Stingray: Fast Concurrent Transactions Without Consensus

Srivatsan Sridhar
Stanford University

Alberto Sonnino
Mysten Labs
University College London

Lefteris Kokoris-Kogias
Mysten Labs

Abstract

Recent advances have improved the throughput and latency of blockchains by processing transactions accessing different parts of the state concurrently. However, these systems are unable to concurrently process (a) transactions accessing the same state, even if they are (almost) commutative, e.g., payments much smaller than an account's balance, and (b) multi-party transactions, e.g., asset swaps. Moreover, they are slow to recover from contention, requiring once-in-a-day synchronization. We present Stingray, a novel blockchain architecture that addresses these limitations. The key conceptual contributions are a replicated bounded counter that processes (almost) commutative transactions concurrently, and a FastUnlock protocol that uses a fallback consensus protocol for fast contention recovery. We prove Stingray's security in an asynchronous network with Byzantine faults and demonstrate on a global testbed that Stingray achieves 10,000 times the throughput of prior systems for commutative workloads.

ACM Reference Format:
Srivatsan Sridhar, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2025. Stingray: Fast Concurrent Transactions Without Consensus. In . ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Blockchain technology has fundamentally transformed the landscape of digital transactions, providing a decentralized and secure framework for managing digital assets. Despite its innovative potential, current blockchains [14, 22, 35] face significant scalability and efficiency challenges. Two key issues are the batch commit of blocks, which necessitates transactions to wait until a block is ready to be proposed, and the sequential execution of transactions, which fails to exploit modern CPU architectures capable of parallel processing. These limitations introduce latency and restrict throughput, impeding the full utilization of available resources.

To mitigate these bottlenecks, recent advances have focused on enhancing concurrency in transaction processing. This has been achieved through improvements in both the agreement module via consensus-less fast paths [10, 21, 30, 42] and the execution module via parallel execution engines [24, 29, 30, 45]. Existing consensus-less fast paths enable concurrency and significantly reduce latency for transactions that have a consensus number of 1 [25], such as payments and transfers, and facilitate easy identification of transactions accessing independent resources for parallel execution [30, 45].

Despite these advancements, existing fast-path protocols and parallel execution engines still have limitations. First, while they parallelize transactions touching independent parts of the state, e.g., payments to/from different accounts, transactions accessing the same account are sequential because they incorporate a version number derived from the preceding transaction on the same account. This lack of concurrency limits performance gains for commutative transactions that do not inherently require sequentiality.

Second, the fast path has been confined to a limited range of transactions. For example, in systems such as Sui [13], FastPay [10] and Astro [21], fast-path transactions are restricted to objects owned by a single entity, allowing simple payments and asset transfers but excluding more complex operations such as asset swaps or multi-signature authorizations. This limitation significantly decreases the potential transaction load that could benefit from the fast path.

Finally, if a user submits concurrent conflicting transactions through the fast path, the system must lock the object until the system can deterministically resolve this deadlock. While it is anticipated that users will not send conflicting transactions, such conflicts may arise due to minor bugs in wallet implementations or malicious collusion among signatories in multi-signature structures. Deployed systems, such as Sui [30], can take up to 24 hours to resolve these conflicts, resulting in a suboptimal user experience.

We address the first limitation by allowing concurrent transactions that are commutative or almost commutative. We were inspired by the database literature, where systems can deduce (or be instructed) that accessing the same resource can occur safely when actions are commutative [26]. This is captured using Conflict-free Replicated Data Types (CRDTs) [36]. However, using CRDTs is insufficient, as any transaction in a blockchain system must pay for gas. This gas payment (and any payment transaction) has a non-commutative comparison with zero. To our knowledge, the only algorithms that bridge this gap between commutativity and comparing a counter (or set) with a bound (i.e., nearly commutative) have been proposed under crash faults [1, 36, 44] and break under Byzantine faults. We resolve this open question in Stingray with *the first Byzantine fault-tolerant bounded counter*. The intuition behind this construction is to (a) provide a local budget per validator for signing transactions but (b) require quorums for approval such that even if every quorum includes all malicious validators the global budget can still not be overspent. This allows users to spend up to half their account's value concurrently before resynchronization. Importantly, resynchronization still happens through a consensus-less fast path, solving the problem in asynchrony using protocols resembling reliable broadcast [15].

Once we identify that the real problem in concurrent distributed ledgers is not the concurrent accesses on the same memory location

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

(i.e., *concurrency*), but the non-monotonic [32] accesses that cannot be reordered (i.e., *contention*), we address the second limitation by extending the fast path to accommodate transactions involving objects owned by different users, such as asset swaps, and support complex authorization structures, including threshold and logical combinations of user authorizations. These transactions are finalized extremely fast and in parallel as long as there is no contention over the resources accessed. However, the absence of contention is no longer guaranteed if one of the users is malicious, risking loss of liveness. The last challenge we solve is alleviating this risk through our novel FastUnlock protocol that leverages the consensus of hybrid blockchains [5, 13] to swiftly resolve conflicting transactions and enable users to submit new transactions within the latency of consensus protocols (e.g., 400ms for Mysticeti [5]).

We implement Stingray on top of the consensus-less fast path of Sui [30], called Mysticeti-FPC [5], and show that it provides significant throughput and latency improvements for transaction loads concurrently accessing the same resources with commutative operations (20,000 tps in 0.5 seconds instead of 2 tps). Additionally, we demonstrated that Stingray incurs no performance penalty for fully parallelizable workloads, both with and without faults, when compared to Sui.

Contributions. This paper makes the following contributions:

- We introduce the first bounded counter with Byzantine fault tolerance, a new object type that supports commutative operations, such as addition and subtraction, and some non-commutative operations, such as comparison with zero for account balances, without consensus.
- We propose a new protocol, FastUnlock, that enables the fast resolution of conflicting transactions within the latency of consensus protocols.
- We present Stingray, a novel system that applies these techniques to existing fast-path protocols. In addition to bounded counters and FastUnlock, Stingray supports on the fast path, transactions involving objects owned by different users, such as asset swaps, and complex authorization structures, including thresholds and other logical combinations.
- We formally prove the safety and liveness of Stingray in the asynchronous network model with Byzantine faults.
- We implement Stingray and evaluate it in a realistic geo-replicated environment to demonstrate that it outperforms the state of the art by 10,000x for a commutative workload.

2 Background

Several consensus-less systems have been proposed in the literature, including FastPay [10], Astro [21], Zef [11], and Linera [33]. In this section, we recap Sui (the Sui Lutris mechanism [30]) as a basis for the Stingray design, as it is the only currently deployed system supporting a consensus-less fast path. Stingray improves upon the Sui design (a) the concurrency admitted between transactions by exploiting commutativity and (b) the scope of object authentication and transactions in the fast path. Finally, Stingray recognizes that these benefits apply to the fast path only in the optimistic case and mitigates this by (c) extending Sui's consensus with FastUnlock to mitigate the worst case of contented transactions.

2.1 Data Structures

Object Types. The Sui blockchain state consists of a set of objects categorized into three types. Sui determines whether to use the fast path or the consensus path for a transaction based on the types of objects involved.

- *Read-only objects* cannot be mutated or deleted and may be used in transactions in either path concurrently by all users.
- *Owned objects* have an owner field that determines access control. The owner is an address representing a public key. A transaction may access the object if it is signed by that key (which can also be a multi-signature). A canonical example is a user's cryptocurrency account. As owned objects are never under contention when the owner is honest, Sui validates transactions accessing only owned or read-only objects using the fast path.
- *Shared objects* do not specify an owner. They can instead be included in transactions by anyone and do not require any authorization. Instead, their authorization logic is enforced by a smart contract. In Sui, such objects are only accessed through consensus to serialize their access.

In Stingray, we introduce two additional types of objects, bounded counters (Sec. 4) and collective objects (Sec. 5.2). Transactions using them are validated using the fast path.

Transactions. A transaction is a signed command that specifies several input objects, a version number per object, and a set of parameters. For owned objects, executing the transaction consumes the input object versions and constructs a set of output objects—which may be the input objects at a later version or new objects. Shared objects do not require a specified version. Instead, the system assigns the version on which the transaction executes based on the consensus sequence. Input objects' versions must be the latest version in validators' databases and must not be re-used across transactions. This limits concurrency because validators must process transactions with the same object sequentially, and we address this using bounded counter objects (Sec. 4).

In Sui, a transaction is signed by a single address and therefore can use one or more objects owned by that address. A single transaction cannot use objects owned by more than one address and must use shared objects instead. In this work, we will allow a transaction to use objects owned by different addresses if the transaction is signed by the all the owners (Sec. 5.1), thus enabling validation of such transactions on the fast path.

Certificates. A *certificate* (Cert) on a transaction contains the transaction and signatures from a quorum of at least $2f + 1$ validators with their identifiers. A certificate may not be unique, and the same logical certificate may be signed by different quorums of validators. However, two different valid certificates on the same transaction are treated as representing semantically the same certificate.

2.2 Processing in the Fast Path and Consensus

Fig. 1 provides an overview of Sui and, by extension, Stingray's common case. A transaction is sent by a user to all validators (❶), who ensure it is correctly authenticated by the owners of all owned objects and that all objects exist (❷). A correct validator rejects any conflicting transaction using the same owned object versions (the

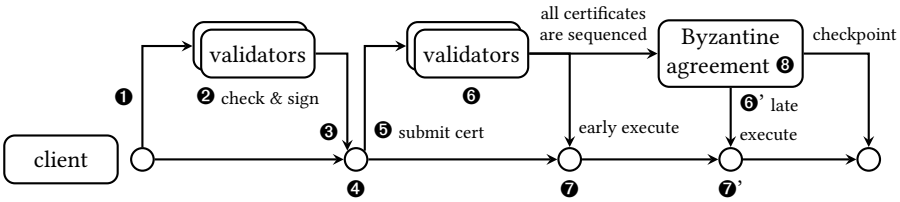

Figure 1: General protocol flow of Sui Lutris [13] fast-path (❶-❼) & consensus failover system (❽,❻',❼').

first transaction using an object acquires a *lock* on it). Validators then countersign the transaction (❸) and return the signatures to the user. A quorum of signatures constitutes a *certificate* for the transaction (❹). Anyone may submit the certificate to the validators (❺) that check it.

At this point, execution may take the fast path: If the certificate only references read-only and owned objects (in Stingray, also owned bounded counter and collective objects) it is executed immediately (❻) and a signature on the effects of the execution is returned to the user. Signatures from $2f + 1$ validators create an *effects certificate* (❼), and the transaction is *finalized*, i.e., it is guaranteed to never be rolled back, even if the set of validators change. If any shared objects are included, execution must wait for them to be assigned versions post-consensus. In all cases, certificates are input into consensus and sequenced (❽). Once sequenced, the system assigns a common version number to shared objects for each certificate, and execution can resume (steps ❻' and ❼') to finalize the transaction. The common sequence of certificates is also used to construct checkpoints, which are guaranteed to include all finalized transactions (❽).

Checkpoints and Reconfiguration. Sui ensures transaction finality before consensus for owned object transactions (❼) or after consensus for shared object transactions (❼'). A reconfiguration protocol ensures that any transaction finalized through the fast path will eventually be included in a checkpoint before the end of the epoch (epochs last roughly for 24 hours).

Limitations of Sui. Sui users are not allowed to submit conflicting transactions that reuse the same owned object versions in steps (❶) and (❷), limiting concurrency. If a misconfigured user client behaves this way, neither transaction may successfully construct a certificate (❹), and the owned object becomes locked until the end of the epoch, harming user experience. To avoid mistrusting users from locking each other's objects for a day, Sui restricts transactions to only contain objects from a single owner, thus limiting the applicability of the fast path. For similar reasons, objects used in the fast path may have at most one owner.

Stingray addresses all these limitations through changes to the fast path (steps ❶-❼). Stingray increases the concurrency in Sui by allowing concurrent transactions on bounded counters. This can be trivially extended to bounded sets and any commutative objects (e.g., add-only counters or PN-sets [41]). We also allow transactions with multi-owner authentication and collective objects that are not expected to have contention to execute in the fast path. Since the risk of contention in such transactions is low but not zero, we finally show how to recover from contention using FastUnlock.

3 System Overview

We introduce Stingray and the setting in which it operates.

3.1 Threat Model and Goals

The adversary is computationally bounded, ensuring that standard cryptographic properties such as the security of hash functions, digital signatures, and other primitives hold. Under this assumption Stingray ensures *validity*, meaning that all transactions that are executed have valid authorization (Def. 3.1).

We consider a message-passing system with $n = 3f + 1$ validators running the Stingray protocol. An adversary can adaptively corrupt up to f validators, referred to as *Byzantine*, who may deviate arbitrarily from the protocol. The remaining validators, called *honest*, follow the protocol. The communication network is asynchronous and messages can be delayed arbitrarily. Given these conditions, Stingray is *safe*, that is, the union (merge) of all transactions executed by honest validators does not lead to invalid state transitions (Def. 3.3). This safety property subsumes the classic safety of blockchains since any fork in the state, if merged, would violate validity predicates such as conservation of value (e.g., in case of double-spends). At the same time, this definition captures concurrent execution.

Finally, assuming messages among honest validators are eventually delivered, Stingray is *live*, meaning honest validators eventually execute certifiably valid user transactions and update their state accordingly, and in the absence of new transactions, eventually converge to the same state (Def. 3.4). Here, liveness is required for transactions that can be executed without causing invalid state transitions, as certified by a quorum of validators (Def. 3.2). In contrast, requiring liveness for all transactions issued by honest users is too strong and unachievable, since the validity of their state transitions depends on the system's current state.

Definition 3.1 (Transaction Validity). A transaction Tx is valid if all its input objects are owned by the transaction's signers.

Definition 3.2 (Certificate Validity). A transaction Tx has a valid certificate if:

- The transaction is valid
- It has a quorum certificate signed by at least $2f+1$ validators

Let $T_p(t)$ denote the set of transactions executed by validator p up to time t. The applications specify a set of predicates $\mathcal{P}$ over sequences of transactions. For example, a predicate may require that no user can spend more than their account balance or that no two transactions perform conflicting state updates.

Definition 3.3 (Safety Properties). For any execution of Stingray with at most f Byzantine validators:

- **Validity:** For all p, t: all Tx $\in T_p(t)$ have valid certifcates.
- **Global safety:** For all t, for all subsets H of honest validators, there exists a sequence T that contains $\bigcup_{p \in H} T_p(t)$ such that for all $P \in \mathcal{P}$: $P(T)$ is true.

Definition 3.4 (Liveness Properties). For any execution of Stingray with at most f Byzantine validators:

- **Progress:** Every transaction with a valid certificate is eventually executed by all honest validators, unless an owner of its input objects equivocates, i.e., signs two transactions with the same object version as input.
- **Eventual consistency:** For all honest validators p_1, p_2 and all time t, there exists a time $t' \geq t$ such that $T_{p_1}(t') \supseteq T_{p_2}(t)$.

Alongside the above properties, our goal is to enable concurrent execution of as many transactions as possible. Stingray uses separate paths for processing commutative/bounded-counter transactions and for other fast-path transactions. We prove the security of the former in Sec. 4.4 and the latter in App. C.

3.2 Motivating Applications of Stingray

We present three example applications of Stingray: one utilizing the bounded counter and two employing multi-owner transactions (whose potential liveness risks are mitigated by FastUnlock).

Concurrent Payments. The first application, concurrent debit or credit of an account, is one of the most common in existing blockchain systems. Not only is this useful for payments, but it is also useful for gas debits, which are required for every transaction. However, it presents significant challenges for parallel execution if not carefully designed. Incoming transactions (credits) only increase the account's balance, so they are commutative. Since they do not cause contention, they can easily be processed concurrently (even though prior blockchains do not do so). Concurrent debits, on the other hand, are more complex due to the need for zero-balance checks. These operations are non-commutative, hence they cause true contention that hinders concurrency.

Concurrent transactions may be achieved in UTXO-style blockchains by dividing one's account into smaller UTXOs. However, keeping track of these smaller UTXOs is cumbersome, and using the same UTXO twice is actually a double-spending attack. In account- or object-based blockchains, one can similarly split their account into smaller accounts and concurrently access them, but it has the same challenges as with UTXOs. For example, users on Sui frequently make mistakes, causing contention, which results in losing access for an entire day. Stingray resolves this issue without introducing such complications by using a mostly-commutative bounded counter, allowing half of the budget to be spent concurrently before requiring a sequential rebalancing transaction.

Atomic swaps. Atomic swaps enable two parties to exchange digital assets without relying on a trusted intermediary. While consensus-based blockchains achieve this through smart contracts, consensus-less environments face the risk of deadlock due to Byzantine users issuing concurrent transactions. Such scenarios can effectively lock both parties' assets. Thus, in Sui, swaps require multiple transactions, with at least one (the swap) relying on consensus.

However, the risk of liveness loss on the fast path only occurs when an active attacker deliberately creates contention. Stingray provides an effective mitigation of this risk with the FastUnlock protocol. This safety net allows Stingray to support multi-owner transactions in the fast path, allowing fast path atomic swaps and other multi-party smart contracts, enhancing the programmability of consensus-less transactions. Although not a real application, the same risk runs for users who inadvertently equivocate on their objects, leading to unexpected deadlocks and a poor user experience. Here too, FastUnlock helps restore liveness quickly, lowering barriers to securely using an ultra-low latency blockchain.

Regulated stablecoins. Regulated stablecoins [20] require the issuer to be able to block an account for regulatory reasons, besides its owner spending from it. This has eluded consensus-less systems since sequencing these potentially conflicting operations requires consensus. Yet, the ability to block objects is nearly never exercised, creating no practical contention. Stingray's collective objects, that may be used by more than one owner (or complex access control) enables such transactions in the fast path.

3.3 Challenges

Stingray defines new object types that allow for higher concurrency. We do not focus on purely commutative data structures for which a Byzantine fault-tolerant CRDT [26] is sufficient but still cannot support transactions for blockchains because of gas payments that require a comparison with zero, a non-commutative operation.

To resolve this, we define the *bounded counter*, which runs in the fast path. The design of this bounded counter creates our first challenge (**Challenge 1**): implementing nearly commutative objects in BFT settings. To achieve this, we depart from prior work in CFT [1] that splits the budget among replicas since in the BFT setting, a single malicious replica could sign infinite transactions. As a result, we need to rely on quorums to distribute the budget collectively. Since there is an exponential number of potential quorums we reduce this to giving a sufficient budget to each validator to spend concurrently but not enough that an overspend can happen if all quorums have a minority of equivocating participants that spend infinite amounts. This tension results in our construction spending half of the bound when not under attack. We then show how we can reset the budget through a consistent read, allowing for the full amount to be concurrently spent with only $\log n$ points of synchronization.

The second challenge we take on is that existing consensus-less blockchains require transactions to operate on state owned by a single user, due to the fear of concurrency. Stingray absolves concurrency for consensus-less transactions and instead identifies contention as the true culprit of correctness violations. Thus, Stingray enhances programmability by defining new types of objects such as collective objects that are owned by multiple users, as well as new types of transactions, such as multi-owner transactions (e.g., an asset swap) that are processed in the fast path. This approach is powerful, and promises reduced latency for such operations that currently require consensus. But this creates our second challenge (**Challenge 2**): users may naturally submit conflicting transactions because contention is now possible. For example, two users may perform a swap and end up locking the objects due to bad timing. To address this challenge, Stingray uses a novel unlocking mechanism, called FastUnlock, that allows users to resolve conflicts quickly,

enabling users to submit new transactions within the latency of a consensus protocol.

4 Concurrency through Commutativiy

We present protocols that allow users to finalize a common category of transactions: updates to a *bounded counter*, without using consensus. This type of object is strictly harder to implement than CRDTs as it has a non-monotonic comparison with zero [32]. Our bounded counter protocol is a modification to the fast-path protocol shown in Fig. 1, while transactions on other owned objects continue to be processed as in Fig. 1.

4.1 The Bounded Counter Object

A bounded counter is an object that has a balance $\mathsf{Bal} \in \mathbb{R}$ as its state and supports transactions with a parameter $\delta \in \mathbb{R}$, allowing additions ($\delta > 0$) and subtractions ($\delta < 0$) on its balance, while maintaining the invariant $\mathsf{Bal} \geq 0$. Our goal is to execute transactions concurrently to the extent possible.

Definition 4.1 (Bounded counter). A bounded counter object has an authorized user called the owner and an initial balance Bal_0, supports transactions Tx with value $\mathsf{Tx}.\delta \in \mathbb{R}$ and has the following properties:

(1) **Global safety**: as in Def. 3.3 with predicate $P(T) = (\mathsf{Bal}_0 + \sum_{\mathsf{Tx} \in T} \mathsf{Tx}.\delta \geq 0)$.
(2) **Progress**: If an honest owner sends a set of transactions T such that $\mathsf{Bal}_0 + \sum_{\mathsf{Tx} \in T} \mathsf{Tx}.\delta \geq 0$, then all validators will eventually execute all transactions in T.
(3) **Validity** and **eventual consistency**: as in Defs. 3.3 and 3.4.

The progress property is a strengthening of Def. 3.4 wherein we specify the conditions under which transactions issued by honest owners get certified and eventually executed. We show a protocol in Sec. 4.3 and prove that it achieves the above properties in Sec. 4.4.

The bounded counter is useful for a common use case of blockchains, cryptocurrencies, where an account's balance can be represented using a bounded counter. The bounded counter can be generalized to a bounded set in which elements can be added or removed, as long as the size of the set does not exceed a predefined bound. This may be useful, for example, to mint a predefined number of limited-edition non-fungible tokens (NFTs). More generally, the bounded counter can be used as a loop counter. For simplicity, we describe the most common case of a real-valued counter bounded below by 0 and consider payments from an account as the canonical use case.

4.2 Key Ideas for the Bounded Counter

Consider a user (owner) who owns an account containing Bal_0 units of money. As an example, suppose that each transaction the user makes credits 1 unit from its account, i.e., $\mathsf{Tx}.\delta = -1$. In this case, safety requires that no honest validator execute more than Bal_0 transactions. Validators execute a transaction only if they see a certificate for the transaction, so it is sufficient to ensure that no more than Bal_0 transactions get certified.

Key Idea 1: Signing Budgets for Validators. Recall that Sui's fast path prevents the certification of two transactions on the same version of an object by ensuring that validators sign at most one transaction per version (Sec. 2). Extending this approach to allow concurrent transactions, we must ensure that each validator signs only a few transactions concurrently. We assign to each validator a *budget* $\mathsf{Bud} = \eta\mathsf{Bal}_0$, where $\eta = \frac{f+1}{2f+1}$, which is the maximum number of transactions the validator can sign. Since each certified transaction is signed by at least $2f+1$ validators, $f+1$ of whom are honest, for each certified transaction, at least $f+1$ is deducted from the total budget of all honest validators. Since the total budget of all honest validators starts at $(2f+1)\mathsf{Bud}$, the number of certified transactions can be at most $\frac{(2f+1)\mathsf{Bud}}{f+1} = \mathsf{Bal}_0$, even if the user and up to f validators are malicious.

Thus, the signing budgets ensure global safety. However, this idea alone does not satisfy liveness. If the f Byzantine validators abstain from signing transactions, each certified transaction requires signatures from $2f+1$ (all) *honest* validators, causing every honest validator to decrease their budget. So, at most $\mathsf{Bud} = \eta\mathsf{Bal}_0$ transactions will get certified, while liveness requires all of them to be certified eventually.

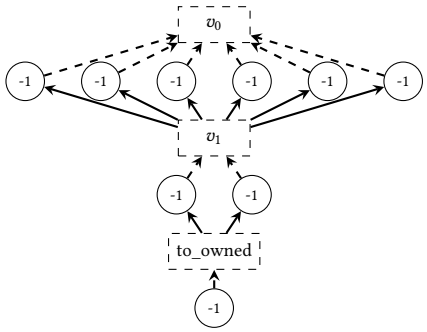

Figure 2: Version updates in the bounded counter. Circles represent unit decrement transactions and dashed boxes versions. The initial balance is $\mathsf{Bal}_0 = 9$, and $f = 1$. For version v_0, each validator has a budget of $\frac{f+1}{2f+1}\mathsf{Bal}_0 = 6$. After 6 transactions are certified, the user sends a version update (dashed box v_1) containing pointers to the 6 certified transactions. The remaining balance is $9 - 6 = 3$ and the validators update their budget to $\frac{f+1}{2f+1} * 3 = 2$. Finally, when the remaining balance is 1, the user converts the bounded counter to a standard owned object and spends the remaining balance.

Key Idea 2: Version Updates. When the user realizes that validators may have exhausted their budget (because Bud transactions have already been certified), the user sends a *version update request* which includes pointers to all the previously certified transactions. Upon seeing a valid version update request, each validator updates their budget to η fraction of the balance remaining after executing the certified transactions. In the above example, this increases each validator's budget from 0 to $\eta(1-\eta)\mathsf{Bal}_0$. The validator also updates its local version to stop signing transactions with the previous version and start signing transactions with the new version (see Fig. 2 for an example).

Note that within a version, transactions get certified concurrently, while transactions across versions are certified sequentially. At each version, the user can spend up to η fraction of the remaining balance, until finally, when the remaining balance is small enough,

the user can convert the bounded counter to a standard owned object and spend the remaining amount in one last transaction (shown in Fig. 2). Thus, with an initial balance of Bal_0 and each transaction spending 1 unit, the user requires only $\log(\mathsf{Bal}_0)$ version update requests, and therefore $O(\log(\mathsf{Bal}_0))$ latency to spend the entire balance. In contrast, this would require $O(\mathsf{Bal}_0)$ latency on Sui's fast path or any other consensus protocol.

4.3 Bounded Counter for Single Owner

Alg. 1 shows the algorithm run by validators for the bounded counter. For simplicity, we specify the algorithm for when all transactions have only one bounded counter object as input, although transactions with multiple bounded counters and other owned objects as inputs can be processed in the fast path.

Processing Transactions. For each bounded counter object, the state maintained by each validator includes the current version Version, the current budget Bud, and the set of transactions SignedTxs the validator has signed (ll. 4 to 6). Upon receiving a new valid transaction, the validator checks that the transaction has the same version as the validator's current version (l. 12). If the transaction's value $\mathsf{Tx}.\delta$ is within budget (l. 13), the validator deducts its budget if required, marks the transaction as signed, and sends the signature to the user. Addition or debit transactions ($\mathsf{Tx}.\delta > 0$) will always be signed, without decreasing the budget, while subtraction or credit transactions ($\mathsf{Tx}.\delta < 0$) will be signed only up to the budget.

Finalizing and Executing Transactions. Upon receiving a valid certificate Cert, the validator broadcasts the certificate to other validators, executes the transaction to update the counter's state locally, and sends an *effects signature* to the user. Unlike standard owned objects where the effects signature contains the new state of the object, for a bounded counter object, the effect signature simply marks the transaction as executed because the validator does not know the counter's correct state yet due to its partial view of other certified transactions. Once the user receives $2f+1$ effects certificates, it considers the transaction finalized.

Version Update Requests. When the validator receives a version update request, it processes the request only if all the transactions to which the request points (v.PrevTxs) are certified (l. 28). Moreover, all the transactions in v.PrevTxs must have the same version as the validator's current version (l. 30). If so, the validator updates its budget. For each certified transaction in v.PrevTxs, the validator increases for additions or decreases for subtractions its budget by η times the transaction's value (l. 33). The validator also reclaims its spent budget for transactions it signed and are now included in v.PrevTxs (l. 35), ensuring that budget is not deducted twice. After the update, the validator's budget is η times the remaining balance of the bounded counter after executing all certified transactions minus the values of the transactions that the validator signed but were not included in v.PrevTxs. Finally, the validator updates its current version to v (l. 37).[1]

Honest Users. The bounded counter protocol allows a user to get transactions certified concurrently under low contention, i.e., when the number of concurrent transactions is low. In Alg. 5 (in the appendix), we specify how an honest user interacts with the bounded counter. In summary, the honest user ensures the following:

(1) The user sends transactions with the same Version until it sends a version update request (Alg. 5 l. 11)
(2) For any version, the user does not send transactions exceeding the validators' budget for that version (Alg. 5 l. 10).
(3) Upon exhausting the budget for a version (Alg. 5 l. 8), the user sends a version update request containing pointers to all transactions it sent for that version (Alg. 5 l. 16).
(4) When the validators' budget falls to 0 for a certain version (perhaps due to rounding), the user sends a transaction requesting to convert the bounded counter object to a standard owned object so that the small amount of remaining balance may be spent through a classic owned object transaction.

An honest user can satisfy these requirements by keeping track of the current version and the validator's budget, all of which can be computed based on the transactions it has sent (see Alg. 5), without any communication. In App. A, we show that liveness holds for users that behave as specified by Alg. 5.

Algorithm 1 Bounded counter for single owner (validator logic)

```
1:  ▷ Initialize a bounded counter object
2:  ▷ Initial version is v_0 and budget is computed from initial balance
3:  procedure InitBC(Bal_0)
4:      Version ← v_0                          ▷ current version for which the validator signs txs
5:      Bud ← η * Bal_0                        ▷ remaining budget for signing txs for Version
6:      SignedTxs ← ∅                          ▷ set of txs signed so far

7:  ▷ Executed upon receiving a transaction
8:  procedure ProcessTx(Tx)
9:      ▷ Ensure same transaction is not proceesed twice
10:     if Tx ∈ SignedTxs then return sign(Tx)
11:     require valid(Tx)                      ▷ check Tx has valid signatures
12:     require Tx.Version = Version
13:     require Bud + Tx.δ ≥ 0
14:     ▷ Decrease the budget if decrement transaction
15:     if Tx.δ < 0 then Bud ← Bud + Tx.δ
16:     SignedTxs ← SignedTxs ∪ {Tx}
17:     return sign(Tx)

18: ▷ Upon receiving valid certificate signed by 2f + 1 validators; ensure same certificate
    not processed twice
19: procedure ProcessCert(Cert)
20:     broadcast(Cert)
21:     wait until areExecuted(Cert.Tx.Version.PrevTxs)
22:     if valid(Cert) then exec(Cert.Tx)      ▷ execute tx, persist object's state
23:     return sign(Cert)                      ▷ send signature to finalize Cert.Tx to user

24: ▷ Upon receiving version update request signed by the user
25: ▷ Ensure same request is not processed twice
26: procedure ProcessVersionUpdateReq(v)
27:     broadcast(v)
28:     require v.PrevVersion = Version and areCertified(v.PrevTxs)
29:     ▷ Check all txs in v.PrevTxs have version PrevVersion
30:     require allVersionsMatch(v.PrevTxs, v.PrevVersion)
31:     ▷ Update budget based on txs in v.PrevTxs
32:     for Tx in v.PrevTxs do
33:         Bud ← Bud + η * Tx.δ
34:         ▷ Regain spent budget for txs included in v.PrevTxs
35:         if Tx ∈ SignedTxs and Tx.δ < 0 then Bud ← Bud − Tx.δ
36:     ▷ Start signing txs for the updated version
37:     Version ← v
```

4.4 Security Proof

Theorem 4.2. *Alg. 1 satisfies validity.*

[1] In reality, the version identifier stored by the validator and included in each transaction can be a hash of the version update request. Collision resistance will bind each version identifier to a unique version update request.

Proof. Validators execute only valid transactions with a valid certificate (Alg. 1 l. 22). □

Theorem 4.3. *Alg. 1 satisfies eventual consistency.*

Proof. If one honest validator executes a transaction (Alg. 1 l. 22), the validator also broadcasts the certificate for the transaction (l. 20). Eventually, all validators also receive the version update request Cert.Tx.Version (along with their certified transactions) corresponding to the certified transaction because at least one honest validator (who signed the certified transaction) received the request and broadcast it (l. 27). Thus, eventually, all honest validators will execute the transaction. □

As a warmup for proving global safety, we first show how Key Idea 1 ensures that within a single version, any subset of the certified transactions does not spend too much balance. Refer to Tab. 2 (in the appendix) for a summary of the notation used in the proof.

Definition 4.4. The value function is defined on a set of transactions $\mathcal{S}$ as $\Delta(\mathcal{S}) = \sum_{\mathsf{Tx}\in\mathcal{S}} \mathsf{Tx}.\delta$.

Lemma 4.5. *For any version v, let $\overline{Bud}_v$ be the average budget of all honest validators at the time when they set Version to v.*[2] *Let C_v be the set of certified transactions with version v. Then, for all $T \subseteq C_v$: $\Delta(T) \geq -\frac{1}{\eta}\overline{Bud}_v$.*

Proof. Let $C_v^- \subseteq C_v$ be the set of decrement transactions (Tx such that $\mathsf{Tx}.\delta < 0$). It is sufficient to show that $\Delta(C_v^-) \geq -\frac{1}{\eta}\overline{\mathsf{Bud}}_v$.

Suppose that $f_\mathrm{r} \leq f$ validators are adversarial (so, $n - f_\mathrm{r} = 3f + 1 - f_\mathrm{r}$ are honest). Each certified transaction has $2f + 1$ signatures, of which at most f_r are from adversarial validators. All other signatories are honest, thus if $\mathsf{Tx} \in C_v^-$ is certified, at least $(2f + 1 - f_\mathrm{r})|\mathsf{Tx}.\delta|$ is deducted from the total budget of all honest validators (note that $\mathsf{Tx}.\delta < 0$). Suppose, for contradiction, that a set of decrement transactions C_v^- are certified such that $\Delta(C_v^-) < -\frac{1}{\eta}\overline{\mathsf{Bud}}_v$. Then, the average budget of all honest nodes is $\overline{\mathsf{Bud}} \leq \overline{\mathsf{Bud}}_v + \frac{2f+1-f_\mathrm{r}}{3f+1-f_\mathrm{r}}\Delta(C_v^-) \leq \overline{\mathsf{Bud}}_v + \frac{f+1}{2f+1}\Delta(C_v^-) < 0$. This is a contradiction because no honest validator signs a transaction that would cause its budget to fall below 0 (Alg. 1 l. 13), so the average budget of honest validators cannot fall below 0. □

Next, we will prove that the version updates introduced as Key Idea 2 preserve global safety. To do this, we first prove that the versions corresponding to certified transactions form a chain in which each version is the PrevVersion of the next (Lem. 4.7). This follows from a quorum-intersection argument and the fact that the versions of transactions signed by any single honest validator form a chain. This property allows us to define a linearly growing history of the bounded counter containing certified transactions from each version update request's PrevTxs (Def. 4.8). Using this, we show that the average budget of honest validators at the time they update their local Version to v is η times the balance after executing transactions in the counter's history, minus any budget deducted for certified transactions that were not included in the history (Lem. 4.9). Combining this with Lem. 4.5, we prove that the certified transactions across all versions do not spend more than the initial balance (Thm. 4.10). Since validators only execute certified transactions, this ensures global safety.

Definition 4.6. For any version $v \neq v_0$, define the parent version $p(v)$ as v.PrevVersion.

Lemma 4.7. *Let $\mathcal{V}$ be the set of versions for which there exists at least one certified transaction. If $\mathcal{V} \neq \emptyset$, then $\mathcal{V} = \{v_0, ..., v_{|\mathcal{V}|-1}\}$ such that for all $i = 1, ..., |\mathcal{V}| - 1$, $p(v_i) = v_{i-1}$.*

Proof. If no transactions have been certified, $\mathcal{V} = \emptyset$. If at least one transaction is certified, then $v_0 \in \mathcal{V}$. This is because initially, honest validators sign only transactions with version v_0 (Alg. 1 ll. 4 and 12) and will not sign transactions for a different version until they receive a version update request containing certified transactions (l. 28). Since a certificate requires at least one honest validator's signature, at least one certified transaction must have version v_0.

For any honest validator j, if it updates Version from v to v', it must be such that $p(v') = v$ (Alg. 1 l. 28). Therefore, for all $v \in \mathcal{V}$, there exists a sequence $v_0, ..., v$ in which each version is the parent of the next version. In other words, the versions in $\mathcal{V}$ form a tree rooted at v_0 with parent links as edges.

Now all that remains to show is that this tree is, in fact, a chain. That is, there is no $v, v' \in \mathcal{V}$ such that $v \neq v'$ and $p(v) = p(v')$. This follows from quorum intersection. If there was $v, v' \in \mathcal{V}$ such that $v \neq v'$ and $p(v) = p(v')$, then for both versions v and v', there is a set of $2f + 1$ validators that signed transactions with that version. These two sets of $2f+1$ validators have at least $2(2f+1) - n = f+1$ validators in common (since $n = 3f + 1$). However, since at most f validators are adversarial, at least one honest validator signed both transactions with version v and v'. However, this is a contradiction because once the honest validator signs a transaction for version v, it will never sign a transaction for version v' since there is no path $v, ..., v'$ in which each is a parent of the next one. □

Definition 4.8. Define the history H_v of a version as $H_{v_0} = \emptyset$ and $H_{v\neq v_0} = H_{p(v)} \cup v$.PrevTxs.

Let $C^i = C_{v_1} \cup ... \cup C_{v_i}$ be the set of certified transactions with versions up to v_i. Let $\widetilde{H}_{v_i} = C^i \setminus H_{v_i}$ be the set of certified transactions not included in the history. For any set of transactions $\mathcal{S}$, $\mathcal{S}^- \subseteq \mathcal{S}$ contains transactions Tx such that $\mathsf{Tx}.\delta < 0$, and $\mathcal{S}^+ = \mathcal{S} \setminus \mathcal{S}^-$.

Lemma 4.9. *For any version $v_i \in \mathcal{V}$, the average budget of all honest validators at the time they upgrade to version v_i satisfies $\overline{Bud}_{v_i} \leq \eta(Bal_0 + \Delta(H_{v_i}) + \Delta(\widetilde{H}_{v_i}^-))$.*

Proof. Throughout the execution, an honest validator i) starts with an initial budget of $\eta\mathsf{Bal}_0$, then ii) decreases its budget for every decrement transaction signed (Alg. 1 l. 15), iii) updates its budget for every certified transaction included in a version update request (l. 33), and iv) reclaims its budget for every certified decrement transaction it had previously signed that is included in a version update request (l. 35). Suppose that $f_\mathrm{r} \leq f$ validators are adversarial (so, $n - f_\mathrm{r} = 3f + 1 - f_\mathrm{r}$ are honest). Combining these four components, the average budget of all honest validators at the time they update

[2] Count a validator's budget as 0 if it never sets Version to v.

to version v_i is

$$\begin{aligned}\overline{\mathsf{Bud}}_{v_i} &\le \eta\mathsf{Bal}_0 + \frac{2f+1-f_r}{3f+1-f_r}\Delta(C^{i^-}) + \eta\Delta(H_{v_i}) - \frac{2f+1-f_r}{3f+1-f_r}\Delta(H^-_{v_i}) \\ &= \eta(\mathsf{Bal}_0 + \Delta(H_{v_i})) + \frac{2f+1-f_r}{3f+1-f_r}\Delta(\widetilde{H}^-_{v_i}) \\ &\le \eta(\mathsf{Bal}_0 + \Delta(H_{v_i}) + \Delta(\widetilde{H}^-_{v_i})).\end{aligned}$$

□

Theorem 4.10. *The bounded counter protocol (validator code: Alg. 1) satisfies global safety.*

Proof. For any given subset of honest validators, let T be the set of transactions executed by some validator in this subset. Let v_k be the latest version in T. Since validators only execute certified transactions (Alg. 1 l. 22), $T \subseteq C^k$. Moreover, validators execute transactions in v_k.PrevTxs before executing transactions with version v_k (Alg. 1 l. 21), so $T \supseteq H_{v_k}$.

Recall that we partitioned $C^{k-1} = H_{v_k} \cup \widetilde{H}^-_{v_k} \cup \widetilde{H}^+_{v_k}$, that is, certified transactions with versions up to v_{k-1} may be in the history of version k, and those that are not may be either increments or decrements. Given these constraints, it is sufficient to prove that $\mathsf{Bal}_0 + \Delta(T) \ge 0$ for the worst case $T = H_{v_k} \cup \widetilde{H}^-_{v_k} \cup C^-_{v_k}$ where all decrement transactions and no increment transactions beyond H_{v_k} are executed.

$$\Delta(T) = \Delta(H_{v_k}) + \Delta(\widetilde{H}^-_{v_k}) + \Delta(C^-_{v_k}) \tag{1}$$

$$\ge \Delta(H_{v_k}) + \Delta(\widetilde{H}^-_{v_k}) - \frac{1}{\eta}\overline{\mathsf{Bud}}_{v_k} \quad \text{(Lem. 4.5)} \tag{2}$$

$$\ge -\mathsf{Bal}_0 \quad \text{(Lem. 4.9)} \tag{3}$$

□

5 Concurrency with Multiple Owners

So far, we have seen designs to improve concurrency for transactions with owned objects. However, these techniques and prior consensus-less systems require that all owned objects accessed in a transaction have the same owner. Stingray enhances object programmability on the fast path using two ingredients: (i) multi-owner transactions, and (ii) collective objects.

5.1 Multi-Owner Transactions

Sui requires all owned objects in a transaction to be 'owned' by the same address [13]. Stingray lifts this restriction: a transaction can reference owned objects from multiple owners. Validators must still ensure that each owned object referenced by a transaction is correctly authorized before signing a transaction. For example, consider an atomic swap transaction that takes object A owned by Alice and object B owned by Bob and exchanges their ownership. For the transaction to be authorized we need two signatures over the full transaction (or a hash), one from an authorized signer of A (i.e., Alice) and one from an authorized signer of B (i.e., Bob).

However, enabling multi-owner transactions makes Stingray more susceptible to owned objects being locked through error or malicious behavior. For example, consider the previous scenario. If Alice signs Tx first, Bob may refuse to sign, denying Alice access to her object. If Alice loses patience and tries to use A in another transaction Tx′, then Bob may sign Tx and race Alice's attempt to build a certificate. Now both Tx and Tx′ contain A and conflict which can lead to A (and B) being locked. Sui unlocks such objects after one day, at epoch change. But multi-owner transactions in Stingray make such conflicts more likely, so the latency of one day is unacceptable. Thus, we develop the FastUnlock protocol described in Sec. 6.1 that provides a resolution in seconds.

5.2 Collective Objects

The second interesting class of objects that Stingray uses are Collective Objects. These objects can be accessed by multiple users concurrently, that is, a transaction on such an object can be authorized by any party (or a subset of parties) from a given set (the set may be infinite). Yet, they are processed on the fast path.

Applications for this include an NFT sale where users can add themselves to a collective set of users that will receive the NFT or an auction where users can add their bids as long as the auction is still running. If there is no limit on the size of the set or the number of bids, then we can simply use an add-only-set data structure to process these transactions concurrently without any contention. However, if there is a limit, we must use a *collective bounded counter*.

5.3 Collective Bounded Counter

Following the owned bounded counter (Sec. 4), our collective bounded counter allows multiple owners to send transactions and version updates. This works well as long as the concurrent transactions sent by all the owners together never exceed validators' budgets; then they all get certified, irrespective of the order in which they arrive. However, the bounded counter may get locked and lose liveness under any of these conditions:

- The owners attempt to spend more than the validators' budget in any version: In this case, honest validators may exhaust their budget by signing different transactions so that no transaction gets certified.
- The owners send two conflicting version update requests, i.e., neither request transitively includes transactions from the other version. In this case, honest validators may be split across the two versions, with neither group of validators willing to switch to the other version, causing neither version's transactions to get certified.

These circumstances would not occur if a single owner sending transactions follows the protocol specifications (Alg. 5). However, a misconfigured owner or multiple owners who do not coordinate may cause such scenarios.

In this section, we show how owners can unlock the bounded counter object, without consensus under optimistic conditions, by issuing a *version merge request* (Alg. 2). When the owners send a *version merge request* that contains a set of versions PrevVersions to merge, a validator processes the request if its current local Version is one of PrevVersions (Alg. 2 l. 5). This allows validators locked on any of the conflicting versions to adopt the merged version while also ensuring that each validator processes versions in a linear order. Just as in version update requests, the validator updates its budget by η times the value of every pending transaction in the history of the merged versions (Alg. 2 l. 8). In this case, the history contains all transactions included in the version update requests for PrevVersions recursively, and pending are the ones for which

Algorithm 2 Collective bounded counter (validator logic)

```
1:  ▷ InitBC, ProcessTx, ProcessCert, ProcessVersionUpdateReq same as Alg. 1

2:  ▷ Upon receiving version merge request signed by the user
3:  procedure ProcessVersionMergeReq(v)
4:      ▷ Check current version is one of the versions being merged
5:      require Version ∈ v.PrevVersions
6:      ▷ Update budget based on all txs in the history of v.PrevVersions, except those
        that have already been considered
7:      for Tx in pendingTxsInHistory(v) do
8:          Bud ← Bud + η * Tx.δ
9:          ▷ Regain spent budget for txs included in v.PrevTxs
10:         if Tx ∈ SignedTxs and Tx.δ < 0 then Bud ← Bud − Tx.δ
11:     ▷ Start signing txs for the updated version
12:     Version ← v
```

the budget has not been updated previously. As in version update requests, the validator reclaims its budget for transactions in the history that it had signed (Alg. 2 l. 10), and finally sets the new version to be v (l. 12).

If the owners accidentally sent two conflicting version update requests, upon sending a version merge request, validators locked on either version will switch to signing transactions for the merged version, restoring liveness. Finally, if the owners sent transactions spending more than the budget, the owners can send a version merge request containing only one previous version. This will cause validators to update their version without changing their budget (no new certified transactions included). The owners can then reissue transactions, ensuring this time not to send transactions spending more than the budget. Thus, in optimistic cases, where a single owner was misconfigured or crashed, or the bounded counter had temporary high contention, the bounded counter can be unlocked without requiring consensus among the validators. In cases of continuously high contention, owners must use FastUnlock (Sec. 6) which uses consensus to unlock their bounded counter. In App. B, we prove the safety of the collective bounded counter.

6 Fast Unlock Protocol

As discussed in Sec. 5, concurrency for multi-owner transactions and collective objects does not come free. They increase the chance that transactions diverge the view of the validators, leading to a loss of liveness. This is already a problem in Sui [30] and Mysticeti [5] for clients that run buggy software and may issue conflicting transactions on an owned object, i.e., transactions operating on the same version of an owned object (cf. Sec. 2.1). Their current solution is to wait until the end of the epoch, at which point they run an atomic snapshot sub-protocol as part of the epoch change and then drop all partial states. As a result, at the start of the new epoch, all validators have exactly the same state, and all objects can be safely accessed again. In Sui, epoch changes occur once per day. Since multi-owner transactions and collective objects make locks more likely, the latency of one day is unacceptable. To remedy this issue, Stingray introduces a FastUnlock functionality. On a high level, FastUnlock is a generalization of the merge functionality introduced in Sec. 5.3. The merge operation in Sec. 5.3 requires an honest owner to drive it to completion and only applies to commutative transactions. For arbitrary transactions, we require consensus to decide which among conflicting transactions to accept.

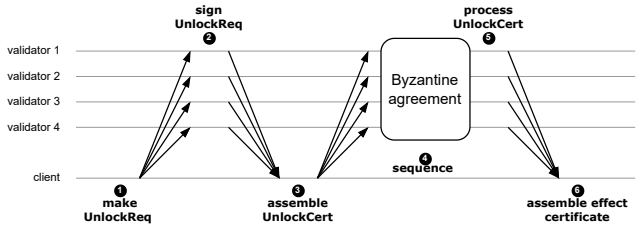

Figure 3: FastUnlock interactions between a user and validators to unlock an object.

Algorithm 3 Process unlock requests

```
1:  ▷ Handle UnlockRqt messages from users.
2:  procedure ProcessUnlockTx(UnlockRqt)
3:      ▷ Check (3.1): Check Auth. (Sec. 6.1)
4:      if !valid(UnlockRqt) then return error
5:      ▷ Step (3.2): Check for certificates.
6:      ObjectKey ← UnlockRqt.ObjectKey
7:      Cert ← LockDb[ObjectKey]                    ▷ can be None
8:      ▷ Step (3.3): Record the decision to unlock.
9:      UnlockVote ← sign(UnlockRqt, Cert)
10:     UnlockDb[ObjectKey] ← Unlocked
11:     return UnlockVote
```

6.1 Baseline FastUnlock Protocol

For simplicity, we show how the user can unlock a single object by executing a no-op or adopting one of the conflicting transactions. App. D extends the basic protocol to execute a new transaction instead of a no-op. Additionally, we describe at the end of this section how validators can detect two conflicting transactions on the system and *automatically* trigger an unlock.

New Persistent Data Structures. Each Stingray validator maintains a set of persistent tables abstracted as key-value maps, with the usual contains, get, and set operations. The table

$$\text{LockDb}[\text{ObjectKey}] \rightarrow \text{Cert or } \textbf{None}$$

maps ObjectKey = (ObjectId, Version), an object's identifier and version, to a certificate Cert, or **None** if the object's version exists but the validator does not hold a certificate for it. The map

$$\text{UnlockDb}[\text{ObjectKey}] \rightarrow \textbf{Unlocked}, \textbf{Confirmed}, \text{ or } \textbf{None}$$

records whether a transaction over the specified object version is involved in a current FastUnlock instance (**Unlocked**), has been sequenced by consensus (**Confirmed**), or none of the above (**None**).

All new owned object entries start with UnlockDb[ObjectKey] set to **None**. Once a transaction certificate is sequenced through consensus, it is always executed (whether it is for a shared object transaction or an owned-object-only transaction) and all owned object entries have UnlockDb[ObjectKey] set to **Confirmed**.

FastUnlock Protocol Description. To safely unlock an object, the user interactively constructs a proof, called a *no-commit certificate*, that no transaction modifying that object has been committed or will be committed on the fast path. This proof consists of a message signed by a quorum of validators attesting that they have not already executed a transaction on ObjectKey, and promising that they will not execute any transaction on ObjectKey in the fast path. After that, only certificates sequenced over consensus may affect such an ObjectKey.

Fig. 3 illustrates the FastUnlock protocol allowing a user to instruct validators to unlock a specific object. A user first creates an *unlock request* specifying the object they wish to unlock:

$$\mathsf{UnlockRqt}(\mathsf{ObjectKey}, \mathsf{Auth})$$

This message contains the object's key ObjectKey to unlock (accessible as UnlockRqt.ObjectKey) and an authenticator Auth ensuring the user is authorized to unlock ObjectKey. The authenticator is composed of two parts: (i) a transaction that mutates the object in question (and potentially additional objects) which is signed by the object owner, and (ii) a proof that the party requesting the unlock can modify the object in question. The authenticator prevents rogue unlock requests for objects that are either not under contention (the transaction shows there exists a transaction that uses the object) or by parties not authorized to act on the objects. The user broadcasts this UnlockRqt message to all validators (Fig. 3 ❶).

Each validator handles the UnlockRqt as follows (Alg. 3). A validator first checks (**Check (3.1)**) the validity of UnlockRqt by verifying the authenticator Auth. Specifically, Auth must contain a valid transaction including ObjectKey, and a signature on the transaction by the owner of ObjectKey. Otherwise, the validator stops processing. The validator attempts to retrieve a certificate Cert for a transaction on ObjectKey if it exists (**Step (3.2)**), or sets Cert to **None**. Then, the validator records that the object in UnlockRqt can only be included in transactions in the consensus path (l. 10) by setting its entry in UnlockDb[ObjectKey] to **Unlocked** (**Step (3.3)**). It finally returns a signed *unlock vote* UnlockVote to the user:

$$\mathsf{UnlockVote}(\mathsf{UnlockRqt}, \textbf{Option}(\mathsf{Cert}))$$

This message contains the authorized UnlockRqt and the certificate Cert for some transaction consuming ObjectKey that the validator executed (Fig. 3 ❷). If the validator has not executed any transaction on ObjectKey, then Cert = **None**.

$$\mathsf{UnlockCert}(\mathsf{UnlockRqt}, \textbf{Option}(\mathsf{Cert})).$$

There are two cases in the creation of UnlockCert:

(1) At least one UnlockVote carries a certificate. This scenario indicates that a correct validator has already executed a transaction, which implies that the object is not locked. However, this is not a proof of finality and subsequent steps may invalidate this execution.
(2) No UnlockVote carries a certificate. This scenario is a 'no-commit' proof as there are $f + 1$ honest validators that will not process certificates (UnlockDb holds **Unlocked**), thus no certificate will be executed in the fast path.

The user submits this UnlockCert for sequencing by the consensus engine (❸).

Algorithm 4 Process unlock certificates

```
1:  ▷ Handle UnlockCert message from consensus.
2:  procedure ProcessUnlockCert(UnlockCert)
3:      ▷ Check (4.1): Check no transaction already processed (Sec. 6.1).
4:      if UnlockDb[ObjectKey] = Confirmed then return
5:      ▷ Check (4.2): Check cert validity (Sec. 6.1).
6:      if !valid(UnlockCert) then return error
7:      ▷ Execute Cert or None (4.3).
8:      Cert ← UnlockCert.Cert
9:      if Cert ≠ None then Tx ← Cert.Tx
10:     else Tx ← No-Op
11:     EffectSign ← exec(Tx, UnlockCert)
12:     ▷ Prevent execution overwrite.
13:     UnlockDb[ObjectKey] ← Confirmed
14:     return EffectSign
```

All correct validators observe a consistent sequence of UnlockCert messages output by consensus (❹) and process them in order as follows (Alg. 4). A validator performs the following checks and if any fail, they ignore the certificate:

- **Check (4.1)** They ensure they did not already process another transaction to completion (i.e. UnlockDb is not **Confirmed**) or a different UnlockCert for the same ObjectKey.
- **Check (4.2)** They check UnlockCert is valid, that is, (i) it is correctly signed by a quorum of authorities, and (ii) the certificate Cert it contains is valid or **None**.

The validator then executes the transaction referenced by Cert (Step 4.3) if one exists. Otherwise, if Cert is **None**, the validator undoes any transaction locally executed on the object[3], then executes a no-op, that is, the object contents remain unchanged but its version number increases by one. The validator finally marks every object key as **Confirmed** to prevent future unlock certificates or checkpoint certificates from overwriting execution (l. 13) and returns an EffectSign to the user (❺). The user assembles a quorum of $2f + 1$ EffectSign messages into an *effect certificate* EffectCert that determines finality (❻).

App. D.1 details the use of gas objects in the context of FastUnlock and App. C proves the safety and liveness of the Stingray system using FastUnlock. The key insight is that an UnlockCert forces transactions on the owned object to go through consensus. There, either a transaction certificate or an unlock certificate will be sequenced first and executed. If a transaction is finalized, an unlock certificate will always cause the execution of that transaction.

Auto-Unlock. The basic FastUnlock scheme presumes that the request to unlock an object is authenticated by the owner(s) of the object. This ensures that only authorized parties can interfere with the completion of a transaction, but it also restricts who can initiate unlocking in case of loss of liveness. Alternatively, an 'AutoUnlock' can be issued by validators if the fast-path protocol is embedded in the consensus protocol, as proposed by Mysticeti [5]. In such a protocol, the presence of conflicting transactions in the causal history of a consensus block is evidence of loss of liveness. Upon seeing such evidence, validators can start locally processing a virtual unlock request posting the signed unlock requests as transactions in the consensus protocol and forming unlock certificates.

[3]The UnlockCert with Cert being **None** ensures such execution could not have been finalized; only a single layer of execution can ever be undone, and no cascading aborts can happen.

7 Implementation

We base our implementation of Stingray on Sui [30] as it is, to our knowledge, the only blockchain currently supporting consensus-less transactions. Specifically, we fork the research codebase of Mysticeti [31], which is a fork of the production codebase of Sui, but without irrelevant features such as Admission control, RPC endpoints, support for light clients, enforcement of correct genesis, etc. Our implementation only modifies the block and transaction processing logic by adding the bounded counter, keeping the networking, storage, and cryptography layers untouched. We open-source our implementation of Stingray and our orchestration tools to ensure reproducibility of our results[4].

8 Evaluation

We evaluate the throughput and latency of Stingray through experiments conducted on Amazon Web Services (AWS), demonstrating its performance improvements over the state-of-the-art.

We compare Stingray with the consensus-less fast path of Sui [30], called Mysticeti-FPC [5] as to our knowledge, Sui is the only blockchain supporting consensus-less transactions. We did not compare with other consensus-less systems, including FastPay [10], Astro [21], Zef [11], and Brick [4] because they only support payments and are thus not adapted to showcase loads under high concurrency[5]. Furthermore, these systems lack a mechanism to unlock transactions and thus cannot optimistically handle contention.

Our evaluation demonstrates the following claims:

- **C1**: Clients of Stingray submitting a commutative load experience a lower latency and higher throughput than those of the Sui baseline.
- **C2**: There is no noticeable performance difference between Sui using owned objects and Stingray using bounded counters for loads with no contention (i.e., parallel). In other words, there is no performance trade-off in adopting Stingray.
- **C3**: Operations under (crash) faults do not overly penalize Stingray in comparison to Sui. That is, both systems observe similar performance degradation when validators are faulty.

Note that evaluating the performance of BFT protocols in the presence of Byzantine faults is an open research question [8], and state-of-the-art evidence relies on formal proofs.

8.1 Experimental Setup

We deploy all systems on a geo-distributed network of validators, each running on a dedicated machine. App. E details the precise machine specs, validator configuration, and the network setup.

In the following graphs, each data point is the p50 latency and the error bars represent the p90 latency (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 5 minutes. We increase the load of transactions sent to the systems and record the throughput and latency. As a result, each plot illustrates the 'steady state' latency of the system under low load and the maximum throughput it can serve after which latency grows steeply. Transactions in the benchmarks contain 512 bytes. The ping latency between the validators varies from 50ms to 250ms.

By *latency*, we mean the time between when the client submits the transaction and when the transaction is finalized by the validators. By *throughput*, we mean the number of transactions finalized per second during the run.

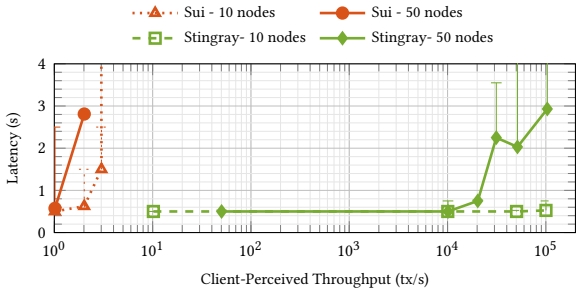

Figure 4: Comparing throughput and latency of Sui and Stingray with a *commutative* load. WAN measurements with 10 and 50 validators.

Protocols	1 tx	10 tx	100 tx
Sui	400-500ms	4-5s	≈ 50s
Stingray	< 500ms	< 500ms	< 500ms

Table 1: Average total time required to submit a load of 1, 10, and 100 commutative transactions to Sui and Stingray.

8.2 Benchmark under Commutative Load

Fig. 4 compares the throughput and latency experienced by clients of Sui and Stingray submitting a load of *commutative* transactions. In Sui, these transactions are implemented using operations on the same owned object [30], whereas, in Stingray, they rely on bounded counter withdrawals with values much lower than the available balance (Sec. 4). Specifically, we measure the maximum rate at which a client can submit these transactions and the corresponding end-to-end latency. Both systems operate in a failure-free wide-area network (WAN) environment, configured with committees of 10 and 50 validators to reflect small and large committee setups.

As expected, Sui clients can submit only about two commutative transactions per second. This is because Sui fails to exploit the commutativity of these transactions and processes them sequentially because it detects false dependencies based on memory access patterns. Consequently, a client must wait approximately 500ms for one transaction to complete before submitting the next. Despite Sui's low baseline latency [5, 30], this commutative load results in a latency proportional to the number of transactions submitted. Consequently, clients perceive significantly higher overall latency as the transaction count increases. For example, as shown in Tab. 1, a single transaction incurs the state-of-the-art latency of 500 ms, but submitting 100 of these transactions causes the total latency to grow linearly to 50 seconds.

In contrast, Fig. 4 demonstrates that Stingray enables parallel submission of commutative transactions, maintaining latencies under

[4] https://github.com/asonnino/mysticeti/tree/stingray (commit `0ae4bb5`)
[5] Parallelizing payments issued from an account can be achieved by splitting the available balance into multiple accounts before initiating the payments.

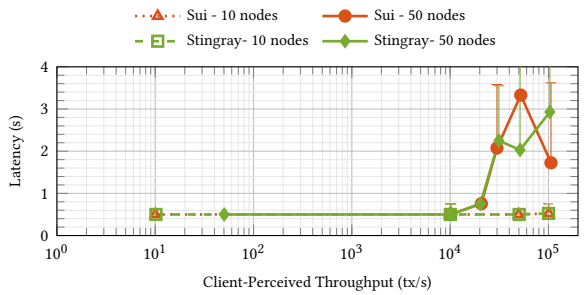

Figure 5: Comparing throughput and latency of Sui and Stingray with a *parallel* load. WAN measurements with 10 and 50 validators.

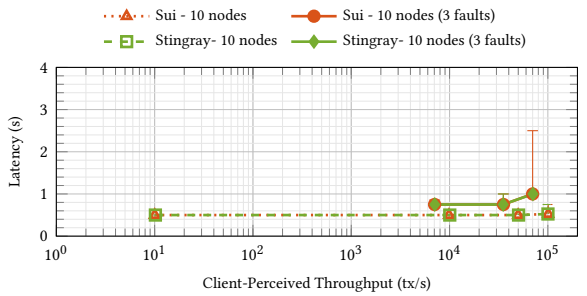

Figure 6: Comparing throughput and latency of Sui and Stingray with a *parallel* load. WAN measurements with 10 validators, 3 faults.

500 ms. This performance holds even for workloads of 10-20k commutative transactions with a large committee of 50 validators (note the log scale on the x-axis) or 100k transactions with a smaller committee of 10 validators. This improvement stems from Stingray's use of the bounded counter, which processes transactions concurrently. As a result, transaction latency remains unaffected by the submission rate until the system reaches saturation. Throughout these benchmarks, the CPU utilization of the validators of both systems remains roughly below 20% and the validators consume less than 10GB of memory (when experiencing the highest loads).

These results validate our claim **C1**: clients of Stingray experience lower latency and higher throughput than those of the Sui baseline when handling commutative transaction loads.

8.3 Benchmark under Parallel Load

Fig. 5 compares the throughput and latency experienced by clients of Sui and Stingray submitting a load of *independent* transactions. Both systems operate in a failure-free wide-area network (WAN) environment, configured with committees of 10 and 50 validators.

In contrast to the previous benchmark, the transactions in this benchmark are implemented using operations on different owned objects. As a result, both systems process these transactions concurrently, and the throughput and latency experienced by the clients of both systems are similar. In both cases, clients experience a latency of less than 500 ms and a throughput of 10-20k transactions per second with a large committee of 50 validators and 100k transactions per second with a small committee of 10 validators.

This result validates our claim **C2**: there is no noticeable performance difference between Sui and Stingray for loads with no contention (in this case, both systems use owned objects), i.e., there is no performance trade-off in adopting Stingray.

8.4 Benchmark under Faults

Fig. 6 compares the throughput and latency experienced by clients of Sui and Stingray submitting a load of independent transactions when a committee of 10 validators experiences 3 (crash) faults, which is the maximum number of faults that can be tolerated in this systems' configuration. The results show that both systems observe similar performance degradation when validators are faulty. In both cases, the throughput drops to about 70k transactions per second, and the latency increases to about 1 second. This result validates our claim **C3**: operations under (crash) faults do not overly penalize Stingray's clients in comparison to Sui's.

9 Related Work

Stingray is closely related to three research directions: consensus-less (fast path) blockchains, parallel execution engines, and replicated data types. Consensus-less blockchains were originally proposed for payments both theoretically [3, 21, 25] and in practice [10] and achieve the lowest possible latency. However, these systems support only payments and no programmability, and do not support validators' reconfiguration. Similarly, Groundhog [39] forego consensus for commutative transactions but doesn't allow non-commutative ones. The Sui Lutris system [13], implemented in the Sui blockchain [30], combines FastPay [10] and the Bullshark [43] consensus protocol to deliver low-latency payments and full programmability. Further, Mysticeti [5] addresses Sui's redundant broadcasting and high signature verification costs. However, all these works adopt a conservative approach to the consensus-less path, limiting it to transactions involving state owned by a single account which avoids contention unless the account owner equivocates. The generic broadcast [38, 40] framework uses consensus-less broadcast for non-conflicting messages and consensus for others, but does not specify which transactions conflict. Stingray expands the design space by allowing transactions that are commutative or have a low risk of contention to run on the consensus-less path. In case of contention, Stingray leverages FastUnlock to restore liveness efficiently.

Parallel execution in blockchains is a relatively new research area led by Solana [45] and FuelVM[6]. On the research side, Block-STM [24] focuses on shared memory and executes blocks of transactions instead of streaming. This creates a tension between high-throughput and low latency as high-throughput needs a high batch size, but collecting this batch increases the latency. On the other hand, Pilotfish [29] focuses on multi-machine execution and follows Sui's streaming architecture. Sui also supports parallel execution in a single machine but only for transactions accessing different memory locations. Stingray improves upon the state of the art by enabling the parallel execution of transactions that access the same memory location, as long as they do not conflict. While this paper emphasizes parallelization on the consensus-less path, the same

[6] https://docs.fuel.network/docs/intro/what-is-fuel/

principles apply even more easily post-consensus as there is no risk of losing liveness due to equivocation.

Another closely related research area is replicated data types. Unlike conflict-free data types (CRDTs) [32, 41], our bounded counter supports non-commutative and non-inflationary state transitions. While some previous work [2, 16–19, 23, 27, 36, 46] developed Byzantine fault-tolerant CRDTs and others developed bounded counters that are safe under no faults [9, 37] and crash faults [7], our work achieves the best of both through the first Byzantine fault-tolerant bounded counter. RDTs with non-commutative operations inherently require coordination among the replicas [6, 12, 28], even without Byzantine faults. Accordingly, our bounded counter also requires coordination but the number of rounds of coordination is at most logarithmic in the counter's initial value.

RapidLane [34] enables concurrent transactions by deferring execution and instead predicting transaction outcomes without reading the object's state. This allows optimistic concurrent processing, including bounded counters. In contrast, Stingray ensures correct execution upfront, avoiding the need for rollbacks due to incorrect predictions. Bazzi et al. [12] enable concurrent payments using small random disjoint quorums to certify each transaction. However, their approach is probabilistic and tolerates only 1/8 faulty validators, while Stingray is deterministic and tolerates up to 1/3 faulty validators.

Acknowledgments

This work is sponsored by Mysten Labs.

References

[1] Paulo Sérgio Almeida and Carlos Baquero. 2019. Scalable eventually consistent counters over unreliable networks. *Distributed Comput.* 32, 1 (2019), 69–89.
[2] Paulo Sérgio Almeida and Ehud Shapiro. 2024. The Blocklace: A Universal, Byzantine Fault-Tolerant, Conflict-free Replicated Data Type. arXiv:2402.08068v3 [cs.DC]
[3] Alex Auvolat, Davide Frey, Michel Raynal, and François Taïani. 2020. Money Transfer Made Simple: A Specification, a Generic Algorithm, and its Proof. *Bull. EATCS* 132 (2020).
[4] Zeta Avarikioti, Eleftherios Kokoris-Kogias, Roger Wattenhofer, and Dionysis Zindros. 2021. Brick: Asynchronous Incentive-Compatible Payment Channels. In *Financial Cryptography (2) (LNCS, Vol. 12675)*. Springer, 209–230.
[5] Kushal Babel, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Arun Koshy, Alberto Sonnino, and Mingwei Tian. 2023. Mysticeti: Reaching the Limits of Latency with Uncertified DAGs. arXiv:2310.14821v4 [cs.DC]
[6] Peter Bailis, Alan D. Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination Avoidance in Database Systems. *Proc. VLDB Endow.* 8, 3 (2014), 185–196.
[7] Valter Balegas, Diogo Serra, Sérgio Duarte, Carla Ferreira, Marc Shapiro, Rodrigo Rodrigues, and Nuno M. Preguiça. 2015. Extending Eventually Consistent Cloud Databases for Enforcing Numeric Invariants. In *SRDS*. IEEE Computer Society, 31–36.
[8] Shehar Bano, Alberto Sonnino, Andrey Chursin, Dmitri Perelman, Zekun Li, Avery Ching, and Dahlia Malkhi. 2020. Twins: BFT Systems Made Robust. arXiv:2004.10617v2 [cs.CR]
[9] Daniel Barbará and Hector Garcia-Molina. 1994. The Demarcation Protocol: A Technique for Maintaining Constraints in Distributed Database Systems. *VLDB J.* 3, 3 (1994), 325–353.
[10] Mathieu Baudet, George Danezis, and Alberto Sonnino. 2020. FastPay: High-Performance Byzantine Fault Tolerant Settlement. In *AFT*. ACM, 163–177.
[11] Mathieu Baudet, Alberto Sonnino, Mahimna Kelkar, and George Danezis. 2023. Zef: Low-latency, Scalable, Private Payments. In *WPES@CCS*. ACM, 1–16.
[12] Rida Bazzi and Sara Tucci-Piergiovanni. 2024. Fractional Payment Transactions: Executing Payment Transactions in Parallel with Less than f+1 Validations. arXiv:2405.05645v1 [cs.DC]
[13] Sam Blackshear, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Xun Li, Mark Logan, Ashok Menon, Todd Nowacki, Alberto Sonnino, Brandon Williams, and Lu Zhang. 2024. Sui Lutris: A Blockchain Combining Broadcast and Consensus. In *CCS*. ACM, 2606–2620.
[14] Ethan Buchman, Jae Kwon, and Zarko Milosevic. 2018. The Latest Gossip on BFT Consensus. arXiv:1807.04938v3 [cs.DC]
[15] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. 2011. *Introduction to reliable and secure distributed programming.* Springer Science & Business Media.
[16] Margarita Capretto, Martín Ceresa, Antonio Fernández Anta, Antonio Russo, and César Sánchez. 2022. Setchain: Improving Blockchain Scalability with Byzantine Distributed Sets and Barriers. In *Blockchain*. IEEE, 87–96.
[17] Margarita Capretto, Martín Ceresa, Antonio Fernández Anta, Antonio Russo, and César Sánchez. 2024. Improving Blockchain Scalability with the Setchain Data-Type. *Distributed Ledger Technol. Res. Pract.* 3, 2 (2024), 12.
[18] Hua Chai and Wenbing Zhao. 2014. Byzantine Fault Tolerance for Services with Commutative Operations. In *IEEE SCC*. IEEE Computer Society, 219–226.
[19] Vicent Cholvi, Antonio Fernández Anta, Chryssis Georgiou, Nicolas Nicolaou, Michel Raynal, and Antonio Russo. 2021. Byzantine-Tolerant Distributed Grow-Only Sets: Specification and Applications. In *FAB (OASIcs, Vol. 92)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2:1–2:19.
[20] Circle. 2025. Fully Backed Digital Dollars. https://www.circle.com/usdc.
[21] Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, and Athanasios Xygkis. 2020. Online Payments by Merely Broadcasting Messages. In *DSN*. IEEE, 26–38.
[22] ethereum.org. 2024. The Complete Guide to Ethereum. https://ethereum.org/en/ Last accessed: Dec 24, 2024.
[23] Davide Frey, Lucie Guillou, Michel Raynal, and François Taïani. 2024. Process-commutative distributed objects: From cryptocurrencies to Byzantine-Fault-Tolerant CRDTs. *Theor. Comput. Sci.* 1017 (2024), 114794.
[24] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. 2023. Block-STM: Scaling Blockchain Execution by Turning Ordering Curse to a Performance Blessing. In *PPoPP*. ACM, 232–244.
[25] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. 2019. The Consensus Number of a Cryptocurrency. In *PODC*. ACM, 307–316.
[26] Martin Kleppmann. 2021. Thinking in Events: From Databases to Distributed Collaboration Software: Keynote at the 15th ACM International Conference on Distributed and Event-Based Systems (DEBS). In *DEBS*. ACM, 15–24.
[27] Martin Kleppmann. 2022. Making CRDTs Byzantine fault tolerant. In *PaPoC@EuroSys*. ACM, 8–15.
[28] Martin Kleppmann and Heidi Howard. 2020. Byzantine Eventual Consistency and the Fundamental Limits of Peer-to-Peer Databases. arXiv:2012.00472v1 [cs.DC]
[29] Quentin Kniep, Lefteris Kokoris-Kogias, Alberto Sonnino, Igor Zablotchi, and Nuda Zhang. 2024. Pilotfish: Distributed Transaction Execution for Lazy Blockchains. arXiv:2401.16292v2 [cs.DC]
[30] Mysten Labs. 2022. Build Without Boundaries. https://sui.io.
[31] Mysten Labs. 2024. Mysticeti. https://github.com/asonnino/mysticeti.
[32] Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, Natacha Crooks, and Joseph M. Hellerstein. 2022. Keep CALM and CRDT On. *Proc. VLDB Endow.* 16, 4 (2022), 856–863.
[33] Linera. 2022. Unlocking the Power of Decentralization. https://linera.io.
[34] George Mitenkov, Igor Kabiljo, Zekun Li, Alexander Spiegelman, Satyanarayana Vusirikala, Zhuolun Xiang, Aleksandar Zlateski, Nuno P. Lopes, and Rati Gelashvili. 2024. Deferred Objects to Enhance Smart Contract Programming with Optimistic Parallel Execution. arXiv:2405.06117v1 [cs.DC]
[35] Satoshi Nakamoto. 2008. *Bitcoin: A Peer-to-Peer Electronic Cash System.* https://bitcoin.org/bitcoin.pdf Last accessed: Dec 24, 2024.
[36] Pezhman Nasirifard, Ruben Mayer, and Hans-Arno Jacobsen. 2023. OrderlessChain: A CRDT-based BFT Coordination-free Blockchain Without Global Order of Transactions. In *Middleware*. ACM, 137–150.
[37] Patrick E. O'Neil. 1986. The Escrow Transactional Method. *ACM Trans. Database Syst.* 11, 4 (1986), 405–430.
[38] Fernando Pedone and André Schiper. 2002. Handling Message Semantics with Generic Broadcast Protocols. *Distributed Comput.* 15, 2 (2002), 97–107.
[39] Geoffrey Ramseyer and David Mazières. 2024. Groundhog: Linearly-Scalable Smart Contracting via Commutative Transaction Semantics. arXiv:2404.03201v1 [cs.DC]
[40] Pavel Raykov, Nicolas Schiper, and Fernando Pedone. 2011. Byzantine Fault-Tolerance with Commutative Commands. In *OPODIS (LNCS, Vol. 7109)*. Springer, 329–342.
[41] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *SSS (LNCS, Vol. 6976)*. Springer, 386–400.
[42] Jakub Sliwinski and Roger Wattenhofer. 2019. ABC: Proof-of-Stake without Consensus. arXiv:1909.10926v3 [cs.CR]
[43] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. Bullshark: DAG BFT Protocols Made Practical. In *CCS*. ACM, 2705–2718.

Algorithm 5 Bounded counter for single owner (user logic)

```
1:  procedure Init(Bal_0)
2:      Version ← v_0
3:      Bud ← η * Bal_0
4:      sentTxs ← ∅

5:  ▷ Invoked when application requires updating the bounded counter
6:  procedure Update(δ)
7:      if Bud + δ < 0 then                          ▷ If update exceeds the budget
8:          VersionUpdate()
9:      ▷ Abort if update exceeds the budget even after version update
10:     if Bud + δ < 0 then return Error
11:     Tx ← {Version: Version, δ: δ}
12:     if δ < 0 then Bud ← Bud + Tx.δ
13:     sendToValidators(Tx)
14:     sentTxs ← sentTxs ∪ {Tx}

15: procedure VersionUpdate()
16:     v ← {PrevVersion : Version, PrevTxs : sentTxs}
17:     for Tx in sentTxs do
18:         Bud ← Bud + η * Tx.δ
19:         if Tx.δ < 0 then Bud ← Bud − Tx.δ
20:     ▷ Send a version update request if there is enough remaining budget, otherwise
        request to convert the bounded counter to an owned object
21:     if Bud ≥ minBudget then sendToValidators(v)
22:     else sendToValidators(convertToOwnedObject(sentTxs))
23:     Version ← v
```


[44] Matthew Weidner and Paulo Sérgio Almeida. 2022. An Oblivious Observed-Reset Embeddable Replicated Counter. In *PaPoC@EuroSys*. ACM, 47–52.

[45] Anatoly Yakovenko. 2018. Solana: A New Architecture for a High Performance Blockchain v0. 8.13. *Whitepaper* (2018).

[46] Wenbing Zhao. 2016. Optimistic Byzantine Fault Tolerance. *Int. J. Parallel Emergent Distributed Syst.* 31, 3 (2016), 254–267.

A Algorithms and Proofs for the Bounded Counter

In Alg. 5, we provide a pseudocode of how an honest user interacts with the bounded counter (corresponding to the description in Sec. 4.3).

Refer to Tab. 2 for a summary of the notation used in the security proof for the bounded counter (proof in Sec. 4.4).

f	maximum number of adversarial validators
Bal_0	initial balance of bounded counter
Bud	validator's signing budget
$\overline{\mathsf{Bud}}_v$	average budget of honest validators for version v
$\eta \triangleq \frac{f+1}{2f+1}$	fraction of Bal_0 assigned to Bud
$\mathsf{Tx}.\delta$	quantity added by Tx to the bounded counter
$\Delta(\mathcal{S})$	sum of the quantities of transactions in the set $\mathcal{S}$
v_0	initial version of the bounded counter
$p(v)$	parent version of v, same as v.PrevVersion
$\mathcal{V}$	set of versions with at least one certified transaction
C_v	set of certified transactions with version v
H_v	history of v: transactions included in the version update requests of v and all previous versions
$\widetilde{H}_v$	certified transactions with v and all previous versions, except those in H_v
$\mathcal{S}^+, \mathcal{S}^-$	Increment, decrement transactions in the set $\mathcal{S}$

Table 2: Table of notation

A.1 Liveness Proof for the Owned Bounded Counter

Theorem A.1. *The bounded counter protocol (validator code: Alg. 1, user code: Alg. 5) satisfies liveness. If the user sends only decrement transactions, all transactions sent by an honest user will be executed by honest validators in $O(\log(\mathsf{Bal}_0))$ rounds.*

Proof. First, we show that when the user runs Alg. 5, the budget Bud computed by the user matches the budget computed by each honest validator. Let Bud_v^c be the budget computed by the user and let Bud_v^j be the budget computed by a validator j at the time when they each update their local Version to v. We will first show that for all validators j, $\mathsf{Bud}_v^j = \mathsf{Bud}_v^c = \eta(\mathsf{Bal}_0 + H_v)$.

To show this, first note that for every honest validator j, $\mathsf{Bud}_v^j = \eta(\mathsf{Bal}_0 + H_v)$. This can be seen from Lem. 4.9 along with the observation that $\widetilde{H}_v^- = \emptyset$, i.e., there are no certified transactions not included in the history of the latest version update, because the user includes all transactions it sent in the version update (Alg. 5 l. 16). Second, we see that $\mathsf{Bud}_v^c = \eta(\mathsf{Bal}_0 + H_v)$. This is because, as argued above, H_v is exactly the set of transactions ever sent by the user, and for each sent transaction, the user's budget updates in Alg. 5 ll. 12, 18 and 19 have the net effect of updating the budget by $\eta * \mathsf{Tx}.\delta$ for each transaction Tx sent by the user.

Finally, for any given version v, the set of decrement transactions $\mathcal{S}$ the user sends satisfies $\mathsf{Bud}_v^c + \Delta(\mathcal{S}) \geq 0$ (Alg. 5 ll. 10 and 12). Therefore, every honest validator signs these transactions eventually (once all messages are delivered). If at most f validators are adversarial, every transaction gets certified, and thus eventually executed by honest validators.

Within a given version, all transactions sent by the user get certified concurrently, without any additional rounds of communication. At each version update, validators must wait to execute transactions with the previous version before executing transactions with the new version, thus successive versions are processed sequentially. If the user sends only decrement transactions, at each version update, the budget gets scaled by a factor $\eta < 1$. Thus, after $O(\log(\mathsf{Bal}_0))$ version updates, the budget will fall below a pre-specified bound minBudget, after which the user can convert the bounded counter object to a standard owned object to spend the remaining balance in a single transaction. □

B Safety Proof for the Collective Bounded Counter

In this section, we prove the safety properties of the collective bounded counter (described in Sec. 5.3, Alg. 2).

Eventual consistency (Thm. 4.3) continues to hold because when an honest validator executes a transaction upon seeing a certificate, it broadcasts the certificate to all other validators. Similarly, validity continues to hold for the collective bounded counter.

To prove global safety, we begin by noting that Lem. 4.5 holds for the collective bounded counter in the same way as for the owned bounded counter. That is, within a single version, any subset of the certified transactions does not spend too much. For clarity, Lem. 4.5 is recapped below. The proof is the same as for the owned bounded counter because the rules for signing a transaction within a given version are the same in the collective bounded counters.

Lemma B.1. *For any version v, let $\overline{Bud}_v$ be the average budget of all honest validators at the time when they set Version to v. Let C_v be the set of certified transactions with version v. Then, for all $T \subseteq C_v$: $\Delta(T) \geq -\frac{1}{\eta}\overline{Bud}_v$.*

We now extend the remainder of the owned bounded counter's security proof (Sec. 4.4) to the collective bounded counter. Since the collective bounded counter allows merging multiple versions to create a new version, we consider the directed acyclic graph (DAG) formed by the versions, in which, unlike the owned bounded counter, each version may have multiple parents.

In the collective bounded counter, versions can be changed through a version update request (as in Alg. 1) or through a version merge request (as in Alg. 2). Def. B.5 generalizes the definition of a version's parent to capture both these cases.

Definition B.2. For any version $v \neq v_0$, define the set of parent versions $P(v)$ as $\{v.\text{PrevVersion}\}$ if v is a version update and v.PrevVersions is v is a version merge.

Even though the structure of the DAG formed by the versions is different, the key property of Lem. 4.7 continues to hold. That is, the set of versions for which there exists at least one certified transaction forms a chain. However, these versions may not be consecutive in the chain.

Definition B.3. There exists a path from v to v', indicated by $v \rightarrow v'$, if for some $k \geq 1$, there exists a sequence $v_1, ..., v_k$ such that $v_1 = v'$, $v_k = v$, and for all $1 < i \leq k$, $v_{i-1} \in P(v_i)$. If $v \rightarrow v'$ doesn't hold, we write $v \not\rightarrow v'$.

Lemma B.4. *Let $\mathcal{V}$ be the set of versions for which there exists at least one certified transaction. If $\mathcal{V} \neq \emptyset$, then $\mathcal{V} = \{v_1, ..., v_{|\mathcal{V}|}\}$ such that $v_1 \rightarrow v_0$ and for all $i = 2, ..., |\mathcal{V}|$, $v_i \rightarrow v_{i-1}$.*

Proof. If no transactions have been certified, $\mathcal{V} = \emptyset$. If at least one transaction is certified, then $v_1 \rightarrow v_0$. This is because initially honest validators sign only transactions with version v_0 (Alg. 1 ll. 4 and 12) and will not sign transactions for a different version until they receive a version update request containing certified transactions (Alg. 1 l. 28) or a version merge request in which $v_0 \in$ PrevVersions (Alg. 2 l. 5).

For any honest validator j, if it updates Version from v to v', it must be such that $v \in P(v')$ (Alg. 1 l. 28, Alg. 2 l. 5). Therefore, for all $v \in \mathcal{V}$, there exists a sequence $v_0, ..., v$ in which there is a path from each version to the next version. In other words, the versions in $\mathcal{V}$ are part of a tree rooted at v_0 with parent links as edges.

Now all that remains to show is that this tree is, in fact, a chain. That is, there is no $v, v' \in \mathcal{V}$ such that $v \not\rightarrow v'$ and $v' \not\rightarrow v$. This follows from quorum intersection. If there was $v, v' \in \mathcal{V}$ such that $v \not\rightarrow v'$ and $v' \not\rightarrow v$, then for both versions v and v', there is a set of $2f + 1$ validators that signed transactions with that version. These two sets of $2f + 1$ validators have at least $2(2f + 1) - n = f + 1$ validators in common (since $n = 3f + 1$). However, since at most f validators are adversarial, at least one honest validator signed both transactions with version v and v'. However, this is a contradiction because an honest validator will not sign transactions for both versions v and v' since there is no path from v to v' or from v' to v. □

Further, we generalize the history of a version (Def. 4.8) to capture version update and version merge requests. Like the owned bounded counter, the collective bounded counter too updates validators' budgets accounting for all transactions in the history of the new version. So, the proof of Lem. 4.9 carries forward similarly in Lem. B.6.

Definition B.5. Define the history H_v of a version as $H_{v_0} = \emptyset$, $H_{v \neq v_0} = H_{P(v)} \cup v$.PrevTxs if v is a version update, and $H_{v \neq v_0} = \bigcup_{v' \in P(v)} H_{v'}$.

Lemma B.6. *For any version $v_i \in \mathcal{V}$, the average budget of all honest validators at the time they upgrade to version v_i satisfies $\overline{Bud}_{v_i} \leq \eta(Bal_0 + \Delta(H_{v_i}) + \Delta(\widetilde{H}^-_{v_i}))$.*

Proof. Throughout the execution, an honest validator i) starts with an initial budget of ηBal_0, then ii) decreases its budget for every decrement transaction signed (Alg. 1 l. 15), iii) updates its budget for every certified transaction included in a version update request (Alg. 1 l. 33) or transitively included in a version merge request (Alg. 2 l. 8), and iv) reclaims its budget for every certified decrement transaction it had previously signed that is included in a version update request (Alg. 1 l. 35) or transitively included in a version merge request (Alg. 2 l. 10). Suppose that $f_r \leq f$ validators are adversarial (so, $n - f_r = 3f + 1 - f_r$ are honest). Combining these four components, the average budget of all honest validators at the time they update to version v_i is

$$\begin{aligned}\overline{\text{Bud}}_{v_i} &\leq \eta\text{Bal}_0 + \frac{2f+1-f_r}{3f+1-f_r}\Delta(C^{i-}) + \eta\Delta(H_{v_i}) - \frac{2f+1-f_r}{3f+1-f_r}\Delta(H^-_{v_i}) \\ &= \eta(\text{Bal}_0 + \Delta(H_{v_i})) + \frac{2f+1-f_r}{3f+1-f_r}\Delta(\widetilde{H}^-_{v_i}) \\ &\leq \eta(\text{Bal}_0 + \Delta(H_{v_i}) + \Delta(\widetilde{H}^-_{v_i})).\end{aligned}$$

□

Theorem B.7. *The collective bounded counter protocol (validator code: Alg. 2) satisfies global safety.*

Proof. For any given subset of honest validators, let T be the set of transactions executed by some validator in this subset. Let v_k be the latest version in T. Since validators only execute certified transactions (Alg. 1 l. 22), $T \subseteq C^k$. Moreover, validators execute transactions in v_k.PrevTxs before executing transactions with version v_k (Alg. 1 l. 21), so $T \supseteq H_{v_k}$.

Recall that we partitioned $C^{k-1} = H_{v_k} \cup \widetilde{H}^-_{v_k} \cup \widetilde{H}^+_{v_k}$, that is, certified transactions with versions up to v_{k-1} may be in the history of version k, and those that are not may be either increments or decrements. Given these constraints, it is sufficient to prove that $\text{Bal}_0 + \Delta(T) \geq 0$ for the worst case $T = H_{v_k} \cup \widetilde{H}^-_{v_k} \cup C^-_{v_k}$ where all decrement transactions and no increment transactions beyond H_{v_k} are executed.

$$\Delta(T) = \Delta(H_{v_k}) + \Delta(\widetilde{H}^-_{v_k}) + \Delta(C^-_{v_k}) \tag{4}$$

$$\geq \Delta(H_{v_k}) + \Delta(\widetilde{H}^-_{v_k}) - \frac{1}{\eta}\overline{\text{Bud}}_{v_k} \quad \text{(Lem. B.1)} \tag{5}$$

$$\geq -\text{Bal}_0 \quad \text{(Lem. B.6)} \tag{6}$$

□

C Security Arguments for FastUnlock

We argue about the safety and liveness of FastUnlock. Intuitively, FastUnlock does not invalidate the finality guarantees of the normal fast path operations. That is, a client holding an effect certificate can be assured that its transaction will never be reverted.

Theorem C.1. *If there exists an effect certificate EffectCert over a transaction Tx, the execution of Tx is never reverted.*

Proof. We assume that the execution of Tx is reverted and show a contradiction. The transaction can only be reverted if there exists an UnlockCert carrying an empty certificate over an ObjectKey modified by Tx. From Check (3.2) of Alg. 3 a correct validator only signs an UnlockVote with an empty Cert only if it has not executed anything for ObjectKey. From our assumption that ObjectKey did admit a no-op there should be $f + 1$ honest validators that did not partake in the generation of the EffectCert of Tx and hence passed the check. Additionally, for the EffectCert to exist by definition it has $2f + 1$ signatories over the ObjectKey in question, at least $f + 1$ of them being honest. This implies a total of at least $f+1+f+1+f = 3f + 2 > 3f + 1$ validators, hence a contradiction. □

The converse also applies, that is, if an UnlockCert exists, then no EffectCert over the ObjectKey will be generated in the fast path. The proof works analogously by adding an extra check during EffectCert generation in which correct validators refuse to process certificates when they recorded **Unlocked** in their UnlockDb.

Next, we show that validators that might process on the consensus path both a Cert (through checkpointing) and UnlockCert will arrive at the same execution result. We prove the case where an UnlockCert is ordered first. For this, we need to enhance the protocol of checkpointing in Sui to check the value of UnlockDb[ObjectKey] and ignore a Cert that tries to process a **Confirmed** ObjectKey, which is a straightforward change.

Theorem C.2. *If a correct validator executes an UnlockCert certificate over ObjectKey as sequenced by the SMR engine, no correct validator will subsequently execute a conflicting Cert as sequenced by the SMR engine.*

Proof. The proof directly follows from the safety property of the SMR engine that all validators will process certificates in the same order. Hence, upon processing UnlockCert, all honest validators mark the execution of ObjectKey as confirmed by setting UnlockDb[ObjectKey] ← **Confirmed** (l. 13 of Alg. 4). Then, Check (4.1) of Alg. 4 (and its dual added at the checkpoint algorithm) ensures that if any further Cert or UnlockCert with a conflict is given as input to the execution engine it is rejected. □

The converse can be proven in the same manner since we enhance the execution of Cert during the checkpoint process with updating UnlockDb[ObjectKey] ← **Confirmed** after processing. Then all UnlockCert on the ObjectKey will be rejected at the Check (4.1) of Alg. 4.

Based on the above theorems, we can prove safety of the overall Stingray system that uses FastUnlock.

Theorem C.3. *Stingray satisfies safety (Def. 3.3).*

Proof. Validity holds because validators execute only transactions with valid certificates.

Next, we prove global safety. For any given object O, let $T_p^O(t)$ be the set of transactions executed on that object by validator p up to time t. Due to Thm. C.2, for any two validators p, q, $T_p^O(t) \subseteq T_q^O(t)$ or $T_q^O(t) \subseteq T_p^O(t)$. Thus, $\bigcup_{p \text{ honest}} T_p^O(t)$ is the set executed by some honest validator. Consider the sequence T^O made by arranging transactions in this set in the order of the object version. Then, T^O contains all transactions in $\bigcup_{p \text{ honest}} T_p^O(t)$ and this sequence respects the application's validity constraint since an honest validator executed transactions in this sequence. Finally, let T be the merged sequence of T^O for all objects, where the merge preserves the partial order for each object. Due to the independence of different objects, T also satisfies the validity predicate, thus proving global safety. □

Liveness argument. Intuitively, we argue that FastUnlock—and its composition with normal fast path operations—neither deadlocks nor enables unjustified aborts (which could starve an object from progress).

Lemma C.4 (Unlock Certificate Availability). *A correct user can obtain an unlock certificate UnlockCert over a valid ObjectKey.*

Proof. A correct validator always signs UnlockVote if it passes the check of Alg. 3. Well-formed UnlockRqt always come with a valid authentication path (Check (3.1)), and Check (3.2) always returns an UnlockVote. As a result, if UnlockRqt is disseminated to $2f + 1$ correct validators by a correct user, they will eventually all return an UnlockVote. The user then aggregates those votes into a unlock certificate UnlockCert over ObjectKey. □

Theorem C.5 (FastUnlock Liveness). *If a correct and authorized user initiates a fast-unlock protocol, the ObjectKey in question will eventually admit a new transaction.*

Proof. A correct and authorized user will eventually generate an unlock certificate by Lem. C.4. Additionally from the liveness property of SMR the unlock certificate will either eventually be added as part of the SMR output or the epoch will end. If the first happens by agreement of consensus the UnlockCert will be executed by all validators, leading to the termination of the fast-unlock protocol and an updated ObjectKey. If the epoch ends, all locks are dropped and liveness of all ObjectKey are automatically available for processing. □

Thm. C.5 is sufficient for correct users as either they will manage to no-op an incorrect invocation of ObjectKey, drive the tranasction of a correct Tx to completion, or the epoch end will automatically unblock them. This means that there will always be an available ObjectKey to be modified.

Now that we proved that an authorized user will succeed into unblocking the ObjectKey we also need to show that an unauthorized user will not succeed into starving legitimate users from progress through abusing fast-unlock.

Theorem C.6 (Starvation Freedom). *No user can successfully initiate a fast-unlock on an ObjectKey it cannot produce an Auth for.*

Proof. All honest validators check the authorization vector Auth of the requesting user (l. 4 in Alg. 3). This means that no honest party will lock an object without an authorization, including slow parties that have not yet seen the ObjectKey which will reject or cache the request for later processing. As a result, by the model, there will never be sufficient UnlockVote to generate an UnlockCert driven by an unauthorized user. □

Theorem C.7. *Stingray satisfies liveness (Def. 3.4).*

Proof. First we prove progress (Def. 3.4). Every transaction with a valid certificate will be eventually executed unless there is an UnlockCert containing Cert = **None**. Moreover, if the owners of the transaction's input objects do not equivocate, there will be no UnlockRqt for those object (recall that only the object owners can issue an UnlockRqt). This ensures progress.

Next, we prove eventual consistency (Def. 3.4). If a validator p_1 executes a transaction, it must have seen the transaction finalized, i.e., $2f + 1$ validators signed a certificate for that transaction. Therefore, at least $f + 1$ honest validators must have seen a certificate Cert for that transaction. If there is no UnlockCert for the transaction's input objects, then eventually, all honest validators will receive $2f + 1$ signatures on the certificate, and thereafter execute the transaction. If there is an UnlockCert for one of the transaction's input objects, then UnlockCert must contain Cert because at least one honest validator whose signature is in UnlockCert must have seen Cert. Therefore, even in this case, all honest validators will eventually execute the transaction. □

Generalization to multi-object unlock. The multi-object unlock protocol can be seen as a composition of many single-object unlock protocols (one per object) as well as a single commit protocol (for the accompanied transaction). As a result, the safety of the protocol follows from the fact that objects are independent of each other so if at least one has a prior certificate then the commit flow will lead to committing that prior certificate (which iteratively applies to all objects with prior certificates). If on the other hand, no object has a prior certificate then the workflow is the combination of the simple FastUnlock per object together with the shared-object path of committing the transactions of Sui which is safe as proven in the original Sui paper [13]. Second, we explore liveness. There are two cases: (1) all objects can be unlocked, (2) one or more objects are already certified. The first case is exactly the same as the simple protocol of Sec. 6 and a proof would follow exactly the same structure. For the second case, we first look into the base case of a single object that is already certified which is already proven in the previous sections. For more than one objects we can see that since the validator adds all certificates in their reply and then processes each certificate separately when handling the unlock cert then there is no interaction between certificate processing and can be considered a batch of independent requests.

Finally, for liveness the accompanied transaction might need to acquire locks. This is also an independent invocation of the Sui fast-path. As a result if the transaction is valid it will either succeed or blocks. In the latter case, the user will have to invoke fast-unlock again including in the set of to-unlock objects the newly blocked objects of the transaction. Given that there is a finite number of objects a user holds an unlock request will eventually succeed.

Algorithm 6 Process unlock requests (multi-object)

```
1:  ▷ Handle UnlockRqt messages from clients.
2:  procedure ProcessUnlockTx(UnlockRqt)
3:      ▷ Check (6.1): Check authenticator.
4:      if !valid(UnlockRqt) then return error
5:      ▷ Collect certificates.
6:      c ← None
7:      for ObjectKey ∈ UnlockRqt.ObjectKeys do
8:          c ← c ∪ LockDb[ObjectKey]
9:      UnlockVote ← sign(UnlockRqt, c)
10:     ▷ Record the decision to unlock.
11:     if c == None then
12:         for ObjectKey ∈ UnlockRqt.ObjectKeys do
13:             UnlockDb[ObjectKey] ← Unlocked
14:     return UnlockVote
```

D Contention Mitigation

The basic FastUnlock protocol speeds up recovery from loss of liveness due to mistakes. However, Stingray aims to support workloads on the fast path that are truly under contention. In this case, the basic protocol in Sec. 6 is insufficient, since it can result in multiple rounds of locking and no-op unlocking without any user transaction being committed. We present a protocol that proposes a new transaction during the unlock phase that is executed once the unlock is sequenced, ensuring liveness.

In the following protocol, we additionally allow users to unlock multiple objects at once. The multi-object unlock protocol follows the same general flow as the single-object unlock protocol described in Sec. 6. We now describe steps ❶-❻ depicted in Fig. 3 for the multi-unlock protocol.

Protocol description. The user first creates an *unlock request* specifying a set of objects to unlock:

$$\mathsf{UnlockRqt}([\mathsf{ObjectKey}], \mathsf{Tx}, \mathsf{Auth})$$

This message contains a list of the object's keys [ObjectKey] to unlock (accessible as UnlockRqt.ObjectKeys), a new transaction Tx to execute if the unlock process succeeds, and an authenticator Auth ensuring the sender is authorized to access all objects in [ObjectKey]. The user broadcasts this message to all validators (❶).

Alg. 6 describes how each validator handles this unlock request UnlockRqt. They first perform Check (6.1) l. 4 to check the authenticator Auth is valid with respect to all objects. This check ensures that the user is authorized to mutate all the objects referenced by UnlockRqt and to lock all owned object referenced by Tx. The validator then collects any certificates for the objects referenced by UnlockRqt (l. 8) and adds them to the response as Cert. The validator then marks the object in UnlockRqt as reserved for transaction execution through consensus only (l. 13).

The validator finally returns an *unlock vote* UnlockVote to the user:

$$\mathsf{UnlockVote}(\mathsf{UnlockRqt}, [\textbf{Option}(\mathsf{Cert})])$$

This message contains the unlock message UnlockRqt itself and possibly a set of certificates [Cert] on transactions including the object keys referenced by UnlockRqt (possible empty) (❷). If Cert

Algorithm 7 Process unlock certificates (multi-object)

```
1:  ▷ Handle UnlockCert messages from consensus.
2:  procedure ProcessUnlockCert(UnlockCert)
3:      ▷ Check (7.1): Check no transaction already processed.
4:      for ObjectKey ∈ UnlockCert.ObjectKeys do
5:          if UnlockDb[ObjectKey] = Confirmed then
6:              return
7:      ▷ Check (7.2): Check message validity.
8:      if !valid(UnlockCert) then return error
9:      ▷ Check (7.3): Can we execute the tx?
10:     v ← [ ]
11:     if UnlockCert.Cert = [ ] then
12:         Tx ← UnlockCert.UnlockRqt.Tx
13:         EffectSign ← exec(Tx, UnlockCert)
14:         v ← EffectSign
15:         for ObjectKey ∈ UnlockCert.ObjectKeys do
16:             UnlockDb[ObjectKey] = Confirmed
17:     else
18:         for Cert ∈ UnlockCert.Cert do
19:             EffectSign ← exec(Cert)
20:             v ← v ∪ EffectSign
21:             for ObjectKey ∈ Cert.ObjectKeys do
22:                 UnlockDb[ObjectKey] = Confirmed
23:     return v
```

is not empty the certified transactions may have been finalized, and should be executed instead of the new transaction.

The user collects a quorum of $2f + 1$ UnlockVote over the same UnlockRqt message and assembles them into an *unlock certificate* UnlockCert:

$$\text{UnlockCert}(\text{UnlockRqt}, \text{Cert})$$

where UnlockRqt is the user-created certified unlock message and UCert is the unions of all set of certificates received in UnlockRqt responses. The user submits this message to the consensus engine (❸) The consensus engine sequences all UnlockCert messages; all correct validators observe the same output sequence (❹).

Alg. 7 describes how validators process these UnlockCert messages after they are sequenced by the consensus engine. The validator first ensures they did not already process another UnlockCert or Cert through checkpoint for the same objects keys (l. 5). They then check UnlockCert is valid, that is, the validator ensures (i) it is correctly signed by a quorum of authorities, and (ii) that all certificates [Cert] it contains are valid (l. 8). The validator can only execute the transaction Tx specified by the user if UnlockCert.Cert is empty (l. 11). The validator then marks every object key of [ObjectKey] as **Confirmed** to prevent any future unlock requests on the ObjectKey from overwriting execution with a different transaction (l. 22) and returns a set of EffectSign to the user (❺).

The user assembles an EffectSign from a quorum of $2f + 1$ validators into an *effect certificate* EffectCert that determines finality (❻).

D.1 Handling Gas Objects

Typical transactions not only mutate objects but also consume a gas object to pay for the computation. If, however, the transaction is equivocated then this gas is locked as well. For this reason Stingray requires a fresh gas-object in order for consensus to process the unlock request. Specifically together with Alg. 3, the parties should provide a fresh gas object for their request. This gas object is checked for validity along with the check in l. 4 and locked for the unlock transaction in l. 10. When the user collects the no-commit proof in the second step of the protocol, the $2f + 1$ collected signatures also serve as a certificate for the gas object. The consensus then checks the validity of the certificate and spends it locally before entering Alg. 4. Then when consensus executes the transaction, one of three scenarios may happen:

- The unlock request is valid and includes a certificate. Then the execution happens as usual and both the gas object for the unlock and the gas object for the execution are consumed.
- The unlock request is valid and comes with a no-op. Then the gas object for unlock is consumed. If there was some locked transaction racing the FastUnlock then the accompanying gas object is potentially blocked. The user can then explicitly unlock that gas object by running FastUnlock.
- The unlock request is not processed because a checkpoint certificate already executed a transaction. Then the gas object is still consumed without altering the state of the ObjectKey.

Note that if gas objects are implemented using bounded counters, the same gas object can be spent concurrently for the transaction and the unlock, thus the above problem wouldn't exist.

E Detailed Experimental Setup

This section complements Sec. 8.1 by specifying the network setup and machine specs used in the benchmarks presented in Sec. 8.

We deploy all systems 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 uniformly distributed across those regions. Each machine provides 10 Gbps of bandwidth, 32 virtual CPUs (16 physical cores) on a 3.1 GHz Intel Xeon Skylake 8175M, 128 GB 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 satisfy the minimum required to run a Sui node as recommended by the Sui Foundation[7].

[7] https://docs.sui.io/guides/operator/validator-config