Permissionless and Asynchronous Asset Transfer

Petr Kuznetsov
LTCI, Télécom Paris, Institut Polytechnique de Paris, France

Yvonne-Anne Pignolet
DFINITY, Zürich, Switzerland

Pavel Ponomarev
ITMO University, Saint Petersburg, Russia

Andrei Tonkikh
National Research University Higher School of Economics, Saint Petersburg, Russia

Abstract

Most modern asset transfer systems use *consensus* to maintain a totally ordered chain of transactions. It was recently shown that consensus is not always necessary for implementing asset transfer. More efficient, *asynchronous* solutions can be built using *reliable broadcast* instead of consensus. This approach has been originally used in the closed (permissioned) setting. In this paper, we extend it to the open (*permissionless*) environment. We present Pastro, a permissionless and asynchronous asset-transfer implementation, in which *quorum systems*, traditionally used in reliable broadcast, are replaced with a weighted *Proof-of-Stake* mechanism. Pastro tolerates a *dynamic* adversary that is able to adaptively corrupt participants based on the assets owned by them.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Asset transfer, permissionless, asynchronous, dynamic adversary

Digital Object Identifier 10.4230/LIPIcs.DISC.2021.28

Related Version This paper is an extended abstract of the technical report [19].
Technical Report: `https://arxiv.org/abs/2105.04966`

Funding *Petr Kuznetsov*: supported by TrustShare Innovation Chair.

1 Introduction

Inspired by advances in peer-to-peer data replication [25, 31], a lot of efforts are currently invested in designing an algorithm for consistent and efficient exchange of assets in *dynamic* settings, where the set of participants, actively involved in processing transactions, varies over time.

Sometimes such systems are called *permissionless*, emphasizing the fact that they assume no trusted mechanism to regulate who and when can join the system. In particular, permissionless protocols should tolerate the notorious *Sybil attack* [7], where the adversary creates an unbounded number of "fake" identities.

Sharing data in a permissionless system is a hard problem. To solve it, we have to choose between consistency and efficiency. Assuming that the network is synchronous and that the adversary can only possess less than half of the total computing power, Bitcoin [25] and Ethereum [31] make sure that participants reach *consensus* on the order in which they access and modify the shared data. For this purpose, these systems employ a *proof-of-work* (PoW) mechanism to artificially slow down active participants (*miners*). The resulting algorithms are notoriously slow and waste tremendous amounts of energy.

Other protocols obviate the energy demands using *proof-of-stake* [2,4,18], *proof-of-space* [9], or *proof of space-time* [24]. However, these proposals still resort to synchronous networks, randomization, non-trivial cryptography and/or assume a trusted setup system.

In this paper, we focus on a simpler problem, *asset transfer*, enabling a set of participants to exchange assets across their *accounts*. It has been shown [15, 16] that this problem does not require consensus. Assuming that every account is operated by a dedicated user, there is no need for the users to agree on a total order in which transactions must be processed: one can build an asset transfer system on top of the *reliable broadcast* abstraction instead of consensus. Unlike consensus [11], reliable broadcast allows for simple *asynchronous* solutions, enabling efficient asset transfer implementations that outperform their consensus-based counterparts [6].

Conventionally, a reliable broadcast algorithm assumes a *quorum system* [12, 22]. Every delivered message should be *certified* by a *quorum* of participants. Any two such quorums must have a correct participant in common and in every run, at least one quorum should consist of correct participants only. In a static f-resilient system of n participants, these assumptions result in the condition $f < n/3$. In a *permissionless* system, where the Sybil attack is enabled, assuming a traditional quorum system does not appear plausible. Indeed, the adversary may be able to control arbitrarily many identities, undermining any quorum assumptions.

In this paper, we describe a permissionless asset-transfer system based on *weighted* quorums. More precisely, we replace traditional quorums with certificates signed by participants holding a sufficient amount of assets, or *stake*. One can be alerted by this assumption, however: the notion of a "participant holding stake" at any given moment of time is not well defined in a decentralized consensus-free system where assets are dynamically exchanged and participants may not agree on the order in which transactions are executed. We resolve this issue using the notion of a *configuration*. A configuration is a partially ordered set of transactions that unambiguously determines the active system participants and the distribution of stake among them. As has been recently observed, configurations form a *lattice order* [20] and a *lattice agreement* protocol [10, 20, 21] can be employed to make sure that participants properly reconcile their diverging opinions on which configurations they are in.

Building on these abstractions, we present the Pastro protocol to settle asset transfers, despite a *dynamic adversary* that can choose which participants to compromise *during* the execution, taking their current stake into account. The adversary is restricted, however, to corrupt participants that together own less than one third of stake in any *active candidate* configuration. Intuitively, a configuration (a set of transactions) is an active candidate configuration if all its transactions *can be* accepted by a correct process. At any moment of time, we may have multiple candidate configurations, and the one-third stake assumption must hold for each of them.

Note that a *superseded* configuration that has been successfully replaced with a new one can be compromised by the adversary. To make sure that superseded configurations cannot deceive slow participants that are left behind the reconfiguration process, we employ a *forward-secure* digital signature scheme [1, 8], recently proposed for Byzantine fault-tolerant reconfigurable systems [21]. The mechanism allows every process to maintain a single public key and a "one-directional" sequence of matching private keys: it is computationally easy to compute a new private key from an old one, but not vice versa. Intuitively, before installing a new configuration, one should ask holders of $> 2/3$ of stake of the old one to upgrade their private keys and destroy the old ones.

We believe that Pastro is the right alternative to heavy-weight consensus-based replicated state machines, adjusted to applications that do not require global agreement on the order on their operations [20, 27]. Our solution does not employ PoW [25, 31] and does not rely on

complex cryptographic constructions, such as a common coin [4,26]. More importantly, unlike recently proposed solutions [14,28], Pastro is resistant against a dynamic adversary that can choose which participants to corrupt in a dynamic manner, depending on the execution.

In this paper, we first present Pastro in its simplest version, as our primary message is a possibility result: a permissionless asset-transfer system can be implemented in an asynchronous way despite a dynamic adversary. We then discuss multiple ways of improving and generalizing our system. In particular, we address the issues of maintaining a dynamic amount of assets via an inflation mechanism and the performance of the system by delegation and incremental updates.

Road map. We overview related work in Section 2. In Section 3, we describe our model and recall basic definitions, followed by the formal statement of the asset transfer problem in Section 4. In Section 5, we describe Pastro and outline its correctness arguments in Section 6. We conclude with practical challenges to be addressed as well as related open questions in Section 7. Detailed proofs and the discussion of optimizations as well as delegation, fees, inflation, and practical aspects of using forward-secure digital signatures are deferred to the appendix.

2 Related Work

The first and still the most widely used permissionless cryptocurrencies are blockchain-based [5]. To make sure that honest participants agree (with high probability) on the order in which blocks of transactions are applied, the most prominent blockchains rely on proof-of-work [25,31] and assume synchronous communication. The approach exhibits bounded performance, wastes an enormous amount of energy, and its practical deployment turns out to be hardly decentralized (`https://bitcoinera.app/arewedecentralizedyet/`).

To mitigate these problems, more recent proposals suggest to rely on the *stake* rather than on energy consumption. E.g., in next version of Ethereum [30], random *committees* of a bounded number of *validators* are periodically elected, under the condition that they put enough funds at stake. Using nontrivial cryptographic protocols (verifiable random functions and common coins), Algorand [13] ensures that the probability for a user to be elected is proportional to its stake, and the committee is reelected after each action, to prevent adaptive corruption of the committee.

In this paper, we deliberately avoid reaching consensus in implementing asset transfer, and build atop asynchronous *reliable broadcast*, following [15,16]. It has been recently shown that this approach results in a simpler, more efficient and more robust implementation than consensus-based solutions [6]. However, in that design a static set of of processes is assumed, i.e., the protocol adheres to the *permissioned* model.

In [14], a reliable broadcast protocol is presented, that allows processes to join or leave the system without requiring consensus. ABC [28] proposes a direct implementation of a cryptocurrency, under the assumption that the adversary never corrupts processes holding 1/3 or more stake. However, both protocols [14,28] assume a static adversary: the set of corrupted parties is chosen at the beginning of the protocol.

In contrast, our solution tolerates an adversary that *dynamically* selects a set of corrupted parties, under the assumption that not too much stake is compromised in an *active candidate* configuration. Our solution is inspired by recent work on *reconfigurable* systems that base upon the reconfigurable lattice agreement abstraction [20,21]. However, in contrast to these *general-purpose* reconfigurable constructions, our implementation is much lighter. In

our context, a configuration is just a distribution of stake, and every new transaction is a configuration update. As a result, we can build a simpler protocol that, unlike conventional asynchronous reconfigurable systems [20, 21, 29], does not bound the number of configuration updates for the sake of liveness.

3 Preliminaries

Processes and channels. We assume a set Π of potentially participating *processes*. In our model, every process acts both as a *replica* (maintains a local copy of the shared data) and as a *client* (invokes operations on the data).[1] In the proofs and definitions, we make the standard assumption of existence of a global clock not accessible to the processes.

At any moment of time, a process can be *correct* or *Byzantine*. We call a process *correct* as long as it faithfully follows the algorithm it has been assigned. A process is *forever-correct* if it remains correct forever. A correct process may turn *Byzantine*, which is modelled as an explicit event in our model (not visible to the other processes). A Byzantine process may perform steps not prescribed by its algorithm or prematurely stop taking steps. Once turned Byzantine, the process stays Byzantine forever.

We assume a *dynamic* adversary that can choose the processes to corrupt (to render Byzantine) depending on the current execution (modulo some restrictions that we discuss in the next section). In contrast, a *static* adversary picks up the set of Byzantine processes *a priori*, at the beginning of the execution.

In this paper we assume that the computational power of the adversary is bounded and, consequently, the cryptographic primitives used cannot be broken.

We also assume that each pair of processes is connected via *reliable authenticated channel*. If a forever-correct process p sends a message m to a forever-correct process q, then q eventually receives m. Moreover, if a correct process q receives a message m from a correct process p, then p has indeed sent m to q.

For the sake of simplicity, we assume that the set Π of potentially participating processes is finite.[2]

Weak reliable broadcast (WRB). In this paper we assume a *weak reliable broadcast primitive* to be available. The implementation of such primitive ensures the following properties:

- If a correct process delivers a message m from a correct process p, then m was previously broadcast by p;
- If a forever-correct process p broadcasts a message m, then p eventually delivers m;
- If a forever-correct process delivers m, then every forever-correct process eventually delivers m.

This weak reliable broadcast primitive can be implemented via a gossip protocol [17].

Forward-secure digital signatures. Originally, *forward-secure digital signature schemes* [1, 8] were designed to resolve the key exposure problem: if the signature (private) key used in the scheme is compromised, then the adversary is capable to forge any previous (or future)

[1] We discuss how to split the processes into replicas and clients in the technical report [19].

[2] In practice, this assumptions boils down to requiring that the rate at which processes are added is not too high. Otherwise, if new stake-holders are introduced into the system at a speed prohibiting a client from reaching a sufficiently large fraction of them, we cannot make sure that the clients' transactions are eventually accepted.

signature. Using forward secure signatures, it is possible for the private key to be updated arbitrarily many times with the public key remaining fixed. Also, each signature is associated with a timestamp. This helps to identify messages which have been signed with the private keys that are already known to be compromised.

To generate a signature with timestamp t, the signer uses secret key sk_t. The signer can update its secret key and get sk_{t_2} from sk_{t_1} if $t_1 < t_2 \leq T$. However "downgrading" the key to a lower timestamp, from sk_{t_2} to sk_{t_1}, is computationally infeasible. As in recent work on Byzantine fault-tolerant reconfigurable systems [21], we model the interface of forward-secure signatures with an oracle which associates every process p with a timestamp st_p. The oracle provides the following functions:

- `UpdateFSKey`(t) sets st_p to t if $t \geq st_p$;
- `FSSign`(m, t) returns a signature for a message m and a timestamp t if $t \geq st_p$, otherwise $\perp$;
- `FSVerify`(m, p, s, t) returns *true* if $s \neq \perp$ and it was generated by process p using `FSSign`(m, t), *false* otherwise.

In most of the known implementations of forward-secure digital signature schemes the parameter T should be fixed in advance, which makes number of possible private key updates finite. At the same time, some forward-secure digital schemes [23] allow for an unbounded number of key updates ($T = +\infty$), however the time required for an update operation depends on the number of updates. Thus, local computational time may grow indefinitely albeit slowly. We discuss these two alternative implementations and reason about the most appropriate one for our problem in the technical report [19].

Verifiable objects. We often use *certificates* and *verifiable objects* in our protocol description. We say that object $obj \in O$ is *verifiable* in terms of a given *verifying function* $\texttt{Verify} : O \times \Sigma_O \to \{true, false\}$, if it comes together with a *certificate* $\sigma_{obj} \in \Sigma_O$, such that $\texttt{Verify}(obj, \sigma_{obj}) = true$, where Σ_O is a set of all possible certificates for objects of set O. A certificate σ_{obj} is *valid* for obj (in terms of a given verifying function) iff $\texttt{Verify}(obj, \sigma_{obj}) = true$. The actual meaning of an object "verifiability", as well as of a certificate validness, is determined by the verifying function.

4 Asset Transfer: Problem Statement

Before defining the asset transfer problem formally, we introduce the concepts used later.

Transactions. A *transaction* is a tuple $tx = (p, \tau, D)$. Here $p \in \Pi$ is an identifier of a process inside the system that initiates a transaction. We refer to p as the *owner* of tx. The map $\tau\colon \Pi \to \mathbb{Z}_0^+$ is the *transfer function*, specifying the amount of funds received by every process $q \in \Pi$ from this transaction.[3] D is a finite set of transactions that tx *depends on*, i.e., tx spends the funds p received through the transactions in D. We refer to D as the *dependency set* of tx.

Let $\mathcal{T}$ denote the set of all transactions. The function $value : \mathcal{T} \to V$ is defined as follows: $value(tx) = \sum_{q \in \Pi} tx.\tau(q)$. $\mathcal{T}$ contains one special "initial" transaction $tx_{init} = (\perp, \tau_{init}, \emptyset)$ with $value(tx_{init}) = M$ and no sender. This transaction determines the initial distribution of the total amount of "stake" in the system (denoted M).

[3] We encode this map as a set of tuples (q, d), where $q \in \Pi$ and $d > 0$ is the amount received sent to q in tx.

For all other transactions it holds that a transaction tx is *valid* if the amount of funds spent equals the amount of funds received: $value(tx) = \sum_{t \in t.D} t.\tau(tx.p)$. For simplicity, we assume that $\mathcal{T}$ contains only valid transactions: invalid transactions are ignored.

Transactions defined this way, naturally form a *directed graph* where each transaction is a node and edges represent dependencies.

We say that transactions tx_i and tx_j issued by the same process *conflict* iff the intersection of their dependency sets is non-empty: $tx_i \neq tx_j, tx_i.p = tx_j.p$ and $tx_i.D \cap tx_j.D \neq \emptyset$. A correct member of an asset-transfer system does not attempt to *double spend*, i.e., it never issues transactions which share dependencies and are therefore conflicting.

Transactions are equipped with a boolean function $\texttt{VerifySender} : \mathcal{T} \times \Sigma_{\mathcal{T}} \to \{true, false\}$. A transaction tx is *verifiable* iff it is verifiable in terms of the function $\texttt{VerifySender}(tx, \sigma_{tx})$. Here σ_{tx} is a certificate that confirms that tx was indeed issued by its owner process p. One may treat a transaction's certificate as p's digital signature of tx.

Asset-transfer system. An *asset-transfer* system (AS) maintains a partially ordered set of transactions T_p and exports one operation: $\texttt{Transfer}(tx, \sigma_{tx})$ adding transaction tx to the set T_p. Recall that $tx = (p, \tau, D)$, where p is the owner of transaction, τ is a transfer map, D is the set of dependencies, and σ_{tx} is a matching certificate.

In a distributed AS implementation, every process p holds the set of "confirmed" transactions T_p it is aware of, i.e., a local copy of the state. A transaction is said to be *confirmed* if some correct process p adds tx to its local copy of the state T_p. Set T_p can be viewed as the log of all confirmed transactions a process p is aware of. Let $T_p(t)$ denote the value of T_p at time t.

An AS implementation then satisfies the following properties:

Consistency: For every process p correct at time t, $T_p(t)$ contains only verifiable and non-conflicting transactions. Moreover, for every two processes p and q correct at time t and t' resp.: $(T_p(t) \subseteq T_q(t')) \vee (T_q(t') \subseteq T_p(t))$.

Monotonicity: For every correct process p, T_p can only grow with time: for all $t < t'$, $T_p(t) \subseteq T_p(t')$.

Validity: If a forever-correct process p invokes $\texttt{Transfer}(tx, \sigma_{tx})$ at time t, then there exists $t' > t$ such that $tx \in T_p(t')$.

Agreement: For a process p correct at time t and a forever-correct process q, there exists $t' \geq t$ such that $T_p(t) \subseteq T_q(t')$.

Here, $\sigma_{tx} \in \Sigma_{\mathcal{T}}$ is a certificate for transaction $tx = (p, \tau, D)$. A certificate protects the users from possible theft of their funds by other users. As we assume that cryptographic techniques (including digital signatures) are unbreakable, the only way to steal someone's funds is to steal their private key.

A natural convention is that a correct process never submits conflicting transactions. When it comes to Byzantine processes, we make no assumptions. Our specification ensures that if two conflicting transactions are issued by a Byzantine process, then at most one of them will ever be confirmed. In fact, just a single attempt of a process to cheat may lead to the loss of its funds as it can happen that neither of the conflicting transactions is confirmed and thus may preclude the process from making progress.

Transaction set as a configuration. For simplicity, we assume that the total amount of funds in the system is a publicly known constant $M \in \mathbb{Z}^+$ (fixed by the initial transaction tx_{init}) that we call *system stake*.[4]

[4] In the technical report [19], we discuss how to maintain a dynamic system stake.

A set of transactions $C \in 2^{\mathcal{T}}$ is called a *configuration*.

A configuration C is *valid* if no two transactions $tx_i, tx_j \in C$ are conflicting. In the rest of this paper, we only consider valid configurations, i.e., invalid configurations appearing in protocol messages are ignored by correct processes without being mentioned explicitly in the algorithm description.

The *initial configuration* is denoted by C_{init}, and consists of just one initial transaction ($C_{init} = \{tx_{init}\}$).

A valid configuration C determines the *stake* (also known as *balance*) of every process p as the difference between the amount of assets sent to p and the amount of assets sent by p in the transactions of C: $stake(q, C) = \sum_{tx \in C} tx.\tau(q) - \sum_{tx \in C \wedge tx.p = q} value(tx)$. Intuitively, a process joins the asset-transfer system as soon as it gains a positive stake in a configuration and leaves once its stake turns zero.

Functions *members* and *quorums* are defined as follows: $members(C) = \{p \mid \exists tx \in C$ such that $tx.\tau(p) > 0\}$, $quorums(C) = \{Q \mid \sum_{q \in Q} stake(q, C) > \frac{2}{3}M\}$. Intuitively, members of the system in a given configuration C are the processes that received money at least once, and a set of processes is considered to be a quorum in configuration C, if their total stake in C is more than two-thirds of the system stake.

Configuration lattice. Recall that a *join-semilattice* (we simply say a *lattice*) is a tuple $(\mathcal{L}, \sqsubseteq)$, where $\mathcal{L}$ is a set of *elements* provided with a partial-order relation $\sqsubseteq$, such that for any two elements $a \in \mathcal{L}$ and $b \in \mathcal{L}$, there exists the *least upper bound* for the set $\{a, b\}$, i.e., an element $c \in \mathcal{L}$ such that $a \sqsubseteq c$, $b \sqsubseteq c$ and $\forall d \in \mathcal{L}$: if $a \sqsubseteq d$ and $b \sqsubseteq d$, then $c \sqsubseteq d$. The least upper bound of elements $a \in \mathcal{L}$ and $b \in \mathcal{L}$ is denoted by $a \sqcup b$. $\sqcup$ is an associative, commutative and idempotent binary operator on $\mathcal{L}$. It is called the *join operator*.

The *configuration lattice* is then defined as $(\mathcal{C}, \sqsubseteq)$, where $\mathcal{C}$ is the set of all valid configurations, $\sqsubseteq = \subseteq$ and $\sqcup = \cup$.

5 Pastro Asset Transfer: Algorithm

In this section we present Pastro – an implementation of an asset-transfer system. We start with the main building blocks of the algorithm, and then proceed to the description of the Pastro protocol itself.

We bundle parts of Pastro algorithm that are semantically related in building blocks called *objects*, each offering a set of operations, and combine them to implement asset transfer.

Transaction Validation. The *Transaction Validation* (TV) object is a part of Pastro ensuring that transactions that a correct process p adds to its local state T_p do not conflict.

A correct process p submits a transaction $tx = (p, \tau, D)$ and a matching certificate to the object by invoking an operation `Validate`(tx, σ_{tx}). The operation returns a set of transactions $txs \in 2^{\mathcal{T}}$ together with a certificate $\sigma_{txs} \in \Sigma_{2^{\mathcal{T}}}$. We call a transaction set *verifiable* iff it is verifiable in terms of a function `VerifyTransactionSet`(txs, σ_{txs}). Intuitively, the function returns *true* iff certificate σ_{txs} confirms that set of transactions txs is *validated* by sufficiently many system members.

Formally, TV satisfies the following properties:

TV-Verifiability: If an invocation of `Validate` returns $\langle txs, \sigma_{txs} \rangle$ to a correct process, then `VerifyTransactionSet`$(txs, \sigma_{txs}) = true$;

TV-Inclusion: If `Validate`(tx, σ_{tx}) returns $\langle txs, \sigma_{txs} \rangle$ to a correct process, then $tx \in txs$;

TV-Validity: The union of returned verifiable transaction sets consists of non-conflicting verifiable transactions.

In our algorithm, we use one Transaction Validation object *TxVal*.

Adjustable Byzantine Lattice Agreement. Our asset-transfer algorithm reuses elements of an implementation of *Byzantine Lattice Agreement* (BLA), a Lattice Agreement [10] protocol that tolerates Byzantine failures. We introduce *Adjustable Byzantine Lattice Agreement* (ABLA), an abstraction that captures safety properties of BLA. An ABLA object is parameterized by a lattice $(\mathcal{L}, \sqsubseteq)$ and a boolean function $\texttt{VerifyInputValue} : \mathcal{L} \times \Sigma_{\mathcal{L}} \to \{true, false\}$. An input value of a given ABLA object is *verifiable* if it complies with $\texttt{VerifyInputValue}$.

ABLA exports one operation: $\texttt{Propose}(v, \sigma_v)$, where $v \in \mathcal{L}$ is an input value and $\sigma_v \in \Sigma_{\mathcal{L}}$ is a matching certificate. It also exports function $\texttt{VerifyOutputValue} : \mathcal{L} \times \Sigma_{\mathcal{L}} \to \{true, false\}$. The $\texttt{Propose}$ operation returns a pair $\langle w, \sigma_w \rangle$, where $w \in \mathcal{L}$ is an output value and $\sigma_w \in \Sigma_{\mathcal{L}}$ is a matching certificate. An output value w of an ABLA object is *verifiable* if it complies with function $\texttt{VerifyOutputValue}$.

An ABLA object satisfies the following properties:

ABLA-Validity : Every verifiable output value w is a join of some set of verifiable input values;

ABLA-Verifiability: If an invocation of $\texttt{Propose}$ returns $\langle w, \sigma_w \rangle$ to a correct process, then $\texttt{VerifyOutputValue}(w, \sigma_w) = true$;

ABLA-Inclusion: If $\texttt{Propose}(v, \sigma_v)$ returns $\langle w, \sigma_w \rangle$, then $v \sqsubseteq w$;

ABLA-Comparability: All verifiable output values are comparable.

In our algorithm, we use two ABLA objects, *ConfigLA* and *HistLA*.

Combining TV and ABLA. Let *ConfigLA* be an ABLA object parameterized as follows: $(\mathcal{L}, \sqsubseteq) = (2^{\mathcal{T}}, \subseteq)$, $\texttt{VerifyInputValue}(v, \sigma_v)$ is defined as $\texttt{VerifyTransactionSet}(v, \sigma_v)$, that is a part of TV. The function $\texttt{VerifyConfiguration}(C, \sigma_C)$ is an alias for $\textit{ConfigLA}.\texttt{VerifyOutputValue}(C, \sigma_C)$. Using this object, the processes produce *comparable configurations*, i.e., transaction sets related by containment ($\subseteq$).

HistLA is an ABLA object used to produce *sets of configurations* that are all related by containment. Under the assumption that every input (a set of configurations) to the object only contains comparable configurations, the outputs are related by containment and configurations in these sets are comparable. We can see these sets as *sequences* of ordered configurations. Such sets are called *histories*, as was recently suggested for asynchronous Byzantine fault-tolerant reconfiguration [21]. It was shown that histories allow us to access only $O(n)$ configurations, when n configurations are concurrently proposed. A history h is called *verifiable* if it complies with $\texttt{VerifyHistory}(h, \sigma_h)$.

Formally, the ABLA object *HistLA* is parameterized as follows: $(\mathcal{L}, \sqsubseteq) = (2^{2^{\mathcal{T}}}, \subseteq)$. The requirement that the elements of an input are comparable is established via the function $\texttt{VerifyInputValue}(\{\langle C_1, \sigma_1 \rangle, \ldots, \langle C_n, \sigma_n \rangle\}) = \bigwedge_{i=1}^{n} \texttt{VerifyConfiguration}(C_i, \sigma_i)$. $\texttt{VerifyHistory}(H, \sigma_H)$ is an alias for $\textit{HistLA}.\texttt{VerifyOutputValue}(H, \sigma_H)$.

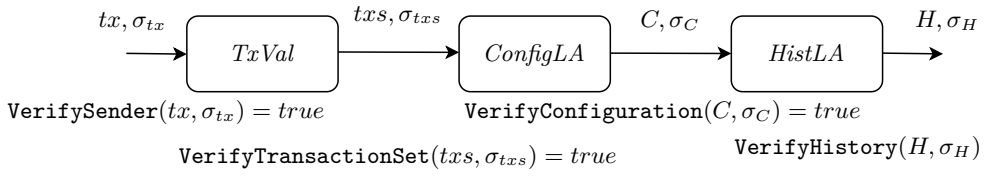

Figure 1 PASTRO pipeline.

Thus, the only valid input values for the *HistLA* object consist of verifiable output values of the *ConfigLA* object. At the same time, the only valid input values for the *ConfigLA* object are verifiable transaction sets, returned from the *TxVal* object. And, the only valid inputs for a *TxVal* object are signed transactions. In this case, all verifiable histories are related by containment (comparable via $\subseteq$), and all configurations within one verifiable history are related by containment (comparable via $\subseteq$) as well. Such a pipeline helps us to guarantee that all configurations of the system are comparable and contain non-conflicting transactions. Hence, such configurations can act as a consistent representation of stake distribution that changes with time.

To get a high-level idea of how Pastro works, imagine a conveyor belt (Figure 1). Transactions of different users are submitted as inputs, and as outputs, they obtain sets of configurations. Then one can choose the configuration representing the largest resulting set and install it in the system, changing the funds distribution in the system.

Algorithm overview. We assume that blocks of code (functions, operations, callbacks) are executed sequentially until they complete or get interrupted by a wait condition (**wait for** ...). Some events, e.g., receiving a message, may trigger callbacks (marked with keyword **upon**). However, they are not executed immediately, but firstly placed in an event queue, waiting for their turn.

By "**let** *var* = *expression*" we denote an assignment to a local variable (which can be accessed only from the current function, operation, or callback), and by "*var* $\leftarrow$ *expression*" we denote an assignment to a global variable (which can be accessed from anywhere by the same process).

We denote calls to a weak reliable broadcast primitive with **WRB-broadcast**$\langle \dots \rangle$ and **WRB-deliver**$\langle \dots \rangle$. Besides, we assume a *weak uniform reliable broadcast (WURB)* primitive, a variant of uniform reliable broadcast [3]. WURB ensures an additional property compared to WRB: if the system remains static (i.e., the configurations stop changing) and a correct process delivers message m, then every forever-correct process eventually delivers m. The primitive helps us to ensure that if configuration C is never replaced with a greater one, then every forever-correct process will eventually learn about such a configuration C. To achieve this semantics, before triggering the **deliver** callback, a correct process just needs to ensure that a quorum of replicas have received the message [3]. The calls to WURB should be associated with some configuration C, and are denoted as **WURB-broadcast**$\langle \dots, C \rangle$ and **WURB-deliver**$\langle \dots, C \rangle$.

The algorithm uses one TV object *TxVal* and two ABLA objects: *ConfigLA* and *HistLA*. We list the pseudocode for the implementation of the *TxVal* object in Appendix A, Figure 5. The implementations of *ConfigLA* and *HistLA* are actually the same, and only differ in their parameters, we therefore provide one generalized implementation for both objects (Appendix A, Figure 6).

Every process maintains variables *history*, T_p and C_{cur}, accessible everywhere in the code. T_p is the latest installed configuration by the process p, by C_{cur} we denote the configuration starting from which the process needs to transfer data it stores to greater configurations, and *history* is the current verifiable history (along with its certificate σ_{hist}).

The main part of Pastro protocol (depicted in Figure 2) exports one operation `Transfer`(tx, σ_{tx}).

In this operation, we first set the local variables that store intermediate results produced in this operation, to null values $\perp$. Next, the `Validate` operation of the Transaction Validation object *TxVal* is invoked with the given transaction tx and the corresponding certificate σ_{tx}. The set of transactions txs and certificate σ_{txs} returned from the TV object are then used as an input for `Propose` operation of *ConfigLA*. Similarly, the result returned

from *ConfigLA* "wrapped in" a singleton set, is used as an input for `Propose` operation of *HistLA*. The returned verifiable history is then broadcast in the system. We consider operation `Transfer`(tx, σ_{tx}) to *complete* by a correct process p once process p broadcasts message $\langle$**NewHistory**$, h, \sigma\rangle$ at line 9 or if p stops the current `Transfer` by executing line 17. Basically, the implementation of `Transfer` operation follows the logic described before and depicted in Figure 1.

typealias SignedTransaction = Pair$\langle \mathcal{T}, \Sigma_{\mathcal{T}} \rangle$
typealias $\mathcal{C}$ = Set$\langle \mathcal{T} \rangle$

Global variables:
TxVal = TV()
ConfigLA = ABLA($\mathcal{L} = 2^{\mathcal{T}}, \sqsubseteq = \subseteq, v_{init} = C_{init}$)
ConfigLA.VerifyInputValue(v, σ) = VerifyTransactionSet(v, σ)
VerifyConfiguration(v, σ) = *ConfigLA*.VerifyOutputValue(v, σ)
HistLA = ABLA($\mathcal{L} = 2^{2^{\mathcal{T}}}, \sqsubseteq = \subseteq, v_{init} = \{C_{init}\}$)
HistLA.VerifyInputValue$(\{v\}, \sigma)$ = VerifyConfiguration(v, σ)
VerifyHistory(v, σ) = *HistLA*.VerifyOutputValue(v, σ)

$T_p = C_{init}$
$C_{cur} = C_{init}$
history = $\{C_{init}\}$, $\sigma_{hist} = \bot$

txs $= \bot$, $\sigma_{txs} = \bot$
$C = \bot$, $\sigma_C = \bot$
$H = \bot$, $\sigma_H = \bot$
curTx $= \bot$

operation Transfer(tx: $\mathcal{T}$, σ_{tx}: $\Sigma_{\mathcal{T}}$): void
1 $txs \leftarrow \bot$, $\sigma_{txs} \leftarrow \bot$
2 $C \leftarrow \bot$, $\sigma_C \leftarrow \bot$
3 $H \leftarrow \bot$, $\sigma_H \leftarrow \bot$
4 $curTx \leftarrow tx$
5 $txs, \sigma_{txs} \leftarrow$ *TxVal*.Validate(tx, σ_{tx})
6 $C, \sigma_C \leftarrow$ *ConfigLA*.Propose(txs, σ_{txs})
7 $H, \sigma_H \leftarrow$ *HistLA*.Propose$(\{C\}, \sigma_C)$
8 $curTx \leftarrow \bot$
9 **WRB-broadcast** $\langle$**NewHistory**$, H, \sigma_H\rangle$

upon WRB-deliver $\langle$**NewHistory**$, h, \sigma\rangle$ from *any*:
10 **if** VerifyHistory(h, σ) **and** history $\subset h$ **then**
11 **trigger** event NewHistory(h) { $\forall C \in h : C$ *is a candidate configuration* }
12 history $\leftarrow h$, $\sigma_{hist} \leftarrow \sigma$
13 **let** C_h = HighestConf(history)
14 UpdateFSKey(height(C_h))
15 **if** $curTx \neq \bot$ **then** { *There is an ongoing* `Transfer` operation }
16 **if** $curTx \in C_h$ **then** { *The last issued transaction by p is included in verifiable history* }
17 CompleteTransferOperation() { *Stops the ongoing* `Transfer` *operation if any* }
18 **else if** *txs* $= \bot$ **then** { *Operation* `Transfer` *is ongoing and txs has not been received* }
19 *TxVal*.Request($\emptyset$) { *Restarts transaction validation by accessing* C_h }
20 **else if** $C = \bot$ **then** { *Operation* `Transfer` *is ongoing and C has not been received yet* }
21 *ConfigLA*.Refine($\emptyset$) { *Restarts verifiable configuration reception by accessing* C_h }
22 **else if** $H = \bot$ **then** { *Operation* `Transfer` *is ongoing and H has not been received yet* }
23 *HistLA*.Refine($\emptyset$) { *Restarts verifiable history reception by accessing* C_h }

Figure 2 PASTRO: code for process p.

If a correct process delivers a message $\langle$**NewHistory**$, h, \sigma\rangle$, where σ is a valid certificate for history h that is greater than its local estimate *history*, it "restarts" the first step that it has not yet completed in Transfer operation (lines 18-23). For example, if a correct process p receives a message $\langle$**NewHistory**$, h, \sigma_h\rangle$, where σ_h is a valid certificate for h and *history* $\subset h$

while being in *ConfigLA*, it restarts this step in order to access a greater configuration. The result of *ConfigLA* will still be returned to p in the place it has called *ConfigLA*.`Propose`. Intuitively, we do this in order to reach the most "up-to-date" configuration ("up-to-date" stake holders).

The State Transfer Protocol (Figure 3) helps us to ensure that the properties of the objects (*TxVal*, *ConfigLA* and *HistLA*) are satisfied *across configurations* that our system goes through. As system stake is redistributed actively with time, quorums in the system change as well and, hence, we need to pass the data that some quorum knows in configuration C to some quorum of any configuration $C' : C \sqsubset C'$, that might be installed after C. The protocol is executed by a correct process after it delivers a verifiable history h, such that $C \in h$ and $C_{cur} \sqsubset C$.

The *height* of a configuration C is the number of transactions in it (denoted as $height(C) = |C|$). Since all configurations installed in Pastro are comparable, height can be used as a unique identifier for an installed configuration. We also use height as the *timestamp* for forward-secure digital signatures. When a process p answers requests in configuration C, it signs the message with timestamp $height(C)$. The process p invokes UpdateFSKey($height(C')$) when it discovers a new configuration $C' : C \sqsubset C'$. Thus, processes that still consider C as the current configuration (and see the corresponding stake distribution) cannot be deceived by a process p, that was correct in C, but not in a higher installed configuration C' (e.g., p spent all its stake by submitting transactions, which became part of configuration C', thereby lost its weight in the system, and later became Byzantine).

The implementation of verifying functions (Figure 4) and a description of the auxiliary functions used in the pseudocode are delegated to Appendix A.

Implementing Transaction Validation. The implementation of the TV object *TxVal* is depicted in Figure 5 (Appendix A). The algorithm can be divided into two phases.

In the first phase, process p sends a message request that contains a set of transactions *sentTxs* to be validated to all members of the current configuration. Every correct process q that receives such messages first checks whether the transactions have actually been issued by their senders. If yes, q adds the transactions in the message to the set of transactions it has seen so far and checks whether any transaction from the ones it has just received conflicts with some other transaction it knows about. All conflicting transactions are placed in set *conflictTxs*. After q validates transactions, it sends a message $\langle$**ValidateResp**, *txs*, *conflictTxs*, *sig*, *sn*$\rangle$. Here, *txs* is the union of verifiable transactions received by p from q just now and all other non-conflicting transactions p is aware of so far. Process p then verifies a received message $\langle$**ValidateResp**, txs_q, $conflictTxs_q$, sig_q, *sn*$\rangle$. The message received from q is considered to be valid by p if q has signed it with a private key that corresponds to the current configuration, all the transactions from txs_q have valid certificates, and if for any verifiable transaction tx from *conflictTxs* there is a verifiable transaction tx also from *conflictTxs*, such that tx conflicts with tx'. If the received message is valid and txs_q equals *sentTxs*, then p adds signature of process q and its validation result to its local set $acks_1$. In case $sentTxs \subset txs_q$, the whole phase is restarted. The first phase is considered to be completed as soon as p collects responses from some quorum of processes in $acks_1$.

Such implementation of the first phase makes correct process p obtain certificate not only for its transaction, but also for other non-conflicting transactions issued by other processes. This helping mechanism ensures that transactions of forever-correct processes are eventually confirmed and become part of some verifiable configuration.

```
upon C_cur ≠ HighestConf(history):
24    let C_next = HighestConf(history)
25    let S = {C ∈ history | C_cur ⊑ C ⊏ C_next}
26    seqNum ← seqNum + 1
27    for C in S:
28          send ⟨UpdateRead, seqNum, C⟩ to members(C)
29          wait for (C ⊏ C_cur or ∃Q ∈ quorums(C) responded with seqNum)
30    if C_cur ⊏ C_next then
31          C_cur ← C_next
32          WURB-broadcast ⟨UpdateComplete, C_next⟩

upon receive ⟨UpdateRead, sn, C⟩ from q:
33    wait for C ⊏ HighestConf(history)
34    let txs = TxVal.seenTxs
35    let values_1 = ConfigLA.values
36    let values_2 = HistLA.values
37    send ⟨UpdateReadResp, txs, values_1, values_2, sn⟩ to q

upon receive ⟨UpdateReadResp, txs, values_1, values_2, sn⟩ from q:
38    if VerifySenders(txs) and ConfigLA.VerifyValues(values_1)
39              and HistLA.VerifyValues(values_2) then
40          TxVal.seenTxs ← TxVal.seenTxs ∪ txs
41          ConfigLA.values ← ConfigLA.values ∪ values_1.filter(⟨v, σ_v⟩ ⇒ v ∉ ConfigLA.values.firsts())
42          HistLA.values ← HistLA.values ∪ values_2.filter(⟨v, σ_v⟩ ⇒ v ∉ HistLA.values.firsts())

upon WURB-deliver ⟨UpdateComplete, C⟩ from quorum Q ∈ quorums(C):
43    wait for C ∈ history
44    if T_p ⊏ C then
45          if C_cur ⊏ C then C_cur ← C
46          T_p ← C { Update set of "confirmed" transactions }
47          trigger event InstalledConfig(C) { C is an installed configuration }
```

Figure 3 State Transfer Protocol: code for process p.

In the second phase, p collects signatures from a weighted quorum of the current configuration. If p successfully collects such a set of signatures, then the configuration it saw during the first phase was *active* (no greater configuration had been installed) and it is safe for p to return the obtained result. This way the so-called "slow reader" attack [21] is anticipated.

If during any of the two phases p receives a message with a new verifiable history that is greater (w.r.t. $\subseteq$) than its local estimate and does not contain last issued transaction by p, the described algorithm starts over. We guarantee that the number of restarts in $TxVal$ in Pastro protocol is finite only for *forever-correct* processes (please refer to the technical report [19] for a detailed proof). Note that we cannot guarantee this for *all* correct processes as during the protocol execution some of them can become Byzantine.

In the implementation, we assume that the dependency set of a transaction only includes transactions that are confirmed (i.e., included in some installed configuration), otherwise they are considered invalid.

Implementing Adjustable Byzantine Lattice Agreement. The generalized implementation of ABLA objects *ConfigLA* and *HistLA* is specified by Figure 6 (Appendix A). The algorithm is generally inspired by the implementation of Dynamic Byzantine Lattice Agreement from [21], but there are a few major differences. Most importantly, it is tailored to work even if the number of reconfiguration requests (i.e., transactions) is infinite. Similarly to the Transaction Validation implementation, algorithm consists of two phases.

In the first phase, process p sends a message that contains the verifiable inputs it knows to other members and then waits for a weighted quorum of processes of the current configuration to respond with the same set. If p receives a message with a greater set (w.r.t. $\subseteq$), it restarts the phase. The validation process performed by the processes is very similar to the one used in Transaction Validation object implementation.

The second phase, in fact, is identical to the second phase of the Transaction Validation implementation. We describe it in the pseudocode for completeness.

As with Transaction Validation, whenever p delivers a verifiable history h, such that it is greater than its own local estimate and h does not contain last issued transaction by p, the described algorithm starts over. Similarly to *TxVal*, it is guaranteed that the number of restarts a forever-correct process make in both *ConfigLA* and *HistLA* is finite.

6 Proof of Correctness

In this section, we outline the proof that Pastro indeed satisfies the properties of an asset-transfer system. First, we formulate a restriction we impose on the adversary that is required for our implementation to be correct. Informally, the adversary is not allowed to corrupt one third or more stake in a "candidate" configuration, i.e., in a configuration that can potentially be used for adding new transactions. The adversary is free to corrupt a configuration as soon as it is superseded by a strictly higher one. We then sketch the main arguments of our correctness proof (the detailed proof is available in [19]).

6.1 Adversarial restrictions

A configuration C is considered to be *installed* if some correct process has triggered the special event InstalledConfig(C). We call a configuration C a *candidate* configuration if some correct process has triggered a NewHistory(h) event, such that $C \in h$. We also say that a configuration C is *superseded* if some correct process installed a higher configuration C'. An installed (resp., candidate) configuration C is called an *active* (resp., an *active candidate*) configuration as long as it is not superseded. Note that at any moment of time t, every active installed configuration is an active candidate configuration, but not vice versa.

We expect the adversary to obey the following condition:

Configuration availability: Let C be an active candidate configuration at time t and let $correct(C, t)$ denote the set of processes in $members(C)$ that are correct at time t. Then C must be *available* at time t:

$$\sum_{q \in correct(C,t)} stake(q, C) > 2/3M.$$

Note that the condition allows the adversary to compromise a candidate configuration once it is superseded by a more recent one. As we shall see, the condition implies that our algorithm is live. Intuitively, a process with a pending operation will either eventually hear from the members of a configuration holding "enough" stake which might allow it to complete its operation or will learn about a more recent configuration, in which case it can abandon the superseded configuration and proceed to the new one.

6.2 Proof outline

Consistency. The consistency property states that (1) as long as process p is correct, T_p contains only verifiable non-conflicting transactions and (2) if processes p and q are correct at times t and t' respectively, then $T_p(t) \subseteq T_q(t')$ or $T_q(t') \subseteq T_p(t)$. To prove that Pastro

satisfies this property, we show that our implementation of *TxVal* meets the specification of Transaction Validation and that both *ConfigLA* and *HistLA* objects satisfy the properties of Adjustable Byzantine Lattice Agreement. Correctness of *HistLA* ensures that all verifiable histories are related by containment and the correctness of *ConfigLA* guarantees that all verifiable configurations are related by containment (i.e., they are *comparable*). Taking into account that the only possible verifiable inputs for *HistLA* are sets that contain verifiable output values (configurations) of *ConfigLA*, we obtain the fact that all configurations of any verifiable history are comparable as well. As all installed configurations (all C such that a correct process triggers an event InstalledConfig(C)) are elements of some verifiable history, they all are related by containment too. Since T_p is in fact the last configuration installed by a process p, we obtain (2). The fact that every p stores verifiable non-conflicting transactions follows from the fact that the only possible verifiable input values for *ConfigLA* are the output transaction sets returned by *TxVal*. As *TxVal* is a correct implementation of Transaction Validation, then union of all such sets contain verifiable non-conflicting transactions. Hence, the only verifiable configurations that are produced by the algorithm cannot contain conflicting transactions. From this we obtain (1).

Monotonicity. This property requires that T_p only grows as long as p is correct. The monotonicity of Pastro follows from the fact that correct processes install only greater configurations with time, and that the last installed configuration by a correct process p is exactly T_p. Thus, if p is correct at time t', then for all $t < t'$: $T_p(t) \subseteq T_p(t')$.

Validity. This property requires that a transfer operation for a transaction tx initiated by a forever-correct process will lead to tx being included in T_p at some point in time. In order to prove this property for Pastro, we show that a forever-correct process p may only be blocked in the execution of a `Transfer` operation if some other process successfully installed a new configuration. We argue that from some moment of time on every other process that succeeds in the installation of configuration C will include the transaction issued by p in the C. We also show that if such a configuration C is installed then eventually every forever-correct process installs a configuration $C' : C \sqsubseteq C'$. As T_p is exactly the last configuration installed by p, eventually any transaction issued by a forever-correct process p is included in T_p.

Agreement. In the end we show that the Pastro protocol satisfies the agreement property of an asset-transfer system. The property states the following: for a correct process p at time t and a forever-correct process q, there exists $t' \geq t$ such that $T_p(t) \subseteq T_p(t')$. Basically, it guarantees that if a transaction tx was considered confirmed by p when it was correct, then any forever-correct process will eventually confirm it as well. To prove this, we show that if a configuration C is installed by a correct process, then every other forever-correct process will install some configuration C', such that $C \sqsubseteq C'$. Taking into account the fact that T_p is a last configuration installed by a process p, we obtain the desired.

7 Concluding Remarks

Pastro is a permissionless asset transfer system based on proof of stake. It builds on lattice agreement primitives and provides its guarantees in asynchronous environments where less than one third of the total stake is owned by malicious parties.

Enhancements and optimizations. To keep the presentation focused, so far we described only the core of the Pastro protocol. However, there is a number of challenges that need to be addressed before the protocol can be applied in practice. In particular the communication, computation and storage complexity of the protocol can be improved with carefully constructed messages and signature schemes. Also, mechanisms for stake delegation as well as fees and inflation are necessary. Note that these are non-trivial to implement in an asynchronous system because there is no clear agreement on the order in which transactions are added to the system or on the distribution of stake at the moment when a transaction is added to the system. We discuss these topics as well as the practical aspects of using forward-secure signatures in the technical report [19].

Open questions. In this paper, we demonstrated that it is possible to combine asynchronous cryptocurrencies with proof-of-stake in presence of a dynamic adversary. However, there are still plenty of open questions. Perhaps, the most important direction is to study hybrid solutions which combine our approach with consensus in an efficient way in order to support general-purpose smart contracts. Further research is also needed in order to improve the efficiency of the solution and to measure how well it will behave in practice compared to consensus-based solutions. Finally, designing proper mechanisms in order to incentivize active and honest participation is a non-trivial problem in the harsh world of asynchrony, where the processes cannot agree on a total order in which transactions are executed.

References

1. Mihir Bellare and Sara K Miner. A forward-secure digital signature scheme. In *Annual International Cryptology Conference*, pages 431–448, Berlin, 1999. Springer.
2. Iddo Bentov, Ariel Gabizon, and Alex Mizrahi. Cryptocurrencies without proof of work. In *Financial Cryptography and Data Security – FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers*, pages 142–157, Berlin, 2016. Springer.
3. Christian Cachin, Rachid Guerraoui, and Luis Rodrigues. *Introduction to Reliable and Secure Distributed Programming.* Springer Publishing Company, Incorporated, 2nd edition, 2011.
4. Jing Chen and Silvio Micali. Algorand: A secure and efficient distributed ledger. *Theor. Comput. Sci.*, 777:155–183, 2019.
5. CoinMarketCap. Cryptocurrency prices, charts and market capitalizations, 2021. , accessed 2021-02-15. URL: `https://coinmarketcap.com/`.
6. 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 *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020, Valencia, Spain, June 29 – July 2, 2020*, pages 26–38. IEEE, 2020.
7. John R. Douceur. The sybil attack. In *Peer-to-Peer Systems, First International Workshop, IPTPS 2002, Cambridge, MA, USA, March 7-8, 2002, Revised Papers*, pages 251–260, Heidelberg, 2002. Springer.
8. Manu Drijvers, Sergey Gorbunov, Gregory Neven, and Hoeteck Wee. Pixel: Multi-signatures for consensus. In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, 2020. USENIX Association. URL: `https://www.usenix.org/conference/usenixsecurity20/presentation/drijvers`.
9. Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. Proofs of space. In *Advances in Cryptology – CRYPTO 2015 – 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, pages 585–605, 2015.

10 Jose M. Falerio, Sriram K. Rajamani, Kaushik Rajan, G. Ramalingam, and Kapil Vaswani. Generalized lattice agreement. In Darek Kowalski and Alessandro Panconesi, editors, *ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012*, pages 125–134. ACM, 2012.

11 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *JACM*, 32(2):374–382, 1985.

12 David K. Gifford. Weighted voting for replicated data. In *SOSP*, pages 150–162, 1979.

13 Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68, 2017.

14 Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Yvonne Anne Pignolet, Dragos-Adrian Seredinschi, and Andrei Tonkikh. Dynamic Byzantine reliable broadcast. In *OPODIS*, 2020.

15 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. In *PODC*, 2019. `arXiv:1906.05574`.

16 Saurabh Gupta. A Non-Consensus Based Decentralized Financial Transaction Processing Model with Support for Efficient Auditing. Master's thesis, Arizona State University, USA, 2016.

17 Anne-Marie Kermarrec and Maarten van Steen. Gossiping in distributed systems. *SIGOPS Oper. Syst. Rev.*, 41(5):2–7, 2007. `doi:10.1145/1317379.1317381`.

18 Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Advances in Cryptology – CRYPTO 2017 – 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, pages 357–388, 2017.

19 Petr Kuznetsov, Yvonne-Anne Pignolet, Pavel Ponomarev, and Andrei Tonkikh. Permissionless and asynchronous asset transfer [technical report]. *arXiv preprint*, 2021. `arXiv:2105.04966`.

20 Petr Kuznetsov, Thibault Rieutord, and Sara Tucci Piergiovanni. Reconfigurable lattice agreement and applications. In Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller, editors, *23rd International Conference on Principles of Distributed Systems, OPODIS 2019, December 17-19, 2019, Neuchâtel, Switzerland*, volume 153 of *LIPIcs*, pages 31:1–31:17, 2019.

21 Petr Kuznetsov and Andrei Tonkikh. Asynchronous reconfiguration with byzantine failures. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179 of *LIPIcs*, pages 27:1–27:17, 2020.

22 Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed Computing*, 11?(4):203–213, 1998.

23 Tal Malkin, Daniele Micciancio, and Sara Miner. Efficient generic forward-secure signatures with an unbounded number of time periods. In *Advances in Cryptology – Eurocrypt 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 400–417, Amsterdam, The Netherlands, April 28 – May 2 2002. IACR, Springer-Verlag.

24 Tal Moran and Ilan Orlov. Proofs of space-time and rational proofs of storage. *IACR Cryptology ePrint Archive*, 2016:35, 2016.

25 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

26 Michael O Rabin. Randomized byzantine generals. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, pages 403–409. IEEE, 1983.

27 Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *SSS*, pages 386–400, 2011.

28 Jakub Sliwinski and Roger Wattenhofer. ABC: asynchronous blockchain without consensus. *CoRR*, abs/1909.10926, 2019. `arXiv:1909.10926`.

29 Alexander Spiegelman, Idit Keidar, and Dahlia Malkhi. Dynamic reconfiguration: Abstraction and optimal asynchronous solution. In *DISC*, pages 40:1–40:15, 2017.

30 Paul Wackerow, Ryan Cordell, Tentodev, Alwin Stockinger, and Sam Richards. Ethereum proof-of-stake (pos), 2008. accessed 2021-02-15. URL: `https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/`.

31 Gavin Wood. Ethereum: A secure decentralized generalized transaction ledger. White paper, 2015.

A Pseudocode

Verifying and auxiliary functions. We provide implementations of required verifying functions used in *ConfigLA* and *HistLA* in Figure 4. The implementation of function `VerifySender`(tx, σ_{tx}) is not presented there, as it simply consists in verifying that σ_{tx} is a valid signature for tx under $tx.p$'s public key.

We use the following auxiliary functions in the pseudocode to keep it concise:

- `VerifySenders`(txs) – returns *true* iff $\forall \langle tx, \sigma_{tx}\rangle \in txs$: `VerifySender`$(tx, \sigma_{tx}) = true$, otherwise returns *false*;
- `VerifyValues`(vs) – returns *true* if $\forall \langle v, \sigma\rangle \in vs$: `VerifyInputValue`$(v, \sigma) = true$, otherwise returns *false*;
- `ContainsQuorum`$(acks, C)$ – returns *true* if $\exists Q \in quorums(C)$ such that $\forall q \in Q \langle q, \dots\rangle \in acks$, otherwise returns *false*;
- `HighestConf`(h) – returns the highest (w.r.t. $\sqsubseteq$) configuration in given history h;
- `firsts`() – method that, when invoked on a set of tuples S, returns a set of first elements of tuples from set S;
- `ConflictTransactions`(txs) – for a given set of verifiable transactions txs returns a set $conflictTxs$ such that $conflictTxs \subseteq txs$ and for any $\langle tx, \sigma_{tx}\rangle \in conflictTxs$ there exists $\langle tx', \sigma_{tx'}\rangle \in conflictTxs$ such that tx conflicts with tx';
- `CorrectTransactions`(txs) – returns a set of transactions $correctTxs$ such that $correctTxs \subseteq txs$, $\langle tx, \sigma_{tx}\rangle \in correctTxs$ iff $\langle tx, \sigma_{tx}\rangle \in txs$, σ_{tx} is a valid certificate for tx and $\nexists \langle tx', \sigma_{tx'}\rangle \in txs$, such that tx conflicts with tx'.

---

fun VerifyTransactionSet(txs : Set$\langle \mathcal{T}\rangle, \sigma_{txs} : \Sigma_{2^{\mathcal{T}}}$) : Bool
48 **let** $\langle sentTxs, acks_1, acks_2, h, \sigma_h\rangle = \sigma_{txs}$
49 **let** C = HighestConf(h)
50 **return** VerifyHistory(h, σ_h) **and** txs = $sentTxs$.firsts() \ $acks_1$.getConflictTxs().firsts()
51 **and** ContainsQuorum($acks_1, C$) **and** ContainsQuorum($acks_2, C$)
52 **and** $acks_1$.forAll($\langle q, sig, conflictTxs\rangle \Rightarrow$
53 FSVerify($\langle$**ValidateResp**, $sentTxs, conflictTxs\rangle, q, sig, height(C)$))
54 **and** $acks_1$.forAll($\langle q, sig, conflictTxs\rangle \Rightarrow$
55 $conflictTxs$.forAll($\langle tx, \sigma_{tx}\rangle \Rightarrow tx$ conflicts with tx' such that $\langle tx', \sigma_{tx'}\rangle \in conflictTxs$))
56 **and** $acks_2$.forAll($\langle q, sig\rangle \Rightarrow$ FSVerify($\langle$**ConfirmResp**, $acks_1\rangle, q, sig, height(C)$))

fun ABLA.VerifyOutputValue($v : \mathcal{L}, \sigma : \Sigma$) : Bool
57 **if** $\sigma = \perp$ **then return** $v = v_{init}$
58 **let** $\langle values, acks_1, acks_2, h, \sigma_h\rangle = \sigma$
59 **let** C = HighestConf(h)
60 **return** VerifyHistory(h, σ_h) **and** $v = \bigsqcup values$.firsts()
61 **and** ContainsQuorum($acks_1, C$) **and** ContainsQuorum($acks_2, C$)
62 **and** $acks_1$.forAll($\langle q, sig\rangle \Rightarrow$ FSVerify($\langle$**ProposeResp**, $values\rangle, q, sig, height(C)$))
63 **and** $acks_2$.forAll($\langle q, sig\rangle \Rightarrow$ FSVerify($\langle$**ConfirmResp**, $acks_1\rangle, q, sig, height(C)$))

fun VerifyConfiguration(v, σ) = *ConfigLA*.VerifyOutputValue(v, σ)

fun VerifyHistory(v, σ) = *HistLA*.VerifyOutputValue(v, σ)

Figure 4 Verifying functions: code for process p.

Global variables:
$seenTxs, correctTxs, sentTxs$: Set⟨SignedTransaction⟩ = $\{\langle tx_{init}, \bot\rangle\}$
$acks_1 = \emptyset$, $acks_2 = \emptyset$, $status = inactive$

operation Validate($tx : \mathcal{T}, \sigma : \Sigma_{\mathcal{T}}$): Pair⟨Set⟨$\mathcal{T}$⟩, $\Sigma_{2^{\mathcal{T}}}$⟩
64 Request($\{\langle tr, \sigma\rangle\}$)
65 **wait for** ContainsQuorum(acks$_2$, HighestConf($history$))
66 $status \leftarrow inactive$
67 **let** $\sigma = \langle sentTxs, acks_1, acks_2, history, \sigma_{hist}\rangle$
68 **return** $\langle sentTxs$.firsts() $\setminus$ $acks_1$.getConflictTxs().firsts(), $\sigma\rangle$

fun Request(txs: Set⟨SignedTransaction⟩): void
69 $seenTxs \leftarrow seenTxs \cup txs$, $correctTxs \leftarrow$ CorrectTransactions($seenTxs$)
70 $sentTxs \leftarrow correctTxs$, $acks_1 \leftarrow \emptyset$, $acks_2 \leftarrow \emptyset$
71 $seqNum \leftarrow seqNum + 1$, $status \leftarrow requesting$
72 **let** C = HighestConf($history$)
73 **send** ⟨**ValidateReq**, $sentTxs$, $seqNum, C\rangle$ to members(C)

upon ContainsQuorum(acks$_1$, HighestConf($history$)):
74 status $\leftarrow$ *confirming*, **let** C = HighestConf($history$)
75 **send** ⟨**ConfirmReq**, $acks_1, seqNum, C\rangle$ to members(C)

upon receive ⟨**ValidateReq**, $txs, sn, C\rangle$ from q:
76 **wait for** $C = T_p$ **or** HighestConf($history$) $\not\sqsubseteq C$
77 **if** VerifySenders(txs) **then**
78 $seenTxs \leftarrow seenTxs \cup txs$, $correctTxs \leftarrow$ CorrectTransactions($seenTxs$)
79 **if** C = HighestConf($history$) **then**
80 **let** $conflictTxs$ = ConflictTransactions($seenTxs$)
81 **let** sig = FSSign(⟨**ValidateResp**, $txs \cup correctTxs$, $conflictTxs\rangle$, height(C))
82 **send** ⟨**ValidateResp**, $txs \cup correctTxs$, $conflictTxs$, sig, $sn\rangle$ to q

upon receive ⟨**ValidateResp**, $txs, conflictTxs, sig, sn\rangle$ from q **and** $sn = seqNum$ **and** $status = requesting$:
83 **let** C = HighestConf($history$)
84 **let** $isValid$ = FSVerify(⟨**ValidateResp**, txs, $conflictTxs\rangle, q, sig,$ height(C))
and VerifySenders($txs \cup conflictTxs$) **and** $conflictTxs$.forAll($\langle tx, \sigma_{tx}\rangle \Rightarrow$
tx conflicts with some tx' such that $\langle tx', \sigma_{tx'}\rangle \in conflictTxs$)
85 **if** $isValid$ **and** $txs = sentTxs$ **then** $acks_1 \leftarrow acks_1 \cup \{\langle q, sig, conflictTxs\rangle\}$
86 **else if** $isValid$ **and** $sentTxs \subseteq txs$ **then** Request(txs)

upon receive ⟨**ConfirmReq**, $acks$, sn, $C\rangle$ from q:
87 **wait for** $C = T_p$ **or** HighestConf($history$) $\not\sqsubseteq C$
88 **if** C = HighestConf($history$) **then**
89 **let** sig = FSSign(⟨**ConfirmResp**, $acks\rangle$, height(C))
90 **send** ⟨**ConfirmResp**, sig, $sn\rangle$ to q

upon receive ⟨**ConfirmResp**, sig, $sn\rangle$ from q **and** $sn = seqNum$ **and** $status = confirming$:
91 **let** C = HighestConf($history$)
92 **let** $isValidSig$ = FSVerify(⟨**ConfirmResp**, $acks_1\rangle$, q, sig, height(C))
93 **if** $isValidSig$ **then** $acks_2 \leftarrow acks_2 \cup \{\langle q, sig\rangle\}$

Figure 5 Transaction Validation: *TxVal* implementation.

Parameters:
Lattice $(\mathcal{L}, \sqsubseteq)$ and initial value $v_{init} \in \mathcal{L}$
Function `VerifyInputValue`(v, σ_v)
Global variables:
$values$: Set$\langle$Pair$\langle\mathcal{L}, \Sigma\rangle\rangle = \{\langle v_{init}, \bot\rangle\}$
$acks_1 = \emptyset$, $acks_2 = \emptyset$, $status = inactive$

operation Propose($v : \mathcal{L}, \sigma : \Sigma$): Pair$\langle\mathcal{L}, \Sigma\rangle$
94 Refine($\{\langle v, \sigma\rangle\}$)
95 **wait for** ContainsQuorum($acks_2$, HighestConf($history$))
96 $status \leftarrow inactive$
97 **let** $\sigma = \langle values, acks_1, acks_2, history, \sigma_{hist}\rangle$
98 **return** $\langle(\bigsqcup values.\text{firsts}()), \sigma\rangle$

fun Refine(vs : Set$\langle$Pair$\langle\mathcal{L}, \Sigma\rangle\rangle$): void
99 $acks_1 \leftarrow \emptyset$, $acks_2 \leftarrow \emptyset$
100 $values \leftarrow values \cup vs$.filter$(\langle v, \sigma_v\rangle \Rightarrow v \notin values.\text{firsts}())$
101 $seqNum \leftarrow seqNum + 1$, $status \leftarrow proposing$
102 **let** C = HighestConf($history$)
103 **send** $\langle$**ProposeReq**, $values$, $seqNum, C\rangle$ to members(C)

upon ContainsQuorum($acks_1$, HighestConf($history$))
104 status $\leftarrow$ *confirming*, **let** C = HighestConf($history$)
105 **send** $\langle$**ConfirmReq**, $acks_1, seqNum, C\rangle$ to members(C)

upon receive $\langle$**ProposeReq**, vs, sn, $C\rangle$ from q:
106 **wait for** $C = T_p$ **or** HighestConf($history$) $\not\sqsubseteq C$
107 **if** VerifyValues($vs \setminus values$) **then**
108 $values \leftarrow values \cup vs$.filter$(\langle v, \sigma_v\rangle \Rightarrow v \notin values.\text{firsts}())$
109 **let** $replyValues = vs \cup values$.filter$(\langle v, \sigma_v\rangle \Rightarrow v \notin vs.\text{firsts}())$
110 **if** C = HighestConf($history$) **then**
111 **let** sig = FSSign($\langle$**ProposeResp**, $replyValues\rangle$, height(C))
112 **send** $\langle$**ProposeResp**, $replyValues$, sig, $sn\rangle$ to q

upon receive $\langle$**ProposeResp**, vs, sig, $sn\rangle$ from q **and** $sn = seqNum$ **and** $status = proposing$:
113 **let** C = HighestConf($history$)
114 **let** $isValidSig$ = FSVerify($\langle$**ProposeResp**$, vs\rangle$, q, sig, height(C))
115 **if** $isValidSig$ **then**
116 **if** $values = vs$ **then** $acks_1 \leftarrow acks_1 \cup \{\langle q, sig\rangle\}$
117 **else if** VerifyValues($vs \setminus values$) **then** Refine($vs \setminus values$)

upon receive $\langle$**ConfirmReq**, $acks$, sn, $C\rangle$ from q:
118 **wait for** $C = T_p$ **or** HighestConf($history$) $\not\sqsubseteq C$
119 **if** C = HighestConf($history$) **then**
120 **let** sig = FSSign($\langle$**ConfirmResp**, $acks\rangle$, height(C))
121 **send** $\langle$**ConfirmResp**, sig, $sn\rangle$ to q

upon receive $\langle$**ConfirmResp**, sig, $sn\rangle$ from q **and** $sn = seqNum$ **and** $status = confirming$:
122 **let** C = HighestConf($history$)
123 **let** $isValidSig$ = FSVerify($\langle$**ConfirmResp**$, acks_1\rangle$, q, sig, height(C)))
124 **if** $isValidSig$ **then** $acks_2 \leftarrow acks_2 \cup \{\langle q, sig\rangle\}$

Figure 6 Adjustable Byzantine Lattice Agreement: generalized implementation of *ConfigLA* and *HistLA*.