Mangrove: Fast and Parallelizable State Replication for Blockchains

Anton Paramonov
ETH Zurich

Yann Vonlanthen
ETH Zurich

Quentin Kniep
ETH Zurich

Jakub Sliwinski
ETH Zurich

Roger Wattenhofer
ETH Zurich

Abstract

Mangrove is a novel scaling approach to building blockchains with parallel smart contract support. Unlike in monolithic blockchains, where a single consensus mechanism determines a strict total order over all transactions, Mangrove uses separate consensus instances per smart contract, without a global order. To allow multiple instances to run in parallel while ensuring that no conflicting transactions are committed, we propose a mechanism called Parallel Optimistic Agreement. Mangrove is optimized for performance under *optimistic* conditions, where there is no misbehavior and the network is synchronous. Under these conditions, our protocol can achieve the latency of 2 communication steps between creating and executing a transaction.

2012 ACM Subject Classification Computer systems organization → Distributed architectures; Security and privacy → Distributed systems security

Keywords and phrases Blockchain, Parallelization, Low-latency, Consensus

1 Introduction

Scalability remains a major challenge for blockchain systems. Arguably, no single blockchain currently offers sufficient throughput to support traditional Web 2.0 applications in a trustless manner [46]. In the modern blockchain landscape, users are fragmented across numerous Layer-1, Layer-2, and even Layer-3 chains. This fragmentation raises concerns about the interoperability and latency of decentralized applications built on top of these chains.

Techniques like sharding, where network nodes are divided into groups to process transactions in parallel [55, 4], have gained considerable attention as potential solutions to the problem. Although sharding has been proven to enhance system performance, the techniques come with drawbacks and limitations. Specifically, sharded systems experience significant latency (or abortion rate) when dealing with smart contracts with high contention. Furthermore, they require cross-shard agreement for transactions spanning multiple shards.

Since it has been established that consensus is not necessary for applications like payments [34, 32], the research on consensus-less payment systems [13] has offered a remarkably simple alternative solution to the problem of scaling. Foregoing consensus, these systems offer a model in which every validator can parallelize processing and execution of all transactions without limitation. In contrast, sharding protocols assume a rigid division of validators that complicates the system and limits the potential for parallelizability. However, consensus-less systems can support only a limited range of applications, as consensus is necessary for general smart contracts.

Our Contributions. We address the following research question:

> "Can a protocol supporting consensus exhibit the advantages of consensus-less payment systems?"

and answer in the positive. We summarize our contributions in the following.

- We propose the Replicated Actor Model, a novel execution model for blockchain systems based on externally owned accounts (user actors) and smart contracts (reactive actors), that makes dependencies explicit with parallelizability in mind.
- We introduce Parallelizable Optimistic Agreement (POA), a consensus primitive that can be used to achieve consensus on the stream of incoming transactions for individual smart contracts. POA instances are designed to run in parallel, while ensuring that conflicting (i.e., double-spending) transactions cannot be committed.
- The resulting system, called Mangrove, combines limitless parallelization, low latency, and support for general smart contracts. In Mangrove, no single validator can delay the entire system's progress, and congestion at one smart contract does not impede the rest of the system. In other words, the transaction throughput of different smart contracts can be scaled horizontally by validators. Additionally, our system achieves optimal latency in optimistic conditions, where users or validators do not misbehave, and the network is synchronous. Under these conditions, a block producer can commit a transaction to a smart contract in *two* communication steps.

2 Related Work

Consensusless Systems. Blockchain-based systems use consensus mechanisms as their core building block. However, consensus tolerating Byzantine faults [39] is inherently slow. First blockchain systems such as Bitcoin [42] and Ethereum [18] are notorious for their limited performance.

However, it has been established that consensus is not necessary for many applications, such as payments [34, 32]. Designs such as Fastpay [13], Astro [22], and Accept [41] propose remarkably simple solutions that inherently parallelize the workload. Validators in these systems can easily add computational resources to process more transactions. Tonkikh et al. [54] and Bazzi et al. [14] show that even dependencies between transactions of the same issuer can be resolved without consensus. Other systems, such as Groundhog [45], Setchain [20], and Pod [8] avoid consensus by using commutative semantics. Frey et al. [30] and Sridhar et al. [52] recently introduced new consensus-free objects, inspired by Byzantine fault-tolerant CRDTs [35]. Mangrove is orthogonal to these efforts, as it focuses on the interplay between all types of objects (with- or without consensus). Albouy et al. [6] show how a consensusless system can also provide anonymity properties in a lightweight fashion.

Despite their numerous advantages, consensusless systems are inherently unsuitable for many applications, as general smart contracts require consensus [7]. The CoD [48] primitive alleviates this problem to a limited extent, by mixing payment system-like logic with consensus as fallback. In turn, it exhibits the problems of consensus systems, like poor parallelizability.

Sui [16, 37] is a modern blockchain system that recognizes the mentioned problems and incorporates a consensusless component in its design. Sui relies on consensus only for complex tasks, like ordering accesses to the same shared objects. We further increase scalability, by allowing parallelism even for shared objects. In addition, Mangrove outperforms Sui in the number of communication rounds needed for transaction execution. Our proposed system

shares many characteristics with Basil [53], namely parallel execution of non-conflicting transactions and fast commits under optimistic conditions. Contrary to Basil, in Mangrove users do not have to drive progress themselves and benefit from lower latency.

Fast Byzantine Consensus. Martin and Alvisi [40] present an algorithm that achieves Byzantine Consensus in just two communication steps under optimistic conditions, specifically assuming an honest leader and a synchronous network. They introduce a parameter $p \leq f$, which represents the number of failures supported by the fast path and show a resilience bound of $n \geq 3f + 2p + 1$. Subsequent work explores the potential and pitfalls of fast consensus, both for single-shot consensus and state machine replication [49, 1, 33]. Recently, Kuznetsov et al. [38] and Abraham et al. [2] revisited this topic, pointing out that for the category of protocols where the set of proposers is a subset of the validators, a lower resilience bound of $n \geq max(3f + 1, 3f + 2p - 1)$ can be achieved.

In contrast to prior [38] that limits the set of proposers to a subset of validators, our protocol allows all users to act as proposers. This generality requires us to meet the more stringent resilience bound of $n \geq 3f+2p+1$ by Martin and Alvisi. Our protocol, Mangrove, matches this bound and thus achieves optimal resilience in this setting.

Parallel Execution. Parallelization is a natural way to improve scalability and can be applied at various levels in blockchains. Parallel execution [31, 10, 26, 43] aims to accelerate the local execution of transactions by the validator across cores, while distributed execution [36] leverages multiple machines per validator. Both approaches concentrate on improving local execution, whereas our work targets the elimination of the bottleneck of the single agreement mechanism.

Sharding. Sharding [3, 9, 24, 4] is a technique of splitting the system state among disjoint groups of validators called shards. As shown in [4], while sharding is efficient when a transaction only accesses a part of the system within one shard, it can result in high latency or abortion rate [5] for transactions that span multiple shards, especially for highly contested actors. In Mangrove, "popular" actors do not slow the progress of the whole system. Instead, actors progress independently, ensuring that the system's overall performance remains unaffected by the contention of individual actors.

3 Replicated Actor Model

Today, it is common for blockchains to define a total order over all transactions [42, 18]. Thus, the ensuing sequential and atomic execution of transaction bundles is often the only considered execution model. The obtained atomic composability property allows for (potentially counterintuitive) applications such as flash loans [44].

In this work, we challenge the status quo of this execution model. To this end, we define the replicated actor model, which foregoes global sequential ordering and is more suited for parallelism. In Appendix A we discuss the model's expressiveness and even propose an extension, which optionally allows for the reintroduction of (targeted) atomic composability.

Our model is closely related to the object model of Sui [16]. We differentiate between the following four types of components.

Actors. A *user actor* is associated with a digital signature key pair. We say that a key pair (user) *controls* a user actor. *Reactive actors* are analogous to smart contracts and can be

thought of as a Turing machine with arbitrary state that can be changed via computations. Actors can emit new transactions.

Objects. *Owned objects* are objects owned by some actor, e.g. gas, tokens, or NFTs. Every type of owned object is associated with a set of actions that can be performed over it. These actions are specified in a global *read-only object* containing the type definition. For example, for gas objects or tokens, those actions may include splitting and merging. Ownership of objects can be transferred. Assume actor A owns a gas object O worth 10 coins and wants to transfer 2 coins to actor B. Then A might perform `split([8, 2])` on O to receive two objects worth 8 and 2 coins respectively, and transfer the latter to B.

Users. A *user* is an external entity associated with a key pair, who interacts with the system by creating and giving instructions to actors through their user actors.

Validators. The *validators* are the network participants in charge of running Mangrove. Validators participate in the broadcast and consensus algorithms and are responsible for keeping a consistent state of the system by maintaining the ownership records of each owned object and the state of actors.

3.1 Validators

We consider a set of n validators, denoted by $\mathcal{V}$, which we assume to be known to all users and validators. We require that at most f of them are Byzantine [39]. We call non-Byzantine validators *honest*. Additionally, under optimistic conditions and when less than p validators are Byzantine, a transaction can be committed in two communication steps. Mangrove assumes the participation of $n \geq 3f + 2p + 1$ validators.

Each validator maintains a dedicated state associated with each actor in the system which we call an *entity*. We denote an entity of validator V corresponding to an actor A with $V.A$. Different entities may be located on different machines at the validator's discretion and can communicate with each other.

Entities of different validators communicate via Outer Links, and entities within a validator communicate via Inner Links. The Inner and Outer Links implement Perfect Links [19] and expose the following interface:

- `function` $Send(m, A)$: sends message m to A
- `callback` $Deliver(m, A)$: fired upon receiving message m from A

Apart from direct messages, validators use two agreement primitives: Parallel Optimistic Agreement (POA), which is used for transactions involving reactive actors, and Parallel Optimistic Broadcast (POB), which is used for transactions that involve only user actors. Both primitives contain a fast path, consisting of (i) a broadcast step and (ii) a single voting step. In case of a failure (and only in case of failure), the fast path is followed by a failover mechanism (slow path) that ensures safety and liveness. Both primitives run in consecutive instances and are described in Section 5 and Appendix C respectively.

Through these agreement primitives, we ensure that honest validators have a consistent view of the state of each reactive actor and the ownership of owned objects.

3.2 Transactions

Every transaction in our system *consumes* owned objects, meaning that once a transaction is executed, the objects it consumed no longer exist. Every transaction would consume a *gas* object to pay for computation fees, the economics of which we leave outside the scope of this work. Every transaction has a $Code$ field, that specifies a list of commands to perform when a transaction is being executed. These commands can be (a) actions over owned objects a transaction consumed or created, (b) creating owned objects, (c) creating new actors, and (d) issuing new transactions. The model differentiates between user and reactive actor transactions.

User Actor Transactions. Users can instruct a user actor they control to issue a transaction. Transactions issued by the user actor are categorized based on the presence of a recipient. If a transaction does not have a recipient, it is referred to as a UA transaction. If it does have a recipient, which is always a reactive actor, it is referred to as a UA-RA transaction.

A UA transaction is of the form $\langle A, sn, [O_1, \ldots, O_k], Code\rangle$ where A, a user actor, is the sender, $sn \in \mathbb{N}$ is a sequence number and it *consumes* owned objects $O_1, \ldots, O_k$ (that must be owned by A) and performs actions specified in $Code$ over them. $Code$ is allowed to create new owned objects at other user actors (arbitrarily many of those), but not reactive actors. It may also spawn new reactive actors. The reason a UA transaction can create new owned objects at user actors but not reactive actors is that the former is commutative whereas the latter requires agreement on the order.

UA-RA transactions are of the form $\langle A, sn, X, [O_1, \ldots, O_k], Code_{\mathsf{pre}}, Call, Code_{\mathsf{post}}\rangle$ instead. That is, they additionally include a recipient X that must be a reactive actor and a $Call$ field specifying a function call to perform on X. A reactive actor might issue its own transactions as a result of processing a $Call$. $Code_{\mathsf{pre}}$ can operate over the consumed objects and prepare them for input into the function call. $Code_{\mathsf{post}}$ can instead operate over the objects returned by the function call and, for example, decide whether and which additional transactions to spawn based on them. The pre- and post-processing $Code$ blocks may spawn transactions of their own and can be executed in parallel to any other transaction since they do not have access to the internal state of the reactive actor.

▶ **Definition 1** (Conflicting Transactions). *Two user transactions tx and tx' with $tx \neq tx'$ and $tx.sender = tx'.sender$ are said to be* conflicting *if $tx.sn = tx'.sn$.*

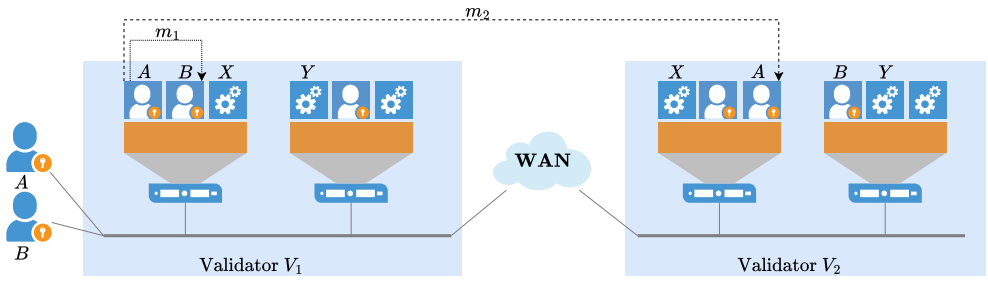

Figure 1 System entities. The validators V_1 and V_2 maintain the user actors corresponding to users A and B, and reactive actors, e.g., X and Y. The message m_1 is sent from $V_1.A$ to $V_1.B$ via Inner Links and a message m_2 to $V_2.A$ via Outer Links. Entities can be spread across multiple machines and organized independently by each validator.

Users are responsible for issuing non-conflicting transactions with consequent sequence numbers. Moreover, users are also responsible for making sure that a user actor they are issuing a transaction for will eventually own the objects consumed by the transaction. We highlight that a user failing to adhere to these requirements has no global impact on the system, but just on the actors controlled by them.

Reactive Actor Transactions. Unlike user actors, reactive actors only issue transactions as a potential result of executing an incoming transaction.

A reactive actor transaction (RA-RA) is always sent to another reactive actor and shares the same structure as a UA-RA transaction, except it omits the sequence number. This omission is because the order of outgoing RA-RA transactions is determined by the order of incoming transactions. The sender is the reactive actor that produced the transaction. Upon receiving either a UA-RA or an RA-RA transaction, a reactive actor might perform computation to change its state based on the current state and *Call* data specified in the transaction.

3.3 Network and Computation Model

The partial synchrony settings of Global Stabilization Time (GST) and Unknown Delta [27] constitute the gold standard of assumptions under which modern blockchains are designed to operate. MANGROVE is designed to function in the GST network model, i.e., correctness is ensured even in complete asynchrony, and liveness is achieved after an arbitrary point in time called Global Stabilization Time, or GST for short. Before GST, messages can be delayed with arbitrary delay, but every message sent at time t must be delivered by $\max(t + \Delta, GST + \Delta)$. The parameter Δ is assumed to be known and fixed.

We assume the time required for local computations and messaging internal to a validator to be negligible.

4 Mangrove Overview

The core principle behind MANGROVE is to have a dedicated agreement mechanism *per actor*. In practice, agreement is achieved differently for the different types of transactions.

- For user actors, validators should agree on *outgoing transactions* for each given sequence number. This, paired with sequential transaction execution, guarantees them a consistent view of the system [22]. To this end, UA transactions are disseminated through Parallel Optimistic Broadcast (POB), and the agreement follows either from the fast path if the optimistic conditions are met, or from the failover mechanism in case they are not.
- Instead, for reactive actors, the validators must agree on *incoming transactions*. Thus, both UA-RA and RA-RA transactions go through the Parallel Optimistic Agreement (POA) mechanism of the reactive actor of the transaction recipient. The consecutive POA instances provide a total order of *incoming transactions* to execute on a reactive actor. Importantly, since agreement is inherited from the POA properties, a dedicated agreement on the user actor can be optimistically skipped, thus also providing two step finalization latency in the fast path.

Since POB is simpler and less general than POA, we describe POA in Section 5 and POB in Appendix C. The complete process for handling UA and UA-RA transactions is detailed in Section 7.2, while the processing of RA-RA transactions is described in Section 7.3.

Both POA and POB were designed with three main goals in mind:

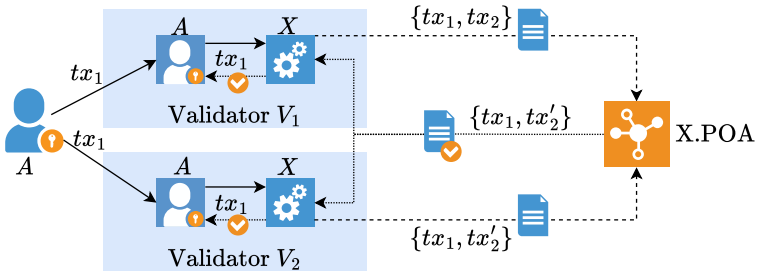

Figure 2 Mangrove UA-RA transaction processing. First, user A broadcasts tx_1 to all $V_i.A$, who relay it to $V_i.X$ (full arrows). Then all $V_i.X$ propose tx_1 and other transactions they have to X.POA (dashed arrows). Finally, when tx_1 is included in a block decided by X.POA, all $V_i.X$ notify $V_i.A$ of the decision (dotted arrows).

(i) They have a low good-case latency, that is, they terminate in two communication steps under optimistic conditions.

(ii) A system running multiple instances in parallel can be sure that at most one transaction for each pair of user and sequence number is decided across all instances.

(iii) A system running multiple instances in parallel does not suffer bottlenecks.

Wait-Free Locking.

To achieve these goals, the POA and POB instances do not communicate directly but instead obtain locks via Inner Links to the user actor entities introduced in Section 3. These entities provide security against conflicting transactions by (locally) tying a sequence number to a transaction. To describe this process more precisely, we introduce the following notation, also used throughout the remainder of this work:

Notation. U denotes an arbitrary user, $V.A$ denotes an arbitrary user actor entity of an arbitrary validator, and $V.X$ denotes an arbitrary reactive actor entity of an arbitrary validator.

User actor entities maintain the following two data structures.

▶ **Definition 2** (Slow-Path Locked). *Each $V.A$ maintains a map $SPLocked$, mapping sequence numbers to transactions. If $V.A.SPLocked[tx.sn] = tx$, we say that $V.A$* SP-locked *(slow-path locked) tx.*

▶ **Definition 3** (Fast-Path Locked). *Each $V.A$ maintains a map $FPLocked$, mapping sequence numbers to transactions. If $V.A.FPLocked[tx.sn] = tx$, we say that $V.A$* FP-locked *(fast-path locked) tx.*

Intuitively, SP-locking a transaction ensures that $V.X$ will never propose a conflicting transaction in the slow path, whereas FP-locking a transaction ensures $V.X$ will never vote for a proposal containing a conflicting transaction in the fast path. However, note that V can FP-lock tx and SP-lock tx' with tx and tx' being conflicting. That might happen in a case V received tx from a user but received a lot of votes for a proposal containing tx', which "forces" V to propose tx' in the slow path.

We describe the interplay between the different building blocks of Mangrove and their correctness in Section 7 and in Appendix F.

5 Parallel Optimistic Agreement

This section describes the algorithm used by validators to agree on a block B of transactions to be executed at a reactive actor X. For every validator V and reactive actor X, $V.X$ has a *pool*, that is, a set of transactions that $V.X$ wants to execute on X. This algorithm is defined assuming a designated validator L, called leader. The Parallel Optimistic Agreement (POA) primitive has an interface consisting of:

- `function` INITIATE$(k, pool)$: start the k-th instance with a transaction *pool*
- `callback` DECIDE(k, B): decide a block B in k-th instance

First, in the *fast path* the leader broadcasts its proposed block and all other validators cast their votes. A validator decides on a block once it receives *enough* votes, a process we refer to as a fast-path decision. Following the fast path, and only if necessary, a *slow path* (whose components are described in Section 6) is initiated to ensure liveness in cases where users or the leader misbehave or the network experiences asynchrony. We refer to a decision made in the slow path as a slow-path decision.

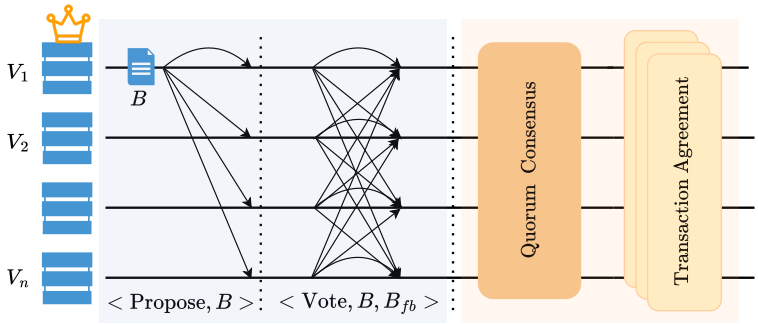

Figure 3 POA scheme. In the fast path (blue rectangle), the leader (in this case V_1) broadcasts their proposal B. Then, validators cast their vote on the proposal, and append their own transactions B_{fb}. In case the fast path fails, validators participate in the slow path (orange box), which consists of one instance of Quorum Consensus and multiple instances of Transaction Agreement.

5.1 Properties

A single instance of POA satisfies the following properties.

▶ **Property 4** (Agreement). *If two honest validators decide blocks B and B' respectively, then $B = B'$.*

▶ **Property 5** (Termination). *Every honest validator eventually decides a block.*

▶ **Property 6** (Fast Termination). *If L is honest, at most p validators misbehave, the system has reached GST, and for every UA-RA transaction $tx \in L.X.pool$ the user who issued tx is honest, then all honest processes decide and stop sending messages in two communication steps.*

▶ **Definition 7** (Emitted transaction). *We say that a transaction tx is* emitted *if it is either (i) a user transaction from an actor A signed and broadcast at the moment where the whole system is in the state such that there exists an honest validator $V.A$ who will eventually*

execute a transaction with a sequence number $tx.sn - 1$ and who will eventually own all objects consumed by tx or (ii) produced by a reactive actor at an honest validator.

▶ **Property 8** (Validity). *(I) If a transaction tx is present in the pool of all honest validators at the start of the protocol and L is honest, then tx is included in the decided block. (II) If a transaction is included in the decided block, it was emitted.*

▶ **Property 9** (No Conflict). *Conflicting transactions cannot be simultaneously decided within POA instances, even if those correspond to different reactive actors.*

5.2 Multi-Instancing

Leader Oracle. POA is designed to run in multiple instances, and to ensure liveness of the system, it is essential to have honest leaders. We assume all validators have access to a common *leaderOracle* function, which given an instance number outputs a leader for that instance. We require this function to output an honest leader infinitely often. This can be achieved by a random choice (assuming a common random source) or by a round-robin approach.

Switching instances. A validator who decides in a POA instance at time t, broadcasts their decision along with the proof (in practice it can be a single aggregate signature). By time $\max(t + \Delta, GST + \Delta)$, every honest validator will receive a proof. Upon receiving such a valid proof, validators rebroadcast the proof and decide. This mechanism allows for the following property.

▶ **Property 10** (Common Termination). *If an honest validator decides in the k-th POA instance at time t, then every honest validator decides in this instance by at most* $\max(t+\Delta, GST+\Delta)$.

The following property guarantees the progress of each validator.

▶ **Property 11** (Multi-Termination). *For all honest validators V and reactive actors X, $V.X$ eventually decides in the k-th instance of POA for X.*

5.3 Algorithm

At the start of the protocol, $L.X$ forms a block consisting of all $tx \in L.X.pool$ and broadcasts it. Upon receiving a block B an honest validator $V.X$ broadcasts its vote plus its own block (called a *fallback block*) consisting of all transactions $tx \in V.X.pool$. The vote is for B in case (i) for every UA-RA transaction $tx \in B$, tx is correctly signed, (ii) $V.A$ manages to FP-lock tx, ensuring there is no conflicting transaction and tx can be executed and (iii) V emitted every RA-RA transaction from B. Otherwise, $V.X$ broadcasts a negative vote.

If at any time a validator $V.X$ receives $n - p$ votes for some block B, $V.X$ decides B, we call it a fast-path decision. Upon fast-path deciding, a validator broadcasts a proof that B can be safely decided (in practice, this can be an aggregated threshold signature [47] of $n - p$ votes). In case a validator does not receive $n - p$ votes in 3Δ time or those votes are for different blocks, they wait until they have $n - f$ votes and propose in the slow path.

The slow path consists of two steps: Quorum Consensus (described in detail in 6.1), followed by the parallel Transaction Agreements (described in 6.2) for every UA-RA transaction decided in Quorum Consensus. A validator proposes to Quorum Consensus according to the following cases.

Algorithm 1 Parallel Optimistic Agreement **on V.X** (High Level)

Uses: Outer Links, Inner Links, Quorum Consensus, Transaction Agreement

function INITIATE(k, pool) ▷ For Leader
    $proposalBlock \leftarrow pool$
    Broadcast via Outer Links $proposalBlock$

upon once $time \leq 3\Delta$ **and** received proposal B **do**
    for all tx : UA-RA transaction $\in B$ **do** ▷ In parallel
        $A \leftarrow tx.sender$
        Request $V.A$ via Inner Links to FP-lock tx

upon once all $V.A$ responded to an FP-lock request **do**
    if all FP-locks are successful **then** $Vote \leftarrow B$
    else $Vote \leftarrow \perp$
    $fallBackBlock \leftarrow pool$
    Broadcast via Outer Links $\langle Vote, fallBackBlock \rangle$

upon once exists a block B with at least $n - p$ votes **do**
    Decide B in POA

upon once $Time > 3\Delta$ **and** exists block B' with at least $n - p - 2f$ votes **do**
    for all tx : UA-RA transaction $\in B'$ **do** ▷ In parallel
        $A \leftarrow tx.sender$
        Request $V.A$ via Inner Links to SP-lock tx

upon once all $V.A$ responded to an SP-lock request **and** all SP-locks succeeded **do**
    Propose B' to Quorum Consensus

upon once $Time > 3\Delta$ **and** (received $n - f$ votes) **and** (there is no block with $n - p - 2f$ votes **or** not all SP-Locks succeeded) **do**
    $B'' \leftarrow$ all transactions present in at least $n - 2f$ fallback blocks
    Attempt to (analogously) SP-lock all UA-RA transactions in B''
    $B'' \leftarrow B'' \setminus \{$transactions that failed to SP-lock$\}$
    Propose B'' to Quorum Consensus

upon event $\langle$decide in Quorum Consensus $\mid B_{qc}\rangle$ **do**
    for all tx : UA-RA transaction $\in B_{qc}$ **do** ▷ In parallel
        $A \leftarrow tx.sender$
        Request $V.A$ via Inner Links to propose tx to Transaction Agreement

upon once all $V.A$ responded to an Transaction Agreement request **do**
    $Fails \leftarrow$ all UA-RA transactions from B_{qc} which $V.A$ did not decide in Transaction Agreement
    $B_{POA} \leftarrow B_{qc} \setminus Fails$
    Decide B_{POA} in POA

Algorithm 2 Parallel Optimistic Agreement **on V.A** (High Level)

1: **upon event** ⟨FP-lock request | tx, X⟩ **do**
2: **if** a conflicting transaction is FP-locked **then** respond *fail* to $V.X$
3: **if** execution preconditions for tx are not met **then** respond *fail* to $V.X$
4: Respond *success* to $V.X$

5: **upon event** ⟨SP-lock request | tx, X⟩ **do**
6: **if** a conflicting transaction is SP-locked **then** respond *fail* to $V.X$
7: Respond *success* to $V.X$

8: **upon event** ⟨Transaction Agreement request | tx, X⟩ **do**
9: Try to SP-Lock tx
10: Propose an SP-Locked transaction with sequence number $tx.sn$ to Transaction Agreement instance $tx.sn$ of A

11: **upon event** ⟨decide in Transaction Agreement | tx', sn⟩ **do**
12: **if** $tx' = tx$ **then** respond "Transaction Agreement sn Success" to $V.X$
13: **else** respond "Transaction Agreement sn Failure" to $V.X$

(i) There was some block B for which $V.X$ received at least $n - p - 2f$ votes. In this case, it is possible that some other validator received $n - p$ votes for B, so $V.X$ attempts to SP-lock all UA-RA transactions in B, and if it succeeds, proposes B to Quorum Consensus.

(ii) If either there was no block for which $V.X$ saw $n - p - 2f$ votes or $V.X$ was not able to SP-lock some transaction in the block, $V.X$ forms a block to propose to Quorum Consensus based on the fallback blocks received.

In particular, a transaction tx is included in the Quorum Consensus proposal of $V.X$ if and only if $V.X$ saw tx in at least $n - 2f$ fallback blocks and (in the UA-RA case) was able to SP-lock tx.

After deciding a block B in the Quorum Consensus, for each UA-RA transaction $tx \in B$ from user A, $V.X$ sends tx to $V.A$, who proposes SP-Locks and proposes it to A's Transaction Agreement instance number $tx.sn$. If a conflicting transaction tx' was SP-Locked before, then tx' is proposed.

Upon deciding in the Transaction Agreement instance number $tx.sn$, $V.A$ notifies $V.X$ whether tx was decided or not. After receiving all such notifications, $V.X$ decides on the block B_{POA}, which is formed from B but omitting those UA-RA transactions which were not decided in their corresponding Transaction Agreement instances.

Intuition. The reason to broadcast a fallback block is for verifiers to know the pools of each other, which would allow for a "good" proposal to Quorum Consensus in case the fast path fails. More precisely, knowing each other's pools, verifiers will propose "popular" transactions to Quorum Consensus, making them committed in the slow path and thus ensuring liveness.

The 3Δ threshold consists of Δ for the leader's broadcast, Δ for verifiers' votes, and Δ for the possible shift in times at which the leader and verifier initiate the primitive.

One needs a Transaction Agreement after Quorum Consensus to preclude committing conflicting UA-RA transactions on different reactive actors. Note that if the fast path

succeeds, it ensures that no conflicting transaction can be committed; hence, we only need the Transaction Agreement in case the fast path fails.

We provide a high-level pseudocode for POA for a reactive actor X in Algorithms 1 and 2. For a precise description, please see Appendix D.

6 Slow Path

In this section, we describe the two components of the slow path of POA and POB: Quorum Consensus and Transaction Agreement.

6.1 Quorum Consensus

- `function` PROPOSE(k, B): start the k-th instance with proposal B
- `callback` DECIDE(k, B): decide a block B in k-th instance

A Quorum Consensus is a partial-synchrony consensus algorithm, meaning it exposes an interface of proposing and deciding a block, and it satisfies Agreement, Termination, and Quorum Validity:

▶ **Property 12** (Quorum Validity). *(I) If a transaction tx is such that tx is in the proposal of every honest validator, then tx is included in the decided block.*

(II) If some transaction tx is included in the decided block, then tx is in the proposal of at least $n - 3f$ honest validators.

▷ Claim 13. There exists an algorithm that solves consensus with Quorum Validity.

Proof. The proof applies Theorem 5 and Definition 2 of [21] to Quorum Validity. Throughout the proof we use terms and notation from [21].

Consider some input configuration $c \in \mathcal{I}_{n-f}$. We claim that a block B consisting of transactions that are present in at least $n-2f$ proposals in $\mathcal{I}_{n-f}$ belongs to $\bigcap_{c' \in sim(c)} val(c')$. Consider an input configuration c' with at least $n - f$ elements (if there are less than $n - f$ elements in c', then any block is admissible). We deduce that $|\pi(c) \cap \pi(c')| \geq n-2f$, therefore, for each $tx \in B$, tx is present in at least $n - 3f$ proposals of c', therefore, can be included in the block decided for c'. Conversely, if some transaction is present in every proposal of c', then it is present in at least $n - 2f$ proposal of c and hence included in B. Thus, $B \in val(c')$. ◀

In Appendix B we discuss how modern high-throughput BFT protocols could be adapted to instantitate practical high-throughput Quorum Consensus.

6.2 Transaction Agreement

Transaction Agreement is a simple consensus primitive used to provide agreement on which transaction tx should be committed at sequence number sn for a given user A. Though it is possible to agree on this proactively, i.e., prior to submitting a transaction to POA, that would imply an additional latency for a UA-RA transaction under optimistic conditions, hence, MANGROVE does it *retroactively* instead, i.e., after the decision in the Quorum Consensus has been made.

- `function` PROPOSE(tx): propose transaction tx
- `callback` DECIDE(tx): decide a transaction tx

The Transaction Agreement primitive should satisfy the specification of a partially synchronous multi-valued Strong Byzantine Agreement, namely:

- Every process can propose and decide a (not necessarily binary) value.
- *Agreement.* If two honest processes decide v and v' respectively, then $v = v'$.
- *Quorum Validity.* If all honest processes propose the same value, only that value can be decided. Furthermore, if the value is decided, it was proposed by at least $n - 3f$ honest validators.
- *Termination.* Given a partially synchronous network, every correct process eventually decides.

This agreement primitive can be implemented as a special case of Quorum Consensus with blocks of size one.

7 Transaction Processing

7.1 Transaction Execution Properties

First, we informally present several properties of transaction processing in Mangrove. These are grouped into *correctness* properties, which are common among many distributed systems, *Fast Execution* properties, demonstrating the ability to process transactions with low latency, and *Parallelization* properties, showing the ability to make progress independently at different actors. Full formal definitions as well as proofs of all properties are given in Appendix F.

Correctness Properties. Reactive actors adhere to the Agreement, Validity, Total Order, and Integrity properties of Total Order Broadcast [25], with respect to emission and execution of transactions. User actors instead follow the Agreement, Validity, and Integrity properties of Byzantine Reliable Broadcast [17].

Fast Execution Properties. Mangrove is designed so that under optimistic conditions, transactions are executed with low latency. Table 1 summarizes conditions needed for a transaction tx of a given type to be executed fast. Those conditions are: *Honest author* (in case of user transactions) — the user who issued a transaction is honest, *synchrony* — the system is after GST, *honest leader* (in case of UA-RA or RA-RA with recipient X) — there exists an honest validator L who will start a POA instance for X as a leader as soon as it has tx in its pool, *good pool* — for every UA-RA transaction $tx' \in L.X.pool$ a user who emitted tx' is honest. The *latency* column shows how many communication steps are needed before every honest validator executes tx. The *resilience* column shows how many misbehaving validators the system can tolerate.

Transaction Type	Honest Author	Synchrony	Honest Leader	Good Pool	Latency	Resilience
UA	✓	✓			2δ	$\leq p$
RA-RA		✓	✓	✓	2δ	$\leq p$
UA-RA	✓	✓	✓	✓	$2\delta^*$ / $3\delta^\dagger$	$\leq p$

Table 1 Conditions and execution latency for the fast-path of different transaction types. Latency across validators is denoted δ, while latency within a validator is ignored. * When emitted by the leader. † For other validators.

There are instances where the *good pool* condition is not met due to user misbehavior, preventing fast transaction execution. Specifically, this occurs only when honest validators see conflicting transactions (Line 14, Algorithm 6). In such cases, an honest validator will have evidence of the user's misbehavior (namely, two signed conflicting transactions) and may initiate punishment (e.g. destroy the gas object). We emphasize that users can only affect the liveness of the fast path and in this case, all honest transactions are still committed in the slow path (that is, Validity I of POA holds no matter the user's misbehavior).

Parallelization Property. Classical blockchain systems address the double-spending problem [42] by totally ordering transactions. While effective, this introduces significant redundancy because many transactions are non-conflicting, meaning they can be executed in any order without affecting the outcome and therefore do not require total ordering.

Mangrove achieves optimal parallelization by ordering only the transactions that require it. It avoids ordering UA transactions entirely since these only create objects at user actors, and such operations are commutative. For reactive actor transactions, Mangrove maintains a partial order by keeping a separate ordered chain for each reactive actor. For example, given two reactive actors X and Y and a set of transactions $\{tx_1^X, \ldots, tx_k^X, tx_1^Y, \ldots, tx_l^Y\}$ (where tx_i^A is a transaction with receiver A), Mangrove maintains two ordered sets $\{tx_1^X, \ldots, tx_k^X\}$ and $\{tx_1^Y, \ldots, tx_l^Y\}$ but does not impose an order between these sets.

For a system with a transaction set T and for a reactive actor X, denote $T_X \subseteq T$ the subset of transactions with X as a receiver. The longest ordered chain in Mangrove is then $\max_{X \in RAs} |T_X|$, whereas in classical systems like Bitcoin and Ethereum, the chain length is $|T|$. This length, $\max_{X \in RAs} |T_X|$, is optimal unless additional assumptions are made about reactive actor behavior.

The creation of multiple short chains is beneficial in two ways. First, each chain can be maintained by a separate machine (running V.X) at each validator. This allows throughput to be scaled horizontally. Secondly, the throughput of an application (corresponding to an RA actor) depends solely on the length of its associated chain, and doesn't degrade when other applications experience high load.

7.2 User Transactions

Each user keeps a sequence number of the last issued transaction for each user actor it controls. When issuing a new transaction tx from a user actor A, a user must first check that A has all owned objects consumed by tx. If a user fails to do so, tx may never be executed, precluding all consecutive transactions from A. Note, though, that this only halts A and not the rest of the system. After a user checks owned objects for tx, it assigns a new sequence number to it, signs it, and broadcasts it to all $V.A$-s using Parallel Optimistic Broadcast. In Parallel Optimistic Broadcast, a user sends tx to all $V.A$-s, those attempt to FP-lock it, and if the FP-lock is successful, broadcast a vote for tx. Once a validator obtains $n - p$ votes tx, it fast-path decides tx, broadcasts a proof and stops sending messages. When unable to decide in the fast-path, validators invoke Transaction Agreement to ensure safety and liveness in case of asynchrony and validators' misbehavior. For pseudocode, see Appendix C.

For a UA transaction tx from A with a sequence number sn and consumed objects $O_1, \ldots, O_k$, it is *executed* if (i) tx is decided in Parallel Optimistic Broadcast, (ii) a transaction from A with a sequence number $sn - 1$ is executed and (iii) A owns $O_1, \ldots, O_k$. See the pseudocode for UA transactions in Algorithm 3.

For a UA-RA transaction tx from A to X with a sequence number sn and consumed

Algorithm 3 UA Transaction Processing

Uses: Parallel Optimistic Broadcast *pob*
upon event ⟨pob[A, k].Decide | $sender, tx$⟩ **do** ▷ On $V.A$
  if $k = tx.sn$ **and** VerifySig(A, tx) **then**
    $pending \leftarrow pending \cup \{tx\}$
upon exists $tx \in pending$: tx is UA **and** $executed[tx.sn - 1]$ **and** $tx.consumedObjects \subseteq ownedObjects$ **do** ▷ On $V.A$
  $pending \leftarrow pending \setminus \{tx\}$
  effects $\leftarrow$ VM.Execute$(tx.Code)$
  $ownedObjects \leftarrow ownedObjects \setminus tx.consumedObjects \cup$ effects.$createdObjects$
  $executed[tx.sn] \leftarrow true$

objects $O_1, \ldots, O_k$, $V.A$ *sends* it to $V.X$ if tx is correctly signed by A, $V.A$ have not seen any other transactions with sequence number sn, and (ii) and (iii) hold. tx is *executed* as soon as it is decided in the POA of X. Note that execution crucially does not rely on (ii) and (iii) to hold. We would like to highlight here that UA-RA transactions *do not use* an agreement mechanism on the sending user actor. See the pseudocode for UA-RA transactions in Algorithm 4.

Algorithm 4 UA-RA Transaction Processing

Uses: Outer Links *ol*, Inner Links *il*, POA *poa*
upon event ⟨Emit UA-RA Transaction | $A, X, [O_1, \ldots, O_k], Code, Call$⟩ **do** ▷ On U
  if not $executed[A][sns[A] - 1]$ **or** $\{O_1, \ldots, O_k\} \not\subseteq ownedObjects[A]$ **then**
    return Error("Not possible to emit")
  $tx \leftarrow$ Sign$(\langle A, sns[A], X, [O_1, \ldots, O_k], Code, Call\rangle)$
  $sns[A] \leftarrow sns[A] + 1$
  for all $V \in \mathcal{V}$ **do trigger** ⟨ol.Send | V, tx⟩
upon event ⟨ol.Deliver | tx : UA-RA Transaction⟩ **do** ▷ On $V.A$
  if not VerifySign(A, tx) **or** $FPLocked[tx.sn] \neq \perp$ **then return**
  $FPLocked[tx.sn] \leftarrow tx$
  await $executed[tx.sn - 1]$ **and** $tx.consumedObjects \subseteq ownedObjects$
  trigger ⟨il.Send | $tx.receiver, tx$⟩
upon event ⟨il.Deliver | tx, A⟩ **do** ▷ On $V.X$
  if $tx \notin executed$ **then** $pool \leftarrow pool \cup \{tx\}$

7.3 Reactive Actor Transactions

When a reactive actor $V.X$ decides a block, it starts executing transactions of that block in a deterministic order. Some transactions might create outgoing transactions when executed. Upon creating an outgoing transaction tx that consumes owned objects $O_1, \ldots, O_k$, $V.X$ checks its owned objects. If it owns all required objects, $V.X$ sends tx to the receiver via inner links, and receiver adds tx to the pool. If some objects that transaction consumes are missing, then this transaction is immediately dropped and has no effect. For the pseudocode, please see Algorithm 7 in Appendix E.

8 Discussion and Outlook

In general, a transaction may result in a *cascade* of multiple subsequent transactions. Classical systems guarantee that such cascades appear to be executed atomically and in isolation. Moreover, they immediately execute the whole cascade of a transaction after agreeing on the initial transaction, whereas Mangrove (in the worst case) performs a separate agreement for each individual transaction in the cascade. Therefore, in the case of deep cascades, we expect classical solutions to complete the execution of the cascade faster. Empirical study of the workload types under which either approach performs better is subject to future research.

Finally, we acknowledge that the bit complexity and message complexity of POA are relatively high. However, the primary objective of this work is to demonstrate the feasibility of a system that supports smart contracts while enabling maximal parallelization. Optimizing these complexities is an important consideration, which we leave as future work.

References

1 Ittai Abraham, Guy Gueta, Dahlia Malkhi, and Jean-Philippe Martin. Revisiting fast practical byzantine fault tolerance: Thelma, velma, and zelma. *arXiv preprint arXiv:1801.10022*, 2018.

2 Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Good-case latency of byzantine broadcast: A complete categorization. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 331–341, 2021.

3 Ramesh Adhikari and Costas Busch. Lockless blockchain sharding with multiversion control. In *International Colloquium on Structural Information and Communication Complexity*, pages 112–131. Springer, 2023.

4 Ramesh Adhikari, Costas Busch, and Miroslav Popovic. Fast transaction scheduling in blockchain sharding. *arXiv preprint arXiv:2405.15015*, 2024.

5 Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. Chainspace: A sharded smart contracts platform. *arXiv preprint arXiv:1708.03778*, 2017.

6 Timothé Albouy, Emmanuelle Anceaume, Davide Frey, Mathieu Gestin, Arthur Rauch, Michel Raynal, and François Taïani. Asynchronous bft asset transfer: Quasi-anonymous, light, and consensus-free. *arXiv preprint arXiv:2405.18072*, 2024.

7 Orestis Alpos, Christian Cachin, Giorgia Azzurra Marson, and Luca Zanolini. On the synchronization power of token smart contracts. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 640–651. IEEE, 2021.

8 Orestis Alpos, Bernardo David, and Dionysis Zindros. Pod: An optimal-latency, censorship-free, and accountable generalized consensus layer. *arXiv preprint arXiv:2501.14931*, 2025.

9 Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. Sharper: Sharding permissioned blockchains over network clusters. In *Proceedings of the 2021 international conference on management of data*, pages 76–88, 2021.

10 Parwat Singh Anjana, Sweta Kumari, Sathya Peri, Sachin Rathor, and Archit Somani. An efficient framework for optimistic concurrent execution of smart contracts. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 83–92. IEEE, 2019.

11 Balaji Arun, Zekun Li, Florian Suri-Payer, Sourav Das, and Alexander Spiegelman. Shoal++: High throughput DAG BFT can be fast! *arXiv preprint arXiv:2405.20488*, 2024.

12 Kushal Babel, Andrey Chursin, George Danezis, Lefteris Kokoris-Kogias, and Alberto Sonnino. Mysticeti: Low-latency DAG consensus with fast commit path. *arXiv preprint arXiv:2310.14821*, 2023.

13 Mathieu Baudet, George Danezis, and Alberto Sonnino. Fastpay: High-performance byzantine fault tolerant settlement. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 163–177, 2020.

14 Rida Bazzi and Sara Tucci-Piergiovanni. The fractional spending problem: Executing payment transactions in parallel with less than f+ 1 validations. In *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing*, pages 295–305, 2024.

15 Sam Blackshear, Evan Cheng, David L Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Dario Russi Rain, Stephane Sezer, et al. Move: A language with programmable resources. *Libra Assoc*, page 1, 2019.

16 Sam Blackshear, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Xun Li, Mark Logan, Ashok Menon, Todd Nowacki, Alberto Sonnino, et al. Sui lutris: A blockchain combining broadcast and consensus. *arXiv preprint arXiv:2310.18042*, 2023.

17 Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

18 Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform, 2014. URL: https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf.

19 Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming.* Springer Science & Business Media, 2011.

20 Margarita Capretto, Martín Ceresa, Antonio Fernández Anta, Antonio Russo, and César Sánchez. Setchain: Improving blockchain scalability with byzantine distributed sets and barriers. In *2022 IEEE International Conference on Blockchain (Blockchain)*, pages 87–96. IEEE, 2022.

21 Pierre Civit, Seth Gilbert, Rachid Guerraoui, Jovan Komatovic, and Manuel Vidigueira. On the validity of consensus. In *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing*, pages 332–343, 2023.

22 Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, and Athanasios Xygkis. Online payments by merely broadcasting messages. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 26–38. IEEE, 2020.

23 George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and Tusk: a DAG-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 34–50, 2022.

24 Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 international conference on management of data*, pages 123–140, 2019.

25 Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys (CSUR)*, 36(4):372–421, 2004.

26 Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding concurrency to smart contracts. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 303–312, 2017.

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

28 Aptos Foundation. Aptos blockchain, 2024. Accessed: 2024-10-11. URL: https://aptosfoundation.org.

29 Sui Foundation. Sui blockchain, 2024. Accessed: 2024-10-11. URL: https://sui.io.

30 Davide Frey, Lucie Guillou, Michel Raynal, and François Taïani. Process-commutative distributed objects: From cryptocurrencies to byzantine-fault-tolerant crdts. *Theoretical Computer Science*, 1017:114794, 2024.

31 Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. Block-STM: Scaling blockchain execution by turning ordering curse to a performance blessing. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 232–244, 2023.

32 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovič, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 307–316, 2019.

33 Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: A scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 568–580. IEEE, 2019.

34 Saurabh Gupta. *A non-consensus based decentralized financial transaction processing model with support for efficient auditing*. Arizona State University, 2016.

35 Martin Kleppmann. Making crdts byzantine fault tolerant. In *Proceedings of the 9th Workshop on Principles and Practice of Consistency for Distributed Data*, pages 8–15, 2022.

36 Quentin Kniep, Lefteris Kokoris-Kogias, Alberto Sonnino, Igor Zablotchi, and Nuda Zhang. Pilotfish: Distributed transaction execution for lazy blockchains. *arXiv preprint arXiv:2401.16292*, 2024.

37 Lefteris Kokoris-Kogias, Alberto Sonnino, and George Danezis. Cuttlefish: Expressive fast path blockchains with fastunlock, 2023. URL: https://arxiv.org/abs/2309.12715, `arXiv:2309.12715`.

38 Petr Kuznetsov, Andrei Tonkikh, and Yan X Zhang. Revisiting optimal resilience of fast byzantine consensus. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 343–353, 2021.

39 Leslie Laport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

40 J-P Martin and Lorenzo Alvisi. Fast byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, 2006.

41 Max Mathys, Roland Schmid, Jakub Sliwinski, and Roger Wattenhofer. A Limitlessly Scalable Transaction System. In *6th International Workshop on Cryptocurrencies and Blockchain Technology (CBT), Copenhagen, Denmark*, September 2022.

42 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008. Accessed: 2025-01-20.

43 Ray Neiheiser, Arman Babaei, Giannis Alexopoulos, Marios Kogias, and Eleftherios Kokoris Kogias. Pythia: Supercharging parallel smart contract execution to guide stragglers and full nodes to safety. *Workshop on Scalability & Interoperability of Blockchains (SIB)*, 2024.

44 Kaihua Qin, Liyi Zhou, Benjamin Livshits, and Arthur Gervais. Attacking the defi ecosystem with flash loans for fun and profit. In *International conference on financial cryptography and data security*, pages 3–32. Springer, 2021.

45 Geoffrey Ramseyer and David Mazières. Groundhog: Linearly-scalable smart contracting via commutative transaction semantics. *arXiv preprint arXiv:2404.03201*, 2024.

46 saikatdas0790. Lament: A tale of constant struggle of what it's like trying to scale on icp, 2024. Accessed: 2024-10-11. URL: https://forum.dfinity.org/t/lament-a-tale-of-constant-struggle-of-what-its-like-trying-to-scale-on-icp/35829.

47 Victor Shoup. Practical threshold signatures. In *Advances in Cryptology—EUROCRYPT 2000: International Conference on the Theory and Application of Cryptographic Techniques Bruges, Belgium, May 14–18, 2000 Proceedings 19*, pages 207–220. Springer, 2000.

48 Jakub Sliwinski, Yann Vonlanthen, and Roger Wattenhofer. Consensus on demand. In *International Symposium on Stabilizing, Safety, and Security of Distributed Systems*, pages 299–313. Springer, 2022.

49 Yee Jiun Song and Robbert van Renesse. Bosco: One-step byzantine asynchronous consensus. In *International Symposium on Distributed Computing*, pages 438–450. Springer, 2008.

50 Alexander Spiegelman, Balaji Arun, Rati Gelashvili, and Zekun Li. Shoal: Improving DAG-BFT latency and robustness. *arXiv preprint arXiv:2306.03058*, 2023.

51 Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: DAG BFT protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2705–2718, 2022.

52 Srivatsan Sridhar, Alberto Sonnino, and Lefteris Kokoris-Kogias. Stingray: Fast concurrent transactions without consensus. *arXiv preprint arXiv:2501.06531*, 2025.

53 Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. Basil: Breaking up bft with acid (transactions). In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 1–17, 2021.

54 Andrei Tonkikh, Pavel Ponomarev, Petr Kuznetsov, and Yvonne-Anne Pignolet. Cryptoconcurrency: (almost) consensusless asset transfer with shared accounts. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1556–1570, 2023.

55 Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 931–948, 2018.

A Model Expressiveness

Most common applications can be easily adapted from something resembling Ethereum's smart contract model to our model with a gain in scalability and without a loss in expressiveness apart from the obvious loss of free atomic composability. For the most part, this is equivalent to how these applications would be designed for the Move VM [15], on the Sui [29] and Aptos [28] blockchains.

- **Token:** A token is primarily defined by a data type for owned objects that adheres to a specific interface. Additionally, a token might have an associated reactive actor responsible for minting new tokens or handling a reserve.
- **Decentralized Exchange:** Each liquidity pool of a decentralized exchange should be its own reactive actor. This allows asynchronous access to different liquidity pools. One popular heavily congested contract does not impede access to any of the other liquidity pools.
- **NFT Marketplace:** The marketplace would be a reactive actor owning all the (NFT) objects currently for sale. Buyers can interact by sending accepted tokens as payment and receiving ownership of the desired object. Sellers can interact by sending it new objects to put up for sale or delisting items, receiving back the object.

More generally, we argue the replicated actor model (Section 3) does not lose expressiveness. This can be shown by providing a general framework of translating applications from Ethereum's smart contract model to our model. However, atomic composability is not inherently guaranteed at the protocol level. If it is intended, it has to be specifically designed into applications and users could be charged additional fees at the application layer for the privilege.

Gas. To incentivize validators to perform computations, our system uses *gas objects*, a type of owned object. Each transaction needs to consume at least a gas object. As validators process the *Call* and *Code* fields, they are compensated through fees deducted from the provided gas object. The amount of gas required depends on the complexity of the transaction, ensuring fair compensation for resource-intensive tasks. Design and study of specific game-theoretic mechanisms in our system model is outside the scope of this work.

Atomic Composability. Full atomic composability could be trivially achieved by keeping all composable state in a single reactive actor. However, this is not realistic when considering an open, diverse and growing ecosystem of applications. More importantly, it does not make use of the scalability advantages of our model. We can instead give users the option to request composability across reactive actors on-demand.

Locking. To this end we define a *locking* pattern that applications may independently opt into. Locking is unique in that it allows idle blocking while waiting for other calls to return. A reactive actor that supports locking has associated state $lockHolder$, $lockCollateral$, $lockStartTime$, $lockPrice$, and exposes functions $lock()$, $unlock()$, $getLockHolder()$, and $getLockPrice()$. The locking fee is the product of the current price for locking and the time the lock is held. The lock is valid until the required fee has exceeded the posted collateral. The price for locking can be updated by the reactive actor and may depend on the application as well as current usage statistics. For example, a decentralized exchange may count accesses or trading volume to a given liquidity pool and set the fee proportional to its recent popularity.

Any $lockPrice$ larger than 0 prevents a permanent deadlock on the reactive actor. However, the application protocol designers need to ensure that at any time the price is fair, to prevent abuse of this feature. The price for holding a lock could even increase super-linearly in the time it is held. This would further discourage continuously preventing others from acquiring the lock.

This would require some support on the protocol side as well. At least, each transaction should receive a timestamp agreed upon by the validators. This would serve as the basis to determine how long each lock is held. Also, execution of other transactions at that reactive actor should be delayed until after the lock is released. Maybe with the exception of a call checking whether a lock is currently being held.

Hard Example: Flash Loan. Some particularities of the strong atomic transaction model are not easily translatable. Locking is necessary but not sufficient to enable full atomic composability. For example, there is no way to implement a flash loan using just the above locking interface.

A flash loan is a loan where there is no risk of default because the loan is only valid within an atomic and isolated transaction. In this case, the lender needs to be sure that the entire transaction may only commit if the loan is paid back. If not, it should be reverted and the loan paid back.

Fully Atomic Transaction Cascades. Locking can be extended to provide atomic and isolated execution of entire transaction cascades. To be able to revert transactions, an atomic transaction execution context is needed. Most importantly, objects stay within the execution context until the transaction commits. Otherwise, other transactions might be able to see partial effects of the cascade before it commits. This would also make reversion on abort impossible without affecting other transactions.

Within an atomic transaction context all additional calls that are made are considered to be atomic and part of the same execution context. These calls need to hold a lock of the recipient reactive actor. The context is passed along the entire transaction cascade. Any transaction within an atomic transaction context may only spawn new transactions within the same context. Locks can only be freed at the end of the transaction (i.e. once it commits or aborts). Conversely, if any of the locks run out of collateral, the transaction

aborts. If a particular transaction aborts, the entire transaction cascade aborts. Importantly, all consumed objects are returned to the sender.

Using this extension of our model even flash loans can be realized. The only additional thing that is required for safety is that the `lend()` method of the flash loan reactive actor can indicate that it needs to be called from an atomic execution context and may revert.

B High-Throughput Quorum Consensus

A reactive actor that is at the core of a popular application may experience high contention over a long time period. Continuously trying to get transactions accepted on the fast-path in this case inhibits both throughput and latency. On the other hand, continuously running a chained high-throughput consensus protocol at each reactive actor, where most instances only produce empty blocks, is a significant and unnecessary burden. To alleviate this, reactive actors should be able to individually toggle between single-shot POA and a high-throughput consensus mechanism. This could happen automatically based on certain heuristics (configured either at the system level or by each RA). Switching from uncontested to contested mode can be done by deciding a special message in the same manner as any transaction. Switching from contested back to uncontested mode can be done via the high-throughput consensus' epoch closing mechanism.

Existing high-throughput consensus protocols, like Narwhal & Tusk [23], Bullshark [51], Shoal/Shoal++ [50, 11] and Mysticeti [12] can be extended to achieve part (II) of Quorum Validity, which is what differentiates it from regular Validity. For this, a simple consensus rule can be added. This rule restricts whether an ordered transaction is actually delivered. In addition to the voting on blocks necessary for ordering, validators directly vote on transactions. Honest validators vote for a transaction if and only if there are no conflicting transactions in their pool. Only transactions reaching a threshold of direct votes are delivered for execution. If this threshold ensures a quorum intersection of at least $f+1$, no conflicting transactions can be committed. This is the same rule that is added in Mysticeti-FPC, to make the general consensus compatible with the reliable broadcast fast-path allowed for owned-object-only transactions.

C Parallel Optimistic Broadcast

The Parallel Optimistic Broadcast (POB) primitive has an interface consisting of:

- `function` Broadcast(sn, tx): broadcast sn-th transaction
- `callback` Decide(sn, tx): decide transaction tx

To issue a UA transaction tx, user A broadcasts it among all $V.A$-s. Subsequently, $V.A$-s FP-lock tx and vote for it, and if some $V.A$ obtains $n-p$ votes, they can decide tx and stop sending messages. After receiving $n-p-2f$ votes for tx, $V.A$ SP-locks tx and proposes it to a Transaction Agreement instance number $tx.sn$ of A.

Parallel Optimistic Broadcast adheres to the following properties.

▶ **Property 14** (Agreement). *If two honest validators decide tx_1 and tx_2 in the same instance of Parallel Optimistic Broadcast, then $tx_1 = tx_2$.*

Proof. If both validators decide in the Transaction Agreement, the Agreement follows from the Agreement of Transaction Agreement.

If both validators decide in the fast path (here, we also call a decision done after receiving a proof on Line 25 a fast-path decision), that means that each of them obtained $n-p$ votes

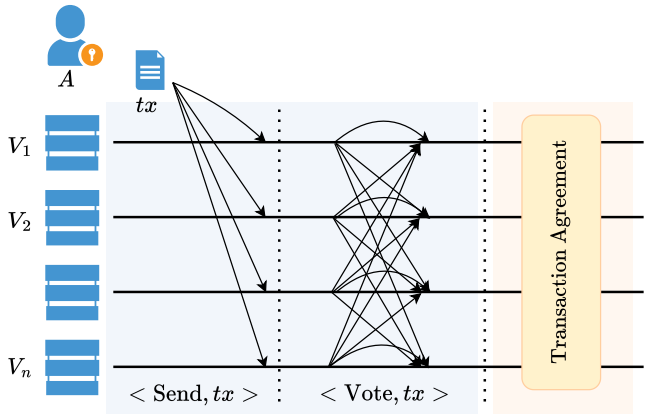

Figure 4 POB scheme. In the fast path (blue rectangle), a user A broadcasts their transaction tx. Then, validators cast their vote. In case the fast path fails, validators participate in the slow path (orange box), which consists of one instance of Transaction Agreement.

for tx (Line 19), meaning there is at least one common honest vote, and hence $tx_1 = tx_2$ since no honest validator issues conflicting votes (Line 6).

If tx_1 was decided in the fast path, it means it got $n-p$ votes, so no conflicting transaction can obtain $n-p-2f$ votes (the intersection is $(n-p)+(n-p-2f)-n = n-2p-2f \geq f+1$), hence no honest verifier will propose it to Transaction Agreement, hence, by the Validity property of the Transaction Agreement, tx_2 can't be decided. ◀

▶ **Property 15** (Integrity). *A transaction tx is decided at most once in the given instance and only if it was broadcast by the user.*

Proof. The only once part is ensured through checks (Lines 14 and 32), and if the user didn't emit the transaction, it will receive at most f votes, which is not sufficient neither for the fast path ($n-p > f$), nor to propose it to the Transaction Agreement ($n-p-2f > f$). ◀

▶ **Property 16** (Validity). *If an honest user broadcasts a transaction tx, it is eventually decided.*

Proof. Since an honest user does not issue conflicting transactions, all honest verifiers will eventually FP-lock tx (Line 6) and broadcast their vote. Now, if at any time an honest verifier gets $n-p$ votes, it broadcasts the proof, hence all honest verifiers will eventually decide. If none of the honest verifiers receive $n-p$ votes, since the user is honest, they all will eventually get $n-f \geq n-p-2f$ votes for tx and hence will propose it to the Transaction Agreement. Therefore, by the Validity property of Transaction Agreement, tx will be eventually decided in it and hence in Parallel Optimistic Broadcast (Line 34). ◀

▶ **Property 17** (Fast Termination). *Given the system is after GST, a user issuing the transaction is honest and at most p validators misbehave, every validator decides and stops sending messages in 2δ time after a user broadcasts its transaction.*

Proof. Assume an honest user broadcasts a transaction tx at time t. Then, since the system is after GST, all honest verifiers will receive it, and, since an honest user can not issue conflicting transactions, will FP-lock it (Line 9) and broadcast their vote for it (Line 12). Therefore, by the time $t+2\delta$, every honest verifier will receive at least $n-p$ votes for tx and will fast-path decide it (Line 19). ◀

Algorithm 5 Parallel Optimistic Broadcast

Uses: Perfect Links ol, Transaction Agreement ta

upon event $\langle$broadcast $\mid tx\rangle$ **do** ▷ On User A
  for all $V \in \mathcal{V}$ **do**
    trigger $\langle ol.Send \mid V.A, tx\rangle$

upon event $\langle$ol.Deliver $\mid A, tx\rangle$ **do** ▷ On $V.A$
  if $FPLocked[tx.sn] \neq \perp$ **and** $FPLocked[tx.sn] \neq tx$ **then**
    $vote \leftarrow \mathsf{Sign}(\perp)$
  else
    $FPLocked[tx.sn] \leftarrow tx$
    $vote \leftarrow \mathsf{Sign}(\mathsf{Vote}(tx))$
  for all $V \in \mathcal{V}$ **do**
    trigger $\langle ol.Send \mid V.A, vote\rangle$

upon once received $n - p$ votes for tx **do** ▷ On $V.A$
  if $pobDecided$ **then**
    return
  $proof(tx) \leftarrow aggregate(n - p\ votes)$
  for all $V \in \mathcal{V}$ **do**
    trigger $\langle ol.Send \mid V.A, proof(tx)\rangle$
  trigger $Decide(tx)$
  $pobDecided \leftarrow True$

upon event $\langle$ol.Deliver $\mid V', proof(tx)\rangle$ **do** ▷ On $V.A$
  if $verify(proof(tx))$ **then**
    for all $V'' \in \mathcal{V}$ **do**
      trigger $\langle ol.Send \mid V''.A, proof(tx)\rangle$
    trigger $Decide(tx)$

upon once received $n - p - 2f$ votes for tx **do** ▷ On $V.A$
  if $SPLocked[tx.sn] \neq \perp$ **and** $SPLocked[tx.sn] \neq tx$ **then**
    return
  $SPLocked[tx.sn] \leftarrow tx$
  trigger $ta[tx.sn].Propose(tx)$

upon event $\langle$ta[tx.sn].Decide $\mid tx\rangle$ **do** ▷ On $V.A$
  if $pobDecided$ **then**
    return
  trigger $Decide(tx)$
  $pobDecided \leftarrow True$

Algorithm 6 Parallel Optimistic Agreement (Part 1)

Uses: Outer Links ol, Inner Links il, Quorum Consensus qc, Transaction Agreement ta
function INITIATE(k, pool) ▷ On $V.X$
  $poaInstance \leftarrow k$
  if $leaderOracle(k) = V$ **then**
    for all $V' \in \mathcal{V}$ **do** $ol.Send(V'.X, \langle k, B \coloneqq pool\rangle)$
  $Timer.Restart()$

upon event $\langle ol.Deliver \mid L, \langle k, B\rangle\rangle$ **do** ▷ On $V.X$
  if $leaderOracle(k) = L$ **then** $block[k] \leftarrow B$

upon once $poaInstance = k$ **and** $block[k] \neq \bot$ **and** $Timer \leq 3\Delta$ **and** $\forall tx$ RA-RA transaction $\in block[k] : tx \in pool$ **do** ▷ On $V.X$
  $repliesFPLock[k] \leftarrow 0$
  $successFPLock[k] \leftarrow true$
  for all tx : UA-RA transaction $\in block[k]$ **do** ▷ In parallel
    trigger $\langle il.Send \mid FPLock(tx), tx.sender\rangle$

upon event $\langle il.Deliver \mid FPLock(tx), X\rangle$ **do** ▷ On $V.A$
  if not $\mathsf{VerifySig}(tx.sender, tx)$ **then**
    trigger $\langle il.Send \mid \textbf{false}, X\rangle$; **return**
  if $FPLocked[tx.sn] = \bot$ **then** $FPLocked[tx.sn] \leftarrow tx$
  if $FPLocked[tx.sn] \neq tx$ **then**
    trigger $\langle il.Send \mid \textbf{false}, X\rangle$; **return**
  if within Δ time $executed[tx.sn - 1]$ **and** $tx.objects \subseteq ownedObjects$ **then**
    trigger $\langle il.Send \mid \textbf{true}, X\rangle$
  else
    trigger $\langle il.Send \mid \textbf{false}, X\rangle$

upon event $\langle il.Deliver \mid status, A\rangle$ **do** ▷ On $V.X$
  $repliesFPLock[k] \leftarrow repliesFPLock[k] + 1$
  $successFPLock[k] \leftarrow successFPLock[k] \wedge status$

upon once $repliesFPLock[k] == |\{tx$: UA-RA transaction $\in block[k]\}|$ **do** ▷ On $V.X$
  $Vote \leftarrow block[k]$
  if not $successFPLock[k]$ **then** $Vote \leftarrow \bot$
  $B_{\mathsf{fb}} \leftarrow pool$
  for all $V' \in \mathcal{V}$ **do**
    trigger $\langle ol.Send \mid [k, \mathsf{Vote}, B_{fb}], V'.X\rangle$

D Parallel Optimistic Agreement (Extended)

This section provides the full pseudocode and proofs showing that Algorithm 6, as presented in Section 5.3, achieves the properties laid out in Section 5.1.

▶ **Lemma 18.** *Given $tx \in B$ was fast-path decided, no honest validator can SP-Lock a*

Algorithm 6 Parallel Optimistic Agreement (Part 2)

33: **upon event** $\langle ol.Deliver \mid U, \langle k, \mathsf{Vote}, B_{\mathsf{fb}} \rangle \rangle$ **do** ▷ On $V.X$
34: $votes[k] \leftarrow votes[k] \cup \{\mathsf{Vote}\}$
35: **if** $\mathsf{Vote} \neq \bot \wedge |\{v \in votes[k] \mid v = \mathsf{Vote}\}| = n - p$ **then**
36: $proof(B) \leftarrow aggregate(n - p\ votes)$
37: **for all** $V'' \in \mathcal{V}$ **do**
38: **trigger** $\langle ol.Send \mid V''.X, proof(B) \rangle$
39: **trigger** $\langle poa.Decide \mid k, B \rangle$
40: $fallbackBlocks[k] \leftarrow fallbackBlocks[k] \cup \{B_{\mathsf{fb}}\}$

41: **upon event** $\langle$ol.Deliver $\mid V', proof(tx) \rangle$ **do** ▷ On $V.X$
42: **if** $verify(proof(tx))$ **then**
43: **for all** $V'' \in \mathcal{V}$ **do**
44: **trigger** $\langle ol.Send \mid V''.A, proof(tx) \rangle$
45: **trigger** $\langle poa.Decide \mid k, B \rangle$

46: **upon once** $poaInstance = k$ **and** $Timer > 3\Delta$ **and** $|votes[k]| = n - f$ **do** ▷ $V.X$
47: **if** $\nexists B : |\{v \in votes[k] \mid v = B\}| \geq n - p - 2f$ **then**
48: $needFallbackBlocks \leftarrow true$ **return**
49: $candidateB[k] \leftarrow B$ such that $|\{v \in votes[k] \mid v = B\}| \geq n - 3f$
50: $repliesSPLock[k] \leftarrow 0$
51: $successSPLock[k] \leftarrow true$
52: **for all** tx : UA-RA transaction $\in candidateB[k]$ **do** ▷ In parallel
53: **trigger** $\langle il.Send \mid SPLock(tx), tx.sender \rangle$

54: **upon event** $\langle il.Deliver \mid SPLock(tx), X \rangle$ **do** ▷ On $V.A$
55: **if** $SPLocked[tx.sn] = \bot$ **then** $SPLocked[tx.sn] \leftarrow tx$
56: $status \leftarrow SPLocked[tx.sn] == tx$
57: **trigger** $\langle il.Send \mid status, X \rangle$

58: **upon event** $\langle il.Deliver \mid status, A \rangle$ **do** ▷ On $V.X$
59: $repliesSPLock[k] \leftarrow repliesSPLock[k] + 1$
60: $successSPLock[k] \leftarrow successSPLock[k] \wedge status$

61: **upon once** $repliesSPLock[k] == |\{tx$: UA-RA transaction $\in candidateB[k]\}|$ **do** ▷ On $V.X$
62: **if** $successSPLock[k]$ **then**
63: **trigger** $\langle qc.Propose \mid B \rangle$; **return**
64: $needFallbackBlocks \leftarrow true$

conflicting transaction tx'.

Proof. A validator attempts to SP-lock a UA-RA transaction tx' in four cases. Either because it received $n - p - 2f$ votes for a block containing tx' (Line 53), because it received $n - p - 2f$ votes for tx' in Parallel Optimistic Broadcast (Line 29, Algorithm 5), because it received $n - 2f$ fallback blocks containing tx' (Line 70), or because it decided a block

Algorithm 6 Parallel Optimistic Agreement (Part 3)

65: **upon once** $needFallbackBlocks$ **do** ▷ $V.X$
66: $candidateB_2[k] \leftarrow \{tx \mid |\{B_{\mathsf{fb}} \in fallbackBlocks[k] \mid tx \in B_{\mathsf{fb}}\}| \geq n - 2f\}$
67: $SPLockedTxs[k] \leftarrow \emptyset$
68: $repliesSPLock_2[k] \leftarrow 0$
69: **for all** tx : UA-RA transaction $\in candidateB_2[k]$ **do**
70: **trigger** $\langle il.Send \mid SPLock_2(tx), tx.sender\rangle$

71: **upon event** $\langle il.Deliver \mid SPLock_2(tx), X\rangle$ **do** ▷ On $V.A$
72: **if** $SPLocked[tx.sn] = \perp$ **then** $SPLocked[tx.sn] \leftarrow tx$
73: $status \leftarrow SPLocked[tx.sn] == tx$
74: **trigger** $\langle il.Send \mid \langle status, tx\rangle, X\rangle$

75: **upon event** $\langle il.Deliver \mid \langle status, tx\rangle, A\rangle$ **do** ▷ On $V.X$
76: **if** $status$ **then**
77: $SPLockedTxs[k] \leftarrow SPLockedTxs[k] \cup \{tx\}$
78: $repliesSPLock_2[k] \leftarrow repliesSPLock + 1$

79: **upon once** $repliesSPLock_2[k] == |\{tx$: UA-RA transaction $\in candidateB_2[k]\}|$ **do** ▷ On $V.X$
80: $QCPropose \leftarrow SPLockedTxs[k] \cup \{tx \in candidateB_2 \mid tx$ is RA-RA transaction$\}$
81: **trigger** $\langle qc.Propose \mid QCPropose\rangle$

82: **upon event** $\langle qc.Decide \mid B_{qc}\rangle$ **do** ▷ On $V.X$
83: **for all** tx : UA-RA transaction $\in B$ **do**
84: $A \leftarrow tx.sender$
85: **trigger** $\langle il.Send \mid UAInitiate(tx), V.A\rangle$

86: **upon event** $\langle il.Deliver \mid UAInitiate(tx), V.X\rangle$ **do** ▷ On $V.A$
87: **if** $SPLocked[tx.sn] = \perp$ **then** $SPLocked[tx.sn] \leftarrow tx$
88: **trigger** $\langle ta[tx.sn].Propose \mid SPLocked[tx.sn]\rangle$

89: **upon event** $\langle ta[sn].Decide$ |tx'$\rangle$ **do** ▷ On $V.A$
90: **if** $tx' = tx$ **then trigger** $\langle il.Send \mid$ "UA success", $V.X\rangle$
91: **else trigger** $\langle il.Send \mid$ "UA fail", $V.X\rangle$

92: **upon once** received all Transaction Agreement responses **do** ▷ On $V.X$
93: $B_{POA} \leftarrow B_{qc} \setminus \{tx \mid tx$ failed in Transaction Agreement$\}$
94: **trigger** $\langle poa.Decide \mid poaInstance, B_{POA}\rangle$

containing tx' in Quorum Consensus (Line 87). We want to show that given some honest validator Fast-Path decided a block containing tx, none of the above can hold.

Let's start with $n - p - 2f$ votes. If a validator received $n - p - 2f$ votes for a block containing tx', it means that at least $n - p - 3f \geq p + 1$ honest validators FP-locked tx'

(Line 13). The same argument holds for tx' in Parallel Optimistic Broadcast (Line 9). But those wouldn't then vote for block B that contains tx, since an FP-lock attempt in Line 13 would fail. Hence, no validator can receive $n-p$ votes for B, thus no fast decision of B is possible. A contradiction.

Next, assume that a validator SP-locked tx' due to receiving $n-2f$ fallback Blocks with tx'. This implies that at least $n-3f$ honest validators broadcasted a fallback block with tx' and hence at least $n-3f \geq 2p+1$ honest validators have tx' in their pool (Line 30), meaning they have $FPLocked[tx.sn] = tx'$ (Line 10) and therefore can not FP-lock tx, and wouldn't vote for a block containing tx. Thus, a Fast-Path decision of a block containing tx is not possible, contradiction.

Finally, assume that a validator SP-locked tx' due to deciding a block containing tx' in Quorum Consensus. By the part (II) of the Quorum Validity property, that means that at least $n-3f \geq 2p+1$ honest validators proposed tx' to Quorum Consensus, and hence SP-Locked it for one of the first two reasons(Lines 55 or 72) which we've shown to be impossible given tx was fast-path decided. ◀

Agreement Property. If two validators decide blocks B and B' in the fast path, that means that each of them received at least $n-p$ votes (Line 35), hence there must be at least $(n-p)+(n-p)-n = 3f+1$ common votes, therefore at least $2f+1$ honest validators voted for both B and B' meaning $B = B'$ since an honest validator only votes once (Line 27).

Assume two honest validators V and V' decided blocks B and B' respectively in the slow path. By the agreement property of the Quorum Consensus, V and V' decided the same block B_{qc} in Quorum Consensus, hence they have the same set of UA-RA transactions to send to Transaction Agreements and by the Agreement Property of the Transaction Agreement, the same subset S of those was not decided. Thus, $B = B' = B_{qc} \setminus S$.

Therefore, what is left to show is that if an honest validator V decides B in the fast path and another honest validator decides B' in Quorum Consensus, then $B = B'$.

If V decides B in the Fast Path, it means that it received $n-p$ votes for B. Hence among every $n-f$ votes there will be at least $n-p-2f$ votes for B. We would like to show that every honest validator will propose B to Quorum Consensus, and hence B will be decided in Quorum Consensus.

To do so, we need to show that a condition in Line 62 will hold, namely that every UA-RA transaction $tx \in B$ will be successfully SP-locked. We show it by showing that no conflicting transaction tx' can be SP-locked.

So every honest validator will propose B to the Quorum Consensus. Denote with B_{qc} a block decided in Quorum Consensus. By the condition (I) of Quorum Validity, if tx is in the proposal of every honest validator, then tx is in the decided block, that is $tx \in B \rightarrow tx \in B_{qc}$. Denote with $S \subset B_{qc}$ a subset of UA-RA transactions in B_{qc} that were not decided in the corresponding Transaction Agreement. We will show that $tx \in B \rightarrow tx \notin S$. Indeed, assume $tx \in B$. Then, by Lemma 18, all honest validators will propose it (Line 88) to the Transaction Agreement, and by the Quorum Validity of the Transaction Agreement tx will be decided. This allows us to conclude that $B \subseteq B_{qc} \setminus S = B'$

Next, note that $B_{qc} \subseteq B$. Indeed, by condition (II) of Quorum Validity, if tx is in B_{qc}, then it was proposed by at least $n-3f \geq 1$ honest validators, and hence, as we've showed that every honest validator proposes B, tx must be in B. This gives us $B' = B_{qc} \setminus S \subseteq B_{qc} \subseteq B$ ◀

Termination Property. There will either be an honest validator that received $n-p$ votes for some block B, or there will be not. In case there will be, it will broadcast the proof, hence every honest validator will eventually receive it and will eventually decide and stop

sending messages. If none of the honest validators ever receive $n - p$ votes, we argue by the termination of a slow path:

By 3Δ time after the start of the instance (Line 46) an honest validator will propose to the Quorum Consensus (Line 63 or 81). They eventually decide in the Quorum Consensus by the Termination property of Quorum Consensus and will propose to multiple Transaction Agreements (Line 85). By the Termination property of the Transaction Agreement, all instances will terminate and a validator will decide in POA (Line 94). ◀

Fast Termination Property. Let's consider some honest validator V. Given the system is in the synchrony period, by the Common Termination property, $L.X$ will start the instance at most Δ time after $V.X$ started and hence $L.X$'s proposal B will reach $V.X$ at most 2Δ time after $V.X$ started. For every UA-RA transaction $tx \in B$, by assumption, a user who emitted tx is honest, and hence V will successfully FP-lock tx since there is no conflicting transaction and since V will be ready to execute tx by the time 2Δ by Lemma 34. Also, by Lemma 33, for every RA-RA transaction $tx \in B$ V will emit tx at most Δ time after L did. Hence $V.X$ will issue a vote for B.

Therefore, due to synchrony, and by the assumption that at most p validators misbehave, every honest validator will acquire $n - p$ votes for B within 3Δ time after its own start and will Fast-Path decide B. ◀

Validity Property. For simplicity, we only consider the validity of a block decided in the Slow Path, since by the Agreement Property, it's the same block as the one decided in the Fast Path.

Consider a block B decided in the Slow Path. Let's first prove condition (I) of the Validity Property: If a transaction tx is present in the pool of all honest validators at the start of the protocol and L is honest, then tx is included in B.

Consider such tx. An honest validator $V.X$ proposes to the Quorum Consensus either a block B_1 for which V received at least $n - p - 2f$ votes (Line 63) or a block B_2 consisting of transactions that were present in $n - 2f$ fallback blocks (Line 81). Since L is honest, B_1 is its proposal, and it contains tx. So does B_2 because among every $n - f$ fallback blocks there will be at least $n - 2f$ blocks from honest validators and each such block contains tx (by condition to form a Quorum Consensus proposal from fallback blocks at Line 66). Therefore, every honest validator's proposal to Quorum Consensus will contain tx, and by the (I) condition of a Quorum Validity tx will be in the decided block B_{qc} of Quorum Consensus. If tx is an RA transaction, it will trivially be in the decided block, so assume tx is a UA-RA transaction. After deciding B_{qc}, every honest validator will propose tx to the corresponding Transaction Agreement (Line 88), since no conflicting transaction can be SP-Locked since an honest user doesn't issue conflicting transactions. Therefore, tx will be decided in its Transaction Agreement instance by the Quorum Validity, and hence be included in B.

Now let's prove condition (II) of Validity, namely that if a transaction is included in the decided block, it was emitted.

Consider some $tx \in B$. By condition (II) of Quorum Validity, tx is in the proposal of at least $n - 3f \geq 1$ honest validators, and an honest validator proposes a UA-RA transaction tx to Quorum Consensus only if it SP-locked tx (lines 53, 63 or 70, 81), which is only possible if $n - 3f \geq 1$ honest validators have tx in their pool which implies (Line 11) that this transaction was emitted. And an honest validator proposes an RA transaction only if it was present in $n - 2f$ fallback blocks (Line 80), meaning at least $n - 3f \geq 1$ honest validators included it in their fallback blocks, meaning they have it in their pool (Line 30), meaning emitted it (Line 13, Algorithm 4). ◀

No Conflict Property. For the sake of contradiction, assume that there were POA instances POA_1 and POA_2 in which blocks B_1 and B_2 were decided, containing respectively tx_1 and tx_2, which are conflicting. Now, either one of those blocks were decided in the fast path or none were.

Consider the case where w.l.o.g. B_1 was decided in the fast path. This means that at least $n - p - f$ honest validators FP-locked tx_1, and hence B_2 can receive at most $2f + p < n - p$ votes, hence can not be fast-path decided. Moreover, given tx_1 was fast-path decided, by Lemma 18 no honest validator can SP-Lock tx_2, therefore, no honest validator proposes tx_2 to the Transaction Agreement instance number $tx_2.sn$, therefore, by the Quorum Validity of Transaction Agreement, tx_2 can not be decided in the Transaction Agreement, and hence in the slow path.

Finally, assume that both B_1 and B_2 were decided in the slow path. But this implies that tx_1 and tx_2 were both decided in the Transaction Agreement instance number $tx_1.sn$ for user $tx_1.sender$ (which are the same values as $tx_2.sn$ and $tx_2.sender$), which is not possible due to the agreement property of Transaction Agreement. ◀

E Pseudocode for Transaction Execution

Algorithm 7 Transaction Execution (UA-RA and RA-RA)

```
1:  upon event ⟨once poa.Decide | k, B⟩ do                                   ▷ On V.X
2:      pool ← pool \ B
3:      for all tx ∈ DeterministicOrder(B) do
4:          if tx ∈ executed then continue
5:          executed ← executed ∪ {tx}
6:          effects ← VM.Execute(tx.Code_pre, tx.consumedObjects)
7:          effects ← VM.Execute(tx.Call, effects)
8:          effects ← VM.Execute(tx.Code_post, effects)
9:          for all raratx ∈ effects.raratxs do
10:             if raratx.consumedObjects ⊈ ownedObjects then
11:                 continue
12:             ownedObjects ← ownedObjects \ raratx.consumedObjects
13:             trigger ⟨il.Send | raratx.receiver, raratx⟩
14:         if tx is UA-RA then
15:             trigger ⟨il.Send | tx.sender, Executed(tx, effects)⟩
16:     poa.Initiate(k + 1)

17: upon event ⟨il.Deliver | ⟨Executed(tx), effects⟩, V.X⟩ do                ▷ On V.A
18:     if executed[tx.sn] then return
19:     await executed[tx.sn − 1] and tx.consumedObjects ⊆ ownedObjects
20:     ownedObjects ← ownedObjects \ tx.consumedObjects ∪ effects.createdObjects
21:     executed[tx.sn] ← true
```

F System Analysis

F.1 Formal Correctness Properties

Properties for reactive actors follow the schema of Total Order Broadcast [25].

▶ **Property 19** (Reactive Actor Agreement). *For any honest validators V_1 and V_2, a reactive actor X and a transaction tx, if $V_1.X$ executes tx, then $V_2.X$ eventually executes tx.*

▶ **Property 20** (Reactive Actor Validity). *If an honest user emits a UA-RA transaction, it is eventually executed by all correct validators.*

If a reactive actor on any honest validator emits an RA-RA transaction, it is eventually executed by all correct validators.

▶ **Property 21** (Reactive Actor Total Order). *For any reactive actor X, honest validators V_1, V_2, and transactions tx_1, tx_2, if $V_1.X$ executes tx_1 before tx_2, then $V_2.X$ executes tx_1 before tx_2.*

▶ **Property 22** (Reactive Actor Integrity). *For any reactive actor X, honest validator V and transaction tx, $V.X$ executes tx at most once and only if tx was emitted.*

User actor properties follow the schema of Byzantine Reliable Broadcast [17].

▶ **Property 23** (User Actor Validity). *If a correct user emits a user actor transaction, this transaction is eventually executed by every honest validator.*

▶ **Property 24** (User Actor Integrity). *Every correct validator executes each user actor transaction at most once and only if it was emitted.*

▶ **Property 25** (User Actor Agreement). *For any user actor A, honest validators V_1, V_2 and user actor transactions tx_1, tx_2 both with sender A and the same sequence number, if $V_1.A$ executes tx_1 and $V_2.A$ executes tx_2, then $tx_1 = tx_2$.*

F.2 Formal Fast Execution Properties

▶ **Property 26** (Fast UA Transaction Execution). *Given an honest user A issues a UA transaction tx, the system is after GST, and at most p validators are faulty, tx is executed after 2 communication steps.*

▶ **Property 27** (Fast UA-RA Transaction Execution). *Given an honest user A issues a UA-RA transaction tx to a reactive actor X, the system is after GST, the leader L of the next POA instance of X is honest and starts the instance as soon as it receives a transaction from A, and at most p validators misbehave, then tx is executed after 3 communication steps.*

▶ **Property 28** (Fast RA-RA Transaction Execution). *Given a reactive actor issues an RA transaction tx to a reactive actor Y, the system is after GST, the leader L of the next POA instance of Y is honest and starts the instance as soon as it receives a transaction from X, and at most p validators misbehave, then tx is executed after 2 communication steps.*

F.3 Proofs for All System Properties

Reactive Actor Agreement. If $V_1.X$ executes tx, it means that it decided a block B containing tx. Assume $V_1.X$ decided B in the k-th instance of POA. By the Multi-Termination and Agreement properties of POA, $V_2.X$ will eventually decide in the k-th instance of POA for X and its decision will be B. Therefore, $V_2.X$ will also eventually execute tx. ◀

▶ **Corollary 29** (Reactive Actor Agreement). *For two honest validators V_1 and V_2, and a reactive actor X, if $V_1.X$ emits tx, then $V_2.X$ eventually emits tx.*

Proof. Let tx' be a transaction executing a *Call* of which made $V_1.X$ emit tx. By the Reactive Actor Agreement property, $V_2.X$ will eventually execute tx' having the same state and the same set of owned objects as $V_1.X$ had when executing tx'. Therefore, $V_2.X$ will also emit tx. ◀

▶ Remark 30. Definition 7 is independent of a validator since if a transaction was emitted by a reactive actor of one validator, by Corollary 29 of the Reactive Actor Agreement property, every honest validator will eventually emit it.

▶ **Definition 31.** *Consider an execution E and two honest validators V_1 and V_2. We define a* direct ancestor according to V_1 *relation for two transactions tx_1 and tx_2 executed by V_1 in E the following way: tx_1 is a direct ancestor of tx_2 if either (i) tx_1 and tx_2 are emitted by the same user actor and $tx_1.sn = tx_2.sn - 1$ or (ii) tx_2 consumes objects created by tx_1. Define an* ancestor *relation as a transitive closure of a direct ancestor relation.*

▶ **Lemma 32.** *Consider two honest validators V_1 and V_2 and execution E. If V_1 executes some transaction tx at time t in E and V_2 executes all ancestors of tx according to V_1 by at most* $\max(t + \Delta, GST + \Delta)$ *then V_2 executes tx by at most* $\max(t + \Delta, GST + \Delta)$.

Proof. In case tx is either UA-RA, RA-RA or RA, this follows from the Reactive Actor Agreement and Common Termination properties without a need for the ancestor premise. Therefore, we focus on the case of tx being a UA transaction. Denote $A := tx.sender$ and tx' a transaction with $sender = A$ and $tx'.sn = tx.sn - 1$. Note that if V_1 executed tx, it means it decided it in Parallel Optimistic Broadcast (Line 2, Algorithm 3), meaning V_2 will also eventually decide it in Parallel Optimistic Broadcast and will put it into *pending* (Line 4). And V_2 will be ready to execute tx (Line 5) by the time of at most $\max(t + \Delta, GST + \Delta)$ since by the state of the Lemma, by that time V_2 will execute all ancestors of tx. ◀

▶ **Lemma 33.** *If an honest validator executes a transaction tx at time t, then every honest validator executes tx at time at most* $\max(t + \Delta, GST + \Delta)$.

Proof. For UA-RA and RA-RA transactions, this follows from the Reactive Actor Agreement and Common Termination properties. We now give proof for UA transactions.

For the sake of contradiction, consider an execution E of the protocol, two honest validators V_1 and V_2, and a UA transaction tx such that in E V_1 executes tx at time t and V_2 does not execute tx by $\max(t + \Delta, GST + \Delta)$.

By Lemma 32, if V_1 executes tx but V_2 does not, it means that there is a transaction that is an ancestor of tx according to V_1 that is not executed by V_2. Repeat the proof process for that ancestor. Since there is a finite number of ancestors of each transaction and an ancestor relation is transitive, we end up with an ancestor of tx that is not executed by the time $\max(t + \Delta, GST + \Delta)$ but all its ancestors are, which contradicts Lemma 32. ◀

▶ **Lemma 34.** *Consider a user actor A controlled by honest user U and a transaction tx from A.*

If U emits tx at time t, then for every honest validator V, $V.A$ will have $executed[tx.sn-1]$ and $tx.objects \subseteq ownedObjects$ by $\max(t + \Delta, GST + \Delta)$.

Proof. Since U is honest and emits tx, we conclude that U saw effect of some transactions $tx_1, \ldots, tx_k$ that made $executed[A][tx.sn]$ and $tx.objects \subseteq objects[A]$ hold. It means that

$\forall i \in [k]$ U received at least $f+1$ votes for tx_i, meaning there is at least one honest process which executed tx_i, which by Lemma 33 implies that every honest validator will execute tx_i by $\max(t+\Delta, GST+\Delta)$. Therefore, for every honest validator $V.A$ $executed[tx.sn-1]$ and $tx.objects \subseteq ownedObjects$ will hold by $\max(t+\Delta, GST+\Delta)$. ◀

Reactive Actor Validity. First, consider an honest user U issues a UA-RA transaction tx to a reactive actor X. Since U is honest, checks in Line 9 of Algorithm 4 will pass and each honest validator will put tx into its *pending* set. Moreover, by Lemma 34, a condition in Line 11, Algorithm 4 will eventually hold for every honest $V.A$ and it will send tx to $V.X$ (Line 12). So, tx will eventually end up in the pool of every honest validator (Line 14), let's denote this time point with t. Now consider a point in time after t and after GST when an honest validator L becomes a leader of POA for X. If by this time L already executed tx, then by the Reactive Actor Agreement tx will be eventually executed by every honest validator. Otherwise, L will include tx in its proposal and by the (I) part of the Validity property of POA, tx will be decided and then executed (Lines 6 and 7).

Now, we give a prove for an RA-RA transaction from a reactive actor X to a reactive actor Y. Corollary 29 states that if for some honest validator V_1, $V_1.X$ emitted tx, then for every other honest validator V_2, $V_2.X$ will eventually emit tx. Therefore, tx will end up in the pool of $V.Y$ for every honest validator V, and by the same logic as with a UA-RA transaction, tx will be eventually executed by every honest validator. ◀

Reactive Actor Total Order. A reactive actor validator executes a transaction only if it decides a block in POA containing this transaction.

Let B_1 be the block containing tx_1 and k_1 be the number of POA instance in which B_1 is decided. Analogously we define B_2 and k_2 for tx_2.

Given $V_1.X$ executed tx_1 before tx_2 and since processes execute blocks sequentially, we conclude that $k_1 \leq k_2$. If now $k_1 < k_2$, then $V_2.X$ executes tx_1 before tx_2 because of the sequential execution of blocks. And if $k_1 = k_2$, then $B_1 = B_2$, and this block is executed in the same order by $V_1.X$ and $V_2.X$ due to the deterministic block execution. ◀

Reactive Actor Integrity. If tx is executed by $V.X$, it means that $V.X$ decided a block containing tx. $V.X$ can decide either on the Fast Path (Line 39, Algorithm 6) or on the slow path (Line 94, Algorithm 6).

If a block containing tx is decided on the fast path, it means it received at least $n-p-f \geq 1$ honest votes (Line 35), meaning it passed the checks of an honest validator. In case tx is a UA-RA transaction, this means that tx is correctly signed by a user (Line 15) and a validator successfully FP-locked tx (Line 13), meaning it has $executed[tx.sn-1] = true$ and $tx.objects \subseteq ownedObjects$ (Line 20), therefore tx is emitted. In case tx is an RA-RA transaction, it means that it is contained in the pool of an honest validator (Line 9), hence it was issued by a reactive actor of an honest validator, and hence it is emitted.

If tx is included in the block decided on the slow path, then by the (II) part of Quorum Validity, there are $n-3f \geq 1$ honest validators who proposed a block containing tx to the Quorum Consensus. An honest validator includes tx in its proposal either if it received $n-p-2f$ votes for the block containing tx (Line 63) or if it received $n-2f$ fallback blocks containing tx (Line 81). In case it received $n-p-2f$ votes, there were at least $n-p-3f \geq p+1$ honest votes for a block containing tx, which by the argument above implies that tx is emitted. In case it received $n-2f$ fallback blocks containing tx, it means that at least $n-3f \geq 1$ honest validators included tx into their fallback block, meaning tx was in their pool, meaning it was either sent by a reactive actor (Line 13, Algorithm 4) and thus emitted, or it was sent by a user actor (Line 12, Algorithm 4) and hence checked for a correct

user signature (Line 9) and for $executed[tx.sn - 1] = true$ and $tx.objects \subseteq ownedObjects$ (Line 11) and thus emitted.

Every transaction is executed at most once by a reactive actor since it maintains an *executed* set and skips a transaction if it was executed already (Line 4, Algorithm 4). ◀

User Actor Validity. Consider an emitted transaction tx issued by user A with $tx.sender = A$. Given that A is honest, by the Validity Property of Parallel Optimistic Broadcast, tx will eventually be decided in the Parallel Optimistic Broadcast and will end up in a *pending* set of every honest validator (Line 4, Algorithm 3). And by Lemma 34, for every honest $V.A$ $executed[tx.sn - 1]$ and $tx.objects \subseteq ownedObjects$ will eventually hold (Line 5) and thus $V.A$ will execute tx (Line 7). ◀

User Actor Integrity. A correct validator executes a transaction only if it previously put it into a *pending* set (Line 5, Algorithm 3). A correct validator puts a transaction into a *pending* set only if it delivers it in a Parallel Optimistic Broadcast instance (Line 4). By the Integrity property of the Parallel Optimistic Broadcast, at most one transaction with a given sequence number will be decided in the given instance, and hence, at most one transaction with a given sequence number will be executed.

Furthermore, a transaction is put in the *pending* set only if it is correctly signed and is executed only if the condition in Line 5 holds, meaning only if a transaction is emitted. ◀

User Actor Agreement. We consider all possible cases for tx_1 and tx_2 to be either UA or UA-RA transactions.

In case both tx_1 and tx_2 are UA transactions, we can conclude that $V_1.A$ and $V_2.A$ delivered them in the same Parallel Optimistic Broadcast instance, hence $tx_1 = tx_2$ by the Agreement property of Parallel Optimistic Broadcast.

If tx_1 and tx_2 are both UA-RA transactions, we conclude that they were both decided in the POA and hence, due to the No Conflict property of POA, $tx_1 = tx_2$.

Finally, it can not be that tx_1 is a UA transaction, and tx_2 is a UA-RA transaction. That is because, given Agreement properties of Parallel Optimistic Broadcast and POA, we can assume that both tx_1 and tx_2 were decided in the slow path, that is in the instance number $tx_1.sn$ (same as $tx_2.sn$) of Transaction Agreement of A. But then $tx_1 = tx_2$ by the Agreement property of Transaction Agreement.

◀

Fast UA Transaction Execution. Each honest validator will terminate the Byzantine Reliable Broadcast of tx in two communication steps. And by Lemma 34, every honest validator will be ready to execute tx by that time. ◀

Fast UA-RA Transaction Execution. The first communication step is a user's broadcast of tx, so, in particular, L receives it. Then we apply the Fast Termination property of POA which adds two additional communication steps. ◀

Multi-termination. Due to the Termination property of POA, it's sufficient to show that $V.X$ will initiate k-th instance. This we prove by induction on k.

The first POA instance is initiated immediately when the reactive actor entity is initialized. Now, assume a validator decides in the $k-1$ instance. This would trigger a $poa.Decide(k-1, \cdot)$ event (Line 1, Algorithm 4) and thus a $poa.Initiate(k)$ event (Line 16). ◀