Zef: Low-latency, Scalable, Private Payments

Mathieu Baudet∗
mathieu.baudet@linera.io
Linera

Mahimna Kelkar∗
mahimna@cs.cornell.edu
Cornell University

Alberto Sonnino∗
alberto@mystenlabs.com
MystenLabs & University College London

George Danezis∗
george@mystenlabs.com
MystenLabs & University College London

ABSTRACT
Zef is the first Byzantine-Fault Tolerant (BFT) protocol to support payments in anonymous digital coins at arbitrary scale. Zef achieves its performance by forgoing the expense of BFT consensus and using instead Byzantine Consistent Broadcast as its core primitive. Zef is asynchronous, low-latency, linearly-scalable, and powered by partially-trusted sharded authorities. Zef introduces opaque coins represented as off-chain certificates that are bound to user accounts. In order to hide the values of coins when a payment operation consumes or creates them, Zef uses cryptographically hiding commitments and NIZK proofs. Coins creation and spending are unlinked through the Coconut [28] blind and randomizable threshold anonymous credentials scheme. To control storage costs associated with coin replay prevention, Zef allows safe accounts deletion once the account is deactivated. Our extensive benchmarks on AWS confirm textbook linear scalability and demonstrate a confirmation time under one second at nominal capacity. Compared to existing anonymous payment systems based on a blockchain [22, 34], this represents a latency speedup of three orders of magnitude, with no theoretical limit on throughput.

KEYWORDS
privacy-preserving payments, bft, consensus-less

1 INTRODUCTION
Anonymous payment systems have been an exciting research area in cryptography since Chaum’s seminal work [12] on e-cash. Early e-cash schemes [11, 12, 27] however required a centralized issuer to operate, usually in the form of a trusted commercial bank, which hampered their adoption. In recent years, the advent of networks like Bitcoin has sparked renewed interest in privacy-preserving decentralized payment systems. Several anonymous payment systems [7, 21, 22] are now deployed as permissionless blockchains. Compared to traditional global payment infrastructures (aka. RTGS systems [6]), however, decentralized anonymous payment systems have not yet reached performance levels able to sustain large-scale adoption. For instance, due to high computational costs, only 2% of Zcash [34] transactions typically take full advantage of the privacy features offered by the platform [1]. Recently proposed Byzantine fault-tolerant (BFT) consensus-less protocols such as FastPay [5] forgo consensus and use weaker broadcast-based primitives to offer low-latency transfers of assets. This makes them suitable for deployment as a high-performance sidechain of an existing blockchain, but do not provide any privacy guarantee.

Zef achieves the best of both worlds: the privacy of anonymous payment systems like Zcash and the performance of consensus-less systems like FastPay. Furthermore, Zef authorities can be sharded to accommodate arbitrary throughput and innovates in its storage design solving one of the major costs of current anonymous payment systems. Zef is thus the first linearly-scalable BFT protocol for anonymous payments with sub-second confirmation time.

Zef implements digital coins that are opaque and unlinkable (in short anonymous) by combining: (i) randomized commitments and Non-Interactive Zero-Knowledge (NIZK) proofs (e.g., [21]) to provide opacity, that is, to hide payment values, together with (ii) blind and randomizable signatures (e.g., [28]) to provide unlinkability, meaning that the relation between senders and receivers is hidden. The natural approach to preventing double spending in anonymous systems requires authorities to keep track of all the coins that have been spent. Eventually, maintaining such an ever-growing spent list is bound to create performance and storage bottlenecks. Both Monero and Zcash nevertheless follow this approach, suggesting theoretical limitations in throughput and/or system lifetime. In contrast, in a non-anonymous payment system such as Bitcoin, it is sufficient to keep track of the currently unspent coins—known as Unspent Transaction Outputs (UTXOs). A key contribution of Zef is to remove the need for authorities to maintain a list of spent coins. Zef proposes a new concept of accounts supporting both (i) account-oriented operations, such as transferring ownership, receiving funds repeatedly, and creating unique identifiers, and (ii) UTXO-like operations, such as spending and deactivating accounts. Deactivated accounts may be safely cleaned up from the local storage of each authority. Importantly, all the features introduced by Zef still rely exclusively on client-driven broadcast (reliable or not) and do not require a consensus protocol.
Contributions. (1) We propose Zef—a consensus-less system for scalable account operations. Accounts are addressed by unique, non-replayable identifiers (UIDs) and support a variety of operations such as account creation, deactivation, transparent payments, and ownership transfer. Importantly, all account operations in Zef, including generation of system-wide unique identifiers, are linearly scalable, consensus-free, and require only elementary cryptography (hashing and signing). (2) We describe and analyze the first asynchronous BFT protocol for opaque, unlinkable payments with linear (aka “horizontal”) scalability and sub-second latency. (3) We open-source a prototype implementation of Zef in Rust and evaluate both the scalability and the latency of anonymous payments.

2 BACKGROUND AND RELATED WORK

FastPay. FastPay [5] was recently proposed as a sidechain protocol for low-latency, high-throughput payments in the permissioned model with asynchronous communication.

• Sidechain protocol: FastPay is primarily meant as a scalability solution on top of an existing blockchain with smart contracts (e.g. Ethereum [31]).
• Permissioned model (Byzantine-Fault Tolerance): \( N = 3f + 1 \) replicas called authorities are designated to operate the system and process the clients’ requests. A fixed set of at most \( f \) authorities may be malicious (i.e. deviate from the protocol).
• Low latency: Authorities do not interact with each other (e.g. running a mempool or a consensus protocol). Client operations succeed predictable after a limited number of client/authority round trips. Notably, in FastPay, a single round-trip with authorities suffices to both initiate a payment and obtain a certificate proving that the transfer is final.
• Scalable: Each authority operates an arbitrary number of logical shards, across many physical hosts. By design, each client request is processed by a single shard within each authority. Within an authority, communication between shards is minimal and never blocks a client request.
• Asynchronous communication: Malicious nodes may collude with the network to prioritize or delay certain messages. Progress is guaranteed when messages eventually arrive.

In a nutshell, the state of the Fastpay accounts is replicated on a set of authorities. Each account contains a public key that can authorize payments out, a sequence number and a balance. Account owners authorize payments by signing them with their account key and including the recipient amount and payment value. An authorized payment is sent to all authorities, who countersign it if it contains the next sequence number; there are enough funds; and, it is the first for this account and sequence number. A large enough number (to achieve quorum intersection) of signatures constitute a certificate for the payment. Obtaining a certificate ensures the payment can eventually be executed (finality). Anyone may submit the certificate to the authorities that check it and update the sender account and recipient balance.

FastPay does not rely on State-Machine Replication (SMR) in the sense that it does not require authorities to agree on a single global state—as one could expect from a traditional sidechain. Doing so, the protocol avoids performance considerations commonly associated with SMR. Notably, FastPay does not incur the end-to-end latency cost of gathering, disseminating, and executing large blocks of transactions, a de-facto requirement for high throughput with SMR solutions [13, 17, 29, 32].

Despite the benefits listed above, until now, the FastPay protocol has been limited to transparent payments, that is, without any privacy guarantees. In fact, to ensure fund availability in worst-case scenarios, FastPay requires all past money transfers to be publicly available in clear text. This contrasts negatively with traditional retail payments (e.g. credit cards) where individual transactions remain within a private banking network. Another technical limitation of FastPay is that unused accounts cannot be deleted. In a privacy-sensitive setting where users would never re-use the same account twice, this means that storage cost of authorities would grow linearly with the number of past transactions. Appendix C provides a deeper system comparison of FastPay and Zef.

Existing private payment schemes. Compared to payment channels (e.g. [25]), safety in FastPay and Zef does not require any upper bound on network delays and clients to stay connected (aka, a synchrony assumption [15]). Furthermore, the reliability of the lightning network depends on the existence of pairwise channels, with the success of a payment between two random nodes being at most 70%\(^1\). In contrast, coins delegated to a FastPay instance are always immediately transferable to any recipient that possesses a public key (resp. an account identifier in Zef).

Several privacy-preserving payment systems have been proposed in the past, each based on a blockchain consensus and therefore not linearly scalable: Zcash, based on ZeroCash [7], uses a zero-knowledge proof of set inclusion which is expensive to compute instead of an efficient threshold issuance credential scheme. As a result most transactions are unshielded, leading to a degradation in privacy [18]. Monero [22] uses ring signatures to ensure transactions benefit from a small anonymity set. However, intersections attacks and other transaction tracing heuristics are applicable. This results in an uneven degree of privacy [23].

Furthermore, the safety in those systems requires any upper bound on network delays and clients to stay connected (aka, a synchrony assumption [15]), while Zef operates asynchronously.

3 SYSTEM OVERVIEW

Zef is a consensus-less protocol designed to support high-volume, low-latency payments, both anonymous and transparent, on top of a primary blockchain. Zef introduces a new notion of accounts, indexed by a unique identifier (UID) so that deactivated accounts can have their data safely removed.

Authorities and quorums. We assume a primary blockchain which supports smart contracts (e.g., Ethereum [31]). In a typical deployment, we expect Zef to be “pegged” to the primary chain through a smart contract, thereby allowing transfers of assets in either direction [3]. The Zef smart contract holds the reserve of assets (e.g., coins) and delegates their management to a set of external nodes called authorities. For brevity, in the rest of this paper, we focus on the Zef system and omit the description of transfers between

\(^1\)https://diar.co/volume-2-issue-25/
the primary blockchain and Zef. The mechanics of such transfers is similar to “funding” and “redeeming” operations in FastPay [5].

Zef is meant to be Byzantine-Fault Tolerant (BFT), that is, tolerate a subset of authorities that deviate arbitrarily from the protocol. We assume an asynchronous network that may collude with malicious authorities to deliver messages in arbitrary order. The protocol makes progress when message are eventually delivered.

We assume that authorities have shared knowledge of each other’s signing public keys. Each authority is also assigned a voting power, which indicates how much control the authority has within the system. $N$ denotes the total voting power, while $f$ denotes the power held by adversarial authorities. In the simplest setting where each authority has a voting power of 1 unit, $N$ denotes the total number of authorities and $f$ denotes the number of adversarial authorities tolerated by the system. In general, unequal voting powers may be used to reflect different stakes locked by the authorities on the main blockchain. Similar to standard protocols, we require $0 \leq f < \frac{N}{3}$. The system parameters $N$ and $f$, as well as the public key and voting power of each authority are included in the Zef smart contract during setup.

We use quorum to refer to a set of signatures by authorities with a combined voting power of at least $N - f$. An important property of quorums, called quorum intersection, is that for any two quorums, there exists an honest authority that is present in both.

Cryptographic primitives. We assume a collision-resistant hash function hash as well as a secure public-key signature scheme. Informally, a random commitment $cm = com_{\alpha}(\sigma)$ is an expression that provides a commitment over the value $\sigma$ (in particular, is collision-resistant) without revealing any information on $\sigma$, as long as the random seed $r$ is kept secret. A signing scheme supports blinding and unblinding operations iff (i) a signature of a blinded message $B = \text{blind}(M; u)$ with blinding factor $u$ can be turned into a valid signature of $M$ by computing the expression $\text{unblind}(B; u)$, and (ii) provided that $u$ is a secret random value, an attacker observing $B$ learns no information on $M$.

Blind signatures will be used for anonymous coins in Section 5 together with an abstract notion of Non-Interactive Zero-Knowledge (NIZK) proof of knowledge. We will also further assume that a public key $pk_{\text{all}}$ is set up between authorities in such a way that any quorum of signatures on $M$ may be aggregated into a single, secure threshold signature of $M$, verifiable with $pk_{\text{all}}$. (See Appendix B for a concrete instantiation.)

Clients, requests, certificates, and coins. Clients to the Zef protocol are assumed to know the public configuration of the system (see above) including networking addresses of authorities. Network interactions are always initiated by a client request. We distinguish account-based requests, i.e., those targeting a specific account, noted $R$, from free requests $R'$. In what follows, all requests are account-based unless mentioned otherwise. Free requests will be used for coin creation in Section 5.

As illustrated in Figure 1, clients may initiate a particular operation $O$ on an account that they own as follows: (i) broadcast a request $R$ containing the operation $O$ and authenticated by the client’s signature to the appropriate logical shard of each authority $\alpha$; and (ii) wait for a quorum of responses, that is, sufficiently many answers so that the combined voting power of responding authorities reaches $N - f$.

An authority responds to a valid request $R$ by sending back a signature on $R$, called a vote, as acknowledgment (2). After receiving votes from a quorum of authorities, a client forms a certificate $C$, that is, a request $R$ together with a quorum of signatures on $R$. In the rest of this paper, we identify certificates on a same value $V$ and simply write $C = \text{cert}[V]$ when $C$ is a certificate on $V$. Depending on the nature of value $V$ (e.g., anonymous coins in Section 5) and implementation choices, the quorum of signatures in $C$ may be aggregated into a single threshold signature $\sigma$.

We further distinguish regular operations from locking operations. A request $R$ typically contains a regular operation $O$ meant to be executed once. The certificate $C = \text{cert}[R]$ is meant to be broadcast back to authorities as a confirmation (4), thereby triggering the one execution of $O$ (3) and allowing the account behind $R$ to process further requests. A confirmation certificate $C$ also acts as a proof of finality, that is, a verifiable document proving that the transaction (e.g., a payment) can be driven to success. In the case of payments, recipients obtain and verify the certificate themselves before accepting the payment. In contrast, locking operations cannot be confirmed and executed. If a request $R$ contains a locking operation, a certificate on $R$ is called a locking certificate and written $L = \text{cert}[R]$. Such certificate $L$ serves as a proof that the account is locked and cannot process further account operations.

A third type of certificates associates a coin to an account identifier. Section 5 introduces anonymous coins of the form $A = \text{cert}[(id, cm)]$ for some commitment $cm$ on the value $v$ of the coin. Appendix A also introduces transparent coins $T = \text{cert}[(id, v, r)]$.

Accounts and unique identifiers. Zef accounts are replicated across all authorities. For a given authority $\alpha$, we use the notation $X(\alpha)$ to denote the current view of $\alpha$ regarding some replicated data $X$. At a high level, Zef improves upon the notion of a FastPay account and provides the following important features:

- A Zef account is addressed by an unique identifier (or UID for short) designed to be non-replayable. We use id, id1, ... to denote account identifiers.
- Every account includes an optional public key $pk_{\text{id}}(\alpha)$ to authenticate their owner, if any. When $pk_{\text{id}}(\alpha) = \bot$, the account is said to be inactive.

![Figure 1: Request and execution of a regular account operation](image-url)
Identifiers are created whenever the owner of an account id requests a fresh identifier, for themselves or for a third-party. In practice, we define the next available identifier as the concatenation of the account address id and its current sequence number \( n = \text{next\_sequence}id(\alpha) \).

Zef makes it possible to safely and verifiably transfer the control of an account to another user by changing the key \( pkid(\alpha) \).

An account can be deactivated by setting \( pkid(\alpha) = \perp \). This operation is final and effectively consumes the assets controlled by the account. Because identifiers id are never reused for new accounts, accounts that are deactivated may be optionally deleted by authorities to reclaim storage (see discussion in Section 4).

In addition to the public balance \( \text{balance}id(\alpha) \), the owner of an account id may own some anonymous coins \( \lambda = \text{cert}(id, cm) \).

Sharding and cross-shard queries. In order to scale the processing of client requests, each Zef authority \( \alpha \) may be physically divided in an arbitrary number of shards. Every request \( R \) sent to an account id in \( \alpha \) is assigned a fixed shard as a public function of id and \( \alpha \). If a request requires a modification of another target account id', the shard processing the confirmation of \( R \) in \( \alpha \) must issue an internal cross-shard query to the shard of id'. Cross-shard queries in Zef are asynchronous messages within each authority. They are assumed to be perfectly reliable in the sense that are they never dropped, duplicated, or tampered with.

Transfer of anonymous coins. In Zef, anonymous coins are both (i) unlinkable and (ii) opaque in the sense that during an anonymous payment: (i) authorities cannot see or track users across coins being created; (ii) authorities cannot see the values behind the commitments cm of the coin being consumed or created.

Specifically, as illustrated in Figure 2, the owner of an account id may spend all the anonymous coins \( A_{i}^{n} = \text{cert}(id, cm_{i}) \) linked to id altogether and create new anonymous coins \( A_{j}^{a_{i}} \) using two communication round-trips as follows:

- Provided the recipient accounts \( id_{j}^{a_{i}} \) and desired coin values \( v_{j}^{a_{i}} \), compute fresh random commitments \( cm_{j}^{a_{i}} \) and fresh blinded messages \( B_{j} = \text{blind}((id_{j}^{a_{i}}, cm_{j}^{a_{i}}), u_{j}) \).

- Using the knowledge of the seed and coin value behind each random commitment \( cm_{j}^{a_{i}} \), construct an NIZK proof \( \pi \) that the \( B_{j} \) are well-formed—in particular, that the values \( v_{j}^{a_{i}} \) are non-negative and have the expected sum.

- Broadcast a locking request \( R \) on the account id, containing the hash of the proof \( \pi \) and its public inputs \( A_{i}^{n} \) and \( B_{j} \).

- Aggregate the responses from a quorum of authorities into a locking certificate \( L = \text{cert}[R] \).

- Broadcast a suitable request \( R' \) containing the proof \( \pi \) together with \( L \), the coins \( A_{i}^{n} \), and the blinded messages \( B_{j} \).

- Obtain signature shares from a quorum of authorities for each \( B_{j} \), then unblind and aggregate the signatures shares to form new coins \( A_{j}^{a_{i}} = \text{cert}(id_{j}^{a_{i}}, cm_{j}) \).

- Communicate each new coin \( A_{j}^{a_{i}} \), as well as its commitment seed and value, privately to the owner of \( id_{j}^{a_{i}} \).

Section 5 further elaborates on the creation of coins from public balances \( \text{balance}id(\alpha) \) and supporting multiple source accounts.

The Zef protocol also supports the converse operation consisting in transferring private coins into a public balance. Appendix B provides more details on an efficient cryptographic instantiation of blind signatures and NIZK proofs using the Coconut scheme [26, 28]. For comparison, we also describe a simplified protocol for transparent coins (i.e., without blinding and ZK-proofs) in Appendix A.

Bootstrapping account generation. In Zef, creating a new account requires interacting with the owner of an existing parent account. New identifiers are derived by concatenating the identifier of such a parent account with its current sequence number. This derivation ensures that account identifiers are unique while avoiding the communication overhead and the complexity of distributed random coin generation (see e.g., [10]).

While Zef lets any user derive new identifiers from an existing account that they possess, for privacy reasons, we expect certain entities to specialize in creating fresh identifiers on behalf of other users. We call them brokers. The role of brokers may also be assumed by authorities or delegated to third parties. In what follows, we assume that clients have a conventional way to pick an available broker and regularly create many UIDs for themselves ahead of time. The resulting interactions are summarized in Figure 3. To protect their identity, clients may also wish to interact with brokers and Zef privately, say, using Tor\(^2\) or Nym\(^3\).

The fact that the role of broker can be delegated without risking account safety is an important property of the Zef protocol discussed in Section 4. The solution relies on the notion of certificate

\(^2\)https://www.torproject.org

\(^3\)https://nymtech.net
We now describe the details of the Zef protocol when it comes to transfers of account ownership. We assume that the initial configuration of a Zef system always includes one root account id_α per authority α.

Transfers of account ownership. An interesting benefit of using unique identifiers as account addresses is that the authentication key pk_α(id) can be changed. Importantly, the change of key can be certified to a new owner of the account. This unlocks a number of applications:

- **Anonymous coins.** Anonymous coins (see Section 5) are defined as certificates of the form A = cert{(id, cm)} for some commitment value cm. Spending A to create new coins is an unlinkable operation but it reveals the existence of coins controlled by the account id. Account transfers provide an alternative way to transfer anonymous coins that is linkable (and cannot divide coins) but delays revealing the existence of coins altogether.

- **Lower account-generation latency.** While transferring ownership of an account id requires the same number of messages as creating a new account, we will see in Section 4 that it only involves executing an operation within the shard of id itself (i.e., no cross-shard requests are used). Hence, brokers who wish to provide new accounts with the lowest latency may create a pool of accounts in advance then re-assign UIDs to clients as needed.

### 4 ACCOUNT MANAGEMENT PROTOCOL

We now describe the details of the Zef protocol when it comes to account operations. An upshot of our formalism is that it also naturally generalizes the FastPay transfer protocol [5]. Notably, in Zef, the one-time effect of a transaction consists of one of several possible operations, instead of transparent payments only. Additionally, in order to support deletion of accounts, Zef must handle the fact that a recipient account might be deleted concurrently with a transfer.

#### 4.1 Protocol Messages and Operations

**Unique identifiers.** A unique identifier (UID) is a non-empty sequence of numbers written as id = [n_1, ..., n_k] for some 1 ≤ k ≤ k_max. We use ≪ to denote the concatenation of one number at the end of a sequence: [n_1, ..., n_{k+1}] ≪ [n_1, ..., n_k] :: n_{k+1} (k < k_MAX). In this example, we say that id = [n_1, ..., n_k] is the parent of id ≪ n_{k+1}. We assume that every authority α possesses at least one root identifier of length one: id_α = [n_α] such that the corresponding account is controlled by α at the initialization of the system (i.e., for every honest α’, pk^{id_α(α’)}(α) = α).

**Protocol messages.** A message (Tag, arg_1, ..., arg_m) is a sequence of values starting with a distinct marker Tag and meant to be sent over the network. In the remainder of the paper, we use capitalized names to distinguish message markers from mathematical functions (e.g., hash) or data fields (e.g. pk^{id_α(α)}), and simply write Tag(arg_1, ..., arg_m) for a message.

**Account operations.** A regular operation is a message O meant to be executed once on a main account id′. The regular operations supported by Zef include the following messages:

- OpenAccount(id′, pk′) to activate a new account with a fresh identifier id′ and authentication key pk′—possibly on behalf of another user who owns pk′;
- Transfer(id′, value) to transfer value coins in a transparent way to an existing account id′;
- ChangeKey(pk′) to transfer the ownership of an account;
- CloseAccount to remove the account id.

In Section 5, we introduce additional regular operations as well as a non-regular (i.e., locking) operation Spend that is not meant to be executed but instead locks an account until it is eventually deactivated. We introduce the operation CloseAccount now to mimic locking operations introduced later on.

**Account states.** The state of every authority α includes a map accounts(α) that contains the states of the accounts present in α, indexed by their UID. The state of such an account id includes the following data:

- An optional public key pk^{id_α(α)} registered to control id, as seen before.
- A transparent (i.e., public) amount of coins, noted balance^{id_α(α)} (initially equal to balance^{id_α}(init), where balance^{id_α}(init) is 0 except for some special accounts created at the beginning).
- An integer value, written next_sequence^{id_α}(α), tracking the expected sequence number for the next operation on id. (This value starts at 0.)
- pending^{id_α}(α), an optional request indicating that an operation on id is pending confirmation (the initial value being λ).
- A list of certificates, written confirmed^{id_α(α)}(α), tracking all the certificates C_n that have been confirmed by α for requests issued from the account id. One such certificate is available for each sequence number n (0 ≤ n < next_sequence^{id_α(α)}).
- A second list of certificates, written received^{id_α(α)}(α), tracking all the certificates that have been confirmed by α and involving id as a recipient account.

#### 4.2 Protocol Description

**Operation safety and execution.** Importantly, account operations may require some validation before being accepted. We say that an operation O is safe for the account id in α if one of the following conditions holds:

- O = OpenAccount(id′, pk′) and id′ = id, next_sequence^{id_α(α)};
- O = Transfer(id′, value) and 0 ≤ value ≤ balance^{id_α(α)};
- O = ChangeKey(pk′) or O = CloseAccount (no additional verification).
Algorithm 1 Account operations (internal functions)

1: function InitAccount(id, pk)
2:   pk^id ← pk
3:   next_sequence^id ← 0
4:   balance^id ← balance^id(init) \( \rightarrow \) Set to 0 except for special accounts
5:   confirmed^id ← [ ]
6:   received^id ← [ ]

7: function ValidateOperation(id, n, O)
8:   switch O do
9:     case OpenAccount(id', pk'):
10:        ensure id' = id := next_sequence^id
11:        run Init(id', pk') \( \rightarrow \) Create new account
12:        return balance^id' \( \rightarrow \) Updated balance
13:     case Transfer(id', value):
14:        ensure 0 < value \leq balance^id
15:     case ChangeKey(pk') \( \rightarrow \) CloseAccount:
16:        balance \( \rightarrow \) balance - value \( \rightarrow \) Update sender's balance
17:        do asynchronously \( \rightarrow \) Cross-shard request to id'
18:        if id' \in\ accounts then
19:           run Init(id', \perp) \( \rightarrow \) Reactivate account
20:           balance^id' ← balance^id + value \( \rightarrow \) Receiver's balance
21:           return received^id' \( \rightarrow \) Update receiver's log
22:     case ChangeKey(pk'):
23:        pk^id ← pk' \( \rightarrow \) Update authentication key
24:     case CloseAccount:
25:        pk^id ← \perp \( \rightarrow \) Make account inactive
26:     case SpendAndTransfer(id', value, \( \sigma_1 \ldots \sigma_r, v_1 \ldots v_r, r_1 \ldots r_r \)):
27:        do asynchronously \( \rightarrow \) Cross-shard request to id'
28:        if id' \in\ accounts then
29:           run Init(id', \perp) \( \rightarrow \) Reactivate account
30:           balance^id' ← balance^id + value + \( \sum_i\ v_i \)
31:           received^id' \( \rightarrow \) Update received

When a regular operation O for an account id is confirmed (i.e. a suitable certificate C is received), we expect every authority \( A \) to execute the operation O in following way:

- if \( A = \) OpenAccount(id', pk'), then the authority \( A \) uses a cross-shard request to set pk^id'(a) = pk'; if necessary, a new account id' is created first;
- if \( A = \) Transfer(id', value), the authority subtracts value from balance^id(a) and uses a cross-shard request to add value to

Algorithm 2 Account service (message handlers)

1: function HandleRequest(auth\{R\})
2:   let Execute(id, n, O) \( \rightarrow \) Lock(id, n, O) \( \rightarrow \) Allow regular and locking operations
3:   ensure pk^id \neq \perp \( \rightarrow \) The account must be active
4:   verify that auth\{R\} is valid for pk^id \( \rightarrow \) Check authentication
5:   if pending^id \neq R then
6:     ensure pending^id \neq \perp
7:     ensure next_sequence^id = n
8:   ensure ValidateOperation(id, n, O) = R
9:   pending^id ← R \( \rightarrow \) Lock the account on R
10: return Vote(R) \( \rightarrow \) Success: return a signature of the request

11: function HandleConfirmation(C)
12:   verify that C = cert{R} is valid
13:   match Execute(id, n, O) \( \rightarrow \) Allow regular operations only
14:   ensure pk^id \neq \perp \( \rightarrow \) Make sure the account is active
15:   if next_sequence^id = n then
16:      run ExecuteOperation(id, O, C) \( \rightarrow \) Update sequence number
17:      pending^id \neq \perp \( \rightarrow \) Make the account available again
18:      confirmed^id ← confirmed^id \( \rightarrow \) Log certificate

balance^id(a) \( \rightarrow \) If necessary, the account id' is created first using an empty public key pk^id'(a) = \perp:

- if O = ChangeKey(pk'), then the authority sets pk^id(a) = pk';
- if O = CloseAccount, then the authority remove the account id.

These definitions translate to the pseudo-code in Algorithm 1. The pseudo-code also includes the logging of certificates with confirmed^id(a) and received^id(a) as well as additional operations Spend and SpendAndTransfer that will be described in Section 5.

Account management protocol. We can now describe the protocol steps for executing a regular operation O on an account id:

1. A client knowing the signing key of id and the next sequence number n signs a request R = Execute(id, n, O) and broadcasts it to every authority in parallel, waiting for a quorum of responses.
2. Upon receiving an authenticated request R = Execute(id, n, O), an authority \( A \) must verify that R is authenticated for the current account key pk^id(a), that next_sequence^id(a) = n, that the operation O is safe (see above), and that pending^id(a) \in \{\perp, R\}. Then, it sets pending^id(a) = R and returns a signature on R to the client.
3. The client aggregates signatures into a confirmation certificate C = cert{R}.
4. The client (or another stakeholder) broadcasts Confirm(C).
5. Upon receiving Confirm(C) for a valid certificate C of value R = Execute(id, n, O) when O is a regular operation, each authority verifies that pk^id(a) \neq \perp, next_sequence^id(a) = n, then increments next_sequence^id(a), sets pending^id(a) = \perp, adds C to confirmed^id(a), and finally executes the operation O once (see above).

The corresponding pseudo-code for the service provided by each authority \( A \) is summarized in Algorithm 2. Importantly, inactive accounts, i.e., those accounts id satisfying pk^id(a) = \perp, cannot accept any request (step (2)) or execute any confirmed operation (step (5)). The protocol to submit locking operations (such as Spend
in Section 5) is similar except that there is no final execution step (5), and by convention, the request is written \( R = \text{Lock}(id, n, O) \) and its certificate \( L = \text{cert}[R]\). Note that step (1) above implicitly assumes that all authorities are up-to-date with all past certificates. In practice, a client may need to provide each authority with missing confirmation certificates for past sequence numbers. (See also “Liveness considerations” below.)

### 4.3 Security Properties

**Agreement on account operations.** When it comes to the operations executed from one account id, the Zef protocol guarantees that authorities execute the same sequence of operations in the same order. Indeed, the quorum intersection property entails that two certificates \( C \) and \( C' \) must contain a vote by a same honest authority \( \alpha \). If they concern the same account id and sequence number \( n \), the verification by \( \alpha \) in step (2) above and the increment of next_sequence\( ^id(\alpha) \) in step (5) implies that \( C \) and \( C' \) certifies the same (safe) request \( R \).

It is easy to see by induction on the length of \( id = [n_1, \ldots, n_k] \) that each authority can only execute certified operations for a given id by following the natural sequence of sequence numbers (i.e., next_sequence\( ^id(\alpha) = 0, 1, \ldots \)). Indeed, by the induction hypothesis (resp. by construction for the base case), at most one operation of the form \( O = \text{OpenAccount}(id, ..) \) can ever be executed by \( \alpha \) on the parent account of \( id \) (resp. as part of the initial setup if id has no parent). We also note that due to the checks in step (5), no operation can be executed from id while the account id is locally absent or if pk\( ^id(\alpha) = 1 \). Account creation executed by the parent account of id is the only way for pk\( ^id(\alpha) \) to be updated from an empty value \( 1 \). Therefore, if an account id is deleted by \( \alpha \) due to an operation CloseAccount, it is necessarily so after \( \text{OpenAccount}(id, ..) \) was already executed once. The account id may be created again by some operation Transfer(id, value) after deletion, but since \( \text{OpenAccount}(id, ..) \) is no longer possible, pk\( ^id(\alpha) \) will remain empty, thus no more operations will be executed from id at this point. Therefore, due to the checks in step (5), the operations executed on id while pk\( ^id(\alpha) \neq 1 \) follows the natural sequence of sequence numbers.

**Agreement on account states.** Let \( \alpha \) be authority and id be an account such that pending\( (\alpha) = 1 \) and \( \alpha \) has not executed an operation CloseAccount on id yet. We observe that the state of id seen by \( \alpha \) is a deterministic function of the following elements:

- the sequence of operations previously executed by \( \alpha \) on id, that is, the content of confirmed\( ^id(\alpha) \), and
- the (unordered) set of operations previously executed by \( \alpha \) that caused a cross- shard request to id as recipient, that is, the content of received\( ^id(\alpha) \).

Indeed, operations issued by id are of the form \( \text{ChangeKey}(pk) \), \( \text{Transfer}(\ldots, \text{value}^B) \), and \( \text{OpenAccount}(\ldots) \). Similarly, possible operations received by id are of the form \( \text{OpenAccount}(id, pk) \) and \( \text{Transfer}(id, \text{value}^B) \). We can determine the different components of the account id as seen by \( \alpha \) as follows:

- next_sequence\( ^id(\alpha) \) will be the size of confirmed\( ^id(\alpha) \);
- pk\( ^id(\alpha) \) will be the last key set by \( \text{OpenAccount}(id, pk) \) (or an equivalent initial setup for special accounts) then subsequent \( \text{ChangeKey}(pk) \) operations, and otherwise pk\( ^id(\alpha) = 1 \);
- balance\( ^id(\alpha) = \sum_i \text{value}_i^B + \sum_j \text{value}_j^{out} + \text{balance}^id(\text{init}) \), where balance\( ^id(\text{init}) \) denotes a possibly non-zero initial balance for some special accounts. (In the presentation of FastPay [5], additionally terms account for external transfers with the primary blockchain in replacement of balance\( ^id(\text{init}) \).)

The agreement property on account operations (see above) entails that whenever two honest authorities have executed the same operations, they must also agree on the current set of active accounts and their corresponding states. In other words, if for all id, confirmed\( ^id(\alpha) = \text{confirmed}_id(\alpha') \), then for all id such that pk\( ^id(\alpha) \neq 1 \) or pk\( ^id(\alpha') \neq 1 \), we have next_sequence\( ^id(\alpha) = \text{next} \_ \text{sequence}_id(\alpha') \), pk\( ^id(\alpha) = \text{pk}_id(\alpha') \), and balance\( ^id(\alpha) = \text{balance}_id(\alpha') \).

In particular, similar to the proof of FastPay [5], balance\( ^id(\alpha) \geq 0 \) holds for every id once every certified operations has been executed. Indeed, consider an honest authority which accepted to vote at step (2) for the last transfer \( \text{Transfer}(\ldots, \text{value}^B) \) from id.

**Liveness considerations.** Zef guarantees that conforming clients may always (i) initiate new valid operations on their active accounts and (ii) confirm a valid certificate of interest as a sender or as a recipient. We note that question (i) is merely about ensuring that the sequence number of an active sender account can advance after a certificate is formed at step (3). This reduces to the question (ii) of successfully executing step (5) for any honest authority, given a valid certificate \( C \).

If the client, an honest authority \( \alpha \), or the network was recently faulty, it is possible that (a) the sender account id may not be active yet at \( \alpha \), or (b) the sequence number of id may be lagging behind compared to the expected sequence number in \( C \). In the latter case (b), similarly to Fastpay, the client should replay the previously confirmed certificates \( C_1 \) of the same account—defined as \( C_1 \in \text{confirmed}_id(\alpha') \) for some honest \( \alpha' \)—in order to bring an authority \( \alpha \) to the latest sequence number and confirm \( C \). In the case (a) where \( id \) is not active yet at \( \alpha \), the client must confirm the create certificate \( C_0 \) of id issued by the parent account id' = parent(id). This may recursively require confirming the history of \( C' \). Note however that this history is still sequential (i.e. there is at most one parent per account) and the number of parent creation certificates is limited by \( k_{\text{MAX}} \).

Importantly, a certificate needs only be confirmed once per honest authority on behalf of all clients. Conforming clients who initiate transactions are expected to persist past certificates locally and pro-actively share them with all responsive authorities.

In practice, the procedure to bring authorities up-to-date can be implemented in a way that malicious authorities that would always request the entire history do not slow down the protocol. (See the discussion in FastPay [5], Section 5.)

### 4.4 Accounts Deletion and Delegation

**Deletion of deactivated accounts.** We have seen that once deactivated, an account id plays no role in the protocol and that id will
never be active again. Therefore, it is always safe for an authority \( a \) to delete a once-deactivated account.

A simple strategy for an authority \( a \) to take advantage of this fact and reclaim some local storage consists in deleting the account id whenever \( \text{pk}_i^\text{id}(a) \) changes its value to \( \perp \). We note however that this strategy is only a best effort. Effectively reclaiming the maximum amount of storage available in the system requires addressing two questions:

1. If an honest authority \( a \) deletes id, how to guarantee that the account is not recreated later by \( a \).
2. If an honest authority \( a \) deletes id, how to guarantee that every other honest authority \( a' \neq a \) eventually deletes id;

Regarding (1), when a cross-shard request is received for an operation Transfer(id, value), the current version of the protocol may indeed require re-creating an empty account id. This storage cost can be addressed by modifying Zef so that an authority \( a \) does not re-create id (or quickly deletes it again) if it determines that no operation \( \text{O} = \text{OpenAccount}(\text{id},...) \) can occur any more. This fact can be tested in background using \( |\text{id}| \leq k_{\text{max}} \) cross-shard queries. Indeed, consider the opposite fact: an inactive account id = id_0 : \( n \) can become active in \( a \) if it holds that (i) next_sequence\( ^{\text{id}}(a) \leq n \) and (ii) the parent account id_0 is either active or can become active.

Regarding (2), we note that sending and receiving clients in payment operations have an incentive to fully disseminate the confirmation certificates to all authorities—rather than just a quorum of them—whenever possible. (The incentives are respectively to fully unlock the sender’s account and to fully increase the receiver’s balance in the eventuality of future unresponsive authorities.) However, such an incentive does not exist in the case of the CloseAccount operation (resp. the message CreateAnonymousCoins of Section 5). Therefore, in practical deployments of Zef, we expect authorities to either communicate with each other a minima in background, or to incentivize clients to continuously disseminate missing certificates (resp. missing free queries \( R' \) of Section 5) between authorities.

Safety of delegated account generation. In the eventuality of malicious brokers, a client must always verify the following properties before using a new account id’:

- The certificate \( C \) returned by the broker is a valid certificate \( C = \text{cert}[R] \) such that \( R = \text{Execute}(\text{id, n, O}) \) and \( O = \text{OpenAccount}(\text{id'},\text{pk}) \) for the expected public key pk. (Under BFT assumption, this implies \( \text{id'} = \text{id} : n \).)
- If the client did not pick a fresh key pk, it is important to also verify that \( C \) is not being replayed.

To avoid revealing which accounts they own, clients should use a fresh authentication key pk every time and consider communicating with brokers privately (e.g. over Tor). How a client may anonymously purchase their first UID from a broker raises the interesting question of how to effectively bootstrap a fully anonymous payment system. (For instance, a certain number of fresh key-less accounts could be given away regularly for anyone to acquire and reconfigure them over Tor before receiving their very first anonymous payment.) Yet, we should point out that revealing the owner of an account does not contradict the confidentiality and the unlikability properties of the anonymous coins described in Section 5.

## 5 ANONYMOUS PAYMENTS

We now describe the Zef protocol for anonymous payments using generic building blocks. In particular, we use a blind signature scheme, random commitments, and Zero-Knowledge (ZK) proofs in a black-box way. A more integrated realization of the protocol suitable for an efficient implementation is proposed in Appendix B.

### 5.1 Protocol Description

Anonymous coins. An anonymous coin is a triplet \( A = (\text{id, cm, }\sigma) \) where \( \text{id} \) is the unique identifier (UID) of an active account, \( \text{cm} \) is a random commitment on a value \( v \in [0, v_{\text{max}}] \) using some randomness \( r, \text{denoted cm = com}_i(\sigma) \), and \( \sigma \) is a threshold signature from a quorum of authorities on the pair \( (\text{id, cm}) \). Following the notations of Section 3, an anonymous coin \( A \) can also be seen as a certificate \( A = \text{cert}(\text{id, cm}) \). To spend a coin, a client must know the value \( v \), the randomness \( r \), and the authentication key controlling id. Importantly, authorities do not need to store commitments cm or signatures \( \sigma \) but only manage the active accounts id. Looking ahead, spending the same coin twice will be prevented by making sure that the id is only spent once, or in other words, by removing id from the list of active accounts.

New account operation. We extend the account operations of Section 4 with a locking operation \( \text{O} = \text{Spend}(\text{value, hash(P)}) \) meant to be included in a request Lock(id, \( n, O \)) in order to prepare some payment \( P \), withdraw value coins publicly, and eventually deactivate the corresponding account. (See details below and algorithm 1.)

Creating anonymous coins. Suppose that a user owns \( t \) coins \( A_i^{\text{in}} = (\text{id}_i^{\text{in}}, \text{cm}_i^{\text{in}}, \sigma_i^{\text{in}}) \) (\( 1 \leq i \leq t \)) such that all \( \text{id}_i^{\text{in}} \) are registered with the authorities as active, the \( \text{cm}_i^{\text{in}} \) are \( t \) mutually distinct random commitments, and \( \sigma_i^{\text{in}} \) is a blind randomizable signature on \( (\text{id}_i^{\text{in}}, \text{cm}_i^{\text{in}}) \) supporting threshold issuance [28]. Let \( \text{cm}_i^{\text{in}} \) be the value of the coin \( A_i^{\text{in}} \). Let \( v \geq 0 \) be a value that the user wishes to withdraw publicly from the account \( \text{id}_i^{\text{in}} \).

Importantly, we require commitments \( \text{cm}_i^{\text{in}} \) to be distinct but not the identifiers \( \text{id}_i^{\text{in}} \). This allows several coins to be linked to the same account — assuming that their owner agrees to spend them simultaneously. In practice, we expect \( \sum_i v_i \) to be the entire balance of the set of accounts \( \{\text{id}_i^{\text{in}}\} \) about to be closed, to the user’s knowledge.

We define the total input value of the transfer as \( v = \sum_i (v_i^{\text{in}} + \text{value}) \). To spend the coins into \( d \) new coins with values \( v_j^{\text{out}} \) (\( 1 \leq j \leq d \)) such that \( \sum_j v_j^{\text{out}} = v \), the sender requests an UID \( \text{id}_j^{\text{out}} \) from each recipient, then proceeds as follows:

1. First, the sender constructs a payment description \( P \) as follows:
   - For \( 1 \leq j \leq d \), sample randomness \( r_j^{\text{out}} \) and set \( \text{cm}_j^{\text{out}} = \text{com}_j^{\text{out}}(v_j^{\text{out}}) \).
   - For \( 1 \leq j \leq d \), sample random blinding factor \( u_j \) and let \( B_j = \text{blind}((\text{id}_j^{\text{out}}, \text{cm}_j^{\text{out}}; u_j)) \).
For every distinct id ∈ {id\textsuperscript{in}, id\textsuperscript{out}}, the sender collects a quorum of signatures for each id\textsuperscript{in} such that
\begin{itemize}
  \item cm\textsuperscript{in} = \text{com}_{\text{id}\textsuperscript{in}}(v\textsuperscript{in}) and cm\textsuperscript{out} = \text{com}_{\text{id}\textsuperscript{out}}(v\textsuperscript{out})
  \item B\textsubscript{j} = \text{blind}((id\textsuperscript{out}, cm\textsuperscript{out}; u\textsubscript{j})
  \item \(\sum_i v\textsuperscript{in}_i + \sum_i v\textsuperscript{out}_i = \sum_j v\textsuperscript{out}_j\)
\end{itemize}
Each value v\textsuperscript{in} and v\textsuperscript{out} is in [0, \theta_{\text{max}}].

(d) Let \(P = (\pi, A_1, \ldots, A_{\ell}, \text{value}_1, \ldots, \text{value}_t, B_1 \ldots B_d)\).

(2) For every distinct id ∈ {id\textsuperscript{in}}, the sender broadcasts an authenticated request \(R_i = \text{Lock}(\text{id}, n, O)\) where \(O = \text{Spend}(\text{value}, \text{hash}(P))\), \(n\) is the next available sequence number for the account id, and value = \(\sum_i \text{id}\textsuperscript{in}_{i=\text{id}} \text{value}_i\).

(3) Upon receiving an authenticated request \(R = \text{Lock}(\text{id}, n, \text{Spend}(\text{value}, h))\) from the owner of id, an authority \(\alpha\) verifies that next_sequence\textsuperscript{id}(\(\alpha\)) = \(n\), pending\textsuperscript{id}(\(\alpha\)) = 1, and \(0 \leq \text{value} \leq \text{balance}\textsuperscript{id}(\(\alpha\)). Then, \(\alpha\) sets pending\textsuperscript{id}(\(\alpha\)) = \(R\) and responds with a signature on \(R\).

(4) The sender collects a quorum of signatures for each \(R_i\) sent above, thus forming a certificate \(L_i = \text{cert}[R_i]\) for every \(i\). It now sends a free request \(R' = \text{CreateAnonymousCoins}(P, L_1, \ldots, L_\ell)\) to all authorities and waits for a quorum of responses.

(5) Upon receiving a free request of the form \(R' = \text{CreateAnonymousCoins}(P, L_1, \ldots, L_\ell)\) where
\[P = (\pi, A_1, \ldots, A_{\ell}, \text{value}_1, \ldots, \text{value}_t, B_1 \ldots B_d)\]
and \(A_i = (id\textsuperscript{in}_i, \text{cm}_i, \sigma_i)\), each authority \(\alpha\) verifies the following:
\begin{itemize}
  \item The values cm\textsuperscript{in} are mutually distinct.
  \item For every \(i\), \(\sigma_i\) is a valid signature on \((id\textsuperscript{in}_i, \text{cm}_i)\).
  \item Every \(L_i\) is a valid certificate for a request of the form \(R_i = \text{Lock}(\text{id}, n, O)\) and such that we have \(id = id\textsuperscript{in}_i, O = \text{Spend}(\text{value}, \text{hash}(P))\), and value = \(\sum_i id\textsuperscript{in}_{i=\text{id}} \text{value}_i\).
  \item The proof \(\pi\) is valid.
\end{itemize}
The authority then deactivates each account id\textsuperscript{in}_i (if needed), and responds with \(d\) signature shares, one for each \(B_j\) = blind((id\textsuperscript{out}_j, cm\textsuperscript{out}_j); u\textsubscript{j}).

(6) For every \(j\), the sender finally combines the signature shares received by a quorum of authorities, then uses unblind to obtain a signature \(\sigma_j\textsuperscript{out}\) on \((id\textsuperscript{out}_j, cm\textsuperscript{out}_j)\).

(7) The \(j^{th}\) recipient receives \((id\textsuperscript{out}_j, cm\textsuperscript{out}_j, \sigma_j\textsuperscript{out}, r\textsubscript{jout}, \sigma_j\textsuperscript{out})\). She verifies that the values and UIDs are as expected, that the commitments cm\textsuperscript{out} are mutually distinct, that the signatures \(\sigma_j\textsuperscript{out}\) are valid.

We note that finality is achieved as soon as the request \(R'\) is formed by the sender. Importantly, the operation Spend is a locking operation (Section 4), meaning that a certificate \(L_i = \text{cert}[R_i]\) alone is not sufficient to be executed and deactivate the account id\textsuperscript{in}_i.

Appendix B provides a concrete instantiation of this scheme.

The corresponding pseudo-code for coin creation is presented in Algorithm 3.

### Algorithm 3: Coin creation service

```plaintext
Algorithm 3 Coin creation service

1. function HANDLECOINCREATIONREQUEST(R')
   2. let CreateAnonymousCoins(P, L_1, \ldots, L_\ell) = R'
   3. let (\pi, A_1, \ldots, A_{\ell}, \text{value}_1, \ldots, \text{value}_t, B_1 \ldots B_d) = P
   4. let (id, cm, \sigma) = A_j for each i = 1..\ell
   5. for i = 1..\ell
      6.    ensure cm \notin \{cm_k\}_{k<i}
      7.    ensure \sigma_i = \text{cert}([id, cm]) is a valid coin signature
      8.    ensure \text{store}_{i} = \text{cert}[R_i] is a valid certificate
      9.    match \text{lock}(\text{id}, n, \text{Spend}(\text{value}, h)) = R_i
      10.   ensure h = \text{hash}(P)
      11.   ensure id = id_i
      12.   ensure value = \sum_i \text{id}_{i=\text{id}} \text{value}_i
      13.   let v = \sum_i \text{value}_i
      14.   verify the ZK-proof \(\pi\) on inputs \((cm_1, \ldots, cm_k, v, B_1 \ldots B_d)\)
      15.   for i = 1..d
      16.     let \(s_j = \text{SignShare}(B_j)\) for each j = 1..d
      17.     return \((s_1, \ldots, s_d)\) as a blinded signature for each output

\(id\) is a new (regular) account operation
\(O = \text{SpendAndTransfer}(id', \text{value}, \sigma_1, \ldots, \sigma_t, u_1, \ldots, u_t, r_1, \ldots, r_t)\)
meant to be included in a request \(R = \text{Execute}(id, n, O)\). Following the framework of Section 4:
\begin{itemize}
  \item \(O\) is safe iff id ∈ accounts(\(\alpha\)), \(0 \leq \text{value} \leq \text{balance}\textsuperscript{id}(\(\alpha\)), and every signature \(\sigma_i\) is a valid signature of a distinct pair \((id, cm_i)\) with \(cm_i = \text{com}_{\text{cm}}(u_i)\).
  \item Upon receiving a valid confirmation certificate \(\text{C} = \text{cert}[R]\), the execution of \(O\) consists in removing the account id and sending a cross-shard request to add the value \(v = \text{value} + \sum_i u_i\) to balance\textsuperscript{id}\textsuperscript{(\(\alpha\))} (possibly after creating an empty account id').
\end{itemize}
The pseudo-code for redeeming operations is presented in Algorithm 1.

#### 5.2 Security Properties

**Safety of the protocol.** We say that an account id is locked-or-deleted iff there exists a valid certificate of the form
\(L = \text{cert}[R]\) where \(R = \text{Lock}(id, n, O)\) for some locking operation \(O\), or \(C = \text{cert}[R]\) where \(R = \text{Execute}(id, n, O)\) such that \(O\) is an operation SpendAndTransfer or CloseAccount.

If \(O\) is a transfer operation we write value(O) for the value of the transfer, source(O) for the main account, recipient(O) for the recipient account. By extension, we write value(C) for the value of a valid confirmation certificate containing such operation \(O\). Summations over certificates range over all valid certificates for distinct requests or coins.

We define the spendable value of an account id as spendable\textsuperscript{id} = 0 if id is locked-or-deleted, and otherwise:
\[
\text{spendable} = \text{balance} + \sum_{\text{recipient}(C)=\text{id}} \text{value}(C) - \sum_{\text{source}(C)=\text{id}} \text{value}(C) + \sum_{\text{id}(A)=\text{id}} \text{value}(A)
\]
We note that if an authority $\alpha$ is perfectly up to date with the certificates related to $id$ and the account id is not locked-or-deleted:

$$spendable^{id} = balance^{id}(\alpha) + \sum_{id(A)=id} value(A)$$

The protocol for account operations entails that a locked-or-deleted account id can never produce new certificates $C$ (in particular with $\text{recipient}(C) = id' \neq id$). Indeed, the case of account deletion was analyzed in Section 4. Similarly, in the case of a locking request $r = \text{Lock}(id, n, O)$, at least a quorum of authorities $\alpha$ have processed $R$. From this point on, the account id held by such authority $\alpha$ can only (i) process the same locking request $R$ and return the same vote, or (ii) become deleted or inactive. By quorum intersection, no new certificate can be produced.

By inspection of the protocol, we deduce that the total spendable value over all accounts $S = \sum_{id} spendable^{id}$ never increases during regular account operations, coin creation, and redeeming of anonymous coins:

- Regular account operations have been studied in Section 4.
- Redeeming coins with SpendAndTransfer increases the balance of a recipient but changes the account to be locked-or-deleted, effectively "burning" at least an equivalent amount. (To this effect, note that the definition of spendable counts distinct coins and so does the protocol.)
- Creating coins with a free request $R^\star = \text{CreateAnonymousCoins}(P, L_1, \ldots, L_\ell)$ requires changing all the source accounts in $L_1, \ldots, L_\ell$ to be locked-or-deleted. Importantly, $L_\ell$ contains a commit of $P$ and cannot re-used for another payment $P' \neq P$. We also note that replaying $R^\star$ produces the same coins $A_J$ and does not increase $S$.

Liveness considerations. For coin creation, we used $\text{hash}(P)$ instead of $P$ in each $R_i$ both to minimize communication and for liveness. Indeed, it is important for the initiator to keep $P$ secret before collecting all the certificates $L_i$. We note that otherwise an attacker knowing $P$ may intercept the requests $R_i$, use them to obtain the certificates $L_i$, then send $R^\star$ to kill the input accounts $id^{in}$ before the sender has a chance to replay the requests $R_i$. Depending on the archiving policy for $L_i$, this may leave the sender unable to replay $R^\star$ to obtain the signature shares of step (5).

Privacy properties. The protocol to create anonymous coins guarantees the following privacy properties.

- Opacity: Except for the ZK proofs $\pi$, the coin values under the commitments $cm^{in}$ and $cm^{out}$ are never communicated publicly.
- Unlinkability: Assuming that the sender during coin creation is honest, authorities cannot trace back to the origin of an anonymous coin when it is spent.

Regarding unlinkability, we note indeed that the receiver information $id^{out}_j$ and $cm^{out}_j$ are only communicated to authorities in blinded form. Besides, after unblinding, the threshold signature $\sigma_j$ does not depend on values controlled by authorities, therefore is not susceptible to tainting.

To prevent double spending, the protocol must reveal the identifiers $id^{in}_j$ of the coins being spent. This means that the sender who initially created the coins linked to $id^{in}_j$ must be trusted for unlinkability to hold. To mitigate this concern, it is recommended that receivers of anonymous coins re-send their coins anonymously to themselves sometime before spending their coins.

In general, the account identifiers $id^{out}_j$ needed for anonymous transfers should be obtained ahead of time and using anonymity solution such as Tor. For simplicity, we have only described account authentication based on a transparent public key $pk^{id}(\alpha)$. In practice, using a random commitment of a public key and a proof of knowledge of the corresponding secret would allow using the same authentication key for several accounts without degrading privacy.

6 IMPLEMENTATION

We implement a multi-core, multi-shard Zef authority in Rust, based on the existing FastPay codebase\footnote{https://github.com/novifinancial/fastpay} which implements the Byzantine reliable broadcast needed. In particular, we were able to re-use modules based on Tokio\footnote{https://tokio.rs} for asynchronous networking and cryptographic modules based on ed25519-dalek\footnote{https://github.com/dalek-cryptography/ed25519-dalek} for elliptic-curve-based signatures. For simplicity, data-structures in our Zef prototype are held in memory rather than persistent storage. The core of Zef is idempotent to tolerate retries in case of packet loss. Each authority shard is a separate native process with its own networking and Tokio reactor core. We are open-sourcing Zef along with any measurements data to enable reproducible results\footnote{https://github.com/novifinancial/fastpay/tree/extensions/benchmark_scripts}.

We have chosen Coconut credentials\footnote{https://github.com/nymtech/coconut} to implement the blind randomizable threshold-issuance signatures $\sigma^{in}_j$ and $\sigma^{out}_j$ of Section 5. Zero-Knowledge proofs are constructed using standard sigma protocols, made non-interactive through the Fiat-Shamir heuristic\footnote{https://github.com/dalek-cryptography/bulletproofs}. As a result, our implementation of Zef assumes the hardness of LRSW\footnote{https://github.com/novifinancial/fastpay/tree/extensions/bulletproofs} and XDH\footnote{https://github.com/novifinancial/fastpay/tree/extensions/benchmark_scripts} (required by Coconut), and the existence of random oracles\footnote{https://github.com/dalek-cryptography/bulletproofs}. Appendix B presents this protocol in details. Our implementation of Coconut is inspired from Nym’s\footnote{https://github.com/novifinancial/fastpay/tree/extensions/bulletproofs} and uses the curve BLS12-381\footnote{https://github.com/novifinancial/fastpay/tree/extensions/bulletproofs} as arithmetic backend.

We have implemented all range proofs using Bulletproofs\footnote{https://github.com/novifinancial/fastpay/tree/extensions/bulletproofs} as they only rely on the discrete logarithm assumption (which is implied by XDH) and do not require a trusted setup. Unfortunately, we couldn’t directly use Dalek’s implementation of Bulletproofs\footnote{https://github.com/dalek-cryptography/bulletproofs} as it uses Ristretto\footnote{https://github.com/novifinancial/fastpay/tree/extensions/benchmark_scripts} as arithmetic backend. Ristretto is incompatible with Coconut (which requires a pairing-friendly curve). Therefore, we have modified Dalek’s implementation to use curve BLS12-381. This required significant effort as the curve operations are deeply baked into the library. Our resulting library is significantly slower than Dalek’s for two reasons: operations over BLS12-381 are slower than over Ristretto, and we couldn’t take advantage of the parallel formulas in the AVX2 backend present in the original library. We are open-sourcing our Bulletproof implementation over BLS12-381\footnote{https://github.com/novifinancial/fastpay/tree/extensions/bulletproofs}.

7 EVALUATION

We now present our evaluation of the performance of our Zef prototype based on experiments on Amazon Web Services (AWS).
Our focus was to verify that (i) Zef achieves high throughput even for large committees, (ii) Zef has low latency even under high load and within a WAN, (iii) Zef scales linearly when adding more shards, and (iv) Zef is robust when some parts of the system inevitably crash-fail. Note that evaluating BFT protocols in the presence of Byzantine faults is still an open research question [4].

We deployed a testbed on AWS, using m5.8xlarge instances across 5 different AWS regions: N. Virginia (us-east-1), N. California (us-west-1), Sydney (ap-southeast-2), Stockholm (eu-north-1), and Tokyo (ap-northeast-1). Authorities were distributed across those regions as equally as possible. Each machine provided 10Gbps of bandwidth, 32 virtual CPUs (16 physical core) on a 2.5GHz, Intel Xeon Platinum 8175, 128GB memory, and ran Linux Ubuntu server 20.04. We selected these machines because they provide decent performance and are in the price range of “commodity servers”.

In the following sections, each measurement in the graphs is the average of 2 independent runs, and the error bars represent one standard deviation\(^{12}\). We set one benchmark client per shard (collocated on the same machine) submitting transactions at a fixed rate for a duration of 5 minutes.

### 7.1 Regular Transfers

We benchmarked the performance of Zef when making a regular transfer as described in Section 4. When referring to latency in this section, we mean the time elapsed from when the client submits the request (Step 1 in Figure 1) to when at least one honest authority processes the resulting confirmation certificate (Step 5 in Figure 1). We measured it by tracking sample requests throughout the system.

**Benchmark in the common case.** Figure 4 illustrates the latency and throughput of Zef for varying numbers of authorities. Every authority ran 10 collocated shards (each authority runs thus a single machine). The maximum throughput we observe is 20,000 tx/s for a committee of 10 nodes, and lower (up to 6,000 tx/s) for a larger committee of 50. This highlights the importance of sharding to achieve high-throughput. This reduction is due to the need to transfer and check transfer certificates signed by 2\(f\) + 1 authorities; increasing the committee size increases the number of signatures to verify since we do not use threshold signatures for regular transfers.

**Scalability.** Figure 5 shows the maximum throughput that can be achieved while keeping the latency under 250ms and 300ms.

---

\(^{12}\)Error bars are absent when the standard deviation is too small to observe.
We benchmarked the performance of Zef when spending two opaque coins into two new ones, as described in Section 5. When referring to latency in this section, we mean the time elapsed from when the client submits the request (Step 2 in Figure 2) to when it assembles the new coins (Step 6 in Figure 2). We measured it by tracking sample requests throughout the system.

**Microbenchmarks.** We report on microbenchmarks of the single-CPU core time required execute the cryptographic operations. Table 1 displays the cost of each operation in milliseconds (ms); each measurement is the result of 100 runs on a AWS m5.8xlarge instance. The first 3 rows respectively indicate the time to (i) produce a coin creation request meant to spend two opaque coins into two new ones, (ii) verify that request, and (iii) issue a blinded coin share. The last 3 rows indicate the time to unblind a coin share, verify it, and aggregate 3 coin shares into an output coin. The dominant CPU cost is on the user when creating a coin request (438.35ms), which involves proving knowledge of each input coins (1 Bulletproof per coin). However, verifying coin requests (142.31ms) is also expensive: it involves verifying the input coins (1 pairing check per input coin) and the output coins request (1 Bulletproof per coin). Issuing a blinded coin share (1 Coconut signature per output coin) is relatively faster (4.90ms). Unblinding (3.37ms), verifying (9.62ms) and aggregating (1.70ms) coin shares take only a few milliseconds. These results indicate that a single core shard implementation may only settle just over 7 transactions per second—highlighting the importance of sharding to achieve high-throughput.

**Benchmark in the common case.** Figure 7 illustrates the latency and throughput of Zef for varying numbers of authorities. Every authority runs 10 collocated shards. The performance depicted in Figure 7 (anonymous payments) are 3 order of magnitude lower than those depicted in Figure 4 (regular transfers); this is due to the expensive cryptographic operations reported in Table 1. We observe virtually no difference between runs with 10, 20, 30, or even 50 authorities: Zef can process about 50 tx/s while keeping latency under 1s in all configurations. This highlights that anonymous payments operations are extremely CPU intensive and that bandwidth is far from being the bottleneck.

**Scalability.** Figure 8 shows the maximum throughput that can be achieved while keeping the latency under 500ms and 1s. The committee was composed by 4 authorities each running a data-center,

![Figure 7: Throughput-latency graph for anonymous coins. WAN measurements with 10, 20, 30 authorities; 10 collocated shards per authority. No faulty authorities.](image)

![Figure 8: Maximum achievable throughput for anonymous coins while keeping the latency under 500ms and 1s. WAN measurements with 4 authorities; 1 to 10 shards per authority running on separate machines. No faulty authorities.](image)

![Figure 9: Throughput-latency graph for anonymous coins under crash-faults. WAN measurements with 10 authorities; 35 collocated shards per authority; 0, 1, and 3 crash-faults.](image)

8 **CONCLUSION**

Zef is the first linearly-scalable BFT protocol for anonymous payments with sub-second latency. Zef defines authorities as sharded services and by managing singly-owned objects using reliable broadcast rather than consensus. To support anonymous coins without sacrificing storage costs, Zef introduces a new notion of uniquely-identified, spendable account. Users can bind new anonymous coins to their accounts and spend coins in a privacy-preserving way thanks to state-of-the-art techniques such as the Coconut scheme [28]. Despite the CPU-intensive cryptographic operations required to preserve opacity and unlinkability of digital coins, our experiments confirm that anonymous payments in Zef provides unprecedentedly quick confirmation time (sub-second instead of tens of minutes) while supporting arbitrary throughput thanks to the linearly-scalable architecture.
A TRANSPARENT COINS

For comparison purposes, we sketch a simplified version of anonymous coins (Section 5) without opacity and unlinkability. At a high level, the protocol is similar to anonymous coins in terms of communication (Figure 10). Due to the absence of blinding and random commitments, communication channels and validators must be trusted for the privacy of every coin operation.

**Transparent coins.** A transparent coin is a certificate $T = \text{cert}[S]$ on a triplet $S = (id, v, r)$ where $id$ is the UID of an active account, $v \in [0, v_{\max}]$, and $r$ is some random seed value.

Seeds values $r$ are used to distinguish coins of the same value attached to the same id. In what follows, we identify certificates with the same content, that is: $\text{cert}[S] = \text{cert}[S']$ iff $S = S'$.

To spend a transparent coin $T$, a client must possess the authentication key controlling id. Importantly, authorities do not need to store $T$ themselves—although they will observe such certificates occasionally in clear.

**New account operation.** Similar to Section 5, we assume a locking account operation $O = \text{Spend}(\text{value}, \text{hash}(P))$ meant to prepare some payment $P$ (see below), withdraw value coins publicly, and eventually destroy the corresponding account.

**Transparent coin payment protocol.** Suppose that a user owns $f$ mutually distinct transparent coins $T_i^{\text{in}} = \text{cert}[S_i^{\text{in}}]$ where $S_i^{\text{in}} = (id_i^{\text{in}}, v_i^{\text{in}}, r_i^{\text{in}})$ ($1 \leq i \leq f$). Let $value_i \geq 0$ be a value that the user wishes to withdraw publicly from the account $id_i^{\text{in}}$.

Similar to Section 5, we require certificates $T_i^{\text{in}}$ to be distinct but not the UIDs $id_i^{\text{in}}$. In practice, we expect $\Sigma_i value_i$ to be the entire balance of the set of accounts $\{id_i^{\text{in}}\}$.

We define the total input value of the transfer as $\delta = \Sigma_i (v_i^{\text{in}} + value_i)$. To spend the coins into $d$ new coins with values $v_i^{\text{out}}$ ($1 \leq i \leq d$) and $\Sigma_i value_i$ we use the following protocol.

![Figure 10: A payment with transparent coins](image-url)
j ≤ d) such that ∑j o_j = v, the sender requests an UID id'_out from each recipient, then proceeds as follows:

(1) First, the sender constructs a payment description P as follows:
   (a) For every 1 ≤ j ≤ d, sample randomness i_out^j.
   (b) Let P = (T^in_1, ..., T^in_n, value_1, ..., value_1, S_out, ..., S_out) where S_out = (i_out^1, i_out^2, ..., i_out^d).

(2) For every distinct id'_in, the sender broadcasts an authenticated request R_i = Lock(id'_in, n_i, O) where O = Spend(value, hash(P)), n_i is the next available sequence number for the account id'_in, and value = ∑i_in o_i value_k.

(3) Upon receiving an authenticated request R = Lock(id, n, Spend(value, h),) from the owner of id, an authority α verifies that next_sequence(id, α) = n, pending(id, α) = ⊥, and 0 ≤ value ≤ balance(id, α). Then, α sets pending(id, α) = R and responds with a signature on R.

(4) The sender collects a quorum of signatures for each R_i sent above, thus forming a locking certificate L_i = cert[R_i] for every i. It now sends a free request

R' = CreateTransparentCoins(P, L_1, ..., L_d)

to all authorities and waits for a quorum of responses.

(5) Upon receiving a free request of the form R' = CreateTransparentCoins(P, L_1, ..., L_d) such that

P = (T^in_1, ..., T^in_n, value_1, ..., value_1, S_out, ..., S_out) where T^in_i = cert[(id'_in, i_in^1, i_in^2)] and S_out = (i_out^1, i_out^2, ..., i_out^d), each authority α verifies the following:

• The certificates T^in_i are valid and mutually distinct.
• Every L_i is a valid certificate for some request R = Lock(id, n_i) where id = id'_in, O = Spend(value, hash(P)), and value = ∑i_in o_i value_k.
• ∑i_in o_i + ∑i_out o_i = ∑i_out o_out_i.

The authority then destroys each account id'_in (if needed) and responds d signatures, one for each S_out.

(6) For every j, the sender finally combines a quorum of signatures on S_out^j into a new coin T^out_j.

(7) The j^th recipient receives T^out_j = cert[(i_out^1, i_out^2, ..., i_out^d)]. She verifies that the values and UID are as expected, that the random seeds r_out^j are mutually distinct, and that the certificates T^out_j are valid.

**Redeeming transparent coins.** Suppose that a user owns ℓ transparent coins T_i (1 ≤ i ≤ ℓ) linked to the same active account id. We define a new account operation

O = SpendAndTransfer(id', value, T_1, ..., T_ℓ)

meant to be included in a request R = Execute(id, n, O) and following the framework of Section 4:

• O is safe if id ∈ accounts(α), 0 ≤ value ≤ balance(id, α), and every T_j is a valid certificate of some distinct triplet S_j = (id, g_j, r_j).
• Upon receiving a valid certificate C = cert[R], the execution of O consists in removing the account id and sending a cross-shard request to add the value p = value + ∑j j r_j to balance(id) (possibly after creating an empty account id').

**B NIZK PROTOCOL**

In this section, we show one possible efficient instantiation of the anonymous payment protocol from Section 5 by opening up the cryptographic primitives used. Our protocol here makes use of the Coconut threshold credential scheme [28], which is based on the work of Pointcheval and Sanders [24]. Informally, Coconut allows users to obtain credentials on messages with private attributes in a distributed setting using a threshold t out of n authorities.

**B.1 Coconut++**

We start by giving an overview of a suitable variant of the Coconut scheme, nicknamed Coconut++. This variant of Coconut is formally proven secure by Rial and Pietrowinska [26]. At a high level, Coconut allows a user to obtain, from a threshold number of authorities, an anonymous credential on a private attribute m showing that it satisfies some application-specific predicate φ(m) = 1. Later, the user can anonymously prove the validity of this credential to any entity in possession of the verification key. While the standard Coconut scheme works for a single attribute, [28] also includes an extension that allows for credentials on a list of q integer-valued attributes m = (m_1, ..., m_q).

Below, we use the notation X = (X_1, ..., X_q) for any list of q variables X_j (1 ≤ j ≤ q). The scheme Coconut++ consists of the following algorithms:

- Setup(1^λ) → (pp): Choose groups (G_1, G_2, G_T) of order p (a λ-bit prime) with a bilinear map e: G_1 × G_2 → G_T. Let H: G_1 → G_2 be a secure hash function. Let g_1, h_1, ..., h_q be generators of G_1 and let g_2 be a generator of G_2. The system parameters are given as pp = (G_1, G_2, G_T, p, e, H, g_1, g_2, h). Parameters are implicit in the remaining descriptions.
- KeyGen(t, n) → (sk, vk): Pick q + 1 polynomials u, w_1, ..., w_q each of degree t − 1 with coefficients in F_p and set sk = (x, y) = (u(0), w_1(0), ..., w_q(0)). Publish the verification key vk = (f, α, β) = (g_1^y, ..., g_q^y, g_1^x, g_2^y, ..., g_q^y). Also issue to each authority j ∈ {1, ..., n}, the secret key sk_j = (x_j, y_j) = (u(j), w_1(j), ..., w_q(j)) and publish the corresponding verification key vk_j = (f_j, α_j, β_j) = (g_1^{y_j}, ..., g_q^{y_j}, g_2 x_j, g_2 y_j, ..., g_q y_j).
- PrepareBlindSign(m, φ) → (f, A): Pick a random r ∈ F_p. Compute the commitment c_m and group element h as

\[ c_m = g_1^{q+1} \prod_{i=1}^{q} h_i^{m_i} \quad \text{and} \quad h = H(c_m) \]

For all i = 1, ..., q, pick a random r_i ∈ F_p and compute the blinded value c_i as follows:

\[ c_i = h^{m_i} g_1^{r_i} \]

Output (f, A) where \( A = (c_m, c, φ) \) where \( φ \) is defined as:

\[ \pi_φ = \text{NIZK}(\{m, o, φ\}) : \quad \forall i, c_i = h^{m_i} g_1^{r_i} \land c_m = g_1^{q+1} \prod_{i=1}^{q} h_i^{m_i} \land φ(m) = 1 \]
We now instantiate the anonymous transfer protocol from Section 5 to use the Coconut scheme with three attributes $\id_j$, $\sigma_j$, $\phi_j$ to facilitate transactions. The protocol works as follows:

1. **PrepareBlindSign**$(k_{\in}, r_{\in}, q_{\in}, v_{\in})$: The sender uses the Zef account $A_{\in}$ to create an encrypted envelope $\tilde{\sigma}_{\in}$ for each output coin $q_{\in}$ that the recipient $\id_{\in}$ can open. The sender determines a nonce $r_{\in}$ and computes $\tilde{\sigma}_{\in} = \tilde{\sigma}_{\in}(h_{\in}, x_{\in})$, where $h_{\in}$ is the hash of the linked account and $x_{\in}$ is the index of the output coin. The sender then uses the Coconut scheme to blind the value $v_{\in}$ and create a blinded signature $\tilde{\sigma}_{\in}$.

2. **ProveCred**$(\vk, (q_{\in}, \sigma_{\in}), (\sigma_{\in}^{\in}, \phi'))$: For each input coin $q_{\in}$, the sender creates a Zef proof $\Theta_j$ that verifies the correctness of the input coin and binds the NIZKs associated with all input coins.

3. **AggCred**$(\sigma_j) \rightarrow (\sigma)$: The authority $j$ aggregates the proofs into a single proof $\Theta_j$.

4. **Unblind**$(\tilde{\sigma}_{\in}, \tilde{\sigma}_{\in}, \tilde{\sigma}_{\in})$全面推进双花：在同一个账户中拥有同一枚硬币的不同所有权，并防止同一硬币的双重支付。The proof $\tilde{\sigma}_{\in}$ for each output coin is revealed when it is spent to verify coin ownership and prevent double-spending of coins within the same account. We use the third attribute $q$ to guarantee the privacy of the value $v$ even after $k$ is revealed.

Suppose that a sender owns $t$ input coins $A_{\in}^t = (id_{\in}^i, q_{\in}^i, v_{\in}^i, \sigma_{\in}^i) (1 \leq i \leq t)$ and wishes to create $d$ output coins of the form $(id_{\out}^j, x_{\out}^j, q_{\out}^j, v_{\out}^j, \sigma_{\out}^j) (1 \leq j \leq d)$. Let $value_{\out}^j$ denote the public value associated with the Zef account $A_{\out}^j$ (possibly $0$) as in Section 5. The sender must ensure that $\sum_i q_{\in}^i + \sum_j value_{\out}^j = \sum_j q_{\out}^j$ and that the coin indices $(i_{\out}, x_{\out}^j)$ are mutually distinct.

Using Coconut for opaque coin transfers. We present an overview of the changes to the anonymous transfer protocol from Section 5 to implement opaque coins. We present an intuition on how these changes use the Coconut primitives described in Appendix B1); the next paragraphs provide detailed explanations of the opaque coin transfer protocol.

Recall that the sender first constructs a payment description $P$ and uses it to lock the UIDs of the input coins. For this, the sender proceeds as follows. For every $1 \leq i \leq t$, it reveals $k_i^t$. Then, for every $1 \leq i \leq t$ and $1 \leq j \leq d$, it calls

$$
\Theta_i \leftarrow \text{ProveCred}(\vk, (q_{\in}^i, \sigma_{\in}^i), (\sigma_{\in}^{\in}, \phi'))
$$

and

$$(rk_j, r_j, \sigma_j) \leftarrow \text{PrepareBlindSign}(k_j^t, \sigma_j, \phi_j')$$

where $\phi'$ is a predicate satisfied by the input and output coin values and defined as follows: $\phi'(\tilde{\sigma}_{\in}, \tilde{\sigma}_{\in}^{\out}) = \text{true}$ iff

$$
\sum_{i=1}^t q_{\in}^i + \sum_{i=1}^d value_{\out}^j = \sum_{i=1}^d q_{\out}^j \quad \text{and} \quad v_{\out}^j \in [0, v_{\max}]
$$

Effectively, the predicate $\phi'$ binds the NIZKs associated with all ProveCred proofs for the input coins and all PrepareBlindSign proofs for the output coins. It also shows that the value on both sides of the transfer is consistent. The payment description $P$ reveals $id_{\in}^t$ and $x_{\out}^j$ (and thus $k_i^t$), and is now constructed as

$$
P = (id_{\in}^t, x_{\out}^j, \Theta, A_{\in}, A_{\out}^j, \ldots, A_{\out}^d, value_1, \ldots, value_t).
$$

Recall now that the sender uses this $P$ to lock the input UIDs, and after retrieving a locking certificate for each UID from the authorities, it submits the request $R^* = \text{CreateAnonymousCoins}(L_1, \ldots, L_t)$ where the $L_i$ denote locking certificates. On receiving $R^*$ from the sender, an authority $\chi$ now verifies the proofs $\Theta$ and $A_j$ and the predicate $\phi'$ by running $\text{VerifyCred}(\vk, \Theta_j, \phi_j')$ for each $i$ and $\sigma_{\out}^{\in} = \text{BlindSign}(sk_j, A_j, \phi_j')$ for each $j$. If the proofs are valid, it returns $\tilde{\sigma}_{\out}$ to the sender.

After collecting $t$ such responses, the sender can now run Unblind and AggCred to obtain a valid credential on each created output coin. Finally, to complete the transfer, he can send the coin $(id_{\out}^j, x_{\out}^j, q_{\out}^j, v_{\out}^j, \sigma_{\out}^{\in})$ to the $j$th recipient.
Opaque coin construction. We present the cryptographic primitives used by the opaque coins transfer protocol. The Setup and KeyGen algorithms are exactly the same as Coconut.

- **CoinRequest** \( (vk, \sigma^\text{ini}, \sigma^\text{in}, \sigma^\text{out}, \sigma^\text{out}, \text{value}^\text{in}, \text{value}^\text{out}, \ldots, \text{value}^\text{out}_d) \) → \( (\text{rk}, \text{r}_q, \text{r}_v, \Gamma) \):
  Parse \( vk = (y_0, y_1, y_2, \alpha, \beta_0, \beta_1, \beta_2) \). For every input coin \( \sigma^\text{ini}_i \) (1 ≤ \( i \) ≤ \( t \)), parse \( \sigma^\text{ini}_i = (h_i, s_i) \), pick at random \( r_i, s_i, r_j, s_j \) in \( \mathbb{F}_p^2 \) and compute
  \[
  h'_i = h_i^{r_i} \quad \text{and} \quad s'_i = s_i^{r_i} (h'_i)^{s_i}
  \]
  Then set \( \sigma^\text{ini}_i = (h'_i, s'_i) \) and build:
  \[
  k_i = \alpha g_1^{r_i} \beta_1^{s_i} \beta_2^v
  \]
  For every output coin \( j \) (1 ≤ \( j \) ≤ \( d \)), pick a random \( o_j \) in \( \mathbb{F}_p \) and compute the commitments \( cm_j \) and the group elements \( h_j \) as
  \[
  cm_j = g_j^{o_j} h_0^{r_j} \quad \text{and} \quad h_j = H(cm_j)
  \]
  For all 1 ≤ \( j \) ≤ \( d \), pick a random \( \text{rk}, \text{r}_q, \text{r}_v \) in \( \mathbb{F}_p^3 \) and compute the commitments \( (ck_j, cq_j, cv_j) \) as follows:
  \[
  ck_j = h_j^{q_j} g_1^{r_j} \quad \text{and} \quad cq_j = h_j^{r_j} g_1^{q_j} \quad \text{and} \quad cv_j = h_j^{s_j} g_1^{r_j}
  \]
  Output \( (\text{rk}, \text{r}_q, \text{r}_v, \Gamma) \) where \( \Gamma = (\sigma^\text{ini}, \tilde{k}, cm, ck, cq, cv, \pi_r) \) when \( \pi_r \) is defined as:
  \[
  \pi_r = \text{NIZK}((q^\text{in}, q^\text{in}, q^\text{out}, q^\text{out}, r_i, \tilde{r}, \bar{r}, \bar{r}_q, \bar{r}_v) : \forall i, k_i = \alpha g_1^{k_i} \beta_1^{k_i} \beta_2^v \quad \wedge \quad \forall j, cm_j = g_j^{o_j} h_0^{r_j} h_j^{q_j} \quad \wedge \quad \forall j, ck_j = h_j^{q_j} g_1^{r_j} \quad \wedge \quad \forall j, cq_j = h_j^{r_j} g_1^{q_j} \quad \wedge \quad \forall j, cv_j = h_j^{s_j} g_1^{r_j} \quad \wedge \quad \sum_i q^\text{ini}_i + \sum_j q^\text{out}_j = d \quad \wedge \quad \sum_j q^\text{out}_j \in [0, \alpha_{\max}]
  )
  \]

- **IssueBlindCoin** \( (sk, \Gamma, k, \tilde{k}, \text{value}^\text{in}, \ldots, \text{value}^\text{in}, \text{value}^\text{out}, \ldots, \text{value}^\text{out}_d) \) → \( \tilde{\sigma} \): The authority \( \tilde{k} \) parses \( sk_x = (x, y_0, y_1, y_2) \), \( vk = (y_0, y_1, y_2, \alpha, \beta_0, \beta_1, \beta_2) \), and \( \Gamma = (\sigma^\text{ini}, \tilde{k}, cm, ck, cq, cv, \pi_r) \). Recompute \( h_j = H(cm_j) \) for each 1 ≤ \( j \) ≤ \( d \).
  Verify the proof \( \pi_r \) using \( \Gamma, h_j, \text{vk}, \text{value}^\text{in}, \ldots, \text{value}^\text{in}, \text{value}^\text{out}, \ldots, \text{value}^\text{out} \). For each 1 ≤ \( i \) ≤ \( t \), parse \( \sigma^\text{ini}_i = (h'_i, s'_i) \), verify \( h'_i \neq 1 \), and that the following bilinear evaluation holds:
  \[
  e(h'_i, k_i + \beta_0^s) = e(s'_i, g_2)
  \]
  If one of these checks fail, stop the protocol and output \( \perp \). Otherwise, compute:
  \[
  \tilde{s}_j = \tilde{h}_j^x \cdot (\tilde{c}_j)^{k_i} \cdot (\tilde{c}_j)^{q_j} \cdot (cv_j)^{r_j}
  \]

\[\text{PlainVerify}(vk, \sigma, k, q, o) \xrightarrow{\perp} \{\text{true/false}\} \]
Parse \( \sigma = (h, s) \) and \( vk = (y_0, y_1, y_2, \alpha, \beta_0, \beta_1, \beta_2) \). Reconstruct \( \kappa = \alpha^{pk} \beta_1^{q_0} \beta_2^v \), output \( \text{true} \) if \( h \neq 1 \) and \( e(h, \kappa) = e(s, g_2) \); otherwise output \( \perp \).

The user then calls \text{AggCred} and \text{Unblind} over each \( \tilde{s}_j \) exactly as described in Appendix B.1.

C FURTHER COMPARISON WITH FASTPAY

In FastPay, accounts are indexed by the public key \( pk \) that controls payment transfers from the account. Such a key is also called a FastPay address. The state of an account \( pk \) is replicated by every authority \( a \) and includes notably a balance \( \text{balance}^{pk}(a) \) and a sequence number \text{next_sequence}^{pk}(a) used to prevent replay of payment certificates.

The definition of FastPay addresses entails that an account \( pk \) (even with balance 0) can never be removed from the system. Indeed, after the information on the sequence number \text{next_sequence}^{pk}(a) is lost, the account owner may re-create an account for the same public key \( pk \) and exploit \text{next_sequence}^{pk}(a) = 0 to replay all past transfers originating from \( pk \). In a context of privacy-aware applications, we note that users are less likely to re-use a same account \( pk \) many times, thus amplifying the storage impact of unused accounts.

In Zef, accounts are indexed by a UID and can be deleted, thus enabling a variety of applications (see Section 3 and 5). On the down side, new users must interact with a broker or an authority ahead of time to obtain new Zef accounts. Existing users have the additional choice to trade some privacy and derive UIDs from their existing account(s).

Cross-shared queries in both FastPay and Zef are asynchronous in the sense that they do not block a client request to confirm a certificate (see Algorithm 1). This is crucial to guarantee that an authority with a lagging view on a particular account can be brought up to date by providing missing certificate history for this account and its parents only—as opposed to exponentially many accounts. In Zef, this property results from a careful design of the protocol allowing missing recipient accounts to be (re)created with an empty public key \( pk^{\text{id}}(a) = \perp \) whenever needed. The uniqueness property of identifiers guarantees that a deleted account can never be reactivated later on.