Mysticeti: Low-Latency DAG Consensus with Fast Commit Path

Kushal Babel
Cornell Tech, USA
IC3

Andrey Chursin
Mysten Labs, USA

George Danezis
Mysten Labs, UK
University College London

Lefteris Kokoris-Kogias
Mysten Labs, Greece
IST Austria

Alberto Sonnino
Mysten Labs, UK
University College London

Abstract

We introduce Mysticeti-C a byzantine consensus protocol with low-latency and high resource efficiency. It leverages a DAG based on Threshold Clocks and incorporates innovations in pipelining and multiple leaders to reduce latency in the steady state and under crash failures. Mysticeti-FPC incorporates a fast commit path that has even lower latency. We prove the safety and liveness of the protocols in a byzantine context. We evaluate Mysticeti and compare it with state-of-the-art consensus and fast path protocols to demonstrate its low latency and resource efficiency, as well as more graceful degradation under crash failures. Mysticeti is the first byzantine protocol to achieve WAN latency of 0.5s for consensus commit, at a throughput of over 50k TPS that matches the state-of-the-art.

1 Introduction

Several recent blockchains, such as Sui [7, 41], have adopted consensus protocols based on certified directed acyclic graphs (DAG) of blocks, such as Narwhal-Tusk [17], Bullshark [33], and recent proposals such as Shoal [34]. By design, these consensus protocols scale well in terms of throughput with a performance of 100k TPS of raw transactions. Notably, using Bullshark coupled with the MoveVM, Sui has processed a peak of 65m programmable transaction blocks on 27 July 2023, sustained throughput of over 700 TPS over the day, using a single consensus worker [7].

However, certified DAG consensus suffers from three disadvantages: (1) the certified DAG 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 [20, 38]; (2) blocks commit on a "per-wave" basis which means that only once every two rounds (for Bullshark) there is a chance to commit. Hence, some blocks have to wait for the wave to finish increasing the latency of transactions inside the block. This is similar to having big batches of $2f + 1$ blocks. Finally, (3) since all blocks need to be signed by all, signature generation and verification consume a large amount of CPU on each validator which grows with the number of validators. This burden is particularly heavy when a crash-recovered validator syncs to the DAG and is overwhelmed by signature verification.

A separate line of work explores the processing of transactions without or before reaching consensus, such as in FastPay [5], Zef [6], and Astro [15]. These systems use reliable broadcast instead of consensus to commit transactions that only access state controlled by a single party, and refer to this mechanism as a *fast path*. The Sui Lutris [7] mechanism underlying the Sui blockchain combines a fast path with a black-box certified DAG consensus. This composition is generic and leads to very low latencies for fast path transactions (the vast majority of transactions on the peak 27 July 2023 date). But it also leads to (1) increased latencies for other 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 since each transaction needs to be certified separately. The latter means that the validator's CPU is largely devoted to performing cryptographic operations rather than processing transactions. The need to sequence valid certificates into the consensus also reduces its capacity and imposes the CPU-heavy certificate verification burden on the critical path of consensus.

In this work, we present Mysticeti a family of protocols to safely commit distributed transactions in a Byzantine setting that focuses on low-latency and low-CPU operation. Mysticeti-C is a consensus protocol based on a threshold logical clock [18] DAG of blocks, that commits every block as early as it can be decided and does not need to explicitly certify each block. We extend it to include a fast path with Mysticeti-FPC, leading to very low-latency commits, without the need to generate an explicit certificate for each transaction. Both designs yield lower latency as well as lower CPU utilization. Both protocols could be augmented with multiple workers to support higher throughput, but our experiments show that current peaks of activity for all blockchains can comfortably be served by a single worker[1].

Contributions. We make the following contributions:

[1] https://app.artemis.xyz/comparables

- We present Mysticeti-C, a DAG-based byzantine consensus algorithm and its proofs of safety and liveness. Notably, it implements a universal 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 one worker Narwhal-based consensus.
- We also present Mysticeti-FPC that offers feature parity with Sui Lutris [7], i.e. 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 [6] and Fastpay [5] but higher throughput due to lower CPU utilization and batching.
- We fully implement both protocols and perform an experimental evaluation on a Wide Area Network. We show their latency-throughput characteristics are superior to certified DAG-based designs on the consensus mode; and competitive in the fast path while their throughput is far superior due to lower CPU overheads.

2 Background

We consider a message-passing system in which each epoch $n = 3f + 1$ validators process transactions using the Mysticeti protocols. In every epoch, a computationally bound adversary that controls the network 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 rest of the validators (at least $2f + 1$) are *correct* or *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 and the usual security properties of cryptographic hash functions, digital signatures, and other cryptographic primitives hold. Under these assumptions, Section 5 shows that the Mysticeti protocols are safe, in that, no two correct validators commit inconsistent transactions.

validators are communicating over a partially synchronous network. There exists a time called Global Stabilization Time (GST) and a known finite time bound Δ, such that any message sent by a party at time x is guaranteed to arrive by time $\Delta + \max\{\text{GST}, x\}$. 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 [17, 23, 33] we focus on byzantine atomic broadcast for Mysticeti. Additionally for Mysticeti-FPC, we show that the fast-path transactions subprotocol satisfies reliable broadcast within an epoch [7], but allows for recovery of equivocating objects across epochs without losing safety at the epoch boundaries. More formally:

Reliable broadcast. 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 defines properties as if the protocol runs in a single epoch. This, however, is unrealistic as there is validator churn. To this end, we extend all the protocols to also take as a parameter the epoch number and all properties should hold inside a single epoch. Fortunately, the definition of reliable broadcast allows the recovery of liveness for blocked sequence numbers that are equivocated inside an epoch. More specifically we define equivocation tolerance as follows:

Equivocation tolerance If a byzantine 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 $r\_deliver_i(m'', q, v_k, e')$,

3 The Mysticeti-C Protocol

We describe the Mysticeti-C consensus protocol. Section 4 describes Mysticeti-FPC, a variant incorporating a fast path.

3.1 Mysticeti-C overview

Mysticeti-C allows a committee of validators to open a consensus channel for an *epoch*, sequence several messages within it, and then eventually close the channel at the end of the epoch[2]. The Mysticeti-C protocol proceeds in a sequence of *rounds*. At the end of every round, each honest validator broadcasts a *unique signed block* for the round. During a round validators receive transactions from users, as well as blocks from other validators. They construct their block to contain both *references to blocks* from past rounds, always starting from their own latest block; as well as *fresh*

[2]The base Mysticeti-C 3-round commit protocol was proposed and proved correct independently as Cordial Miners [24].

transactions not already included indirectly in the past blocks. Once a block contains references to at least $2f + 1$ validator blocks from the previous round, and after a delay, it can be signed by the validator and disseminated to the other validators.

To commit transactions, the basic variant of Mysticeti-C relies on committing a *common sequence of blocks from leaders* at specific *leader rounds*. Rounds are structured to be a sequence of a leader round, followed by one or more support rounds, and finally a decision round. For each leader round all correct validators determine a leader using a deterministic method based on the round's number. Within the subsequent support rounds, blocks support the first leader block (in case of equivocation) indirectly included in their block. When a block indirectly includes $2f + 1$ validator blocks that support a leader block we say the block certifies the leader block. If a leader block is certified by $2f + 1$ blocks in the decision round, we initiate extending the leader commit sequence. First, any previous uncommitted leader block that is certified by at least one block in the causal history of the leader is committed, before the final leader block is committed.

For the universal commit rule, the basic commit rule is virtualized as if every blocks is a leader block. However, since this can create situation where the ordering of commits is unclear a block can not only be committed but also undecided, for example if there is some support but not $2f + 1$. Then, if a block is undecided, no subsequent block can be committed until the blocks gets decided. In order to speed up the process of deciding, the universal commit rule enables direct skips instead of only direct commits. A direct skip can happen when there is no votes for a block as this means that there will never be a certificate to indirect commit it later.

With both commit rules the sequence of (virtual) leaders committed is consistent across all correct validators. Each leader block commits the full causal history of blocks and contained transactions, that are not already part of a previous commit. The algorithm to transform the sequence of leader commits and expand it to transaction commits can be arbitrary as long as all new transactions in blocks are included in the sequence in a deterministic manner.

3.2 Transactions submission, support, and certificates on consensus blocks

A client submits transactions to a validator who includes it inside their next block. If the transaction does not appear in the consensus output within some time, the client picks another validator and retries. Since Mysticeti-C implements a BAB the transactions at this stage are treated as a payload of bytes which will be forwarded to the execution engine [22, 31] after the transaction first appears in the total ordering.

The main unit of communication in Mysticeti-C is the block, which includes (1) the author A of the block along with their signature on the full block contents, (2) a round number r, (3) a sequence containing at least $2f + 1$ distinct hashes of blocks from previous rounds, and (4) a list of transactions. By convention, the first hash must be to the latest past block from A. A block is valid if it is signed by a valid validator; 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 $r - 1$. 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 correct validator produces at most one unique block per round.

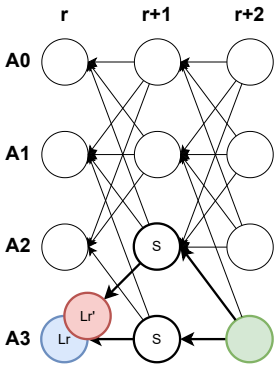

Figure 1. Block $(A_3, r + 2, \_)$ (green) contains support originating from different validators for both (A_3, r, L_r) (blue) and (A_3, r, L'_r) (red) equivocating blocks. If any of the blocks will gather $2f + 1$ support it will be certified, and we show that at most one may do so.

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. Once a block B is supported by $2f + 1$ distinct validator blocks, we say that the block B is *certified*, and any block that links to over $2f + 1$ blocks that support B represents a *certificate* for the block B. As we will see in the next section this concept of support and certificates is critical to achieve safety in Mysticeti-C.

We note that support and certification are properties of blocks and are monotonic properties: if one correct validator ever observes a block supporting another, or certifying another this will remain true forever, and also will be considered true by any other honest validator once they process the block. As Figure 1 illustrates, a block $(A_3, r+2, \_)$ (green) that supports block (A_3, r, L_r) (blue) may contain also support for an equivocating block (A_3, r, L'_r) (red).

3.3 Leaders and basic consensus and commit rule

Mysticeti-C rounds are composed of an ever-repeating sequence of phases which we call a wave: a single leader round, followed by a fixed number of one or more support rounds, and finally a single commit round (there must be at least

three phases in one wave). This pattern of rounds continues until the consensus channel is closed on a final commit round. Without losing generality we present the protocol with a single support round, and 3 rounds in total. This version requires two timeouts, one for the leader round and one for the support round[3].

At each leader round a single validator is deterministically chosen to act as the *leader*: the block from this validator in this round is used – if committed – to extend the sequence of commits, and as a result, the sequence of transactions. The exact algorithm by which leaders are chosen does not matter as long as all correct validators agree on the leader for all leader rounds, and eventually, a correct validator acts as a leader (Section 5).

During the support round, validators generate blocks with sufficient delay to allow a block from the leader to be included in their subDAGs and gain support. Finally, in the commit round, validators wait enough time to witness sufficient support. If $2f + 1$ blocks certify a leader block from the leader round, we start the algorithm to extend the sequence of leader blocks committed. To extend the sequence of committed leader blocks we apply the following algorithm: the latest leader block L will be committed last (since this triggered the commit). Then we consider the previous leaders: we pick the previous uncommitted leader L' with the largest round *that has a certificate in the sub-graph* from block L. And apply the commit algorithm recursively with $L \leftarrow L'$. The algorithm stops when the last committed leader is discovered. Then leaders are committed from the oldest to the newest by unwinding the recursion. This guarantees that even if validators commit leaders at different points in (logical threshold) time they do not diverge. Algorithm 3 describes the base commit algorithm.

Given the sequence of leader blocks committed, a deterministic algorithm can be applied to extend the sequence with all the blocks linked by the leaders, and then expand the blocks to the transactions contained in blocks. For example, each leader block can commit the full dag of blocks it links to that are not yet committed, and all transactions not ordered previously their transactions, ordered by ascending rounds.

Illustration of commit rule. Figure 2 illustrates an example of the commit rule in action. Leader L_r does not have enough certificates in the decision round $r + 2$ to commit since only 2 ($< 2f + 1$) validators issue blocks that certify it. However, leader block L_{r+3} has enough (i.e,. $3 = 2f + 1$ for $f = 1$) certificates in the decision round $r + 5$, to initiate a commit. Before committing block L_{r+3}, the leader block L_r is committed, since L_{r+3} contains a certificate for L_r in its causal history (green block). So the sequence is extended by L_r then L_{r+3}.

[3]An alternative design would be with two support rounds (4 rounds in total) which requires a timeout only for the leader round, but in our initial experiments the 3-round design was faster.

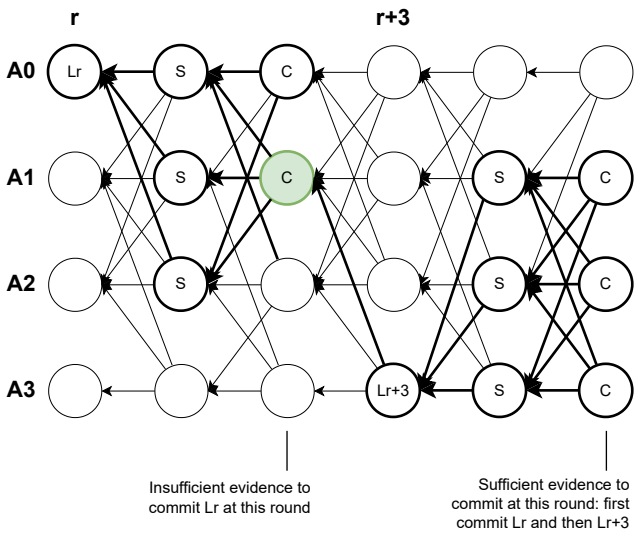

Figure 2. An illustration of the commit rule.

Algorithm 1 Offset Logic

```
1: procedure WaveNumber(r)
2:     return (r−roundOffset)/waveLength

3: procedure LeaderRound(w)
4:     return w ∗ waveLength+roundOffset

5: procedure DecisionRound(w)
6:     return w ∗ waveLength + waveLength − 1+roundOffset

7: procedure GetPredefinedLeader(w)
8:     r_leader ← LeaderRound(w)+LeaderOffset
9:     return PredefinedLeader(r_leader)
```

3.4 The universal commit rule

The basic Mysticeti-C protocol serves as an informative introduction to the underlying protocol's mechanisms. However, it exhibits two notable limitations that contribute to increased latency. Firstly, it commits transactions only once every three rounds, resulting in a five-round commitment duration for some transactions. Furthermore, the occurrence of a leader crash exacerbates latency by at least three additional rounds. To address these issues, we introduce the concept of a universal committer abstraction.

In contrast to the basic Mysticeti-C, which aims to commit one block per wave, the universal Mysticeti-C seeks to commit every block within a wave. To achieve this, we introduce the concept of "slots." A slot represents a tuple (validator, round) and can be either empty or contain the validator's proposal for the respective round. Additionally, a slot can assume one of three states: "to-commit," "to-skip," or "undecided". This approach draws inspiration from Multi-Paxos [27]. The number of slots instantiated per round can be

Algorithm 2 Consensus helper functions

```
1:  procedure GetLeaderBlock(w)
2:      r_leader ← LeaderRound(w)
3:      id ← GetPredefinedLeader(r_leader)
4:      if ∃b ∈ DAG[r_leader] s.t. b.author = id then return b
5:      return ⊥

6:  procedure GetFirstVotingBlocks(w)
7:      r_voting ← LeaderRound(w) + 1
8:      return DAG[r_voting]

9:  procedure GetDecisionBlocks(w)
10:     r_decision ← DecisionRound(w)
11:     return DAG[r_decision]

12: procedure Link(b_old, b_new)
13:     return exists a sequence of k ∈ ℕ blocks b_1, ..., b_k s.t. b_1 =
        b_old, b_k = b_new and ∀j ∈ [2, k] : b_j ∈ ⋃_{r≥1} DAG[r] ∧ b_{j−1} ∈
        b_j.parents

14: procedure IsVote(b_vote, b_leader)
15:     function SupportedBlock(b, id, r)
16:         if r ≥ b.round then return ⊥
17:         for b' ∈ b.parents do
18:             if (b'.author, b'.round) = (id, r) then return b'
19:             res ← SupportedBlock(b', id, r)
20:             if res ≠⊥ then return res
21:         return ⊥
22:     (id, r) ← (b_leader.author, b_leader.round)
23:     return SupportedBlock(b_vote, id, r) = b_leader

24: procedure IsCert(b_cert, b_leader)
25:     res ← |{b ∈ b_cert.parents : IsVote(b, b_leader)}|
26:     return res ≥ 2f + 1

27: procedure SkippedLeader(w)
28:     r_leader ← LeaderRound(w)
29:     id ← GetPredefinedLeader(r_leader)
30:     B ← GetFirstVotingBlocks(w)
31:     res ← |{b ∈ B s.t. ∀b' ∈ b.parents : b'.author ≠ id}|
32:     return res ≥ 2f + 1

33: procedure SupportedLeader(w)
34:     b_leader ← GetLeaderBlock(w)
35:     B ← GetDecisionBlocks(w)
36:     if |{b' ∈ B : IsCert(b', b_leader)}| ≥ 2f + 1 then return b_leader
37:     return ⊥

38: procedure CertifiedLink(b_anchor, b_leader)
39:     w ← WaveNumber(b_leader.round)
40:     B ← GetDecisionBlocks(w)
41:     return ∃b ∈ B s.t. IsCert(b, b_leader) & Link(b, b_anchor)
```

Algorithm 3 Baseline Mysticeti-C

```
1:  procedure TryCommit(r_committed, r_highest)
2:      w_committed ← WaveNumber(r_committed)
3:      w_highest ← WaveNumber(r_highest)
4:      sequence ← [ ]
5:      for w ∈ [w_committed + 1 up to w_highest] do
6:          status ← TryDirectDecide(w)
7:          if status = Commit(b_leader) then
8:              res ← IndirectDecide(w, b_leader)
9:              sequence ← sequence‖res
10:         sequence ← sequence‖status
11:     return sequence

12: procedure TryDirectDecide(w)
13:     if SkippedLeader(w) then return Skip(w)
14:     b_leader ← SupportedLeader(w)
15:     if b_leader ≠⊥ then return Commit(b_leader)
16:     return ⊥

17: procedure IndirectDecide(w, b_anchor)
18:     sequence ← [ ]
19:     b ← b_anchor
20:     w_committed ← WaveNumber(b_anchor.round)
21:     for w' ∈ [w − 1 down to w_committed + 1] do
22:         b_leader ← GetLeaderBlock(w')
23:         if CertifiedLink(b, b_leader) then
24:             sequence ← sequence‖Commit(b_leader)
25:             b ← b_leader
26:         else
27:             sequence ← sequence‖Skip(w')
28:     return sequence
```

configured but remains constant for the entire epoch. The basic committer employs one slot every three rounds, referred to as the leader slot. Employing more than one slot per round can mitigate the performance impact of crash faults, but if the validator within a slot behaves in a Byzantine manner, they 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 slots per round as an effective compromise for our experiments.

The new commit rule operates as follows: Initially, we establish a deterministic total order among all pending slots, aligning with the round ordering. Within a single round, the ordering may either remain fixed or change per round (e.g., round robin). For instance, with four validators (v1, v2, v3, v4) and two rounds (r1, r2), a potential total slot ordering for two slots per round could be represented as: [(v1, r1), (v2, r1), (v2, r2), (v3, r2)]. This order resembles a FIFO queue, with (v1, r1) at the forefront.

All pending "slots" start in an "undecided" state, and the validator subsequently utilizes Algorithm 4 to determine the status of each slot (i.e., whether to mark them as "to-commit" or "to-skip") and to produce a commit sequence.

Step 1: Direct decision rule. The validator initially employs a similar direct decision rule as in the basic Mysticeti-C (Line 9 of Algorithm 4) for each slot, beginning with the highest. A slot is marked as "to-commit" if it accumulates $2f+1$ certificates for that slot. The slot is marked as "to-skip" if it accumulates $2f + 1$ blocks that do not support any of the (potentially equivocating) blocks proposed for that slot (Line 13). Otherwise, the slot remains "undecided." Promptly

marking slots as "to-skip" contributes to the reduction of "undecided" slots following crash-failures.

Step 2: Indirect decision rule. If the direct decision rule fails to determine the slot (Line 10), the validator resorts to the indirect decision rule to attempt to reach a decision for the slot (Line 18). This rule operates in two stages. It initially searches for a "commit anchor," which is the first block with round number ($r' \geq r + \texttt{wave\_length}$) that is marked as either "undecided" or "to-commit". If the commit anchor is marked as "undecided," the validator marks the slot as "undecided" (Line 22). Conversely, if the commit anchor is marked as "to-commit," the validator checks for a certificate in the sub-graph of the commit anchor linking to that slot (Line 25), similar to the basic Mysticeti-C (see Section 3.3). If a certificate is found, the validator marks the slot as "to-commit" (Line 26) and otherwise it marks it as "to-skip" (Line 28).

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.

3.5 Block and commit timestamp

One final functionality we want to have in Mysticeti is that of exposing timestamps. Mysticeti includes a timestamp in each block and for each commit. Validators include the current time in each block they create. When a block is received its timestamp is validated by checking that the time included is greater or equal to the timestamps of included blocks, otherwise, reject the block as invalid. Honest validators will only include blocks into their blocks with past timestamps, and if a block is received with a future timestamp a validator must wait before including (or rejecting) it.

As a result, if a Byzantine validator introduces a block too far in the future, such a block will be rejected. The small variation in the local clocks of validators is mitigated by the implementation, by suspending the block in memory for a short duration if that block's timestamp is only slightly ahead of the current local time.

When Mysticeti consensus emits a commit, it also associates a timestamp with this commit, known as a commit timestamp. Commit timestamp is denoted as a maximum of the timestamp(s) of leader block(s) of such commit and the timestamp of the previous commit. As such, Mysticeti commit timestamps are guaranteed to be monotonically increasing. We have to include the commit timestamp of the previous commit in the maximum, since when pipelining the consecutive commit leader blocks are not necessarily linked by parent-child relationship, and thus cannot guarantee monotonicity.

Algorithm 4 Universal Mysticeti-C

```
1:  procedure TryCommit(r_committed, r_highest)
2:      sequence ← [ ]
3:      for r ∈ [r_highest down to r_committed + 1] do
4:          for l ∈ [k − 1 down to 0] do
5:              i ← r % wave_lenght
6:              c ← BaseCommitter(i, l)
7:              w ← c.WaveNumber(r)
8:              if c.LeaderRound(w) ≠ r then continue
9:              status ← c.TryDirectDecide(w)
10:             if status =⊥ then
11:                 status ← TryIndirectDecide(c, w, sequence)
12:             sequence ← status||sequence
13:     decided ← [ ]
14:     for status ∈ sequence do
15:         if status =⊥ then break
16:         decided ← decided||status
17:     return decided

18: procedure TryIndirectDecide(c, w, sequence)
19:     r_decision ← c.DecisionRound(w)
20:     anchors ← [s ∈ sequence s.t. r_decision < s.round]
21:     for a ∈ anchors do
22:         if a =⊥ then return ⊥
23:         if a = Commit(b_anchor) then
24:             b_leader ← c.GetLeaderBlock(w)
25:             if c.CertifiedLink(b_anchor, b_leader) then
26:                 return Commit(b_leader)
27:             else
28:                 return Skip(w)
29:     return ⊥
```

4 The Mysticeti-FPC Protocol

In Mysticeti-FPC validators include transactions in their blocks as in Mysticeti-C, but also include explicit votes for past transactions within their blocks. A correct validator votes for a transaction if it does not conflict with any other transactions they voted for. A block that causally contains $2f + 1$ votes for a transaction certifies the transaction, and once a transaction is certified validators may execute it unless it requires consensus. When a commit occurs only transactions certified by committed blocks are included in the common sequence of transactions to be executed.

4.1 Fast-path with Mysticeti-FPC

Section 3 discussed how to achieve consensus using Mysticeti-C. Nevertheless, the benefits of Mysticeti-C can also be extended to blockchains that have a consensusless path such as Sui Lutris [7]. These hybrid blockchains use the observation that some objects, such as coins, that only touch state controlled by a single party need not undergo consensus — they can be safely finalized through a fast path utilizing reliable broadcast. Such objects are said to have an 'owned-object' type. We call transactions that have all their inputs as owned objects as fast-path transactions. Unlike prior works [7], the fast path in Mysticeti-FPC is embedded inside the block DAG structure itself. This removes the need for a validator

to sign each fast-path transaction individually. Instead, a validator's fast path votes are piggybacked on its signed blocks being produced already as part of the consensus protocol. This simple optimization has three benefits:

1. The number of signature generation and verification operations is significantly reduced, thus the compute bottleneck is alleviated.
2. A separate post-consensus checkpointing mechanism is no longer required, thus reducing the sync latency. The consensus commits themselves serve as checkpoints.
3. The epoch close mechanism (Section 4.2) is simplified.

Although these are key benefits compared to prior work, there is an interesting tradeoff when we compare Mysticeti-FPC to Mysticeti-C. We now first describe the Mysticeti-FPC protocol and then discuss the latency tradeoff with Mysticeti-C protocol.

In addition to the block contents of Mysticeti-C, blocks in Mysticeti-FPC also contain explicit votes for transactions that have at least one owned-object input. Transactions are received from users, and a validator includes a transaction in its block if it does not conflict with any other transaction that it has voted for in the past. Note that a validator implicitly votes for the transactions included in its blocks. A validator, in its block B, includes an explicit vote for a transaction t, if (1) t appears in the causal history of B; and (2) t does not conflict with any other transaction that it has voted for in the past. In our implementation, we represent the vote for a transaction t appearing in block b at position i as the tuple (b, i). Votes for transactions appearing consecutively from the positions s to e are succinctly represented as the tuple (b, s, e). Of course, votes for a transaction that appears at multiple positions or in multiple blocks are added up against the same transaction.

Fast-path execution. A validator may safely execute a fast path transaction as soon as it observes blocks from $2f + 1$ validators that contain a vote for the transaction. Thanks to the quorum intersection, no correct validator will ever execute two conflicting fast-path transactions.

Fast-path finality. A block is said to certify a transaction if it links to blocks from at least $2f + 1$ validators that contain a vote for the transaction. When blocks from $2f + 1$ validators certify a fast path transaction, it is considered final (see Theorem 5.18), i.e. its effects are guaranteed to persist across epoch boundaries and reconfiguration of validators.

Consensus path. Consensus path transactions are executed after and in the order they are finalized by the commit rule. The commit rule of Mysticeti-FPC is identical to that of Mysticeti-C except for one important detail of the algorithm to convert the committed sequence of blocks to the committed sequence of transactions contained in those blocks. While Mysticeti-C allows this algorithm to commit all the contained transactions, Mysticeti-FPC places one filter on the algorithm — transactions that have at least one owned-object input are committed only if the blocks contain $2f + 1$ votes for the transaction. Thanks to the quorum intersection argument, this filter ensures, in Theorem 5.19, that the transactions executed by the fast path and the consensus path do not conflict with each other. Notice that a fast path transaction that is committed by the consensus path will have enough votes to be executed by the fast path already.

4.2 Epoch close mechanism

Blockchain protocols usually operate in epochs, in order to allow validators to leave and new validators to join the system at epoch boundaries. Additionally, the epoch boundary also serves as a natural point, for protocols that have a consensusless path, to *unlock* those transactions that have lost liveness due to equivocation from the client [7]. This committee *reconfiguration* must ensure one key safety property – transactions finalized in an epoch should continue to persist across subsequent epochs. In other words, no transactions that may be committed in future epochs should conflict with transactions finalized in the current epoch.

The safety of reconfiguration is ensured by including all the finalized transactions from the current epoch in the causal history of the epoch's final commit, which also serves as the starting state for the next epoch. Reconfiguration safety is trivial to guarantee in systems that require all transactions to undergo consensus, such as Mysticeti-C, due to the total ordering property of consensus. A deterministic consensus commit c serves as the epoch boundary between epoch e and $e+1$, such that all transactions finalized in epoch e appear in and before commit c.

Reconfiguration solutions are however non-trivial for systems that have a consensusless fast path, such as Mysticeti-FPC. There is a race between finalized transactions being included in the consensus commits and new transactions being finalized by the fast path. If the epoch is closed trivially, the final commit of the epoch may fail to include all the transactions finalized by the fast path, violating the key safety property of reconfiguration. In what follows, we describe the mechanism for closing an epoch safely in Mysticeti-FPC.

Recall that a block B is said to certify a transaction tx if the causal history of B contains $2f + 1$ votes for tx. However, we now introduce an overriding bit, called *epoch-change bit* in Mysticeti-FPC blocks, which when set to 1 (default set to 0), indicates that the block does not certify any transaction, regardless of the causal history. This epoch-change bit, in effect, allows for pausing the consensusless fast path of Mysticeti-FPC near the closing of the epoch, thus averting the race condition highlighted above.

Epoch change begins at a pre-determined commit, for example, when a smart contract indicates that the new committee is ready to take over. As soon as an honest validator

determines that the epoch change has begun, it stops including transactions or casting votes for any fast path transactions and sets the epoch-change bit to 1 in all its future blocks for this epoch. Additionally, although the validator continues to advance its rounds and participate in consensus, it stops contributing to the processing and finalization of fast-path transactions. Once blocks from $2f + 1$ validators with the epoch-change bit set are committed by the consensus path, the epoch is considered closed. As we prove in Theorem 5.18, this epoch close mechanism guarantees the key safety property that transactions finalized in an epoch continue to persist in all subsequent epochs. The liveness of the algorithm is trivial as it simply piggybacks on the liveness of Mysticeti-C. Once the epoch is considered closed, any continuing validator may unlock the fast path transactions, which could not get finalized due to equivocation by the client, for fresh votes in the subsequent epochs allowing for equivocation tolerance.

5 Security Arguments

In this Section, we show the correctness of Mysticeti. 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 5.1. *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 5.2. *If a correct validator commits some block in a slot s, then no other 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 5.3. *If a correct validator directly commits some block in a slot s, then no other correct validator decides to skip the slot 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 5.2.

Case (b). Let block b' denote the leader 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 5.1.

□

Lemma 5.4. *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 5.5. *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 5.4. □

As a result of Lemma 5.5, we get the following corollary:

Corollary 5.6. *No two correct validators commit distinct blocks for the same slot.*

Lemma 5.7. *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 5.2. 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 the slot i as it is the highest committed slot for X. Due to Lemma 5.3, if Y decides the slot i, then it must also commit the slot i. By Corollary 5.6, 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 5.2. 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 5.8. *All correct validators commit a consistent sequence of leader blocks (i.e., the committed leader sequence of one correct validator is a prefix of another's).*

Proof. The committed sequence of leader blocks is nothing but the sequence of committed blocks before the first undecided slot. The statement is then a direct implication of Lemma 5.7. □

Theorem 5.9 (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 leader blocks. Since a correct validator has all of the causal histories of a block when the block is added to its DAG, and the sequence of committed leader blocks of one validator is a prefix of another's (Lemma 5.8), all correct validators deliver a consistent sequence of blocks, i.e. the sequence of blocks delivered from one validator is a prefix of another. The total order property of BAB immediately follows. □

Theorem 5.10 (Integrity)**.** *Mysticeti-C satisfies the integrity property of Byzantine Atomic Broadcast.*

Proof. The algorithm to linearize the causal history of a committed leader block removes any block with duplicate sequence numbers before delivering the sequence of blocks. □

Lemma 5.11 (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 5.12 (Leader-Proposal)**.** *After GST an honest leader'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 leader (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 5.11). Then the honest leader will directly propose its block. Since the timeout is set to $2 * \Delta$ the leader's proposal of wave w will arrive before the first honest validator times out hence, every honest validator will vote for the leader. □

Lemma 5.13 (Sufficient Votes)**.** *After GST all honest validators will create a certificate for the honest leader.*

Proof. By Lemma 5.12 all honest validators will vote for an honest leader after GST. For an honest validator to propose a block at the decision round it needs to (a) get the proposal of the leader and (b) have $2f+1$ parents. All honest validators receive the leader proposal within Δ since the leader is honest. Additionally from the moment one honest validator advances to the decision round all honest validators will receive its block proposal and adopt the parents within Δ. As a result, by the code, all 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 5.14. *The round-robin schedule of leaders in Mysticeti ensures that in any window of* $3f+3$ *rounds, there are three consecutive rounds with honest primary leaders. A primary leader is the leader 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 leader 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 leaders. □

Lemma 5.15. *After GST any undecided slot eventually gets decided.*

Proof. Let there be an undecided slot s in round r. After GST, due to Lemma 5.14, there will eventually be an honest leader 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 5.13, the honest leader'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 due to the induction hypothesis. □

Theorem 5.16 (Consensus Liveness). *After GST an honest leader's proposal will commit.*

Proof. By Lemma 5.13 there will be $2f+1$ certificates for the leader, one per honest party. By the code an honest validator tries to commit the leader for every block they get so eventually they will get the $2f+1$ certificates. The validator schedules the block to be committed. By Lemma 5.15, all prior undecided blocks will eventually be decided, and the validator will deliver the honest leader's block. □

Theorem 5.17 (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 leader 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 leader in round $r' > r$ as per the leader schedule of Mysticeti-C. Due to Theorem 5.16, after GST, Y will commit the leader's block b'. Due to Lemma 5.8, Y will also commit the leader sequence L before committing b'. Since Y follows an identical deterministic algorithm as X to deliver blocks from the committed sequence of leader blocks, it also delivers b' eventually. □

5.1 Security Arguments of Mysticeti-FPC

Theorem 5.18 (Epoch close safety). *Transactions finalized by Mysticeti-FPC 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 5.19 (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 5.20 (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 $4 * \Delta$ since any round has at most duration of timeout+Δ=$3 * \Delta$. In the next round, every honest validator will reference the $2f+1$ votes in their DAG and execute. □

Theorem 5.21 (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 5.20. 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 5.20. □

6 Implementation

We implement a networked multi-core Mysticeti validator in Rust. It uses `tokio`[4] for asynchronous networking, utilizing TCP sockets for communication without relying on any RPC frameworks. For cryptographic operations, we rely on the efficient `ed25519-consensus`[5] for asymmetric cryptography and `blake2`[6] for cryptographic hashing. To ensure data persistence and crash recovery, we've integrated a Write-Ahead Log (WAL), seamlessly tailored to our specific requirements. We've intentionally avoided key-value stores

[4] https://tokio.rs
[5] https://github.com/penumbra-zone/ed25519-consensus
[6] https://github.com/RustCrypto/hashes

like RocksDB[7] to eliminate associated overhead and periodic compaction penalties. Our implementation optimizes I/O operations by employing vectored writes[8] 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`[9] 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 networking layer 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 and addressing efficiency bottlenecks.

We are open-sourcing our Mysticeti implementation, along with its simulator and orchestration utilities[10].

7 Evaluation

We evaluate the throughput and latency of Mysticeti through experiments on Amazon Web Services (AWS). We then show its performance improvements over several state-of-the-art protocols. Despite the large number of BFT consensus protocols [11, 12, 14, 19, 29, 35, 43], we opt to compare Mysticeti-C with vanilla HotStuff [44], HotStuff-over-Narwhal (called *Narwhal-HotStuff*) [17], and Bullshark [33]. We select these protocols for the availability of open-source implementations and detailed benchmarking scripts, their similarity to Mysticeti, and their adoption in real-world deployments. We specifically select the Jolteon [20] variant of HotStuff as it has been adopted by Flow [39], Diem [4], Aptos [37], and Monad [30]. 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 [7, 41] and in the roadmap for integration within Aptos. We evaluate the 1-worker variants of the Narwhal-based systems (that is, Narwhal-HotStuff and Bullshark). We also evaluate the fast path Mysticeti-FPC against Zef [6] (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 [40].

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 [3], and state-of-the-art evidence relies on formal proofs of safety and liveness (see Section 5).

7.1 Experimental setup

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 10 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 random and contain 512 bytes, and Mysticeti is instantiated with two leaders per round (see Section 3.4).

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 entire duration of the run. Appendix C provides a tutorial to reproduce our experiments.

7.2 Benchmark in ideal conditions

Figure 3 illustrates the Latency (seconds) - Throughput (Transactions per second, TPS) relationship for Mysticeti-C compared with other consensus protocols, for a small deployment of 10 validators (up to 3 tolerable failures) and a larger deployment of 50 validators (up to 16 tolerable failures). The systems run in ideal conditions, without faults.

At a steady state of 50k to 100k 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

[7] https://rocksdb.org
[8] https://linux.die.net/man/3/writev
[9] https://github.com/facebook/sapling/tree/main/eden/scm/lib/minibytes
[10] https://github.com/MystenLabs/mysticeti/tree/paper (commit aee594d)

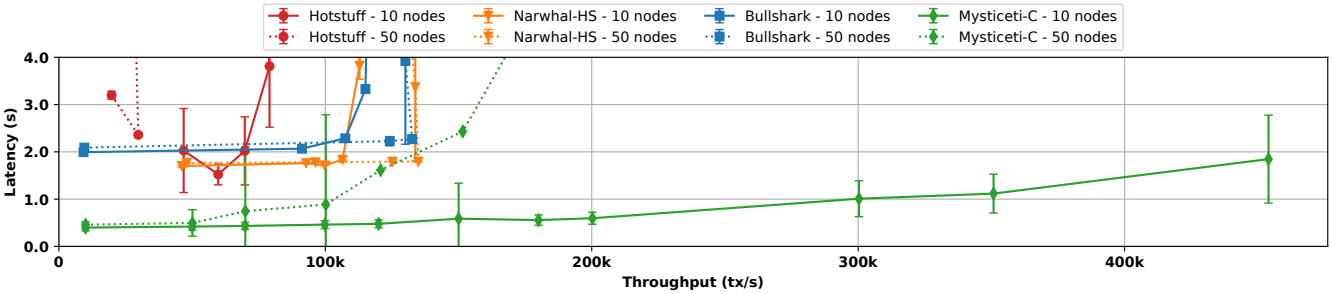

Figure 3. Throughput-Latency graph comparing Mysticeti-C performance with state-of-the-art consensus protocols.

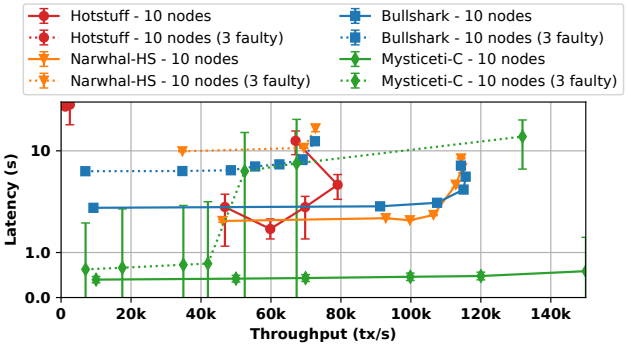

Figure 4. Throughput - Latency under crash faults. Note the log y-axis, for latency.

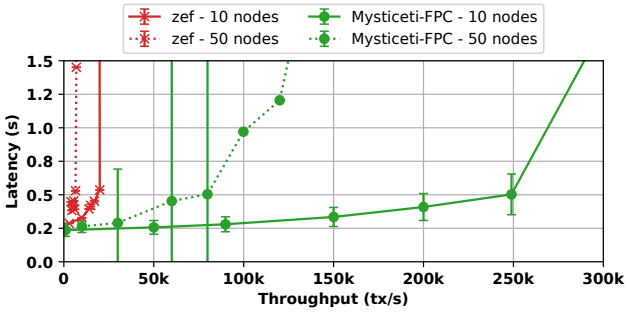

Figure 5. Throughput - Latency comparison the for fast path commits between Mysticeti-FPC and Zef

architecture and is over 3x slower in terms of latency compared with Mysticeti-C for low system loads. In terms of throughput, the smaller Mysticeti-C network scales extremely well and achieves a throughput of over 400k TPS before latency reaches 1.5s, that is, comparable to the latency of state-of-the-art systems. The larger deployment scales to 120k TPS before latency goes over 1.5s, which is comparable to the single worker variant of Narwhal-based designs, and HotStuff variants. This illustrates that the single-host throughput efficiency of Mysticeti-C is higher than for previous designs. Note that current real-world blockchains combined[11] 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 (which we discuss later).

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.

7.3 Benchmark with faults

Figure 4 illustrates the performance of HotStuff, Narwhal-HotStuff, Bullshark, and Mysticeti-C when a committee of 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 60-80k TPS in about 8-10 seconds. In contrast, Mysticeti-C can process up to 50k TPS while maintaining sub-second latency and up to 80k TPS with comparable latency to Narwhal-HotStuff and Bullshark. As a result, Mysticeti-C 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 can handle a similar throughput to state-of-the-art consensus protocols but with sub-second latency despite the presence of crash faults.

7.4 Benchmark of the fast path

Figure 5 illustrates the Latency - Throughput of fast path commits for Mysticeti-FPC, compared with Zef [6] when deployed without privacy protections[12]. 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

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

[12] Zef can also be instantiated to leverage the Coconut threshold credentials system [32] to provide privacy guarantees at the cost of performance.

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.

8 Related Work

Mysticeti is a family of protocols designed to support next-generation distributed ledgers. 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 blockchains [7] which has been productionized by Sui [41]. 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 [5] and Bullshark [33]. 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 [9] for consensus and Reliable Broadcast [8] for consensus-less distributed ledgers with equivocation tolerance.

We now focus on the different variants of Mysticeti, namely Mysticeti-C and Mysticeti-FPC. 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 [5], Zef [6], and Astro [15] in order to get better privacy guarantees.

In terms of consensus, the most recent DagRider [23], Narwhal-Tusk [17], Bullshark [33], DisperseLedger [43] 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. Mysticeti uses instead a DAG of signed but not certified blocks, reducing latency significantly. The base 3-round commit rule of Mysticeti-C was developed concurrently to Cordial Miners [24], without reference to fast path, pipelining or multiple leaders, and without an implementation or an experimental evaluation. However, the proof of safety and liveness apply to both and reassure us of the correctness of the base commit rule. The subsequent concurrent work on Flash [28] also discussed how to leverage a blocklace/DAG to allow for payments akin to the Mysticeti-FPC fast path, without however integrating it with a consensus path for more complex transactions.

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. Shoal [34] defines a pipelined variant of Bullshark and other Narwhal-based protocols, this can be seen as a similar contribution to the universal committer of Mysticeti if set with one leader per round. We plan in our future work to adapt our universal committer to Bullshark in order to have a fair comparison over certified DAGs. Additionally, Shoal and HammerHead [42] propose leader reputation protocols inspired by Carousel [13]. Mysticeti-C could adopt these designs to select more reliable leaders, but for liveness, it would need to adopt a leader rotation schedule where leaders remain static for 3 rounds.

Previous consensus protocols such as Hashgraph [2] and Blockmania [16] also use a DAG of signed but not certified blocks: however they use DAGs that are not structured as threshold clocks [18] making the proofs of safety for them very complex. Notably, Mysticeti-C works in only 3 message communication rounds which match PBFT, and is optimal latency [1, 10] without the use of optimistic methods like Zyzzyva [26]. This is lower than the state-of-the-art Jolteon [21] currently deployed in multiple blockchains [30, 37–39]. 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 DagRider and Narwhal. Finally, non-byzantine variants of consensus have also been defined on threshold clocks, such as Que-Paxa [36] but cannot be deployed as part of distributed ledgers.

9 Conclusion

We introduced Mysticeti, a threshold clock-based byzantine consensus protocol with the lowest WAN latency of 0.5s and the ability to process over 50k TPS at this latency for single-host nodes, far exceeding the needs of blockchains today (which consume in total about 1.2k TPS). Its fast path achieves even lower latency at 0.25s but is over 8x more resource efficient compared with protocols with explicit certificates. It is also more crash tolerant using multiple leaders per round, implemented through a universal commit 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.

Acknowledgements

We would like the thank Dmitry Perelman, Mingwei Tian, Tasos Kichidis, and Xun Li from the MystenLabs engineering team for the great discussions that improved this work. This work was conducted while Kushal Babel was interning with Mysten Labs.

References

[1] Abraham, I., Nayak, K., Ren, L., and Xiang, Z. Good-case latency of byzantine broadcast: A complete categorization. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2021), PODC'21, Association for Computing Machinery, p. 331–341.

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

[3] Bano, S., Sonnino, A., Chursin, A., Perelman, D., Li, Z., Ching, A., and Malkhi, D. Twins: Bft systems made robust. In *25th International Conference on Principles of Distributed Systems* (2022).

[4] Baudet, M., Ching, A., Chursin, A., Danezis, G., Garillot, F., Li, Z., Malkhi, D., Naor, O., Perelman, D., and Sonnino, A. State machine replication in the libra blockchain. *The Libra Assn., Tech. Rep 1*, 1 (2019).

[5] Baudet, M., Danezis, G., and Sonnino, A. Fastpay: High-performance byzantine fault tolerant settlement. In *AFT '20: 2nd ACM Conference on Advances in Financial Technologies, New York, NY, USA, October 21-23, 2020* (2020), ACM, pp. 163–177.

[6] Baudet, M., Sonnino, A., Kelkar, M., and Danezis, G. Zef: Low-latency, scalable, private payments. *CoRR abs/2201.05671* (2022).

[7] Blackshear, S., Chursin, A., Danezis, G., Kichidis, A., Kokoris-Kogias, L., Li, X., Logan, M., Menon, A., Nowacki, T., Sonnino, A., et al. Sui lutris: A blockchain combining broadcast and consensus. Tech. rep., Technical Report. Mysten Labs. https://sonnino.com/papers/sui-lutris. pdf, 2023.

[8] Bracha, G., and Toueg, S. Asynchronous consensus and broadcast protocols. *J. ACM 32*, 4 (1985), 824–840.

[9] Castro, M., and Liskov, B. Practical byzantine fault tolerance. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999* (1999), M. I. Seltzer and P. J. Leach, Eds., USENIX Association, pp. 173–186.

[10] Chan, B. Y., and Pass, R. Simplex consensus: A simple and fast consensus protocol. Cryptology ePrint Archive, Paper 2023/463, 2023. https://eprint.iacr.org/2023/463.

[11] Chan, B. Y., and Shi, E. Streamlet: Textbook streamlined blockchains. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies* (2020), pp. 1–11.

[12] Chen, J., Gupta, S., Sonnino, A., Kokoris-Kogias, L., and Sadoghi, M. Resilient consensus sustained collaboratively. *arXiv preprint arXiv:2302.02325* (2023).

[13] Cohen, S., Gelashvili, R., Kogias, L. K., Li, Z., Malkhi, D., Sonnino, A., and Spiegelman, A. Be aware of your leaders. In *International Conference on Financial Cryptography and Data Security* (2022), Springer, pp. 279–295.

[14] Cohen, S., Goren, G., Kokoris-Kogias, L., Sonnino, A., and Spiegelman, A. Proof of availability & retrieval in a modular blockchain architecture. *Cryptology ePrint Archive* (2022).

[15] Collins, D., Guerraoui, R., Komatovic, J., Monti, M., Xygkis, A., Pavlovic, M., Kuznetsov, P., Pignolet, Y.-A., Seredinschi, D.-A., and Tonkikh, A. Online payments by merely broadcasting messages (extended version). *arXiv preprint arXiv:2004.13184* (2020).

[16] Danezis, G., and Hrycyszyn, D. Blockmania: from block dags to consensus. *arXiv preprint arXiv:1809.01620* (2018).

[17] Danezis, G., Kokoris-Kogias, L., Sonnino, A., and Spiegelman, A. Narwhal and tusk: a dag-based mempool and efficient BFT consensus. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022* (2022), Y. Bromberg, A. Kermarrec, and C. Kozyrakis, Eds., ACM, pp. 34–50.

[18] Ford, B. Threshold logical clocks for asynchronous distributed coordination and consensus. *CoRR abs/1907.07010* (2019).

[19] Gao, Y., Lu, Y., Lu, Z., Tang, Q., Xu, J., and Zhang, Z. Dumbo-ng: Fast asynchronous bft consensus with throughput-oblivious latency. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (2022), pp. 1187–1201.

[20] Gelashvili, R., Kokoris-Kogias, L., Sonnino, A., Spiegelman, A., and Xiang, Z. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In *International Conference on Financial Cryptography and Data Security* (2022), Springer, pp. 296–315.

[21] Gelashvili, R., Kokoris-Kogias, L., Sonnino, A., Spiegelman, A., and Xiang, Z. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In *Financial Cryptography and Data Security - 26th International Conference, FC 2022, Grenada, May 2-6, 2022, Revised Selected Papers* (2022), I. Eyal and J. A. Garay, Eds., vol. 13411 of *Lecture Notes in Computer Science*, Springer, pp. 296–315.

[22] Gelashvili, R., Spiegelman, A., Xiang, Z., Danezis, G., Li, Z., Malkhi, D., Xia, Y., and Zhou, R. Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2023, Montreal, QC, Canada, 25 February 2023 - 1 March 2023* (2023), M. M. Dehnavi, M. Kulkarni, and S. Krishnamoorthy, Eds., ACM, pp. 232–244.

[23] Keidar, I., Kokoris-Kogias, E., Naor, O., and Spiegelman, A. All you need is DAG. In *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021* (2021), A. Miller, K. Censor-Hillel, and J. H. Korhonen, Eds., ACM, pp. 165–175.

[24] Keidar, I., Naor, O., Poupko, O., and Shapiro, E. 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* (2023), R. Oshman, Ed., vol. 281 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 26:1–26:22.

[25] Kokoris-Kogias, L., Sonnino, A., and Danezis, G. Cuttlefish: Expressive fast path blockchains with fastunlock. *arXiv preprint arXiv:2309.12715* (2023).

[26] Kotla, R., Alvisi, L., Dahlin, M., Clement, A., and Wong, E. L. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007* (2007), T. C. Bressoud and M. F. Kaashoek, Eds., ACM, pp. 45–58.

[27] Lamport, L. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001),

51–58.
[28] Lewis-Pye, A., Naor, O., and Shapiro, E. Flash: An asynchronous payment system with good-case linear communication complexity. *CoRR abs/2305.03567* (2023).
[29] Malkhi, D., and Szalachowski, P. Maximal extractable value (mev) protection on a dag. *arXiv preprint arXiv:2208.00940* (2022).
[30] Monad. Monadbft:pipelined two-phase hotstuff consensus. *https://docs.monad.xyz/technical-discussion/consensus/monadbft* (2023).
[31] Patrignani, M., and Blackshear, S. Robust safety for move. In *36th IEEE Computer Security Foundations Symposium, CSF 2023, Dubrovnik, Croatia, July 10-14, 2023* (2023), IEEE, pp. 308–323.
[32] Sonnino, A., Al-Bassam, M., Bano, S., Meiklejohn, S., and Danezis, G. 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* (2019), The Internet Society.
[33] Spiegelman, A., andF Alberto Sonnino, N. G., and Kokoris-Kogias, L. Bullshark: DAG BFT protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022* (2022), H. Yin, A. Stavrou, C. Cremers, and E. Shi, Eds., ACM, pp. 2705–2718.
[34] Spiegelman, A., Aurn, B., Gelashvili, R., and Li, Z. Shoal: Improving DAG-BFT latency and robustness. *CoRR abs/2306.03058* (2023).
[35] Spiegelman, A., Aurn, B., Gelashvili, R., and Li, Z. Shoal: Improving dag-bft latency and robustness. *arXiv preprint arXiv:2306.03058* (2023).
[36] Tennage, P., Basescu, C., Kokoris-Kogias, L., Syta, E., Jovanovic, P., Estrada-Galiñanes, V., and Ford, B. 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* (2023), J. Flinn, M. I. Seltzer, P. Druschel, A. Kaufmann, and J. Mace, Eds., ACM, pp. 281–297.
[37] The Aptos team. Aptos. https://aptoslabs.com, 2023.
[38] The Diem Team. Diembft v4. https://developers.diem.com/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2021-08-17.pdf, 2021.
[39] The Flow Team. The flow blockchain. https://flow.com, 2023.
[40] The Linera Team. Linera. https://linera.io, 2023.
[41] The Sui team. The sui blockchain. http://sui.io, 2023.
[42] Tsimos, G., Kichidis, A., Sonnino, A., and Kokoris-Kogias, L. Hammerhead: Leader reputation for dynamic scheduling. *arXiv preprint arXiv:2309.12713* (2023).
[43] Yang, L., Park, S. J., Alizadeh, M., Kannan, S., and Tse, D. {DispersedLedger}:{High-Throughput} byzantine consensus on variable bandwidth networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (2022), pp. 493–512.
[44] Yin, M., Malkhi, D., Reiter, M. K., Golan-Gueta, G., and Abraham, I. 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* (2019), P. Robinson and F. Ellen, Eds., ACM, pp. 347–356.

A Fast-Unlock of Blocked Transactions in Mysticeti-FPC

One peculiarity of the current design of Mysticeti-FPC is that some fast-path transactions might take more time to execute in Mysticeti-FPC than they would take if we simply run Mysticeti-C. This is because simply ordering a fast-path transaction is not sufficient as a conflicting transaction might have been certified concurrently. Hence in Mysticeti-FPC we need to wait for a certificate to be ordered instead of simply the transaction.

In this Section we discuss how we can add a few extra validity rules in Mysticeti-FPC's block creation rule that create sufficient backpressure such that if a tranasction is ordered by consensus and no conflicting transaction is visible in the DAG at that point in time then it should have also finalized in the fast-path and executed.

The Rule: More specificially we enhance the block creation of Mysticeti-FPC with three additional rules:

- A node includes in their block only tranasctions that can be executed. This means that all their causal depedencies have been decided inside their subDAG
- Since now any included transactions can be decided, every validator needs to cast votes for all transactions in its causal history as part of their block
- Finally, if a negative vote is cast, the validator needs to support this with a pointer (hash) to the conflicting transaction it has instead supported.

Sketch Proof This addition to the rules allows us to show that a transaction without conflicts gets certified the latest at the point that it is ordered by consensus hence there is no latency hit. This cannot be said for transaction with contention where, unlike in Mysticeti-C, they will have to pay the extra latency or resolving the race through the Fast Unlock protocol in Appendix 5.1.

This is true because a non-contended transaction that is included as part of a leader proposal will get positive votes from all participating parties in the voting round of consensus. Since at least $2f + 1$ votes are needed to advance the DAG this will be an implicit certificate. As a result at the decision round of Mysticeti-C the certificate will be included by all parties participating hence by the time the transaction is ordered it should already have been executed through the fast-path.

Reducing the load of the catchup One last issue that this backpressure creates is that struggling parties now need to cast an immense ammount of votes before they can again participate in the consensus protocol. To avoid this we only require votes for transactions that are undecided as far as the view of the leader of the round is concerned. This means that transactions with a certificate and transactions where a conflicting transaction has a certificate need not be explicitly voted again as we already know the decision. Given that most transactions get a decision in 200-500ms then the struggler validator only need to vote for the last part of the DAG which is not a big burder.

B Fast-Unlock of blocked transactions in Mysticeti-FPC

A key challenge with fast-path transactions that exists in all consensus-less systems and carries over at Mysticeti-FPC is that clients must ensure that conflicting transactions are never issued. This limitation can be quite strict, and even

minor software bugs in client implementations can result in assets becoming locked. For instance, if a defective wallet accidentally sends a transaction with a randomized signature twice, it might be interpreted as two conflicting transactions, leading to asset lockup. Similarly, problems can arise when a client miscalculates the required gas for a transaction and tries to reissue it with a higher gas amount.

B.1 Fast-Unlock

Cuttlefish [25] proposed a solution to this problem by introducing a Fast-Unlock protocol. However, as Cuttlefish is build on Sui it requires significant effort to handle client messages and coordinate validators. Mysticeti-FPC in contrast exposes all transactions inside its DAG. As a result we can create simplified and highly efficient unlock protocol.

First, we define two different paths to invoke the unlock protocol, the validator-initiated path and the user-initiated path. Unlocking means (using the notation of [25]) increasing the version number of the contented objects without changing their value. This invalidates the conflicting transactions as they now are accessing an unavailable version and allows for a new transactions to correctly access the fast-path and commit.

The simplest way to invoke the unlock procedure is when a validator witnesses two valid transactions (have correct authentication) which try to access the same object, creating a conflict. If this happens then the validator proposes the "unlock" the object in a new transaction in its DAG which is included in a block that has both conflicting transactions in its causal history. The second way is for the client to sign a cancellation transaction, which needs to be accompanied with an authenticator showing that the client can legitimately access the owned objects of the transaction.

B.2 Voting Rule

Once a valid unlock transaction is included in a block validators treat it as any other consensus transaction with two added requirements. First, when they process the transaction they check that they have not witnessed yet a certificate for the object to be unlocked. If this is true they accept the transaction, which also means that for the object under question they no longer process certificates. As a result, similar to epoch change, the validators history no longer can be considered to support certification for the objects in question even if a certificate appears in their causal history. In cases that in the same round both a fast-unlock transaction is accepted and a certificate is included[13] we follow the explicit parent ordering between blocks to decide if the certificate came first and the unlock should be ignored.

Finally, the validators vote for the unlock transaction as if it is a shared object transaction. There are two possible outcomes for this transaction:

- The unlock transaction was issued by a client but it was too late. Hence a a certificate was already created and witnessed by honest validators. Then these honest validators will not vote for the transaction and it will never get certified. However, this is not an issue as the original certificate will follow the normal operation path and commit the original transaction
- The unlock transaction gets certified. Then we execute as a shared object transaction an no-op the objects in question

Notice that there is a possibility that there is a certificate both for a transaction and for an unlock. Then we let consensus resolve the race, meaning that whichever executes first on consensus is the valid one and the other fails with an "object not available" error.

C Reproducing Experiments

We provide the orchestration scripts [14] used to benchmark the codebase evaluated in this paper on AWS .

Deploying a testbed. The file '~/.aws/credentials' should have the following content:

```
[default]
aws_access_key_id = YOUR_ACCESS_KEY_ID
aws_secret_access_key = YOUR_SECRET_ACCESS_KEY
```

configured with account-specific AWS *access key id* and *secret access key*. It is advise to not specify any AWS region as the orchestration scripts need to handle multiple regions programmatically.

A file 'settings.json' contains all the configuration parameters for the testbed deployment. We run the experiments of Section 7 with the following settings:

```
{
    "testbed_id": "${USER}-hammerhead",
    "cloud_provider": "aws",
    "token_file": "/Users/${USER}/.aws/credentials
        ",
    "ssh_private_key_file": "/Users/${USER}/.ssh/
        aws",
    "regions": [
        "us-east-1",
        "us-west-2",
        "ca-central-1",
        "eu-central-1",
        "ap-northeast-1",
        "ap-northeast-2",
        "eu-west-1",
        "eu-west-2",
        "eu-west-3",
        "eu-north-1",
        "ap-south-1",
        "ap-southeast-1",
        "ap-southeast-2"
    ],
    "specs": "m5d.8xlarge",
```

[13] Which is Byzantine behaviour.

[14] https://github.com/MystenLabs/mysticeti/tree/paper (commit aee594d)

```
    "repository": {
        "url": "https://github.com/AUTHOR/REPO.git
            ",
        "commit": "main"
    }
}
```

where the file '/Users/$USER/.ssh/aws' holds the ssh private key used to access the AWS instances, and 'AUTHOR' and 'REPO' are respectively the GitHub username and repository name of the codebase to benchmark.

The orchestrator binary provides various functionalities for creating, starting, stopping, and destroying instances. For instance, the following command to boots 2 instances per region (if the settings file specifies 13 regions, as shown in the example above, a total of 26 instances will be created):

```
cargo run --bin orchestrator -- testbed deploy --instances 2
```

The following command displays he current status of the testbed instances

```
cargo run --bin orchestrator testbed status
```

Instances listed with a green number are available and ready for use and instances listed with a red number are stopped. It is necessary to boot at least one instance per load generator, one instance per validator, and one additional instance for monitoring purposes (see below). The following commands respectively start and stop instances:

```
cargo run --bin orchestrator -- testbed start
cargo run --bin orchestrator -- testbed stop
```

It is advised to always stop machines when unused to avoid incurring in unnecessary costs.

Running Benchmarks. Running benchmarks involves installing the specified version of the codebase on all remote machines and running one validator and one load generator per instance. For example, the following command benchmarks a committee of 100 validators (none faulty) under a constant load of 1,000 tx/s for 10 minutes (default):

```
cargo run --bin orchestrator -- benchmark \
    --committee 100 fixed-load --loads 1000 --faults 0
```

Monitoring. The orchestrator provides facilities to monitor metrics. It deploys a Prometheus instance and a Grafana instance on a dedicated remote machine. Grafana is then available on the address printed on `stdout` when running benchmarks with the default username and password both set to `admin`. An example Grafana dashboard can be found in the file 'grafana-dashboard.json'[15].

[15] https://github.com/MystenLabs/mysticeti/blob/paper/orchestrator/assets/grafana-dashboard.json