Byzantine consensus protocol to achieve the lower bounds of latency of 3 message rounds. Since Mysticeti-C is built over DAGs it also achieves high resource efficiency and censorship resistance. Mysticeti-C achieves this latency improvement by avoiding explicit certification of the DAG blocks and by proposing a novel commit rule such that every block can be committed without delays, resulting in optimal latency in the steady state and under crash failures. We further extend Mysticeti-C to Mysticeti-FPC, which incorporates a fast commit path that achieves even lower latency for transferring assets. Unlike prior fast commit path protocols, Mysticeti-FPC minimizes the number of signatures and messages by weaving the fast path transactions into the DAG. This frees up resources, which subsequently result in better performance. We prove the safety and liveness in a Byzantine context. We evaluate both Mysticeti protocols and compare them with state-of-the-art consensus and fast path protocols to demonstrate their low latency and resource efficiency, as well as their more graceful degradation under crash failures. Mysticeti-C is the first Byzantine consensus protocol to achieve WAN latency of 0.5s for consensus commit while simultaneously maintaining state-of-the-art throughput of over 200k TPS. Finally, we report on integrating Mysticeti-C as the consensus protocol into the Sui blockchain [1], resulting in over 4x latency reduction.**

I. Introduction

Several recent blockchains, such as Sui [1], [2], have adopted consensus protocols based on certified directed acyclic graphs (DAG) of blocks [3], [4], [5], [6], [7], [8], [9], [10], [11]. By design, these consensus protocols scale well in terms of throughput, with a performance of 100k tx/s of raw transactions and are robust against faults and network asynchrony [12], [3]. This, however, comes at a high latency of around 2-3 seconds, which can hinder user experience and prevent low-latency applications.

Mysticeti-C: the power of uncertified DAGs. Certified DAGs [6], [3], where each vertex is delivered through consistent broadcast [13], have high latency for three main reasons: (1) the certification process requires multiple round-trips to broadcast each block between validators, get signatures, and re-broadcast certificates. This leads to higher latency than traditional consensus protocols [14], [15], [16]; (2) blocks commit on a "per-wave" basis, which means that only once every two rounds (for Bullshark [4]) there is a chance to commit. Hence, some blocks have to wait for the wave to finish increasing the latency of transactions proposed by the block. This phenomenon is similar to committing big batches of $2f+1$ blocks. Finally, (3) since all certified blocks need to be signed by a supermajority of validators, signature generation and verification consume a large amount of CPU on each validator, which grows with the number of validators [17], [18]. This burden is particularly heavy for a crash-recovered validator that typically needs to verify thousands of signatures when trying to catch up with the rest.

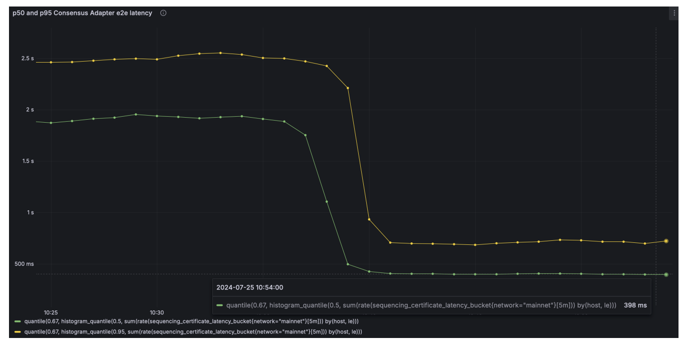


Fig. 1: P50 latency of the Sui blockchain [1] switching from Bullshark (1.9s) to Mysticeti-C (400ms) on 106 independently run validators

These shortcomings come in stark contrast to the early protocols for BFT consensus, such as PBFT [16], which require only 3 message delays to commit (instead of the 6 in Bullshark) and facilitate the pipeline of proposals to commit one every round [19]. They, however, require a high number of authenticated messages to coordinate, which consumes a lot of resources and results in low throughput. Additionally, they are fragile to faults and implementation mistakes due to their complexity, especially the view-change sub-protocols.

This work presents Mysticeti, a family of DAG-based protocols allowing to safely commit distributed transactions in a Byzantine setting that focuses on low-latency and low-CPU operation, achieving the best of both worlds. Mysticeti-C is a consensus protocol based on a threshold logical clock [20] DAG of blocks, that commits every block as early

Network and Distributed System Security (NDSS) Symposium 2025
24 - 28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.240929
www.ndss-symposium.org

as it can be decided. Mysticeti-C solves all of the above challenges as (1) it does not require explicit certificates, committing blocks within the known lower bound [21] of 3 message rounds, (2) commits every block independently and does not need to wait for the wave to finish, and (3) requires a single signature generation and verification per block, minimizing the CPU overhead.

From a production readiness point of view, the protocol tolerates crash failures without any throughput degradation and minimal latency degradation. It uses a single message type, the signed block, and a single multi-cast transmission method between validators, making it easier to understand, implement, test, and maintain. Mysticeti-C has been adopted by the Sui blockchain [1] that switched from the state-of-the-art Bullshark [4] to Mysticeti-C. Figure 1 shows the 80% latency reduction (from 1.9s to 400 ms) that happened at the moment of the deployment on a 106 validator network.

Mysticeti-FPC: supporting consensusless transactions. The power of uncertified DAGs is not limited to consensus protocols. This work generalizes Mysticeti-C to apply uncertified DAGs to BFT systems that process transactions without or before reaching consensus, such as in FastPay [22], Zef [23], Astro [24], and Sui [2]. These systems use reliable broadcast instead of consensus to commit transactions that only access state controlled by a single party.

The only operating protocol of this kind is Sui Lutris [2], which powers the open source Sui blockchain (Linera [25] is under development). Sui combines a consensusless "fast" path with a black-box certified DAG consensus. This composition is generic and leads to low latencies for fast-path transactions. But it also leads to (1) increased latencies for transactions requiring the consensus path and overall increased sync latency due to a separate post-consensus checkpoint mechanism, and (2) additional signature generation and verification for transactions to be certified separately. The latter means that the validator's CPU is largely devoted to performing cryptographic operations rather than executing transactions. To alleviate these challenges, we co-design with Mysticeti-C a fast path-enabled version called Mysticeti-FPC, leading to very low-latency commits without the need to generate an explicit certificate for each transaction. This new design inherits the benefits of lower latency and lower CPU utilization.

Contributions. We make the following contributions:

- We present Mysticeti-C, a DAG-based Byzantine consensus algorithm and its proofs of safety and liveness. Notably, it implements a commit rule where every single block can be directly committed, significantly reducing latency even when failures occur. We show it has a low commit latency and exceeds the throughput of Narwhal-based consensus. Mysticeti-C is already powering the Sui blockchain [1] with more than $1.5B of value under management and 1M Daily Active Accounts.
- We also present Mysticeti-FPC that offers feature parity with Sui Lutris [2], that is, both a fast path and a consensus path, as well as safe checkpointing and epoch close mechanisms. We show that Mysticeti-FPC has a fast path latency comparable with Zef [23] and Fastpay [22] but higher throughput thanks to lower CPU utilization and batching.
- We implement and evaluate both protocols on a wide-area network. We show their performance is superior to certified DAG-based designs both in consensus and consensusless modes due to the need for fewer messages and lower CPU overheads. We also report the experiences and performance benefits of integrating Mysticeti-C into a production blockchain.

II. Overview

This paper presents the design of the Mysticeti protocols, a pair of Byzantine Fault Tolerant (BFT) protocols based on Directed Acyclic Graphs (DAGs) that aim to achieve high performance in a partially synchronous network. Mysticeti-C is a low-latency consensus protocol that commits multiple blocks per round, while Mysticeti-FPC extends Mysticeti-C with a fast path for transactions that do not require consensus.

A. System model, goals, and assumptions

We consider a message-passing system where, in each epoch, $n = 3f + 1$ validators process transactions using the Mysticeti protocols. In every epoch, a computationally bound adversary can statically corrupt an unknown set of up to f validators. We call these validators *Byzantine* and they can deviate from the protocol arbitrarily. The remaining validators (at least $2f+1$) are *honest* and follow the protocol faithfully.

For the description of the protocol, we assume that links between honest parties are reliable and authenticated. That is, all messages among honest parties eventually arrive and a receiver can verify the sender's identity. The adversary is computationally bound hence the usual security properties of cryptographic hash functions, digital signatures, and other cryptographic primitives hold. Under these assumptions, Section V shows that the Mysticeti protocols are safe, in that, no two correct validators commit inconsistent transactions.

Validators communicate over a partially synchronous network. There exists a time called Global Stabilization Time (GST) and a finite time bound Δ, such that any message sent by a party at time x is guaranteed to arrive by time $\Delta + \max\{\mathsf{GST}, x\}$. Within periods of synchrony (after GST) the Mysticeti protocols are also live in that they are guaranteed to commit transactions from correct validators.

Following prior work [6], [4], [3] we focus on byzantine atomic broadcast for Mysticeti. Additionally for Mysticeti-FPC, we show that the fast-path transactions sub-protocol satisfies reliable broadcast within an epoch [2], but allows for recovery of equivocating objects across epochs without losing safety at the epoch boundaries.

More formally, each validator v_k broadcasts messages by calling $r\_bcast_k(m, q)$, where m is a message and $q \in \mathbb{N}$

is a sequence number. Every validator v_i has an output *r_deliver*$_i(m, q, v_k)$, where m is a message, q is a sequence number, and v_k is the identity of the validator that called the corresponding *r_bcast*$_k(m, q)$. The reliable broadcast abstraction guarantees the following properties:

- **Agreement:** If an honest validator v_i outputs *r_deliver*$_i(m, q, v_k)$, then every other honest validator v_j eventually outputs *r_deliver*$_j(m, q, v_k)$.
- **Integrity:** For each sequence number $q \in \mathbb{N}$ and validator v_k, an honest validator v_i outputs *r_deliver*$_i(m, q, v_k)$ at most once regardless of m.
- **Validity:** If an honest validator v_k calls *r_bcast*$_k(m, q)$, then every honest validator v_i eventually outputs *r_deliver*$_i(m, q, v_k)$.

Additionally, for byzantine atomic broadcast, each honest validator v_i can call *a_bcast*$_i(m, q)$ and output *a_deliver*$_i(m, q, v_k)$. A byzantine atomic broadcast protocol satisfies reliable broadcast (agreement, integrity, and validity) as well as:

- **Total order:** If an honest validator v_i outputs $a\_deliver_i(m, q, v_k)$ before $a\_deliver_i(m', q', v'_k)$, then no honest party v_j outputs $a\_deliver_j(m', q', v'_k)$ before $a\_deliver_j(m, q, v_k)$.

Finally, most prior work on consensusless transactions defines properties as if the protocol runs in a single epoch. This setting is unrealistic as it cannot accommodate recovering from equivocation, which is a common benign event for non-expert users. To this end, we extend all the protocols to also take as a parameter the epoch number and all properties should hold within a single epoch. Fortunately, the definition of reliable broadcast allows the recovery of liveness for blocked sequence numbers that are equivocated inside an epoch. Thus, we define equivocation tolerance for consesusless transactions as follows:

- **Equivocation tolerance:** If a validator v_k concurrently called *r_bcast*$_k(m, q, e)$ and *r_bcast*$_k(m', q, e)$ with $m \neq m'$ then the rest of the validators either *r_deliver*$_i(m, q, v_k, e)$, or *r_deliver*$_i(m', q, v_k, e)$, or there is a subsequent epoch $e' > e$ where v_k is honest, calls *r_bcast*$_k(m'', q, e')$ and all honest validators *r_deliver*$_i(m'', q, v_k, e')$,

B. Intuition behind the Mysticeti design

Mysticeti aims to push the latency boundaries of state machine replication in DAG-based blockchains. Achieving BFT consensus typically necessitates at least three message delays [16][1]. This underscores the inherent latency sub-optimality of Narwhal [3], that implements consensus (at least 3 message delays) on certified DAG blocks, when the block certification itself adds a further 3 message delays. Consequently, the first design challenge for Mysticeti is to manage equivocation and ensure data availability [27], without relying on pre-certification of individual blocks.

Moreover, even if we overcome this initial challenge, committing only one block every three messages falls short of the performance potential inherent in DAG-based consensus, which thrives on processing $O(n)$ blocks per round, one per validator, to fully utilize network resources. Therefore, a key objective for Mysticeti is to maximize block commitments per round to align system tail latency closely with the three-message delay. However, achieving this presents a more formidable challenge. Unlike traditional methods that rely on the recursive and elegant commit rules found in DAG-based consensus protocols [6], [3], [4], [7], [8], our approach cannot afford to require sufficient distance between two potential candidate blocks on the DAG to prevent conflicting decisions among validators with divergent sub-DAG views. Implementing such protocols would require at least one gap round, raising the latency to a minimum of four delays.

Mysticeti is not just a consensus protocol but a class of protocols facilitating state machine replication. For now, we only focused on the consensus protocol Mysticeti-C, but section IV extends it to protocols for consensus-less agreement with Mysticeti-FPC. The core contribution of Mysticeti-FPC to prior work is that it is co-designed with Mysticeti-C instead of being a separate path like in Sui [2]. This allows us to avoid the need for generating a majority-signed certificate per transaction, freeing a significant amount of network and CPU resources to be used for actual transactions instead of generating and verifying certificates [17], [18].

Given ourexperience of deploying DAG-based consensus protocols [2], there are some design challenges that relate to engineering. Bullshark [4] requires separate sub-protocols for managing individual block certification, for exchanging certified blocks, and for managing the communication of metadata between nodes. The challenge with Mysticeti is to design a protocol that has a single message type, the signed block, and a single network primitive, by which each block is multi-cast to all other correct validators.

A final point of focus inspired by our deployment is that crash-faults and struggling nodes are a common occurrence and not an exception. This is why we have designed Mysticeti to be able to tolerate crash-faults with as little performance degradation as possible.

C. The structure of the Mysticeti DAG

We present the structure of the Mysticeti DAG. Its main goal is to build an uncertified DAG protocol that provides the same guarantees as a certified DAG.

The Mysticeti protocols operate in a sequence of logical *rounds*. For every round, each honest validator proposes a unique signed *block*; Byzantine validators may attempt to equivocate by sending multiple distinct blocks to different parties or no block. During a round, validators receive transactions from users and blocks from other validators and use them as part of their proposed blocks. A block includes

[1] While some protocols, such as Zyzzyva [26], operate under optimistic assumptions, they often prove fragile in scenarios of asynchrony or faults [3], [12]. Moreover, they are unsuitable for the blockchain environment, characterized by a multitude of unreliable nodes wielding a minor fraction of the total voting power.

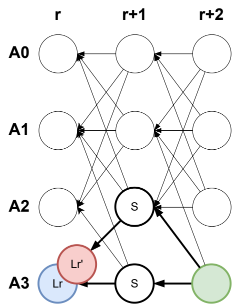


Fig. 2: Block $(A_3, r+2, \cdot)$ (green) may reference blocks from different validators that support both (A_3, r, L_r) (blue) and (A_3, r, L'_r) (red) equivocating blocks. If any of the blocks gathers $2f+1$ support, it will be certified, and we show that at most one may do so.

references to blocks from prior rounds, always starting from their most recent block, alongside *fresh transactions* not yet incorporated indirectly in preceding blocks. Once a block contains references to at least $2f+1$ blocks from the previous round, the validator signs it and sends it to other validators.

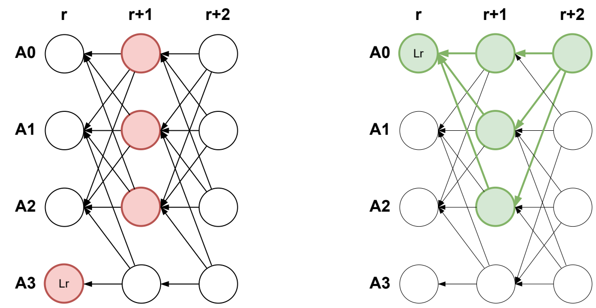
Clients submit transactions to a validator, who subsequently incorporates them into their blocks. In the event that a transaction fails to become finalized within a specified time frame, the client selects an alternative validator for resubmission.

Block correctness. A block should include at a minimum (1) the author A of the block and their signature on the block contents, (2) a round number r, (3) a list of transactions, and (4) at least $2f+1$ distinct hashes of blocks from the previous round, along potentially others from all previous rounds. By convention, the first hash must be to the previous block of A[2]. We index each block by the triplet $B \equiv (A, r, h)$, comprised of the author A, the round r, and the hash h of the block contents. A block is valid if (1) the signature is valid and A is part of the validator set, and (2) all hashes point to distinct valid blocks from previous rounds, the first block links to a block from A, and within the sequence of past blocks, there are $2f+1$ blocks from the previous round $r-1$.

Identifying DAG patterns. We say that a block B' *supports* a past block $B \equiv (A, r, h)$ if, in the depth-first search performed starting at B' and recursively following all blocks in the sequence of blocks hashed, block B is the first block encountered for validator A at round r. As Figure 2 illustrates, a block $(A_3, r+2, \cdot)$ (green) may reference blocks $(A_2, r+1, \cdot)$ and $(A_3, r+1, \cdot)$ from different validators that respectively support block (A_3, r, L_r) (blue) and the equivocating block (A_3, r, L'_r) (red). At most one of these equivocating blocks can gather support from $2f+1$ validators.

Mysticeti-C (Section III) and Mysticeti-FPC (Section IV) operate by interpreting the structure of the DAG to reach decisions using a single type of message, the block. They mainly operate by identifying the following two patterns:

[2]This rule also helps to guarantee the safety of fast pah transactions upon epoch change (Section IV-D).



(a) Illustration of *skip* pattern, blocks $(A_0, r+1, \cdot), (A_1, r+1, \cdot), (A_2, r+1, \cdot)$ do not support (A_3, r, L_r).

(b) Illustration of *certificate* pattern, block $(A_0, r+2, \cdot)$ is a certificate for (A_0, r, L_r).

Fig. 3: Illustration of main DAG patterns identified by validators.

1) The *skip pattern*, illustrated by Figure 3 (left), where at least $2f+1$ blocks at round $r+1$ *do not* support a block (A, r, h). Note that there may be multiple or no proposal for the slot. The skip pattern is identified if for all proposals, we observe 2f+1 subsequent blocks that do not support it (or support no proposal).
2) The *certificate pattern*, illustrated by Figure 3 (right), where at least $2f+1$ blocks at round $r+1$ *support* a block $B \equiv (A, r, h)$. We then say that B is *certified*. Any subsequent block (illustrated at $r+2$) that contains in its history such a pattern is called a *certificate* for the block B.

Using these patterns, we obtain certificates implicitly by interpreting the DAG, and the certification guarantees are identical to Narwhal [3]. That is, a certified block ($2f+1$ support) is available and no other certified block may exist for the same spot (A, r). This counter intuitively means that even if A equivocates and one of its blocks is certified, we process it as being correct – despite the self evident Byzantine behavior. This does not constitute a problem as we only commit blocks that belong to the implicitly certified part of the DAG. We also note that a skip pattern guarantees that a certificate will never exist for a block, and thus it will never be part of the implicitly certified DAG and can be safely skipped.

Liveness intuition. Since we are not using randomization, we need to rely on timeouts for liveness. Although every blocks has the potential of being committed directly in 3 message delays we cannot provide liveness for all of them through timeouts, as this would allow Byzantine validators to slow down the DAG to the point that every round would move at the speed of the timeout instead of network speed.

Instead we only provide guaranteed liveness after GST for one block per round[3]. We deem this block as the primary block of the round r and require that validators at $r+1$ wait a timeout for it to arrive before disseminating their blocks. Additionally, if the block is in the view of a validator at $r+1$

[3]This can be extended to more blocks but it increases the chance that the adversary controls one block causing a full delay for the round.

we further require the validator to wait another timeout for $r + 2$ or until there are $2f + 1$ votes for the primary block of r. This guarantees the existence of a certificate over an honest primary block after GST and provides liveness for Mysticeti-C.

III. The Mysticeti-C Consensus Protocol

Mysticeti-C is the first DAG-based consensus protocol that decides blocks in 3 message delays. It achieves this through foregoing an explicit certification of the blocks and through treating every block as a first-class block that can be proposed and decided directly. Additionally, Mysticeti-C is able to instantly identify and exclude crashed validators, the most frequent failure case in blockchains in the wild.

A. Proposer slots

Mysticeti-C introduces the concept of *proposer slot*. A proposer slot represents a tuple (validator, round) and can be either empty or contain the validator's proposal for the respective round. For instance, in Bullshark [4], there is a single proposer every two rounds, which results in higher latencies. Unfortunately, it is not trivial to increase the number of slots, as the commit rule of Bullshark relies on the fact that every proposer slot has a link to every other proposer slot, something that is not possible even if there is a single proposer per round, let alone n.

We overcome this challenge by introducing multiple *states* for each proposer slot, namely: to-commit, to-skip, or undecided. The to-commit state is the equivalent of the decided state that already exists in the prior work. The most important state is the undecided, which forces all subsequent proposer slots to wait, mitigating the risk of non-deterministic commitments due to network asynchrony without the need for a buffer round as prior work [6], [3], [4], [7]. Finally, the to-skip state allows to exclude proposer slots assigned to crashed validators, thus allowing the subsequent slots to commit.

The number of proposer slots instantiated per round can be configured but for systems with few faults it can be set to n so that every block has a chance to commit in 3 steps. It can also be dynamically adjusted based on the network conditions, following a similar deterministic approach to HammerHead [28] (see Appendix A). Initially, we establish a deterministic total order among all pending proposer slots, aligning with the round ordering. Within a single round, the ordering may either remain fixed or change per round (e.g., round robin). Figure 4 illustrates an example of a Mysticeti DAG with four validators, (A0, A1, A2, A3), four slots per round, and a potential proposer slot ordering represented as (L1a, L1b, L1c, L1d) and (L2a, L2b, L2c, L2d) for the first and second rounds, respectively. This order resembles a FIFO queue.

As discussed in Section II-C, validators await the proposal from the primary validator assigned to the first proposer slot of round r for up to a predetermined delay Δ before generating their own proposal for round $r + 1$. Section V shows that this delay ensures the liveness of the protocol.

B. The Mysticeti-C decision rule

This section describes the decision rule of Mysticeti-C leveraging an example protocol run. Section III-D provides detailed algorithms. As illustrated by Figure 4a, all proposer slots are initially in the undecided state. The end goal of Mysticeti-C is to mark all proposer slots as either to-commit or to-skip by detecting the DAG patterns presented in Section II-C. The Mysticeti-C decision rule operates in three steps:

Step 1: Direct decision rule. Starting with the latest proposer slot (L6d in Figure 4), the validator applies the following *direct decision rule* to attempt to determine the status of the slot. The validator marks a slot as to-commit if it observes $2f + 1$ *commit patterns* for that slot, that is, if it accumulates $2f + 1$ distinct implicit certificate blocks for it (see Section II-C). This is the **first key design point** for lowering the latency as we certify blocks while constructing the DAG by interpreting *certificate patterns*.

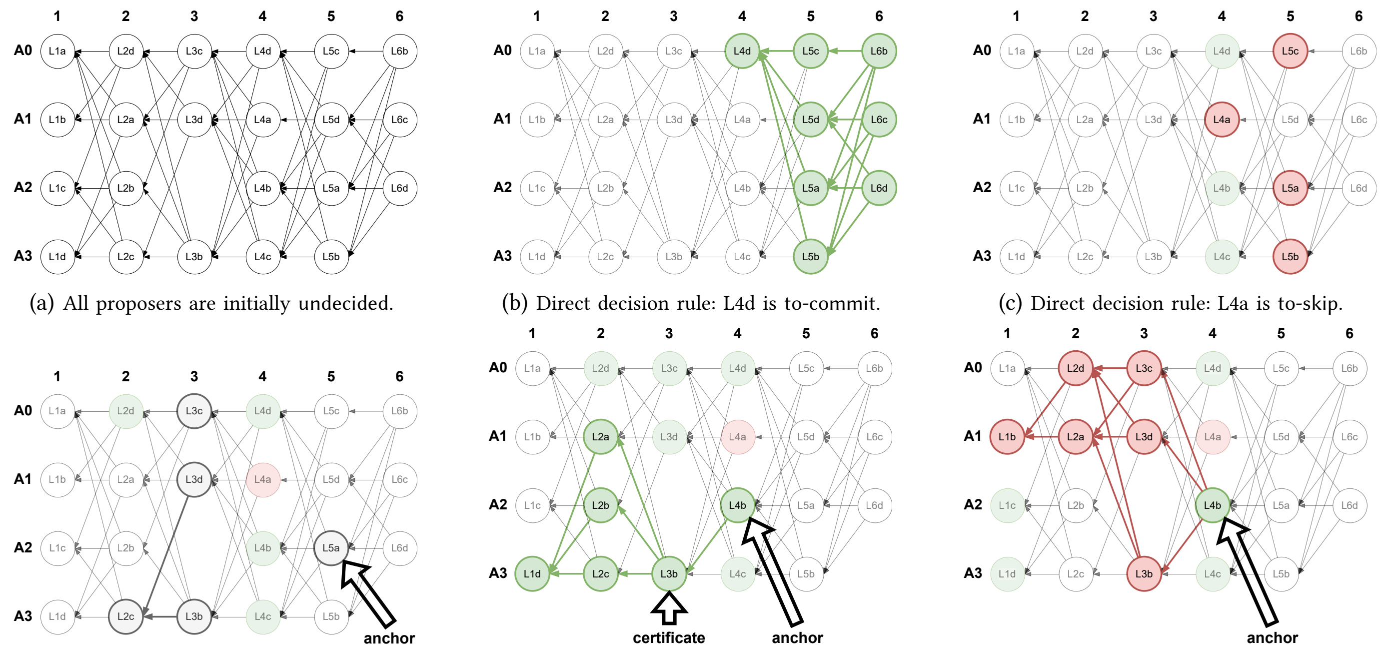
Figure 4b illustrates the direct decision rule applied to L4d, which is marked as to-commit in just 3 messages due to the presence of $2f + 1$ commit patterns. The first message delay is the proposal block; the second message delay is the block(s) supporting and voting/certification; and the third message delay is the block(s) certifying serving as acknowledgment/commitment. The direct decision rule marks a slot as to-skip if it observes a *skip pattern* for that slot. That is for any proposal for the slot (there may be multiple due to potential equivocation) it observes $2f + 1$ blocks that do not support it or support no proposal. Figure 4c demonstrates the direct decision rule applied to L4a, which is marked as to-skip due to the presence of a skip pattern.

Promptly marking slots as to-skip is the **second key design point** that contributes to the reduction of undecided slots following crash-failures and allows Mysticeti-C to tolerate crash-faults virtually for free.

If the direct decision rule fails to mark a slot as either to-commit or to-skip, the slot remains undecided and the validator resorts to the *indirect decision rule* presented in step 2 below. During normal operations, however, we expect the direct decision rule to succeed and to only resort to the indirect decision rule during periods of asynchrony or under attacks.

Step 2: Indirect decision rule. If the direct decision rule fails to determine the slot, the validator resorts to the indirect decision rule to attempt to reach a decision for the slot. This rule operates in two stages. It initially searches for an *anchor*, which is defined as the first slot with the round number $(r' > r + 2)$ that is already marked as either undecided or to-commit[4]. Figure 4d and Figure 4e respectively illustrate the anchor of L2c (marked as undecided) and the anchor of L1d (marked as to-commit).

[4]This section assumes a fixed distance of 3 rounds between a proposer slot which is the minimum secure distance. Section III-D generalize this rule to a variable distance and discusses its tradeoffs.



(a) All proposers are initially undecided.

(b) Direct decision rule: L4d is to-commit.

(c) Direct decision rule: L4a is to-skip.

(d) Indirect decision rule: L2c is undecided. Its anchor (L5a) is undecided, we cannot determine the status of L2c yet.

(e) Indirect decision rule: L1d is to-commit. Its anchor (L4b) is to-commit and there's a certified link from L4b to L1d.

(f) Indirect decision rule: L1b is to-skip. Its anchor (L4b) is to-commit and no certified link (only two links in round 2) from L4b to L1b.

Fig. 4: Example application of the Mysticeti-C decision rule with four validators (A0, A1, A2, A3) and four proposer slots per round.

If the anchor is marked as undecided the validator marks the slot as undecided (Figure 4d). Conversely, if the anchor is marked as to-commit, the validator marks the slot either as to-commit if the anchor causally references a certificate pattern over the slot or as to-skip in the absence of a certificate pattern. Figure 4e illustrates the indirect decision rule applied to L1d, which is marked as to-commit due to the presence of a certificate pattern linking L4b to L1d. On the other hand, Figure 4 demonstrates the indirect decision rule applied to L1b, which is marked as to-skip due to the absence of a certificate pattern linking L4b to L1b.

This is the **third key design point** contributing to the safety of Mysticeti-C without the need for links between proposers. Namely, instead of forcing a direct happened-before relationship between proposer slots, we take advantage of the predefined total ordering of proposer slots to ensure that any decision is recursively carried forward such that no matter the commit pattern, the commit decisions are deterministic.

Step 3: Commit sequence. After processing all slots, the validator derives an ordered sequence of slots. Subsequently, the validator iterates over that sequence, committing all slots marked as to-commit and skipping all slots marked as to-skip. This iteration continues until the first undecided slot is encountered. Section V demonstrates that this commit sequence is safe and that eventually all slots will be classified as either to-commit or to-skip. In the example depicted in Figure 4, the commit sequence is L1a, L1c, L1d, L2a. Appendix A provides a detailed walkthrough of the decision rule applied to the example DAG of Figure 4.

This is the **final key design point** of Mysticeti-C; unlike prior work that commits everything the moment a decision rule exists, Mysticeti-C applies some backpressure through undecided slots to preserve safety. This, however, does not harm performance, as these undecided slots would have not even existed as possible commit candidates in prior designs.

C. Choosing the number of proposer slots

The example presented by Figure 4 assumes a number of proposer slots per round equal to the committee size. While this choice offers the best latency under normal conditions, it may impact performance during periods of extreme asynchrony or under Byzantine attack.

In these cases, the probability that the direct decision rule fails to classify a proposer slot increases when some proposer slots are slow or equivocate. This forces the validator to resort to the indirect decision rule more often. As a result, there can be an increase in the number of undecided slots, which in turn delays the commit sequence. Figure 4 illustrates this example through the classification of L2c and L1b as undecided, preventing the exemplified protocol execution from immediately committing L2d, L3b, L3c, L3d, L4b, L4c, and L4d, which would have been possible under ideal conditions. This is nevertheless an extreme case of the adversary controlling the network and some validators only to slow down the system without any actual profit. After a decade of running blockchains in the wild, this is not something that has been witnessed, as attackers tend to attack in order to break safety and not liveness.

Nevertheless, in order to mitigate it we use HammerHead [28] in order to select $2f+1$ leaders that are best performing as candidate leaders. This strikes a good balance as it does not increase the median latency and only increases

Algorithm 1 Helper functions

procedure GetProposerBlock(w)
  $r_{proposer} \leftarrow$ ProposerRound(w)
  $id \leftarrow$ GetPredefinedProposer($r_{proposer}$)
  if $\exists b \in DAG[r_{proposer}]$ s.t. $b.author = id$ **then return** b
  return $\perp$

procedure GetFirstVotingBlocks(w)
  $r_{voting} \leftarrow$ ProposerRound(w) $+ 1$
  return $DAG[r_{voting}]$

procedure GetDecisionBlocks(w)
  $r_{decision} \leftarrow$ DecisionRound(w)
  return $DAG[r_{decision}]$

procedure Link(b_{old}, b_{new})
  return exists a sequence of $k \in \mathbb{N}$ blocks $b_1, \ldots, b_k$ s.t. $b_1 = b_{old}, b_k = b_{new}$ and $\forall j \in [2, k] : b_j \in \bigcup_{r \geq 1} DAG[r] \wedge b_{j-1} \in b_j.parents$

procedure IsVote($b_{vote}, b_{proposer}$)
  function SupportedBlock(b, id, r)
    if $r \geq b.round$ **then return** $\perp$
    for $b' \in b.parents$ **do**
      if $(b'.author, b'.round) = (id, r)$ **then return** b'
      $res \leftarrow$ SupportedBlock(b', id, r)
      if $res \neq \perp$ **then return** res
    return $\perp$
  $(id, r) \leftarrow (b_{proposer}.author, b_{proposer}.round)$
  return SupportedBlock(b_{vote}, id, r) $= b_{proposer}$

procedure IsCert($b_{cert}, b_{proposer}$)
  $res \leftarrow |\{b \in b_{cert}.parents : \text{IsVote}(b, b_{proposer})\}|$
  return $res \geq 2f + 1$

procedure SkippedProposer(w)
  $r_{proposer} \leftarrow$ ProposerRound(w)
  $id \leftarrow$ GetPredefinedProposer($r_{proposer}$)
  $B \leftarrow$ GetFirstVotingBlocks(w)
  $res \leftarrow |\{b \in B \text{ s.t. } \forall b' \in b.parents : b'.author \neq id\}|$
  return $res \geq 2f + 1$

procedure SupportedProposer(w)
  $b_{proposer} \leftarrow$ GetProposerBlock(w)
  $B \leftarrow$ GetDecisionBlocks(w)
  if $|\{b' \in B : \text{IsCert}(b', b_{proposer})\}| \geq 2f + 1$ **then**
    return $b_{proposer}$
  return $\perp$

procedure CertifiedLink($b_{anchor}, b_{proposer}$)
  $w \leftarrow$ WaveNumber($b_{proposer}.round$)
  $B \leftarrow$ GetDecisionBlocks(w)
  return $\exists b \in B$ s.t. IsCert($b, b_{proposer}$) & Link(b, b_{anchor})

Algorithm 2 DirectDecider Algorithm

`waveLength` ▷ Defaults to 3
`roundOffset`
`proposerOffset`

procedure TryDirectDecide(w)
  if SkippedProposer(w) **then return** `Skip`(w)
  $b_{proposer} \leftarrow$ SupportedProposer(w)
  if $b_{proposer} \neq \perp$ **then return** `Commit`($b_{proposer}$)
  return $\perp$

procedure WaveNumber(r)
  return ($r -$ `roundOffset`)/`waveLength`

procedure ProposerRound(w)
  return $w *$ `waveLength` $+$ `roundOffset`

procedure DecisionRound(w)
  return $w *$`waveLength`$+$`waveLength`$-1+$`roundOffset`

procedure GetPredefinedProposer(w)
  $r_{proposer} \leftarrow$ ProposerRound(w)
  return PredefinedProposer($r_{proposer} +$ `ProposerOffset`)

Algorithm 3 Mysticeti-C

`committeeSize`
`waveLength` ▷ Defaults to 3
`numOfProposers` ▷ Set to 2 in Section VII

procedure TryDecide($r_{committed}, r_{highest}$)
  $sequence \leftarrow [\,]$
  for $r \in [r_{highest}$ down to $r_{committed} + 1]$ **do**
    for $l \in [$`numOfProposers` $- 1$ down to $0]$ **do**
      $i \leftarrow r$ % `waveLength`
      $c \leftarrow$ DirectDecider(`waveLength`$, i, l$)
      $w \leftarrow c$.WaveNumber(r)
      if c.ProposerRound(w) $\neq r$ **then** `continue`
      $status \leftarrow c$.TryDirectDecide(w)
      if $status = \perp$ **then**
        $status \leftarrow$ TryIndirectDecide($c, w, sequence$)
      $sequence \leftarrow status || sequence$
  $decided \leftarrow [\,]$
  for $status \in sequence$ **do**
    if $status = \perp$ **then** break
    $decided \leftarrow decided || status$
  return $decided$

procedure TryIndirectDecide($c, w, sequence$)
  $r_{decision} \leftarrow c$.DecisionRound($w$)
  $anchors \leftarrow [s \in sequence \text{ s.t. } r_{decision} < s.round]$
  for $a \in anchors$ **do**
    if $a = \perp$ **then return** $\perp$
    if $a =$ `Commit`(b_{anchor}) **then**
      $b_{proposer} \leftarrow c$.GetProposerBlock($w$)
      if c.CertifiedLink($b_{anchor}, b_{proposer}$) **then**
        return `Commit`($b_{proposer}$)
      else
        return `Skip`(w)
  return $\perp$

the expected latency by $\frac{1}{3}$ of a delay. Section III-D provides detailed Mysticeti-C algorithms that allow the number of proposer slots per round to be configurable.

D. Mysticeti-C Algorithms

This section presents the detailed algorithms of Mysticeti-C. It can be skipped if a high-level understanding is sufficient.

Algorithm 1 provides base utility functions common to many DAG-based consensus protocols [3], [6], [4]. The function PredefinedProposer($\cdot$) of Algorithm 2 is a determinist leader election function, such as round robin. Mysticeti-C has one type of message; the block and its validity rules are described in Section II-C. Every node simply proposes blocks for every round, and the validity rules make sure this happens at a beneficial pace.

Algorithm 3 presents the Mysticeti-C algorithm that is run every time a valid block is received. Mysticeti-C is instantiated with the following parameters:. (1) The committee size `committeeSize`. (2) The wavelength `wave_lenght`, which the description of Section III assumes to always equal 3. A larger wavelength parameter increases the probability of observing a certificate pattern (Section II-C) over proposer slots during periods of asynchrony but increases the median latency during periods of network synchrony. (3) The number of proposer slots per round, which the example depicted by Figure 4 of Section III assumes to equal the committee size.

The entry point of this algorithm is the procedure TRYDECIDE$(\cdot)$ (Line 4). It operates by instantiating a Direct Decider (Algorithm 2) for each possible proposer slot in each round that applies the direct decision rule (Line 9). Each Direct Decider instance is instantiated with a round offset `roundOffset` $= r$ and a proposer offset `proposerOffset` $= l$, such that each instance operates over a unique proposer slot. These instances try to apply the direct decision rule to their proposer slot by calling the procedure TRYDIRECTDECIDE$(\cdot)$ (Line 12). If the direct decision rule fails, Algorithm 3 resorts to the indirect decision rule (Line 14). The algorithm returns the commit sequence.

IV. THE MYSTICETI-FPC FAST PATH PROTOCOL

For workloads necessitating consensus, the MYSTICETI-C protocol successfully achieves a low latency bound. However, popular workloads [29] such as asset transfers, payments or NFT minting, can be finalized before consensus, through and even lower latency fast path. This section presents MYSTICETI-FPC that extends the consensus protocol with such consensusless transactions.

A. Embedding a fast path into the DAG

The real-world deployment of such hybrid blockchains, exemplified by Sui [2], [1], capitalizes on the insight that certain objects, like coins, solely access state controlled by a single party and need not undergo consensus. These objects can be finalized through a fast path utilizing reliable broadcast. Such objects are classified as having an *owned object* type as opposed to the traditional *shared object* type. Transactions that exclusively involve owned objects as inputs are called *fast path transactions*. Two transactions *conflict* if they take as input the same owned object.

In MYSTICETI-FPC validators include transactions, and explicitly *vote* for causally past transactions, in their blocks. A validator includes a transaction T in its block if it does not conflict with any other transaction for which the validator has previously voted. This is also an implicit vote for the transaction. Other validators, include explicit votes for T in a block B if: (i) T is present in the causal history of B; and (ii) T does not conflict with any other already voted on transaction. In our implementation (Section VI), we denote the vote for a transaction T appearing in block B at position i as the tuple (B, i). Once T has $2f + 1$ votes from distinct validators, we call T *certified*. It is a guarantee that no two conflicting transaction will be certified in the same epoch. This is the basis of the fast path safety. Transaction T is finalized when either (i) there exists $2f + 1$ validators supporting a certificate over T, even before a MYSTICETI-C commit, or (ii) MYSTICETI-C commits through consensus a block that contains a certificate over T in its causal history (see Section IV-B).

In contrast to previous approaches [2], [22], [30], [24], the fast path in MYSTICETI-FPC is integrated within the DAG structure itself. This eliminates the need for additional protocol messages and for validators to individually sign each fast-path transaction. Instead, a validator's fast path votes are embedded within its signed blocks, which are already produced as part of the consensus protocol. Consequently, in addition to the block contents of MYSTICETI-C, blocks in MYSTICETI-FPC also incorporate explicit votes for transactions involving at least one owned object input. This deep embedding in the DAG additionally simplifies checkpoints [2] as it does not require an external sub-protocol to collect all fast-path transactions that have been finalized. Instead, MYSTICETI-FPC simply defines checkpoints as the set of finalized fast path transactions referenced by the causal history of each MYSTICETI-C commit. These can then be used to make sure that all validators have the same state for an epoch change.

To summarize, MYSTICETI-FPC offers several advantages compared to prior work: (i) A reduction in the number of signature generation and verification operations alleviating the compute bottleneck. (ii) Elimination of a separate post-consensus checkpointing mechanism, resulting in reduced synchronization latency, as the consensus commits themselves serve as checkpoints. (iii) Simplification of the epoch close mechanism, as we examine next.

B. Execution and finality

Similarly to Sui [2], MYSTICETI-FPC introduces the distinction between *fast path execution* and *fast path finality*. The former refers to the moment when a transaction is executed by a validator, the execution effects are known, and the validator can execute subsequent transactions over the same object. The latter signifies when a transaction is considered final, ensuring persistence across epoch boundaries and validator reconfigurations.

Fast path execution. A validator can safely execute a fast path transaction once it observes blocks from $2f + 1$ validators that include a vote for the transaction. Due to quorum intersection, no correct validator will ever execute conflicting fast path transactions. Figure 5 illustrates a DAG pattern enabling the validator to safely execute fast path transactions T_1 and T_3. The blocks $(A_0, r, \cdot)$ contain the fast path transactions T_1, T_3, and T_6, while the blocks $(A_0, r+1, \cdot)$, $(A_1, r+1, \cdot)$, and $(A_2, r+1, \cdot)$ support (A_0, r, L_r) and explicitly vote for T_1 and T_3 (but not for T_6[5]). Upon observing these blocks, the validator can safely execute T_1 and T_3. Note that MYSTICETI-FPC transaction execution can be extremely low-latency,

[5]Transaction T_6 may conflict with another transaction for which the validator already voted.

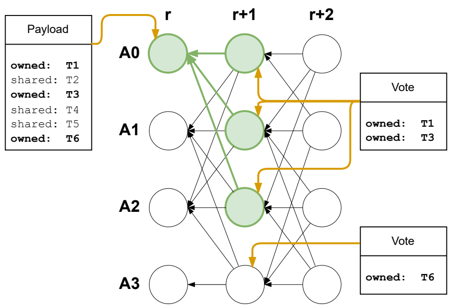


Fig. 5: Illustration fast path transaction execution. The blocks $(A_0, r, \cdot)$ contain the fast path transactions T_1, T_3, and T_6. Blocks $(A_0, r+1, \cdot)$, $(A_1, r+1, \cdot)$, $(A_2, r+1, \cdot)$ support (A_0, r, L_r) and explicitly vote for T_1 and T_3 (but not T_6). Upon observing these blocks, the validator can safely execute T_1 and T_3.

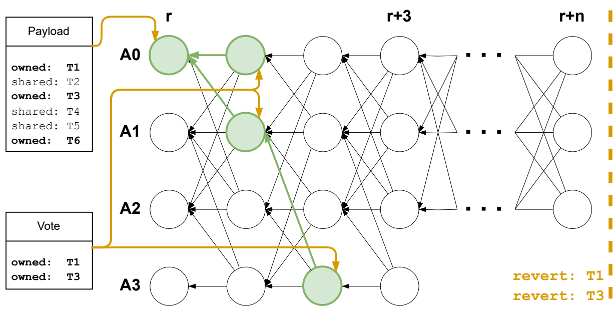
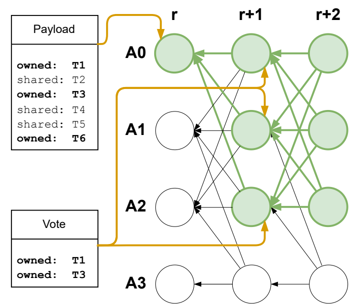
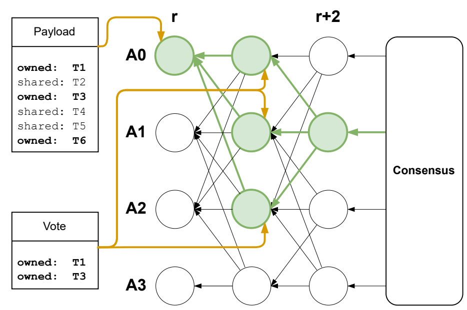


Fig. 6: Illustration of the scenario where transactions T_1 and T_3 are executed by validator A_3 at round $r+2$ but no other validator observes sufficient votes to execute those transactions, and validator A_3 reverts their execution upon epoch change.



(a) Transactions T_1 and T_3 proposed by $(A_0, r, \cdot)$ are finalized at round $r+2$ upon observing the $2f+1$ certificate pattern defined by $(A_0, r+2, \cdot)$, $(A_1, r+2, \cdot)$, and $(A_3, r+2, \cdot)$, referencing the $2f+1$ blocks $(A_0, r, \cdot)$, $(A_1, r, \cdot)$, and $(A_3, r, \cdot)$ that explicitly vote for T_1 and T_3.



(b) Transactions T_1 and T_3 proposed by $(A_0, r, \cdot)$ are finalized after consensus upon committing block $(A_1, r+2, \cdot)$. This block defines a certificate pattern over $(A_0, r, \cdot)$ that contains $(A_0, r+1, \cdot)$, $(A_1, r+1, \cdot)$, and $(A_3, r+1, \cdot)$ that vote for T_1 and T_3.

Fig. 7: Illustration of the two fast path transaction finalization scenarios.

requiring only a single round of communication, as opposed to the 2 rounds required by related work [2], [22], [24], [30].

Fast path finality. Transactions executed by some honest validators can still be reverted since there is no guarantee that other validators will eventually observe sufficient evidence to execute the transaction. For instance, Figure 6 illustrates a scenario where transactions T_1 and T_3 are executed by validator A_3 at round $r+3$, but no proposals from that validators are included into the DAG for rest of the epoch, possibly due to network asynchrony. Consequently, no other validator observes sufficient evidence to execute those transactions, and validator A_3 reverts their execution upon epoch change. Note that reverting execution is a straightforward operation and already supported by the Sui protocol, the only blockchain deploying a fast path.

To ensure that the effects of a fast path transaction endure across epoch boundaries and validator reconfiguration, it must be *finalized*. A fast path transaction is finalized when the validator observes either (1) $2f+1$ certificate patterns over the block proposing the transaction (as detailed in Section II-C), each containing $2f+1$ votes for the transaction, or (2) a single certificate pattern over the block proposing the transaction, which includes $2f+1$ votes for the transaction and is referenced in the causal history of a block committed by the consensus protocol. Figure 7 illustrates these two possible finality pattern for fast path transactions T_1 and T_3.

The finality of a fast-path transaction across epochs is proven by Theorem 5 of Section V. Additionally, Section IV-C outlines how Mysticeti-FPC accommodates transactions containing both owned object and non-owned object inputs.

C. Mixed-objects transactions

Mysticeti-FPC allows for transactions that contain both owned-object and non-owned-object inputs. Such transactions are called *mixed-objects transactions*. Validators execute and finalize these transactions upon observing (1) blocks from $2f+1$ validators that include a vote for the transaction, and (2) a block committed by the consensus protocol referencing these blocks in its causal history.

Figure 8 provides an example illustrating the finalization of a mixed-object transaction. This mechanism intuitively operates in two steps: first, it "locks" the owned-object inputs, and then sequences this lock to prevent the execution of potentially conflicting owned-object transactions. The safety of this approach is guaranteed by Theorem 6 of Section V.

D. Epoch change and reconfiguration

As mentioned in Section III, quorum-based blockchains typically operate in epochs, allowing validators to join and leave the system at epoch boundaries. Moreover, epoch boundaries serve as natural boundaries for protocols with

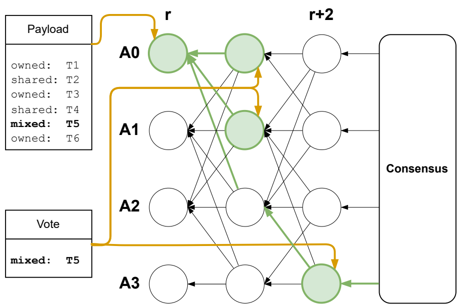


Fig. 8: Illustration of a mixed-objects transaction T_5 that contains both owned-object inputs and non-owned-object inputs. T_5 is proposed as part of $(A_0, r, \cdot)$. Blocks $(A_0, r_1, \cdot)$, $(A_1, r+1, \cdot)$, and $(A_3, r+2, \cdot)$ vote for T_5. The validator can execute and finalize T_5 once block $(A_3, r+2, \cdot)$ is committed by the consensus protocol.

a consensusless path to "unlock" transactions that have lost liveness due to equivocation from the client [2], [31]. This committee reconfiguration process must uphold a critical safety property: transactions finalized in an epoch should persist across subsequent epochs. In other words, transactions finalized in the current epoch should not conflict with transactions committed in future epochs. This holds trivially for consensus protocols which is why we omit the epoch change for Mysticeti-C.

The Mysticeti-FPC epoch-change protocol. The safety of reconfiguration is ensured by including all finalized transactions from the current epoch into the causal history of the epoch's final commit, which also acts as the initial state for the succeeding epoch. Guaranteeing reconfiguration safety is straightforward in systems mandating consensus for all transactions, such as Mysticeti-C, owing to the total ordering property inherent in consensus. A deterministic consensus commit C sets the boundary between epochs e and $e+1$. This makes sure that all transactions completed in epoch e are included in and come before commit C.

However, designing reconfiguration mechanisms for systems with a consensusless fast path, like Mysticeti-FPC, presents non-trivial challenges. There is a race between finalized transactions being incorporated into consensus commits and new transactions being finalized by the fast path. Trivially closing the epoch may result in the final commit of the epoch failing to encompass all transactions finalized by the fast path, thereby violating the safety property of reconfiguration.

To solve this challenge, Mysticeti-FPC introduces an overriding bit called the *epoch-change bit* in all its blocks. When this bit is set to 1 (default set to 0), it signifies that blocks referencing these votes do not contribute to the finalization of fast path transaction, irrespective of its causal history. Effectively, this epoch-change bit allows for the pause of the consensusless fast path of Mysticeti-FPC near the end of the epoch, mitigating the race condition highlighted above.

Epoch change starts at a predefined commit, often signaled by a higher-layer logic (e.g., a smart contract) indicating the readiness of the new committee to take charge. Once an honest validator detects the commencement of epoch change, it ceases to include transactions and to cast votes for any fast-path transactions. Subsequently, it sets the epoch-change bit to 1 in all its future blocks for the current epoch. Furthermore, while the validator continues to progress through rounds and participate in consensus, it stop processing and finalizing fast-path transactions. Upon committing blocks from $2f+1$ validators with the epoch-change bit set via the consensus path, the epoch is considered closed.

Once the epoch ends, any validator participating the committee of the next epoch may unlock fast-path transactions that were blocked due to client equivocations. These transactions can then receive fresh votes in subsequent epochs.

Security intuition. The epoch-change mechanism ensures that transactions finalized in an epoch (including on the fast path before consensus) persist across all subsequent epochs, a critical safety property (more formally in Theorem 5). Informally, by committing $2f+1$ blocks with the epoch-change bit set, we guarantee that every transaction finalized via the fast-path would have a certificate as part of the causal history of the epoch-change commit (due to a quorum intersection argument). Consequently, all validators process the certificate before they end of the epoch and persist execution results across epochs.

The liveness of Mysticeti-FPC directly depends on the liveness of Mysticeti-C. Informally, if the epoch is long enough, a non-conflicting transaction will gather sufficient votes, and then be certified by $2f+1$ blocks with the epoch-change bit unset. Which in turn ensures that it will be included in a commit and persisted across epochs. Section V-B formally proves the safety and liveness of Mysticeti-FPC.

V. Mysticeti Security

We argue the security of Mysticeti-C and Mysticeti-FPC under the Byzantine assumption presented in Section II.

A. Security of Mysticeti-C

This section argues the safety, liveness, and integrity of Mysticeti-C.

Safety of Mysticeti-C. A validator v_k broadcasts messages calling $a\_bcast_k(b, r)$, where b is a block signed by validator v_k and r is the block's round number, i.e., $r = b.round$. Every validator v_i has an output $a\_deliver_i(b, b.round, v_k)$, where v_k is the author of b and the validator that called the corresponding $a\_bcast_k(b, b.round)$.

Lemma 1. *If at a round x, $2f+1$ blocks from distinct authorities certify a block B, then all blocks at future rounds ($> x$) will link to a certificate for B from round x.*

Proof. Each block links to $2f+1$ blocks from the previous round. For the sake of contradiction, assume that a block in round $r(> x)$ does not link to a certificate from round x. If $r = x+1$, by the standard quorum intersection argument, a correct validator equivocated in round x, which is a contradiction. Similarly, if $r > x + 1$, by the standard quorum intersection argument, a correct validator's block in round $r - 1$ does not link to its own block in round x, which is also a contradiction. □

Lemma 2. *If a correct validator commits some block in a slot s, then no correct validator decides to directly skip the slot s.*

Proof. A validator X decides to directly skip a slot s if there is no support during the support rounds for any block corresponding to s. If another validator committed some block b for slot s, at least $f + 1$ correct validators supported b. By the quorum intersection argument, X must have observed at least one validator supporting B, which is a contradiction. □

Lemma 3. *If a correct validator directly commits some block in a slot s, then no correct validator decides to skip s.*

Proof. For the sake of contradiction, assume that a correct validator X directly commits block b in slot s while another correct validator Y decides to skip the slot. Y can decide to skip the slot s in one of two ways: (a) Y directly skipped s because there was no support during the support rounds for any block corresponding to s, or (b) Y skipped s during the recursive commits triggered by a direct commit of a later slot.

Case (a). Direct contradiction of Lemma 2.

Case (b). Let block b' denote the proposer block, committed during the recursive indirect commits, that allowed Y to decide s as skipped. Due to the commit rule, the round number of b' is greater than the decision round of s, and b' does not link to a certificate for b. Since X committed b, there are $2f + 1$ certificates for b in its decision round, leading to a contradiction due to Lemma 1. □

Lemma 4. *For any slot $s \equiv (v, r)$, a correct validator never supports two distinct block proposals from validator v in round r across all of its blocks.*

Proof. By definition, a block can only support at most a single proposal for a particular slot s. Block support is calculated through a depth-first traversal of the referenced blocks, such that the first block corresponding to s encountered during the traversal is supported. Since a correct validator first includes a reference to its own block from the previous round, once a correct validator supports a certain block for s, it continues to support the same block in all of its future blocks. □

Lemma 5. *For any slot, at most a single block will ever be certified, i.e. gather a quorum ($2f + 1$) of support.*

Proof. For contradiction's sake, assume that two distinct block proposals for a slot gather a quorum of support. By the standard quorum intersection argument, a correct validator supports two distinct blocks for the same slot, which is a contradiction of the proved Lemma 4. □

As a result of Lemma 5, we get the following corollary:

Corollary 1. *No two correct validators commit distinct blocks for the same slot.*

Lemma 6. *All correct validators have a consistent state for each slot, i.e. if two validators have decided the state of a slot, then both either commit the same block or skip the slot.*

Proof. Let $[x_i]_{i=0}^{n}$ and $[y_i]_{i=0}^{m}$ denote the state of the slots for two correct validators X and Y, such that n and m are respectively the indices of the highest committed slot. WLOG $n \leq m$. Any slot decided by X higher than n are direct skips and are therefore consistent with Y due to Lemma 2. We now prove, by induction, statement $P(i)$ for $0 \leq i \leq n$: if X and Y both decide the slot i, then both either commit the same block or skip the slot.

Base Case: $i = n$. X directly commits slot i, the highest committed slot for X. From Lemma 3, if Y decides slot i, then it must also commit slot i. By Corollary 1, Y commits the same block.

Assuming $P(i)$ is true for $k + 1 \leq i \leq n$, we now prove $P(k)$. Similar to the base case, if one validator decides to directly commit a block in slot k, then the other validator, if it also decides slot k, decides to commit the same block. If one validator decides to directly skip slot k, then the other validator, if it also decides slot k, decides to skip due to Lemma 2. We now analyze the only remaining case where X and Y indirectly decide the slot k. Let k' denote the first slot $> k$ with a round number higher than the decision round of k. There exist slots $k_x (\geq k')$ and $k_y (\geq k')$ such that X commits block b_x in k_x while skipping all slots in $[k', k_x)]$ and Y commits block b_y in k_y while deciding to skip all slots in $[k', k_y)]$. As $k_x \leq n$, it follows from the induction hypothesis that $k_x = k_y$ and $b_x = b_y = b$. Since the indirect decision of X and Y for slot k depends entirely on the causal history of the same block b, both validators decide the slot k identically. □

Lemma 7. *All correct validators commit a consistent sequence of proposer blocks (i.e., the committed proposer sequence of one correct validator is a prefix of another's).*

Proof. The committed sequence of proposer blocks is nothing but the sequence of committed blocks before the first undecided slot. The statement is then a direct implication of Lemma 6. □

Theorem 1 (Total Order)**.** *Mysticeti-C satisfies the total order property of Byzantine Atomic Broadcast.*

Proof. Correct validators deliver blocks by using an identical deterministic algorithm to order the causal history of committed proposer blocks. Since a correct validator has all the causal histories of a block when the block is added to its DAG, and the sequence of committed proposer blocks of one validator is a prefix of another's (Lemma 7), all correct validators deliver a consistent sequence of blocks, i.e., the sequence of blocks delivered from one validator is a prefix of the sequence delivered by any other validator. The total order property of BAB immediately follows. ☐

Liveness of Mysticeti-C. We show the liveness of Mysticeti-C under partial synchrony (Section II).

Lemma 8 (Round-Synchronization)**.** *After GST all honest parties will enter the same round within Δ.*

Proof. After GST all messages sent before GST deliver within Δ. This means that if r is the highest round any honest validator proposed a block for before GST, then every honest validator will receive the block proposal of the honest validator at $GST + \Delta$ and also enter r. ☐

Lemma 9 (Leader-Proposal)**.** *After GST an honest proposer's proposal will get votes from every honest validator.*

Proof. After GST if an honest validator enters wave w, then it has to broadcast the last block of wave $w - 1$. Within Δ the honest proposer (and every other honest party) will receive the block and adopt the parents, being able to also enter wave w as they are all synchronized (Lemma 8). Then the honest proposer will directly propose its block. Since the timeout is set to $2 \cdot \Delta$ the proposer's block of wave w will arrive before the first honest validator times out hence, every honest validator will vote for the proposer. ☐

Lemma 10 (Sufficient Votes)**.** *After GST all honest validators will create a certificate for the honest proposer.*

Proof. By Lemma 9 all honest validators will vote for an honest proposer after GST. For an honest validator to propose a block at the decision round it needs to (a) get the proposal of the proposer and (b) have $2f + 1$ parents. All honest validators receive the proposer proposal within Δ since the proposer is honest. Additionally once a honest validator advances to the decision round all honest validators will receive its block proposal and adopt the parents within Δ. Consequently, by construction, honest validators wait for $2 \cdot \Delta$ before giving up the certificate creation and will receive the votes from all honest validators witnessing a certificate ☐

Lemma 11. *The round-robin schedule of proposers in Mysticeti ensures that in any window of $3f + 3$ rounds, there are three consecutive rounds with honest primary proposers. A primary proposer is the proposer of the first slot of a round.*

Proof. There are $3f + 1$ groups of three consecutive rounds. Due to the round-robin schedule, each of the honest validators must be the primary proposer in exactly 3 of these groups. As there are $2f + 1$ honest validators, due to the pigeonhole principle, one group must contain $\lceil \frac{3*(2f+1)}{3f+1} \rceil$ = 3 honest proposers. ☐

Lemma 12. *After GST any undecided slot eventually gets decided.*

Proof. Let there be an undecided slot s in round r. After GST, due to Lemma 11, there will eventually be an honest proposer for the first slots s_0, s_1 and s_2 of rounds $k, k + 1$ and $k + 2$ respectively, where $k > r$. By Lemma 10, the honest proposer's blocks will have $2f + 1$ certificates and be scheduled for a commit. We now prove that by induction, all slots in round $\leq k - 1$ get decided. In the base case, any undecided slots in rounds $k - 3, k - 2$ or $k - 1$ get decided by the commits in slots s_0, s_1 and s_2 respectively, as they are the first slots higher than the respective decision rounds. For the induction step, any undecided slot s in round $x \leq k - 4$ also gets decided since s_0 is higher than the decision round of x and there are no undecided slots between s and s_0 (induction hypothesis). ☐

Theorem 2 (Consensus Liveness)**.** *After GST the proposal of an honest proposer will commit.*

Proof. By Lemma 10 there will be $2f + 1$ certificates for the proposer, one per honest party. By the code an honest validator tries to commit the proposer for every block they get so eventually they will get the $2f + 1$ certificates. The validator schedules the block to be committed. By Lemma 12, all prior undecided blocks will eventually be decided, and the validator will deliver the honest proposer's block. ☐

Theorem 3 (Agreement)**.** *Mysticeti-C satisfies the agreement property of Byzantine Atomic Broadcast.*

Proof. If a correct validator outputs $a\_deliver_i(b, r, v_k)$, then it must have committed a sequence of proposer blocks $L = l_0, l1...l_n$ such that the deterministic algorithm to deliver blocks from the sequence L delivers block b. Another correct validator Y that has not delivered b will eventually see a proposal b' from an honest proposer in round $r' > r$ as per the proposer schedule of Mysticeti-C. Due to Theorem 2, after GST, Y will commit the proposer's block b'. Due to Lemma 7, Y will also commit the proposer sequence L before committing b'. Since Y follows an identical deterministic algorithm as X to deliver blocks from the committed sequence of proposer blocks, it also delivers b' eventually. ☐

Integrity of Mysticeti-C. Mysticeti-C guarantees integrity by construction.

Theorem 4 (Integrity)**.** *Mysticeti-C satisfies the integrity property of Byzantine Atomic Broadcast.*

Proof. The algorithm linearizing the causal history of a committed proposer block removes any block with duplicate sequence numbers before delivering the sequence of blocks. ☐

B. Security of Mysticeti-FPC

We argue the safety and liveness of Mysticeti-FPC.

Theorem 5 (Epoch close safety)**.** *Transactions finalized in an epoch continue to persist in all subsequent epochs.*

Proof. It is sufficient to prove that all fast-path transactions that are considered final have one certifying block committed in the current epoch. For contradiction's sake, assume that the epoch closed before any certifying block for a finalized transaction tx could be committed. For the epoch to close, blocks from $2f+1$ validators with the epoch-change bit set must be committed. Since tx is finalized, $2f+1$ validators, by definition, publish a block that certifies the transaction. By quorum intersection, one honest validator v published a block B_1 in round r_1 certifying transaction tx, whereas a block B_2 in round r_2 from v with epoch-change bit set must have been committed. All blocks published by v in rounds $\geq r_2$ also have the epoch-change bit set. Because blocks with the epoch-change bit set, by definition, do not certify any transaction, B_1 is necessarily published in an earlier round than that of B_2 (i.e. $r_1 < r_2$). B_1 is therefore contained in the causal history of B_2, and must also have been committed, which is a contradiction. □

Theorem 6 (Mysticeti-FPC Safety)**.** *An honest validator in Mysticeti-FPC never finalizes two conflicting transactions.*

Proof. Transactions that have an owned object as input require votes from $2f+1$ validators to be finalized. If two conflicting fast paths are finalized, an honest validator must have voted for both transactions (by quorum intersection), hence a contradiction. Using a similar argument, a fast path transaction does not conflict with a consensus path transaction, as the consensus path in Mysticeti-FPC finalizes a transaction with an owned object input only if it has votes from $2f+1$ validators. □

Theorem 7 (Fast-Path Liveness)**.** *An honest fast-path transaction will commit after GST.*

The proof is the same as consistent broadcast. We do it after GST assuming the epoch does not end. If the epoch has infinite length then we can convert all references to Δ with "eventually" and the proof will work in asynchrony.

Proof. An honest validator will submit a fast-path transaction that does not have equivocation. As a result, all honest validators will receive it after Δ and vote. These votes will appear in the DAG after at most $4 \cdot \Delta$ since any round has at most duration of timeout+Δ = $3 \cdot \Delta$. In the next round, every honest validator will reference the $2f+1$ votes in their DAG and execute. □

Theorem 8 (Equivocation-Tolerence)**.** *If a faulty validator v_k concurrently called r_bcast$_k(m,q,e)$ and r_bcast$_k(m',q,e)$ with $m \neq m'$ then the rest of the validators either r_deliver$_i(m,q,v_k,e)$, or r_deliver$_i(m',q,v_k,e)$, or there is a subsequent epoch $e' > e$ where v_k is honest, calls r_bcast$_k(m'',q,e')$ and all honest validators r_deliver$_i(m'',q,v_k,e')$,*

Proof. For the case that validators *r_deliver$_i(m',q,v_k,e)$* it is a direct result of Theorem 7. Otherwise, from the code of the epoch change when the epoch ends all validators forget the locks they have taken on messages without certificates. As a result in a future epoch e' where v_k is honest and does not equivocate it will be able to commit m again from Thereon 7. □

VI. Implementation

We implement a networked multi-core Mysticeti validator in Rust. It uses `tokio` [32] for asynchronous networking, utilizing TCP sockets for communication without relying on any RPC frameworks. For cryptographic operations, we use `ed25519-consensus` [33] for asymmetric cryptography and `blake2` [34] for cryptographic hashing. To ensure data persistence and crash recovery, integrate a Write-Ahead Log (WAL), seamlessly tailored to our specific requirements. We have intentionally avoided key-value stores like RocksDB [35] to eliminate associated overhead and periodic compaction penalties. Our implementation optimizes I/O operations by employing vectored writes [36] for efficient multi-buffer writes in a single syscall. For reading the WAL, we make use of memory-mapped files while carefully minimizing redundant data copying and serialization. We use the `minibytes` [37] crates to efficiently work with memory-mapped file buffers without unsafe code.

While all network communications in our implementation are asynchronous, the core consensus code runs synchronously in a dedicated thread. This approach facilitates rigorous testing, mitigates race conditions, and allows for targeted profiling of this critical code path.

In addition to regular unit tests, we have two supplementary testing utilities. First, we developed a simulation layer that replicates the functionality of the `tokio` runtime and TCP networking. This simulated network accurately simulates real-world WAN latencies, while our tokio runtime simulator employs a discrete event simulation approach to mimic the passage of time. Utilizing this simulator, we can test a wide range of scenarios on a single machine and accurately estimate resulting latencies. It's worth noting that we've found these simulated latencies, such as commit latency, to closely mirror those observed in real-world cluster testing, provided that the cross-validator latency distribution in the simulated network is correctly configured. Second, we created an a command-line utility (called 'orchestrator') designed to deploy real-world clusters of Mysticeti with machines distributed across the globe. The simulator has proven indispensable in identifying correctness defects, while the orchestrator has been instrumental in pinpointing performance bottlenecks. We open-source our Mysticeti implementation, its simulator, and orchestration utilities[6].

VII. Evaluation

We evaluate the throughput and latency of Mysticeti through experiments on AWS to show its performance improvements over the state-of-the-art.

Despite the large number of BFT consensus protocols [11], [9], [38], [39], [8], [7], [5], [10], [40], [41], we opt to compare Mysticeti-C with vanilla HotStuff [42], HotStuff-over-Narwhal (called *Narwhal-HotStuff*) [3], and Bullshark [4]. We select these protocols for the availability of open-source implementations and detailed benchmarking scripts, their

[6]https://github.com/asonnino/mysticeti/tree/paper (commit 96fd831)

similarity to MYSTICETI, and their adoption in real-world deployments. We specifically select the Jolteon [14] variant of HotStuff as it has been adopted by Flow [43], Diem [44], Aptos [45], and Monad [46]. We also select the Narwhal-HotStuff variant as it operates on a structured DAG as MYSTICETI and is the most performant variant of HotStuff. We finally select Bullshark as it is a performant DAG-based protocol adopted by the Sui blockchain [1], [2], Aleo [47], and Fleek [48]. We evaluate the Narwhal-based systems (that is, Narwhal-HotStuff and Bullshark) in their default 1 worker configuration. We also evaluate the fast path MYSTICETI-FPC against Zef [23] (in its default configuration, with 10 shards), which is the state-of-the-art fast path protocol that serves as the foundation for the Linera blockchain [25].

Throughout our evaluation, we particularly aim to demonstrate the following claims. **C1:** MYSTICETI-C has higher throughput and drastically lower latency than the baseline state-of-the-art protocols. **C2:** MYSTICETI-C has a similar throughput to the baseline protocols but maintains sub-second latencies when operating in the presence of crash faults. **C3:** MYSTICETI-FPC maintains the same latency as the baseline state-of-the-art consensus-less protocol but with drastically higher throughput.

Note that evaluating BFT protocols in the presence of Byzantine faults is an open research question [49], and state-of-the-art evidence relies on formal proofs of safety and liveness (which we present in Section V). While there is a need to robustly tolerate Byzantine faults, we note that they are rare in observed delegated proof of stake blockchains, as compared to crash faults that are very common.

A. Experimental setup

We deploy a MYSTICETI testbed on AWS, using `m5d.8xlarge` instances across 13 different AWS regions: N. Virginia (us-east-1), Oregon (us-west-2), Canada (ca-central-1), Frankfurt (eu-central-1), Ireland (eu-west-1), London (eu-west-2), Paris (eu-west-3), Stockholm (eu-north-1), Mumbai (ap-south-1), Singapore (ap-southeast-1), Sydney (ap-southeast-2), Tokyo (ap-northeast-1), and Seoul (ap-northeast-2) . Validators are distributed across those regions as equally as possible. Each machine provides 10Gbps of bandwidth, 32 virtual CPUs (16 physical cores) on a 2.5GHz Intel Xeon Platinum 8175, 128GB memory, and runs Linux Ubuntu server 22.04. We select these machines because they provide decent performance, are in the price range of 'commodity servers', and are the same instance types used by our baselines.

MYSTICETI can employ more than one slot per round to mitigate the performance impact of crash faults and commit more blocks per round, but if the proposer slot behaves in a Byzantine manner, it can still manipulate their slot to remain undecided, resulting in similar latency effects as an unmasked crash fault. Therefore, we have chosen to have two proposer slots per round as an effective compromise for our experiments. To implement the partial synchrony assumption, validators wait up to 1 second to receive a proposal from the first proposer slot of the previous round.

In the following graphs, each data point is the average latency and the error bars represent one standard deviation (error bars are sometimes too small to be visible on the graph). We instantiate several geo-distributed benchmark clients within each validator submitting transactions at a fixed rate for a duration of several minutes. We experimentally increase the load of transactions sent to the systems, and record the throughput and latency of commits. As a result, all plots illustrate the 'stead state' latency of all systems under low load, as well as the maximal throughput they can serve after which latency grows quickly. Transactions in the benchmarks are arbitrary and contain 512 bytes. The ping latency between the validators varies from 50ms to 250ms.

When referring to *latency*, we mean the time elapsed from when the client submits the transaction to when the transaction is committed by the validators. When referring to *throughput*, we mean the number of committed transactions over the duration of the run. **??** provides a tutorial to reproduce our experiments.

B. Benchmark in ideal conditions

Figure 9 illustrates the Latency (seconds) - Throughput (Transactions per second, TPS) relationship for MYSTICETI-C and other consensus protocols, for a small deployment of 10 validators and a larger deployment of 50 validators. The systems run in ideal conditions, without faults.

At a steady state of 50k to 400k TPS for both network sizes MYSTICETI-C exhibits sub-second latency, a factor 2x-3x lower than the fastest protocols, namely HotStuff, and Narwhal-HotStuff. Bullshark uses a certified DAG and worker architecture and is over 3x slower in terms of latency compared with MYSTICETI-C for low system loads. In terms of throughput, both MYSTICETI-C networks scale extremely well and achieves a throughput of over 300k-400k TPS before the latency reaches 1s, that is, well lower than the latency of state-of-the-art systems. This illustrates that the single-host throughput efficiency of MYSTICETI-C is higher than for previous designs. Note that current real-world blockchains combined[7] process fewer than 100M transactions per day, equivalent to about 1.2k TPS, well within the steady state low-latency parameter space for MYSTICETI-C, without any further scaling strategies.

These observations validate our claim **C1** showing that MYSTICETI-C has higher throughput and drastically lower latency than the baseline state-of-the-art protocols.

Throughout these benchmarks, the the CPU utilization of the validators remains below 10% and the validators consumes less than 15GB of memory (when experiencing the highest load of 400k tx/s).

C. Benchmark with faults

Figure 10 illustrates the performance of HotStuff, Narwhal-HotStuff, Bullshark, and MYSTICETI-C when a committee of

[7]Estimates from https://app.artemis.xyz/comparables

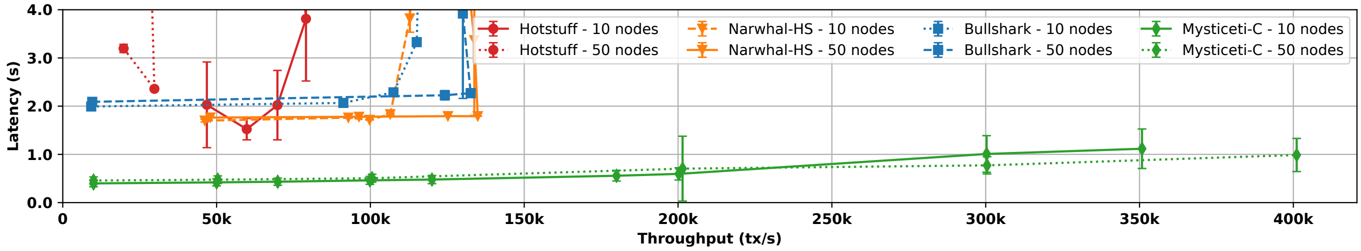


Fig. 9: Throughput-Latency graph comparing Mysticeti-C performance with state-of-the-art consensus protocols.

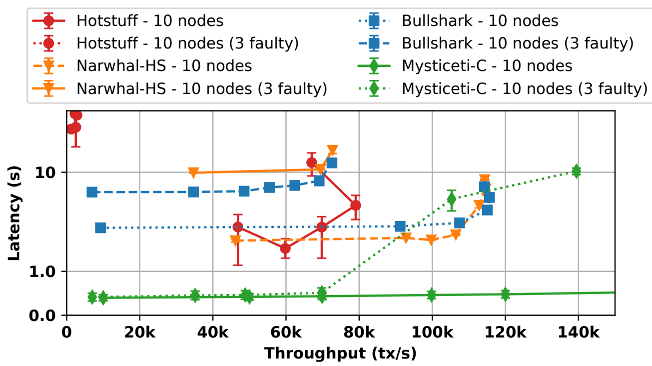


Fig. 10: Throughput - Latency under crash faults.

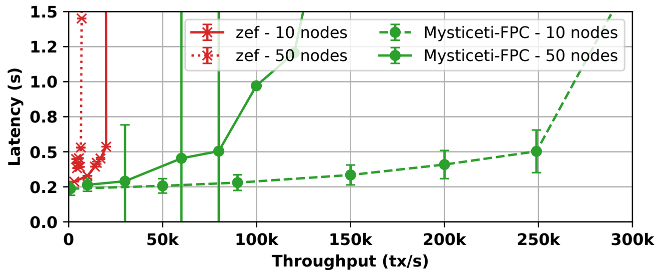


Fig. 11: Throughput - Latency comparison the for fast path commits between Mysticeti-FPC and Zef

10 parties suffers 0 to 3 crash faults (the maximum that can be tolerated in this setting). HotStuff suffers a massive degradation in both throughput and latency. With 3 faults, the throughput of HotStuff drops to a few hundred TPS and its latency exceeds 15s. Narwhal-HotStuff, Bullshark, and Mysticeti-C maintain a good level of throughput: the underlying DAG continues collecting and disseminating transactions despite the faults. Narwhal-HotStuff and Bullshark can process about 70k TPS in about 8-10 seconds. In contrast, Mysticeti-C can process the same load while maintaining sub-second latency. This improvement is due to the ability of Mysticeti to operate with multiple leaders per round. Mysticeti-C thus demonstrates a 15-20x latency improvement compared to the baseline state-of-the-art protocols.

These observations validate our claim **C2** showing that Mysticeti-C handles similar throughput to the state-of-the-art but with sub-second latency despite crash faults.

D. Benchmark of the fast path

Figure 11 illustrates the Latency - Throughput of fast path commits for Mysticeti-FPC, compared with Zef [23] when deployed without privacy protections[8]. Both systems run in ideal conditions, without faults. We observe that for low loads both protocols have a comparable latency of around 0.25s. However, as the load increases a Zef host has to verify and produce an increasing number of signatures, proportional to the throughput times the number of validators. As a result throughput tops at 20k TPS for a small Zef network and 7K TPS for a larger network, at a latency of 0.5s. Mysticeti-FPC avoids the need for individual signature verification for each transaction. At a low load, its latency is similar to Zef at 0.25s. However, as the load increases Mysticeti-FPC can process many more messages on a single host, namely 175k TPS for a small network and 80K for a larger network, at a latency of less than 0.5s. This is a single host throughput improvement of 8x-10x compared with Zef. We acknowledge that the Zef design can scale by adding additional hosts per validator, and sharding. However, this leads to additional hardware cost meaning that Mysticeti-FPC is an order of magnitude more resource efficient for the same latency.

We thus validate our claim **C3** showing that Mysticeti-FPC offers the same latency as state-of-the-art consensus-less protocols but with significantly higher throughput.

VIII. Mysticeti in Production

We collaborated with the Sui team to integrate Mysticeti-C into the Sui blockchain as a replacement for Bullshark [4], which it used for consensus (Figure 1).

There are a number of reasons why Sui is a good fit for using Mysticeti-C. First, Sui maintains a fixed committee consensus during each epoch, which does not require Mysticeti-C to support unscheduled reconfiguration, allowing for a drop in replacement of the consensus component. Secondly, Byzantine behavior in Sui is handled through shifts in stake delegation between epochs. Thus, the priority is to maintain performance under frequent crash faults, as is the case with Mysticeti-C. Byzantine faults need to be handled safely, but it is not critical to maintain extremely high performance while doing so, since they are rare. In the past year, no Byzantine faults involving equivocation have been observed on the Sui mainnet.

Code adaptations. To ensure seamless integration with the existing Sui codebase, we undertook a series of adaptations. We improved system resilience through the addition of new

[8]Zef can also be instantiated to leverage the Coconut threshold credentials system [50] to provide privacy guarantees at the cost of performance.

unit tests, crash recovery mechanisms, and bulk synchronization. We further integrated into Mysticeti-C key Sui features, such as a timestamp service [51] (detailed in Appendix B), facilitating compatibility and interoperability with the higher-level smart contract layer. Finally, we integrated HammerHead [28], adding proposer reputation, to further enhance stability and performance.

From prototype to production. The roadmap spans from the initial experimentation on the prototype code to a production-ready version of Mysticeti-C deployed in Sui.

Explorations on how to integrate Mysticeti-C started in November 2023, with experimentation on the prototype code (Section VI) and an exploration of which existing Sui code components could be reused. We reached a significant milestone in February 2024: deploying a production-ready version of Mysticeti-C onto the geo-distributed private test environment. Initial testing was conducted on a testbeds comprised of 137 validators with voting power that emulated the distribution observed on the Sui mainnet. We run stress tests that simulate typical blockchain traffic, ranging from 100 to 6,000 transactions per second.

While the deployment went smoothly, the system's latency profile initially did not meet expectations. Although Mysticeti-C had sub-second P50 latency, it has noticeably higher tail latency and jitters. We discovered several contributing factors including inefficient synchronization of unevenly broadcasted blocks, and significant degradation under high CPU utilization in the QUIC networking implementation we used. The synchronization issue was not noticeable in the current Sui blockchain due to the higher latency of Bullshark. In an effort to address these challenges, we improved the fault tolerance and effectiveness of block synchronization. Also we migrated to TCP networking for node-to-node communication, with streaming of proposed blocks.

Thorough testing is essential to gain confidence before the deployment of Mysticeti-C into the Sui mainnet. First, we developed and open-sourced a Domain-Specific Language (DSL) to swiftly construct Mysticeti's DAG [9] under various scenarios, which simplifies the creation of diverse DAG structures such as missing proposers, and diverse selection of block ancestors. Also we used the deterministic simulated testing framework built for the Sui project to randomly inject network faults, network latency jitters and thread delays to Mysticeti-C. The system was verified to maintain safety and liveness under these randomly injected failures. In addition, many experiments with Mysticeti-C were carried out over the private testnet environment with high generated load, and sometimes down validators. We made sure the system stay live and latency growth is expected under these conditions.

When deploying Mysticeti-C to Sui staging environments, the environments alternated between running Bullshark, the existing consensus protocol, and Mysticeti-C with each epoch. Devnet epochs last 1 hour, while testnet epochs last 24 hours. This method ensures that both consensus protocols get test coverage in the staging environments, and provides reassurance that protocol upgrades can be executed smoothly upon transitioning to the mainnet. Moreover, it allows for performance comparison between the consensus protocols under the same environment. Sui mainnet validators operated by independent entities voted to switch to Mysticeti-C on July 25th, 2024 PST.

[9]https://github.com/MystenLabs/sui/pull/16436

Protocols	Committee Size	TPS	P50 Latency	P95 Latency
Bullshark	137	5,000	2,890ms	4,600ms
Mysticeti-C	137	5,000	650ms	975ms

TABLE I: Comparison of production performance: bullshark vs. Mysticeti-C deployment within Sui with 137 validators (with equal voting power). Both systems are subjected to a load of 5,000 TPS and observed a sustained throughput of 5,000 TPS. All benchmarks ran for many hours.

Performance assessment. The performance results depicted in Table I are provided by the Sui team. Measurements are obtained from a private deployment on Vultr [52], utilizing `vbm-24c-256gb-amd` instances deployed on 9 different regions: Amsterdam, Frankfurt, Paris, Los Angeles, San Jose (California), Newark (New York), Tokyo, New Delhi, and Johannesburg. Each machine provides 25Gbps of bandwidth, 48 virtual CPUs (24 physical cores) on a 2.85GHz AMD EPYC 7443P, 256GB memory, and runs Linux Ubuntu server 24.04. The partially synchronous assumption is implemented by mandating validators to wait an additional 250ms for the block of the anchor slot of the previous round after receiving $2f+1$ proposals from that previous round.

Sui equipped with the production-ready implementation of Mysticeti-C demonstrates superior latency compared to when equipped with the production-ready implementation of Bullshark, with p50 and p95 latency of 650ms and 975ms for 137 validators, respectively. In contrast, Sui equipped with Bullshark exhibits a p50 and p95 latency of 2.89s and 4.6s for the same configuration. The measurements are taken while both systems run in their steady-state, with a load of 5,000 transactions per second (and exhibiting an equal throughout) for multiple hours. These results demonstrate the substantial latency improvements – of over 4x – brought to the blockchain when swapping Bullshark for Mysticeti-C.

IX. Related Work

Mysticeti is a family of protocols designed to support next-generation distributed ledgers [53], [54], [55], [56]. To this end, its goal is to capture as wide a range of distributed ledgers as possible whether consensus-based or consensus-less. The pioneer on hybrid distributed ledgers is the Sui Lutris blockchain [2] which has been productionized by Sui [1]. However, the design of Sui Lutris focuses on providing a glue between the two distinct use-cases of consensus-based and consensus-less distributed ledgers, or in the production code a glue of FastPay [22] and Bullshark [4]. This design process of starting with the to-be-glued components and ending in a final system has led to significant inefficiencies such as multiple rebroadcasting of the same data as well as signature verification costs. Unlike Sui Lutris, Mysticeti is designed from first principles and as a result shows a potential halving of the latency, matching the lower bounds of PBFT [16] for consensus and Reliable Broadcast [57], [13] for consensusless distributed ledgers with equivocation tolerance.

We already discussed the core benefits of Mysticeti-FPC in terms of much lower CPU cost. In addition, it inherits the ability to change epochs, reconfigure the validator set, and tolerate equivocations from Sui Lutris. These benefits can also be used to embed other broadcast-based protocols like FastPay [22], Astro [24], and Zef [23], to improve privacy.

In terms of consensus, the most recent DagRider [6], Narwhal-Tusk [3], Bullshark [4] were the inspiration for using a structured DAG and defining a safe commit rule on it. However, they all use a DAG of certified blocks which increases both latency and implementation complexity. Although at first glance, certification seems to benefit adversarial cases where nodes can advance the DAG without needing to synchronize, our production experience of Bullshark [4] has shown that this benefit is negated right after consensus is finished and executing transactions starts (which requires all dependencies to be already executed). As a result, the certification benefits only Byzantine Atomic Broadcast protocols but not if used for the common case of powering a State Machine Replication system (e.g., a blockchain). Mysticeti uses instead a DAG of signed but not certified blocks, reducing latency significantly being the fastest DAG-based SMR to date.

Cordial Miners [58] has also proposed a similar DAG-structure to Mysticeti-C. However, their *Blocklace* detects and excludes equivocating miners so that it can eventually converge when there is no misbehavior. Its expected latency is additionally higher than Mysticeti-C as it only commits one proposed block per wave (3 rounds) and it lacks an implementation for us to do some more direct comparison[10]. Mysticeti in comparison has additionally shown how to integrate a fast path as well as how to commit most of the blocks with an expected latency of 3 rounds. The subsequent concurrent work on Flash [59] also discussed how to leverage a blocklace/DAG to allow for payments akin to the Mysticeti-FPC fast path, but without integrating it with a consensus path for complex transactions. Motorway [60] uses a consensus protocol based on 'data lanes', a relaxed notion of a DAG where replicas independently and concurrently disseminate data. Consensus is achieved over metadata through any black-box consensus mechanism, leveraging these data lanes as a data dissemination layer. This approach follows the spirit of Narwhal-HotStuff [3], with the added benefit of increased performance by eliminating the need for validators to progress in strict DAG rounds. Unfortunately, this approach also increases significantly the engineering complexity and foregoes the robustness properties that structured DAG-based protocols have.

As far as the Mysticeti-C commit rule is concerned, the first proposal of having a pipelined and multi-proposer version for quorum-based consensus comes from Multi-Paxos [61]. This work has been studied extensively as well as extended to multiple directions [62], [63], [64]. However, it only addresses crash and omission faults. The core idea can directly be transferred to Byzantine faults as PBFT [16] uses a similar structure to Paxos, and we can see its adoption in Mir-BFT [65]. Blockmania [66] as well as Schett & Danezis [67] further develop the idea for DAG-based consensus, and the recent work Shoal [5] has applied it to certified DAGs with recursive commit rules [4]. Mysticeti's commit rule is the next evolution, extending pipelining into uncertified recursive DAGs in order to achieve simultaneously the lowest latency possible (3 message rounds, according to [21]) as well as the high throughput and censorship resistance of DAGs.

Notably, Narwhal-based designs use a worker-primary architecture to increase throughput. Mysticeti-C can be adapted to this architecture, by acting as a primary for any number of workers in case additional throughput is needed. Additionally, Shoal and HammerHead [28] propose leader reputation protocols inspired by Carousel [68]. Our production implementation of Mysticeti-C adopts these designs to select more reliable proposers (Section VIII), but for liveness, it would need to adopt a proposer slot rotation schedule where slots remain static for 3 rounds.

Previous consensus protocols such as Hashgraph [69] also use a DAG of signed but not certified blocks: however, they use DAGs that are not structured as threshold clocks [20] making their proofs of safety very complex and leaving several open questions regarding practical implementations [3]. Fino [11] generalizes the commit rule of Bullshark to an unstructured certified DAG. BBCA-ledger [10] interweave together a novel low-latency happy path based on a variant of Byzantine Consistent Broadcast and Bullshark as a high-throughput DAG-based fallback path.

Notably, Mysticeti-C works in only 3 message communication rounds, which matches PBFT, and is optimal latency [70], [71] without the use of optimistic methods like Zyzzyva [26]. This is lower than the state-of-the-art Jolteon [14] currently deployed in multiple blockchains [43], [15], [45], [46]. The reason is that these protocols focus on linear communication complexity, whereas Mysticeti-C embraces its cubic cost and amortizes it using the DAG structure as first proposed by Dag-Rider and Narwhal.

X. Conclusion

We introduce Mysticeti-C, a threshold clock-based Byzantine consensus protocol with the lowest WAN latency of 0.5s and the ability to process over 200k TPS at this latency for single-host nodes, far exceeding the needs of blockchains today (which consume in total about 1.2k TPS). We additionally present Mysticeti-FPC, a fast path protocol achieves

[10]The Cordial Miners manuscript publicly available during Mysticeti's development considered a single certificate pattern to be a sufficient condition to commit a block. This is not safe. As we saw in our proofs, there is a need for $2f+1$ blocks certify a block to safely commit it. The published version of the work, that appeared concurrently to this work, fixes this issue.

even lower latency at 0.25s but with over 8x better resource efficiency compared with protocols with explicit certificates. Despite being designed in a BFT setting, both Mysticeti protocols efficiently handle crash faults using multiple proposer slots per round, implemented through a novel decision rule.

We leave several explorations for the future. For use cases requiring higher throughput, we note that Mysticeti-C can be augmented with workers, in a similar way to Tusk and Bullshark. This would allow it to scale without known bounds, at the cost of additional latency (a round trip) to coordinate workers and primaries. An alternative approach would be to run multiple Mysticeti-C instances in parallel, something we feel is under-explored but inspired us to have explicit votes in Mysticeti-FPC. The structure of Mysticeti-FPC has all nodes timestamping transactions through their votes and may be useful for implementing MEV protections.

Finally, we note that as the latency of consensus under low load shrinks (now 0.5s) the latency advantages of the fast path diminish. It is an open industrial question whether use cases that require low latency justify the complexity of dual path systems going forward, as the latency gap closes. It is also an open question whether the worker-primary architecture employed by Narwhal-based designs is useful today since a single worker throughput by far exceeds the capacity needed by blockchains, and does so at a lower latency. This may change in the future as more capacity is needed.

Acknowledgment

This work was conducted while Kushal Babel was interning with Mysten Labs. We would like the thank Dmitry Perelman, Xun Li, and Lu Zhang from the Mysten Labs engineering team for the great discussions that improved this work. We also extend our thanks to Nikita Polianskii for various discussions around potential fast path designs and to Matthias Hallgren for suggesting to dynamically adjust the number of proposer slots per round based on the network's observed behavior. We also thank Ehud Shapiro and Oded Naor for reviewing the manuscript and related work. Finally, we thank Kartik Nayak, Aniket Kate, and Nibesh Shrestha for identifying that our timeout for liveness needs to be 3Δ.

References

[1] The Sui team, "The sui blockchain," http://sui.io, 2023.
[2] S. Blackshear, A. Chursin, G. Danezis, A. Kichidis, L. Kokoris-Kogias, X. Li, M. Logan, A. Menon, T. Nowacki, A. Sonnino *et al.*, "Sui lutris: A blockchain combining broadcast and consensus," *arXiv preprint*, 2023.
[3] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Narwhal and tusk: a dag-based mempool and efficient BFT consensus," in *European Conference on Computer Systems.* ACM, 2022.
[4] A. Spiegelman, N. G. andF Alberto Sonnino, and L. Kokoris-Kogias, "Bullshark: DAG BFT protocols made practical," in *Conference on Computer and Communications Security, CCS.* ACM, 2022.
[5] A. Spiegelman, B. Aurn, R. Gelashvili, and Z. Li, "Shoal: Improving DAG-BFT latency and robustness," *CoRR*, vol. abs/2306.03058, 2023.
[6] I. Keidar, E. Kokoris-Kogias, O. Naor, and A. Spiegelman, "All you need is DAG," in *Principles of Distributed Computing.* ACM, 2021.
[7] Y. Gao, Y. Lu, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Dumbo-ng: Fast asynchronous bft consensus with throughput-oblivious latency," in *Conference on Computer and Communications Security*, 2022.
[8] L. Yang, S. J. Park, M. Alizadeh, S. Kannan, and D. Tse, "Dispersedledger: High-throughput byzantine consensus on variable bandwidth networks," in *Networked Systems Design and Implementation*, 2022.
[9] N. Shrestha, R. Shrothrium, A. Kate, and K. Nayak, "Sailfish: Towards improving latency of dag-based bft," *Cryptology ePrint Archive*, 2024.
[10] C. Stathakopoulou, M. Wei, M. Yin, H. Zhang, and D. Malkhi, "BBCA-LEDGER: High Throughput Consensus meets Low Latency," *arXiv preprint*, 2023.
[11] D. Malkhi and P. Szalachowski, "Maximal extractable value (mev) protection on a dag," *arXiv preprint*, 2022.
[12] G. Giuliari, A. Sonnino, M. Frei, F. Streun, L. Kokoris-Kogias, and A. Perrig, "An Empirical Study of Consensus Protocols' DoS Resilience," in *AsiaCCS*, 2024.
[13] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to reliable and secure distributed programming.* Springer Science & Business Media, 2011.
[14] R. Gelashvili, L. Kokoris-Kogias, A. Sonnino, A. Spiegelman, and Z. Xiang, "Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback," in *Financial Cryptography and Data Security.* Springer, 2022.
[15] The Diem Team, "Diembft v4," https://developers.diem.com/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2021-08-17.pdf, 2021.
[16] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Symposium on Operating Systems Design and Implementation (OSDI).* USENIX Association, 1999.
[17] Z. Li, A. Sonnino, and P. Jovanovic, "Performance of eddsa and bls signatures in committee-based consensus," in *Workshop on Advanced tools, programming languages, and PLatforms for Implementing and Evaluating algorithms for Distributed systems*, 2023.
[18] K. K. Chalkias, J. Lindstrøm, D. Maram, B. Riva, A. Roy, A. Sonnino, and J. Wang, "Fastcrypto: Pioneering cryptography via continuous benchmarking," 2024.
[19] R. Kotla and M. Dahlin, "High throughput byzantine fault tolerance," in *International Conference on Dependable Systems and Networks, 2004*, 2004, pp. 575–584.
[20] B. Ford, "Threshold logical clocks for asynchronous distributed coordination and consensus," *CoRR*, vol. abs/1907.07010, 2019.
[21] J. Martin and L. Alvisi, "Fast byzantine consensus," *IEEE Trans. Dependable Secur. Comput.*, vol. 3, no. 3, pp. 202–215, 2006.
[22] M. Baudet, G. Danezis, and A. Sonnino, "Fastpay: High-performance byzantine fault tolerant settlement," in *Advances in Financial Technologies*, 2020.
[23] M. Baudet, A. Sonnino, M. Kelkar, and G. Danezis, "Zef: Low-latency, scalable, private payments," *CoRR*, vol. abs/2201.05671, 2022.
[24] D. Collins, R. Guerraoui, J. Komatovic, M. Monti, A. Xygkis, M. Pavlovic, P. Kuznetsov, Y.-A. Pignolet, D.-A. Seredinschi, and A. Tonkikh, "Online payments by merely broadcasting messages (extended version)," *arXiv preprint arXiv:2004.13184*, 2020.
[25] The Linera Team, "Linera," https://linera.io, 2023.
[26] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. L. Wong, "Zyzzyva: speculative byzantine fault tolerance," in *Symposium on Operating Systems Principle.* ACM, 2007.
[27] S. Cohen, G. Goren, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Proof of availability and retrieval in a modular blockchain architecture," in *Financial Cryptography and Data Security.* Springer, 2023.
[28] G. Tsimos, A. Kichidis, A. Sonnino, and L. Kokoris-Kogias, "Hammerhead: Leader reputation for dynamic scheduling," *arXiv preprint arXiv:2309.12713*, 2023.
[29] R. Neiheiser, A. Babaei, G. Alexopoulos, M. Kogias, and E. K. Kogias, "Chiron: Accelerating node synchronization without security trade-offs in distributed ledgers," *arXiv preprint arXiv:2401.14278*, 2024.
[30] M. Baudet, A. Sonnino, M. Kelkar, and G. Danezis, "Zef: low-latency, scalable, private payments," in *Proceedings of the 22nd Workshop on Privacy in the Electronic Society*, 2023, pp. 1–16.
[31] L. Kokoris-Kogias, A. Sonnino, and G. Danezis, "Cuttlefish: Expressive fast path blockchains with fastunlock," *arXiv preprint arXiv:2309.12715*, 2023.
[32] T. T. Team, "Tokio," https://tokio.rs, 2024.
[33] H. de Valence, "Ed25519 for consensus-critical contexts," https://crates.io/crates/ed25519-consensus, 2024.
[34] RustCrypto, "Rustcrypto: Hashes," https://github.com/RustCrypto/hashes, 2024.
[35] T. R. Team, "Rocksdb," https://rocksdb.org, 2024.

[36] Die.Net, “writev(3) - linux man page,” https://linux.die.net/man/3/writev, 2024.
[37] Meta, “Sapling (minibytes),” https://github.com/facebook/sapling/tree/main/eden/scm/lib/minibytes, 2024.
[38] J. Chen, S. Gupta, A. Sonnino, L. Kokoris-Kogias, and M. Sadoghi, “Resilient consensus sustained collaboratively,” *arXiv preprint arXiv:2302.02325*, 2023.
[39] S. Cohen, G. Goren, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, “Proof of availability & retrieval in a modular blockchain architecture,” *Cryptology ePrint Archive*, 2022.
[40] D. Malkhi, C. Stathakopoulou, and M. Yin, “BBCA-CHAIN: One-Message, Low Latency BFT Consensus on a DAG,” *arXiv preprint*, 2023.
[41] B. Y. Chan and E. Shi, “Streamlet: Textbook streamlined blockchains,” in *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, 2020, pp. 1–11.
[42] M. Yin, D. Malkhi, M. K. Reiter, G. Golan-Gueta, and I. Abraham, “Hotstuff: BFT consensus with linearity and responsiveness,” in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, P. Robinson and F. Ellen, Eds. ACM, 2019, pp. 347–356.
[43] The Flow Team, “The flow blockchain,” https://flow.com, 2023.
[44] M. Baudet, A. Ching, A. Chursin, G. Danezis, F. Garillot, Z. Li, D. Malkhi, O. Naor, D. Perelman, and A. Sonnino, “State machine replication in the libra blockchain,” *The Libra Assn., Tech. Rep*, vol. 1, no. 1, 2019.
[45] The Aptos team, “Aptos,” https://aptoslabs.com, 2023.
[46] Monad, “Monadbft:pipelined two-phase hotstuff consensus,” *https://docs.monad.xyz/technical-discussion/consensus/monadbft*, 2023.
[47] Aleo, “Zero-knowledge with uncompromised speed and privacy,” https://aleo.org, 2024.
[48] Fleek, “Build Lightning Fast,” https://fleek.xyz, 2024.
[49] S. Bano, A. Sonnino, A. Chursin, D. Perelman, Z. Li, A. Ching, and D. Malkhi, “Twins: Bft systems made robust,” in *25th International Conference on Principles of Distributed Systems*, 2022.
[50] A. Sonnino, M. Al-Bassam, S. Bano, S. Meiklejohn, and G. Danezis, “Coconut: Threshold issuance selective disclosure credentials with applications to distributed ledgers,” in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.
[51] T. S. Team, “Access on-chain time,” https://docs.sui.io/guides/developer/sui-101/access-time, 2024.
[52] Vultr, “Vultr,” https://docs.vultr.com, 2024.
[53] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis, “Sok: Consensus in the age of blockchains,” in *AFT*, 2019.
[54] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis, “Chainspace: A sharded smart contracts platform,” in *NDSS*, 2018.
[55] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, “Omniledger: A secure, scale-out, decentralized ledger via sharding,” in *2018 IEEE symposium on security and privacy (SP)*. IEEE, 2018, pp. 583–598.
[56] A. Sonnino, S. Bano, M. Al-Bassam, and G. Danezis, “Replay attacks and defenses against cross-shard consensus in sharded distributed ledgers,” in *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2020, pp. 294–308.
[57] G. Bracha and S. Toueg, “Asynchronous consensus and broadcast protocols,” *J. ACM*, vol. 32, no. 4, pp. 824–840, 1985.
[58] I. Keidar, O. Naor, O. Poupko, and E. Shapiro, “Cordial miners: Fast and efficient consensus for every eventuality,” in *37th International Symposium on Distributed Computing, DISC 2023, October 10-12, 2023, L’Aquila, Italy*, ser. LIPIcs, R. Oshman, Ed., vol. 281. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, pp. 26:1–26:22.
[59] A. Lewis-Pye, O. Naor, and E. Shapiro, “Flash: An asynchronous payment system with good-case linear communication complexity,” *CoRR*, vol. abs/2305.03567, 2023.
[60] N. Giridharan, F. Suri-Payer, I. Abraham, L. Alvisi, and N. Crooks, “Motorway: Seamless high speed bft,” *arXiv preprint arXiv:2401.10369*, 2024.
[61] L. Lamport, “Paxos made simple,” *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, 2001.
[62] P. Tennage, C. Basescu, L. Kokoris-Kogias, E. Syta, P. Jovanovic, V. Estrada-Galiñanes, and B. Ford, “Quepaxa: Escaping the tyranny of timeouts in consensus,” in *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, J. Flinn, M. I. Seltzer, P. Druschel, A. Kaufmann, and J. Mace, Eds. ACM, 2023, pp. 281–297.
[63] P. Tennage, A. Desjardins, and L. Kokoris-Kogias, “Racs and sadl: Towards robust smr in the wide-area network,” *arXiv preprint arXiv:2404.04183*, 2024.
[64] I. Moraru, D. G. Andersen, and M. Kaminsky, “Egalitarian paxos,” in *ACM Symposium on Operating Systems Principles*, 2012.
[65] C. Stathakopoulou, T. David, M. Pavlovic, and M. Vukolić, “[solution] mir-bft: Scalable and robust bft for decentralized networks,” *Journal of Systems Research*, vol. 2, no. 1, 2022.
[66] G. Danezis and D. Hrycyszyn, “Blockmania: from block dags to consensus,” *arXiv preprint arXiv:1809.01620*, 2018.
[67] M. A. Schett and G. Danezis, “Embedding a deterministic bft protocol in a block dag,” in *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, 2021, pp. 177–186.
[68] S. Cohen, R. Gelashvili, L. K. Kogias, Z. Li, D. Malkhi, A. Sonnino, and A. Spiegelman, “Be aware of your leaders,” in *International Conference on Financial Cryptography and Data Security*. Springer, 2022, pp. 279–295.
[69] L. Baird, “The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance,” 2016.
[70] I. Abraham, K. Nayak, L. Ren, and Z. Xiang, “Good-case latency of byzantine broadcast: A complete categorization,” in *PODC*, 2021.
[71] B. Y. Chan and R. Pass, “Simplex consensus: A simple and fast consensus protocol,” Cryptology ePrint Archive, Paper 2023/463, 2023, https://eprint.iacr.org/2023/463. [Online]. Available: https://eprint.iacr.org/2023/463

Appendix A
Example of Mysticeti-C Execution

This section completes Section III by providing a step-by-step example of a Mysticeti-C execution by leveraging Figure 4a of Section III. This figure illustrates an example of a Mysticeti DAG with four validators, (A0, A1, A2, A3), four slots per round. Initially, all proposers are marked as undecided.

Slots classification. The validator applies the direct decision rule starting with the latest slots, L6d, L6c, L6b, L6a, L5d, L5c, L5b, L5a (in that order), but fails to determine their status due to the absence of both a skip pattern and a certificate pattern. They thus remain undecided.

The direct decision rule then successfully marks L4d as to-commit due to the presence of $2f + 1$ certificate patterns, colored in green in Figure 4b. This reasoning is then applied successively to L4c and L4b, also marked as to-commit. Figure 4c then demonstrates the direct decision rule applied to L4a, resulting in its classification as to-skip due to the presence of a skip pattern. Continuing with L3d, L3c, L3b, and L2d, the direct decision rule categorizes them all as to-commit, similar to L4d, L4c, L4b.

Moving to L2c, Figure 4d shows the direct decision rule failing to classify it. Lacking both a skip and certificate pattern, the validator resorts to the indirect decision rule. It first identifies the *anchor* of L2c, which is the block with the lowest rank and round number r' such that $(r' > r + 2)$ and that is marked as either undecided or to-commit. In this case, L2c’s anchor is L5a. Since L5a is undecided, L2c remains so as well. The same reasoning is applied to L2b which is also marked undecided. Proceeding with L2a, the direct decision rule marks it as to-commit.

However, the direct decision rule cannot classify L1d. Consequently, Figure 4e demonstrates the application of the indirect decision rule to L1d, with L4b as its anchor (note that L4a is marked to-skip and thus cannot be an anchor). Since L4b is marked to-commit, and L1d has a certificate pattern linking to its anchor, L1d is marked to-commit.

After marking L1c as to-commit in the same way as L1d, the validator analyzes L1b. Since the direct decision rule cannot decide it, the indirect decision rule is applied with L4a as its anchor. Unlike L1d, there's no certified link from L4a to L1b, resulting in L1b being marked to-skip (Figure 4). Finally, L1a is marked to-commit via the direct decision rule.

Commit sequence. With as many proposers as possible classified as either to-commit or to-skip, the validator can establish the commit sequence. Beginning with the lowest slot, it outputs slots marked as to-commit while skipping those marked to-skip, halting once an undecided slot is encountered. The resulting commit sequence is L1a, L1c, L1d, L2a. Eventually, the DAG will progress and the slot L5a will be classified either as to-commit or to-skip, allowing the validator to classify L2c and all subsequent slots.

Choosing the number of proposer slots. The example presented by Figure 4 assumes a number of proposer slots per round equal to the committee size. While this choice offers the best latency under normal conditions, it may impact performance during periods of extreme asynchrony or under Byzantine attack. In these cases, the probability that the direct decision rule fails to classify a proposer slot increases when some proposer slots are slow or equivocate. This forces the validator to resort to the indirect decision rule more often. A practical mitigation could be to dynamically adjust the number of proposer slots per round based on the network's observed behavior. This would allow the system to maintain low latency during periods of synchrony while increasing the probability of observing certificate patterns during periods of asynchrony. To ensure all validators interpret the DAG consistently, the number of proposer slots per round should be agreed upon by all validators. This can be achieved by updating the number of proposers slots per round at commit boundaries, following a similar deterministic approach to HammerHead [28]

Appendix B
Exposing Commit Timestamps

As mentioned in Section VIII, the production-ready implementation of Mysticeti-C is equipped to expose timestamps to the higher application layer. Each Mysticeti-C block contains both the timestamp of its proposal and its commit. Validators incorporate their current time in each block they propose. Upon receiving a block, its timestamp undergoes validation by ensuring that the included time is greater than or equal to the timestamps of its parent blocks; otherwise, the block is rejected as invalid. Honest validators will only consider blocks as parents of their proposal if they possess past timestamps with respect to their local time, while if a block arrives with a future timestamp, a validator must wait before including or rejecting it.

Consequently, if a Byzantine validator introduces a block too far into the future, it will be rejected. To mitigate the small variations in the local clocks of validators, our implementation suspends the block in memory for a brief duration if its timestamp is only slightly ahead of the current local time.

When Mysticeti-C outputs a commit, it associates a timestamp with this action, termed a *commit timestamp*. The commit timestamp is defined as the maximum of the timestamp(s) of the proposer block(s) of such commit and the timestamp of the previous commit. Therefore, Mysticeti-C commit timestamps ensure monotonic increase. It's essential to include the commit timestamp of the previous commit in this maximum calculation because successive committed blocks are not necessarily linked by a parent-child relationship, thus unable to guarantee monotonicity. This contrasts with timestamp mechanisms in existing DAG-based consensus protocols, which lack a proposer every round and can thus ensure that each committed block references the previous committed block [3], [6], [4].