Low-latency, Scalable, DeFi with Zef

Mathieu Baudet[1], Alberto Sonnino[1], Michał Król[2]

[1]Novi, Facebook
[2]City, Univeristy of London

1 Introduction

Zef [3] was recently proposed to extend the low-latency, Byzantine-Fault Tolerant (BFT) payment protocol FastPay [1] with anonymous coins. This report explores further extensions of FastPay and Zef beyond payments. We start by off-chain assets (e.g. NFTs) in Section 2. We introduce the idea of on-demand BFT consensus instances throught the example of atomic swaps between account owners in Section 3.

2 Off-chain Assets

In Zef, a coin is defined as a quorum of validator signatures (known as a *certificate*) that binds a user account identifier id with a monetary value v. For instance, the *transparent coins* of Zef are defined as $C = \mathsf{cert}[(\mathsf{id}, s, v)]$ for some seed s. To provide unlinkability and privacy, Zef also defines another type of coin called *opaque coins* along the same lines but using the Coconut scheme [9]. For privacy reasons, all Zef coins (transparent and opaque) are stored off-chain[1] by their owners.

The coins linked to an account id can be spent altogether by the owner of id by issuing an operation Spend that deactivates id permanently. Compared to FastPay, Zef accounts are addressed by a unique identifier id that can never be replayed in future accounts. This makes it possible to effectively remove deactivated accounts and avoid a permanent storage cost every time that coins are spent. Zef coins can be generalized into *assets*, where a certificate binds arbitrary data to an account id:

- Assuming that a distinct data value x needs to be linked to an account id, the asset may simply defined as $A = \mathsf{cert}[(\mathsf{id}, x)]$.
- The procedure to consume and create coins in Zef can be generalized to consume input assets and create new assets according to specific rules of the form

$$(x_1^{out}, \ldots, x_d^{out}) = f_{\mathsf{exec}}(P, x_1^{in}, \ldots, x_\ell^{in})$$

  where f_{exec} is a fixed, deterministic, partial *execution function* and P is a set of parameters.

[1]We use the expressions "off-chain" and "on-chain" for the data outside and inside the Zef authorities, although Zef is not blockchain, strictly speaking.

- Importantly, the Spend operations used to deactivate the inputs accounts $x_1^{in}, \ldots, x_\ell^{in}$ must contain a commitment on P so that any replay of the asset creation request on the same input assets produces exactly the same output assets. (See the coin creation request in Zef [3])

Off-chain assets based on Zef provide storage-free certified execution at scale for single-owner data. This new general framework applies in particular to Non-Fungible Tokens (NFTs) with the following benefits:

- Off-chain storage provides some level of privacy w.r.t other users.
- Any owner-initiated operations such as transferring, combining NFTs, and applying legitimate modifications are supported at scale.

3 Atomic Swaps

We now describe an extension of Zef [3] meant to support the swap of ownership of two accounts in an atomic way.

To prevent race-conditions between operations (say, spending and swapping assets), a correct solution must start by requiring each owner to independently lock their asset into a new instance of the swap protocol. This creates a difficulty as one owner may lock their asset while the other fails to do so, or simply changes their mind. Hence, contrary to the operations described in Zef, an acceptable solution for atomic swap must support two possible outcomes: confirm and abort.

Because authorities must agree on this binary outcome, this raises the interesting question whether a correct solution for atomic swap in the FastPay model must implement a fully-featured, one-shot binary consensus protocol. From the FLP theorem [5], we know that a deterministic, asynchronous solution for a fault-tolerant consensus cannot guarantee both safety and eventual termination.

In this section we start by describing an implementation of atomic swap where termination assumes eventual cooperation between the owners of the two accounts. Importantly, this assumption is only made after both owners have locked their assets. We will discuss an optional refinement of the protocol at the end in order to enforce eventual termination in the partially-synchronous model.

Atomic-swap instances. We augment the state of each authority α (Zef [3], Section 3 and 4) with a new field $\mathsf{atomic\_swaps}(\alpha)$ that maps certain UIDs to the states of ongoing atomic-swap instances. Although atomic-swap instances are addressed by a UID, they have a distinct type (see below) and are not subject to account operations of Section 4 of Zef [3]. For clarity, below, we use swid (rather than id) to denote identifiers in the domain of $\mathsf{atomic\_swaps}(\alpha)$.

New account operations. To create atomic-swap instances and lock assets into an ongoing instances, we extend the protocol of Section 4 of Zef [3] with two new account operations:

- To lock the content of an account id into an atomic-swap instance swid, we introduce operations of the form $O = \mathsf{LockInto}(\mathsf{swid}, i, \mathsf{pk})$ where $i \in \{1, 2\}$ is the role index in the atomic swap, and pk is a key provided for authentication purposes and to be set on the other account in case of success. Such operation is sent in a (locking) request $R = \mathsf{Lock}(\mathsf{id}, n, O)$.

- To create a new instance of an atomic swap identified by a fresh identifier $\mathsf{swid} = \mathsf{id} :: n$, a (regular) request $R = \mathsf{Execute}(\mathsf{id}, n, O)$ may be used with

$$O = \mathsf{StartConsensusInstance}(\mathsf{swid}, \mathsf{id}_1, n_1, \mathsf{id}_2, n_2)$$

  meaning that the owners of id_1 and id_2 ($\mathsf{id}_1 \neq \mathsf{id}_2$) wish to atomically exchange their ownership of id_1 and id_2. The numbers n_1 and n_2 are the expected sequence numbers of the lock certificates of the respective owners for the operation LockInto above.

This new operations are summarized in Algorithm 1. We recall the framework for account requests in Zef in Algorithm 4.

Overview of the protocol. The successive steps of an atomic swap of ownership between two accounts id_1 and id_2 can now be summarized as follows (see also Figure 1):

- The two owners of id_1 and id_2 coordinate off-chain and decide to swap the ownership between id_1 and id_2. After sharing the next sequence numbers n_1 and n_2 of their respective accounts (❶), they decide to ask a *broker* to create a new UID for an atomic swap instance (❷). (This role may also be assumed by one of the owners.)

- The broker broadcasts an authenticated request containing an operation

$$\mathsf{StartConsensusInstance}(\mathsf{swid}, \mathsf{id}_1, n_1, \mathsf{id}_2, n_2)$$

  for a suitable fresh UID swid (❸). After receiving a quorum of answers (❹), this results in a certificate Γ certifying the creation of the instance swid to each client (❺).

- After verifying Γ, each owner broadcasts an authenticated request containing a LockInto operation in order to lock their account into swid (❻). This results in locking certificates L_1 and L_2 to be shared between clients (❽).

- Based on L_1 and L_2, one of the clients (or both as long as they agree on the desired outcome) acting as a *round leader* interact(s) with the consensus instance swid and attempts to drive the completion of the one-shot binary agreement protocol swid in order to confirm or abort the swap (❾).

- Eventually, at least one of the clients receives enough *commit* votes from authorities running the consensus instance swid that it may create a *commit certificate* C^* (defined in the next paragraph). When a commit certificate C^* is broadcast to authorities (❿), each authority issues internal cross-shard requests to the shards of id_1 and id_2, with the following effects:

  1. the sequence number $\mathsf{next\_sequence}^{\mathsf{id}_i}(\alpha)$ is incremented and $\mathsf{pending}^{\mathsf{id}_i}(\alpha)$ is reset to $\bot$ (effectively unlocking the account id_i),
  2. $\mathsf{confirmed}^{\mathsf{id}_i}(\alpha)$ is updated to include C^*, and finally
  3. if the *decision value* is Confirm and C^* is seen for the first time, the authentication key of the account $\mathsf{pk}^{\mathsf{id}_i}(\alpha)$ is set to the appropriate key pk_{3-i} initially chosen by the other owner as part of L_{3-i} (⓫).

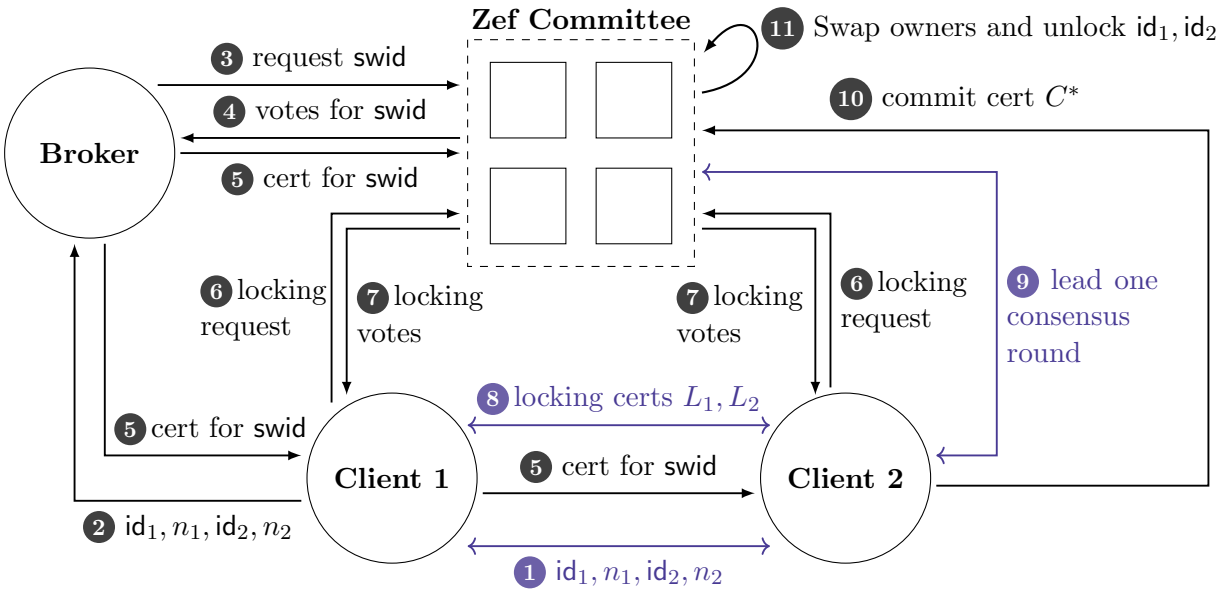

Figure 1: An atomic swap

Data types and notations. We introduce the following definitions:

- A decision value V is either Confirm or Abort.
- A *proposal* is a message $P = \mathsf{Proposal}(\mathsf{swid}, k, V)$ for some *round number* $k \geq 0$ and decision value V. We write $\mathsf{id}(P) = \mathsf{swid}$, $\mathsf{round}(P) = k$ and $\mathsf{decision}(P) = V$.
- A *pre-commit certificate* is a certificate on a proposal of the form $C = \mathsf{cert}[\mathsf{PreCommit}(P)]$.
- A *commit certificate* is a certificate on a proposal of the form $C^* = \mathsf{cert}[\mathsf{Commit}(P)]$.

We extend the notations $\mathsf{id}(\cdot)$, $\mathsf{decision}(\cdot)$, and $\mathsf{round}(\cdot)$ to decision certificates and commit certificates.

By definition, *agreement* holds iff two valid commit certificates for swid always contain the same decision value.

Atomic-swap states. The state of an atomic-swap instance swid as seen by an authority α can be described as follows:

- The two accounts to swap $\mathsf{id}_i^{\mathsf{swid}}(\alpha)$ $(i \in \{1, 2\})$;
- The two expected sequence numbers of lock certificates $n_i^{\mathsf{swid}}(\alpha)$;
- The two optional authentication keys $\mathsf{pk}_i^{\mathsf{swid}}(\alpha)$ (initially $\perp$ until the asset i is *locked*);
- A last *pending proposal* $\mathsf{proposed}^{\mathsf{swid}}(\alpha)$: initially $\perp$ then a proposal P.
- A last *pending pre-commit*: $\mathsf{locked}^{\mathsf{swid}}(\alpha)$: initially $\perp$ then a pre-commit certificate C.

Atomic-swap protocol. An atomic-swap instance swid at α may receive the following requests R from a client:

- A proposal request $R = \mathsf{HandleProposal}(\mathsf{auth}_{\mathsf{pk}}[P], L_1, L_2)$ for some $P = \mathsf{Proposal}(\mathsf{swid}, k, V)$ and optional certificates L_1 and L_2.
- A pre-commit request $R = \mathsf{HandlePreCommit}(C)$ for some pre-commit certificate C.
- A commit request $R = \mathsf{HandleCommit}(C^*, L_1, L_2)$ where C^* is a commit certificate and each L_i is an optional lock certificate.

The handling of such requests to a consensus instance is presented in Algorithm 2 and can be summarized as follows: (Invalid requests are ignored; additional *safety* rules are provided below.)

- Proposal request $R = \mathsf{HandleProposal}(\mathsf{auth}_{\mathsf{pk}}[P], L_1, L_2)$: The receiving authority α must verify that each L_i is either $\bot$ or a *valid lock certificate for the role* $i \in \{1, 2\}$ in the instance $\mathsf{swid} = \mathsf{id}(P)$, that is:
  - $L = \mathsf{cert}[R]$ is a valid certificate for some $R = \mathsf{Lock}(\mathsf{id}, n, \mathsf{LockInto}(\mathsf{swid}, i, \mathsf{pk}))$;
  - $\mathsf{id} = \mathsf{id}_i^{\mathsf{swid}}(\alpha)$ and $n = n_i^{\mathsf{swid}}(\alpha)$;

  After setting $\mathsf{pk}_i^{\mathsf{swid}}(\alpha) = \mathsf{pk}$ for each such non-null L_i, the authority α verifies the following conditions:
  - $P = \mathsf{Proposal}(\mathsf{swid}, k, V)$ is a proposal for swid and the authentication of P by pk is correct;
  - the proposal P is *valid* in the sense that $V = \mathsf{Confirm}$ implies that both input accounts are locked ($\forall i, \mathsf{pk}_i^{\mathsf{swid}}(\alpha) \neq \bot$);
  - the round k is *available* (see discussion below);
  - the proposal P is *safe* (see definition below)

  If the conditions are fulfilled then α sets $\mathsf{proposed}^{\mathsf{swid}}(\alpha)$ to P and returns a signature on $\mathsf{PreCommit}(P)$.
- Pre-commit request $R = \mathsf{HandlePreCommit}(C)$: If $C = \mathsf{cert}[\mathsf{PreCommit}(P)]$ is a valid pre-commit certificate, $\mathsf{id}(C) = \mathsf{swid}$, and C is safe (see below), then the authority sets $\mathsf{locked}^{\mathsf{swid}}(\alpha) = C$ and returns a signature on $\mathsf{Commit}(P)$.
- Commit request $R = \mathsf{HandleCommit}(C^*, L_1, L_2)$: If $C^* = \mathsf{cert}[\mathsf{Commit}(P)]$ is a valid commit certificate and $\mathsf{id}(C^*) = \mathsf{swid}$, then several cross-shard requests are prepared and sent to *selected* accounts as follows:
  - The account id is *selected* iff either (i) it was locked previously in the instance ($\mathsf{id} = \mathsf{id}_i^{\mathsf{swid}}(\alpha)$ and $\mathsf{pk}_i^{\mathsf{swid}}(\alpha) \neq \bot$) or (ii) it holds that $\mathsf{decision}(P) = \mathsf{Abort}$ and a valid certificate $L_i = \mathsf{cert}[R_i]$ such that $R_i = \mathsf{Lock}(\mathsf{id}, n, \mathsf{LockInto}(\mathsf{swid}, i, \mathsf{pk}))$ is part of the request;
  - Cross-shard requests are sent to the selected accounts id to unlock them by resetting $\mathsf{pending}^{\mathsf{id}}(\alpha)$ and incrementing $\mathsf{next\_sequence}^{\mathsf{id}}(\alpha)$. If the decision value $\mathsf{decision}(P)$ is $\mathsf{Confirm}$ and the instance swid still exists, a new authentication key is also set for $\mathsf{pk}^{\mathsf{id}}(\alpha)$ thus fulfilling the desired swap of ownership.

– Finally, the instance swid is destroyed (if it was still present)

Due to the validity condition above on P, the decision value is necessarily Abort if some account was never locked (i.e. $\mathsf{pk}_i^{\mathsf{swid}}(\alpha) = \bot$). The lock certificates L_i in the commit requests ensure that early termination of the consensus instance do not prevent users from unlocking their account in every authority afterwards. To allow immediate deletion of an instance swid, in the case of Abort, we do not enforce consistency between the additional lock certificates L_i and the original data in swid. This bears no consequence since we only allow this behavior after verifying an Abort commit certificate for swid.

Safety rules. To guarantee agreement, an authority α only accepts proposal and proposed certificates that are *safe* at the time of the request (Algorithm 3):

1. A proposal P is safe for α iff the following conditions hold:
   - (a) if $\bot \neq \mathsf{proposed}^{\mathsf{swid}}(\alpha) = P_0$ and $P \neq P_0$, then $\mathsf{round}(P) > \mathsf{round}(P_0)$;
   - (b) if $\bot \neq \mathsf{locked}^{\mathsf{swid}}(\alpha) = C_0$, then $\mathsf{round}(P) > \mathsf{round}(C_0)$ and $\mathsf{decision}(P) = \mathsf{decision}(C_0)$.
2. A proposed certificate C is safe for α iff the following conditions hold:
   - (c) if $\bot \neq \mathsf{proposed}^{\mathsf{swid}}(\alpha) = P_0$, then $\mathsf{round}(C) \geq \mathsf{round}(P_0)$;
   - (d) if $\bot \neq \mathsf{locked}^{\mathsf{swid}}(\alpha) = C_0$, then $\mathsf{round}(C) \geq \mathsf{round}(C_0)$.

Note that by definition of the protocol, $\mathsf{proposed}^{\mathsf{swid}}(\alpha)$ and $\mathsf{locked}^{\mathsf{swid}}(\alpha)$ never go back to $\bot$ once there are set. Rather, these two fields respectively tracks the latest (safe) proposal P and the latest (safe) proposed certificate C that were voted on by α.

Available rounds. In practice, we may wish to prevent requests from using arbitrary round numbers $k \in \mathbb{N}$, because preventing exhaustion of such numbers would then require using unbounded-size infinite-precision integers. To address this issue while avoiding active coordination between authorities, we propose that authorities makes new round numbers available in sequential order at a fixed, given rate.

Client Protocol. Assuming that an instance swid is still running and that no one else is proposing, a client with an asset locked in swid may drive completion as follows:

- Query all the authorities in parallel to retrieve the highest round k of a proposal $P = \mathsf{proposed}^{\mathsf{swid}}(\alpha)$ for some α and/or the highest pre-commit certificate $C = \mathsf{locked}^{\mathsf{swid}}(\alpha)$, if any.
- After a suitable delay δ, if $C \neq \bot$, then broadcast C to obtain a commit certificate. Otherwise, when $k + 1$ is available, make a new proposal, then broadcast the pre-commit certificate.
- Broadcast the final commit certificate C^*.

Proof of safety. The agreement property on commit certificates is derived from the following lemma:

Lemma 3.1. *Assume* $C^* = \mathsf{cert}[\mathsf{Commit}(P_1)]$ *and* $C_2 = \mathsf{cert}[\mathsf{PreCommit}(P_2)]$ *such that* $\mathsf{round}(P_2) \geq \mathsf{round}(P_1)$ *then* $\mathsf{decision}(P_1) = \mathsf{decision}(P_2)$.

Proof. By induction on $\mathsf{round}(P_2) \geq \mathsf{round}(P_1)$.

If $\mathsf{round}(P_2) = \mathsf{round}(P_1)$, since the certificates $\mathsf{cert}[\mathsf{PreCommit}(P_1)]$ and $\mathsf{cert}[\mathsf{PreCommit}(P_2)]$ exist, by quorum intersection, there exists an honest node α that voted for both P_1 and P_2. However, by safety rule (a), honest nodes only vote for new proposals with strictly increasing rounds, therefore $P_1 = P_2$.

Otherwise, assume $\mathsf{round}(P_2) > \mathsf{round}(P_1)$. Let $C_1 = \mathsf{cert}[\mathsf{PreCommit}(P_1)]$. By quorum intersection of C^* and C_2, there exist be a honest node α that voted for both $\mathsf{Commit}(P_1)$ and $\mathsf{PreCommit}(P_2)$.

By rule (d), the round of $\mathsf{locked}^{\mathsf{swid}}(\alpha)$ never decreases. Thus, by rule (c), $\mathsf{round}(P_2) > \mathsf{round}(P_1) = \mathsf{round}(C_1)$ implies that α voted for $\mathsf{Commit}(P_1)$ first, then $\mathsf{PreCommit}(P_2)$.

At the time of voting for $\mathsf{PreCommit}(P_2)$, by (b) and (d), we have that $\mathsf{locked}^{\mathsf{swid}}(\alpha) = C$ for some pre-commit certificate C such that $\mathsf{round}(P_2) > \mathsf{round}(C) \geq \mathsf{round}(P_1)$ and $\mathsf{decision}(P_2) = \mathsf{decision}(C)$. By induction, we conclude $\mathsf{decision}(P_1) = \mathsf{decision}(C)$. □

Discussion on termination. As noted earlier, after the two assets are locked, in theory, one of the owners can prevent the protocol from terminating by indefinitely submitting proposals that conflict with the other client. In addition to forfeiting half of the assets, this requires active communication with at least one honest authority at every new round in the future, when a round becomes available. Specifically, the malicious leader must indefinitely guess or quickly observe whether the other client is proposing Confirm or Abort, and propose the opposite decision value.

A classical approach to enforce strict termination in the partially-synchronous model consists in (1) restricting proposals to be signed by a particular client based on the parity of k and (2) making new rounds available at an exponentially slow rate. In practice, this approach may be activated after a certain delay, when it is clear that the two owners are not collaborating.

Comparison with existing consensus protocols. Our proposal is based on the observation that traditional leader-based consensus protocols do not technically require leaders to be drawn from the entire set of validators or even to have non-zero voting rights—as long as enough leaders can be trusted to make progress. In our case, this means that we can use the owner(s) of locked account(s) as leaders of the consensus protocol instead of Zef validators. Once both accounts are locked, our proposal lets the two leaders coordinate directly outside of the protocol — at least for some time, until a slow, leader selection is (optionally) activated to enforce termination.

Compared to fully-featured implementation of a consensus protocol such as LibraBFT [2], our approach is a one-shot consensus protocol, in particular chains of blocks are not needed. We also do not attempt to provide responsiveness or tight latency guarantees when leaders are not cooperative. However, our proposal is arguably significantly simpler, only uses constant storage, and does not require active coordination between authorities (e.g. broadcasting timeout messages).

Future work. We have presented an atomic swap functionality that changes the owner's keys of the two accounts simultaneously. Arguably, this constitutes the first step towards a more general

framework where multiple users may lock their accounts (and corresponding assets) into a consensus instance in order to execute arbitrary queries/updates on the locked accounts in an atomic way.

4 Auctions

We now describe an extension of Zef [3] meant to support running decentralised auctions. We target support both 1st price and 2nd price auctions:

- In the 1st price auctions, a bidder with the highest bid gets the item and pays a price equivalent to its bid.
- In the 2nd price auctions, a bidder with the highest bid gets the item but pays a price equivalent of the 2nd highest bid in the auction.

The 2nd price auctions provide a truthfulness property[?], where all the bidders are incentivised to provide their true valuation of items (the winner never overpays for an item). However, it comes at a price of requiring additional security mechanisms. A malicious seller may participate in the auction (potentially with Sybil identities) uniquely to become the 2nd highest bid in the auction and thus increase the selling price of the item. Such a behavior can be disincentivised, if every submitted but unrevealed bid is penalized [4].

In a classical setup, auctions require multiple phases that need to be ordered:

- the seller creates an auction
- the bidders need to submit their sealed bids
- the seller stops the bid submission
- the bidders need to reveal their bids
- the seller announces the result of the auction

A correct solution must order those operations to prevent race-conditions between the bidders and the seller (was a bid submitted/revealed before the deadline?).

4.1 Bid submission

We define a Threshold Public Key Encryption (TPKE) system consists of five algorithms [8]:

- $\mathsf{Setup}(n, k, \Lambda)$: Takes as input the number of decryption servers n, a threshold k where $1 \leq k \leq n$, and a security parameter $\Lambda \in \mathcal{Z}$. It outputs a triple $(\mathsf{PK}, \mathsf{VK}, \mathsf{SK})$ where PK is a public key, VK is a verification key, and $\mathsf{SK} = (\mathsf{SK}_1, ..., \mathsf{SK}_n)$ is a vector of n private key shares. Decryption server i is given the private key share (i, SK_i) and uses it to derive a decryption share for a given ciphertext. The verification key VK is used to check validity of responses from decryption servers.
- $\mathsf{Encrypt}(\mathsf{PK}, m)$: Takes as input a public key PK and a message m . It outputs a ciphertext c.

- $\mathsf{ShareDecrypt}(\mathsf{PK}, i, \mathsf{SK}_i, c)$: Takes as input the public key PK, a ciphertext c, and one of the n private key shares in SK. It outputs a decryption share $\mu = (i, \hat{\mu})$ of the enciphered message, or a special symbol $(i, \perp)$.

- $\mathsf{ShareVerify}(\mathsf{PK}, \mathsf{VK}, c, \mu)$: Takes as input PK, the verification key VK, a ciphertext c, and a decryption share μ. It outputs valid or invalid. When the output is valid we say that μ is a valid decryption share of C.

- $\mathsf{Combine}(\mathsf{PK}, \mathsf{VK}, C, \mu_1, ..., \mu_k)$: Takes as input PK, VK, a ciphertext c, and k decryption shares $\mu_1, ..., \mu_k$. It outputs a cleartext M or $\perp$.

We assume that the FastPay authorities jointly execute $\mathsf{Setup}(n, f+1, \Lambda)$ as a part of the bootstrap process. Each authority a_i is given its private key share SK_i, the public key PK and the verification key VK. We assume that no other authority $a_j, j \neq i$ has access to SK_i.

A user u_i willing to participate in the auction chooses an amount it is willing to pay for the item v_i. It also chooses $v_{\mathsf{max}}, v_{\mathsf{max}} \leq v_i$ to back its bid. The user transfers v_{max} to the auction object. The money acts as a deposit that will be returned if the user does not win the auction or used to pay for the object if the user wins the auction.

The user then encrypts its bid by executing $c_i = \mathsf{Encrypt}(\mathsf{PK}, m_i)$, where m_i contains v_i and the auction identifier. The user generates z_i, a proof of correctness of encryption c_i.

The user submit c_i and z_i to the authorities and obtains a submission certificate C_i. Note that receiving a certificate (confirmation from $2f+1$ authorities), means that at least $f+1$ honest authorities have received the encryption and can jointly recover message M_i.

The bidders send their bid submission certificates offline to the seller. The seller then submits those certificates to the system. Effectively, the seller can choose which bids will be allowed to participate in the auction. However, the seller is incentivized to maximize the number of bids in the auction as each additional bid can only increase the selling price of the object (and thus the revenue of the seller).

Once the seller decides that they gathered enough bids, they submits end of bidding message and obtains a certificate on the submission. The end of bidding message contains all the bids already submitted by the seller. After accepting an end of bidding message the authorities stop accepting new bids.

The seller individually contacts each authority presenting a certificate on the end of bidding message. If the certificate is valid, the contacted authority invokes ShareDecrypt on all the bids present in the message and releases its decryption shares μ_i. The seller collects all the decryption shares and locally invokes Combine to recover the values of the bids. The seller includes all the decrypted bids in an end of auction message and submits the message to the authorities (trying to get a certificate).

Once a certificate on end of auction is submitted to an authority, the authority:

- Calculates the highest and the 2nd highest bid

- Assigns the object being sold to the highest bidder

- Deducts the winner's deposit by the value of the 2nd highest bid and transfers this value to the seller

- Returns the deposits to the bidders

- Deletes the item objects

References

[1] Mathieu Baudet, George Danezis, and Alberto Sonnino. "FastPay: High-Performance Byzantine Fault Tolerant Settlement". In: *2nd ACM Conference on Advances in Financial Technologies*. AFT '20. 2020, 163–177.

[2] Mathieu Baudet et al. *State machine replication in the Libra Blockchain*. 2019.

[3] Mathieu Baudet et al. "Zef: Low-latency, Scalable, Private Payments (draft report)". In: (2021).

[4] Matheus VX Ferreira and S Matthew Weinberg. "Credible, truthful, and two-round (optimal) auctions via cryptographic commitments". In: *Proceedings of the 21st ACM Conference on Economics and Computation*. 2020, pp. 683–712.

[5] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. "Impossibility of Distributed Consensus with One Faulty Process". In: *Journal of the ACM* 32.2 (1985), 374–382.

[6] Marc Shapiro et al. "A comprehensive study of Convergent and Commutative Replicated Data Types". In: ().

[7] Marc Shapiro et al. "Conflict-Free Replicated Data Types". In: *Stabilization, Safety, and Security of Distributed Systems*. Ed. by Xavier Défago, Franck Petit, and Vincent Villain. 2011, pp. 386–400.

[8] Victor Shoup and Rosario Gennaro. "Securing threshold cryptosystems against chosen ciphertext attack". In: *Journal of Cryptology* 15.2 (2002), pp. 75–96.

[9] Alberto Sonnino et al. "Coconut: Threshold issuance selective disclosure credentials with applications to distributed ledgers". In: *arXiv preprint arXiv:1802.07344* (2018).

A Generalized Account Operations

In FastPay and Zef, each account state contains a balance, noted $\mathsf{balance}^{\mathsf{id}}(\alpha) \in \mathbb{Z}$. The execution of a payment operation can be seen as applying a pair of updates $(-x, +x)$ to the sender and the recipient states, respectively. Namely, $-x < 0$ is the *local* update removing funds and $+x > 0$ is the *remote* update adding funds. An authority accepts to validate an (authenticated) payment operation $(-x, +x)$ created by the owner of id iff the resulting local state $\mathsf{balance}^{\mathsf{id}}(\alpha) - x \geq 0$ is *valid*. We note that remote updates $+x$ are always *safe* in the sense that they never make a valid state invalid.

This leads us to propose the following general axioms for account states and updates:

Generalized states and updates. Let $\mathcal{S}$ be a set of *state values* and $\mathcal{U}$ be a set of *updates*. We assume a validity predicate $\mathsf{is\_valid}$ on $\mathcal{S}$, a safety predicate $\mathsf{is\_safe}$ on $\mathcal{U}$ and an operator $(\cdot)$ from $\mathcal{S} \times \mathcal{U}$ to $\mathcal{S}$ such that the following holds:

1. $\forall s \in \mathcal{S}$, $\forall u_1, u_2 \in \mathcal{U}$, $s \cdot u_1 \cdot u_2 = s \cdot u_2 \cdot u_1$;
2. $\forall s \in \mathcal{S}$, $\forall u \in \mathcal{U}$, $\mathsf{is\_valid}(s)$ and $\mathsf{is\_safe}(u) \implies \mathsf{is\_valid}(s \cdot u)$.

The commutative updates (1) as well as the notion of eventually consistency described in the proof of Zef ([3], Section 4) draws similarities to the notion of Commutative Replicated Data Types (CmRDTs) in the field of distributed databases [6, 7].

Generalized account operations. We may generalize the protocol for direct payments of FastPay and Zef as follows:

- The balance is replaced by a field $\mathsf{state}^{\mathsf{id}}(\alpha) \in \mathcal{S}$ such that the initial value of a new account is always valid.
- A new account operation $O = \mathsf{Apply}(\mathsf{id}', u_-, u_+)$ is *safe* to be issued by the owner of the account id as seen by an authority α iff $\mathsf{is\_safe}(u_+)$ and $\mathsf{is\_valid}(\mathsf{state}^{\mathsf{id}}(\alpha) \cdot u_-)$. (Note that in practice, additional constraints may apply on u_- and u_+ for O to be validated by α.)
- The execution of an update $\mathsf{Apply}(\mathsf{id}', u_-, u_+)$ sent by account id consists in setting $\mathsf{state}^{\mathsf{id}}(\alpha) := \mathsf{state}^{\mathsf{id}}(\alpha) \cdot u_-$ and $\mathsf{state}^{\mathsf{id}'}(\alpha) := \mathsf{state}^{\mathsf{id}'}(\alpha) \cdot u_+$.

The same argument as in Section 4 of [3] shows that whenever two honest authorities have executed the same certified updates then the two authorities agree on the states of active accounts. Besides, after every certified update has been executed by an authority α, $\mathsf{is\_valid}(\mathsf{state}^{\mathsf{id}}(\alpha))$ holds for every id.

This formalism lets us address the following applications:

- A single NFT may be represented on-chain using $\mathcal{S} = \mathbb{Z}$ and $\mathcal{U} = \{-1, +1\}$. (Here, $\mathcal{S} = \mathbb{Z}$ ensures proper definitions of the operator $(\cdot)$. In practice, state values would range in $\{-1, 0, 1\}$, the invalid value -1 being a temporary state.)
- To support severals NFTs and several currencies at the same time, we note that independent updates on $\mathcal{S}_1 \times \mathcal{U}_1$ and $\mathcal{S}_2 \times \mathcal{U}_2$ may composed by defining a product operator $(\cdot)$ on $\mathcal{S} \times \mathcal{U}$ with $\mathcal{S} = \mathcal{S}_1 \times \mathcal{S}_2$ and $\mathcal{U} = \mathcal{U}_1 \biguplus \mathcal{U}_2$.
- A (multi)-set of objects or coins with monotonic requirements (e.g. to own X, one must be own parent(X) and 3 coins). (This approach is seen in CRDTs for data-structures such as trees, directed graphs, etc.)

Further generalization. In the case of multi-currency states, we note that the formalism does not force u_- and u_+ to be in the same unit of currency. Each validator may accept conversion requests up to a certain *most favorable rate* that may differ from other validators and may fluctuate over time.

This approach paves the way for automated marker makers (AMM) in the Zef system: every shard of each Zef authority may maintain the current conversion rate(s) in a local cache and accept rate updates ("push") from a dedicated blockchain tracking the past transactions (using real-time aggregated counters in each authority) and continuously re-computing the rate(s).

Storage Cost A well-known issue of CmRDTs is the storage cost given that each update is individually synchronized and logged across replicas[2]. In Zef, this issue is mitigated by the fact

[2] `https://github.com/protocol/research-grants/blob/master/RFPs/rfp-005-optimized-CmRDT.md`

that accounts can be deleted after transferring their state to a new account, thereby effectively compressing the history of the account.

Algorithm 1 Account operations (core Zef + atomic swap)

```
1: function ValidateOperation(id, n, O)                    ▷ Internal validation of account operation
2:     switch O do
3:         case OpenAccount(id′, pk′):
4:             ensure id′ = id :: next_sequence^id
5:         case Transfer(id′, value):
6:             ensure 0 < value ≤ balance^id
7:         case ChangeKey(pk′):
8:             pass
9:         case StartConsensusInstance(swid, id_1, pk_1, id_2, pk_2):
10:            ensure swid = id :: next_sequence^id
11:            ensure id_1 ≠ id_2
12:        case LockInto(swid, i, pk):                     ▷ Temporarily transfer the management of id to swid
13:            return Lock(id, n, O)                       ▷ O is valid and locking.
14:    return Execute(id, n, O)                            ▷ If we reach this, O is valid and regular.

15: function ExecuteOperation(id, O, C)                    ▷ Execution of account operation (unchanged from Zef)
16:    switch O do
17:        case OpenAccount(id′, pk′):
18:            do asynchronously                           ▷ Cross-shard request to id′
19:                run InitAccount(id', pk')               ▷ Create new account
20:                received^id′ ← received^id′ :: C        ▷ Log certified request in recipient's account
21:        case Transfer(id′, value):
22:            balance^id ← balance^id − value             ▷ Update sender's balance
23:            do asynchronously                           ▷ Cross-shard request to id′
24:                if id′ ∉ accounts then
25:                    run InitAccount(id′, ⊥)             ▷ Create receiver's account if needed
26:                balance^id′ ← balance^id′ + value       ▷ Update receiver's balance
27:                received^id′ ← received^id′ :: C
28:        case ChangeKey(pk′):
29:            pk^id ← pk′                                 ▷ Update authentication key
30:        case StartConsensusInstance(swid, id_1, n_1, id_2, n_2):
31:            do asynchronously                           ▷ Cross-shard request to swid
32:                run InitInstance(swid, id_1, n_1, id_2, n_2, C)   ▷ Create a new consensus instance
```

Algorithm 2 Consensus service

function InitInstance($\mathsf{swid}, \mathsf{id}_1, n_1, \mathsf{id}_2, n_2, C$) ▷ Create a consensus instance for an atomic swap
  for $i \leftarrow 1..2$ **do** $(\mathsf{id}_i^{\mathsf{swid}}, n_i^{\mathsf{swid}}, \mathsf{pk}_i^{\mathsf{swid}}) \leftarrow (\mathsf{id}_i, n_i, \bot)$
  $(\mathsf{proposed}^{\mathsf{swid}}, \mathsf{locked}^{\mathsf{swid}}, \mathsf{received}^{\mathsf{swid}}) \leftarrow (\bot, \bot, C)$

function HandleProposal($\mathsf{auth}_{\mathsf{pk}}[P], L_1, L_2$) ▷ Handle a proposal with optional lock certificates
  verify $\mathsf{auth}_{\mathsf{pk}}[P]$
  match $\mathsf{Proposal}(\mathsf{swid}, k, V) = P$
  for $i \leftarrow 1..2$ **do**
    if $L_i \neq \bot$ **then**
      verify $\mathsf{cert}[R_i] = L_i$
      match $\mathsf{Lock}(=\mathsf{id}_i^{\mathsf{swid}}, =n_i^{\mathsf{swid}}, \mathsf{LockInto}(=\mathsf{swid}, =i, \mathsf{pk}_i)) = R_i$
      $\mathsf{pk}_i^{\mathsf{swid}} \leftarrow \mathsf{pk}_i$ ▷ Record the public key of role i
  ensure $\mathsf{pk} \in \{\mathsf{pk}_1^{\mathsf{swid}}, \mathsf{pk}_2^{\mathsf{swid}}\}$ ▷ Only users with a locked account can propose
  ensure $V = \mathsf{Abort}$ or $\forall i, \mathsf{pk}_i^{\mathsf{swid}} \neq \bot$ ▷ Enforce validity of the swap
  ensure IsRoundAvailable(swid, k) ▷ Available round values are restricted at a given time
  ensure IsSafeProposal(P) ▷ Enforce safety rule
  $\mathsf{proposed}^{\mathsf{swid}} \leftarrow P$ ▷ Record the proposal for future safety checks
  return Vote($\mathsf{PreCommit}(P)$) ▷ Success: return a signature meant to pre-commit P

function HandlePreCommit(C) ▷ Handle a pre-commit request
  verify $\mathsf{cert}[\mathsf{PreCommit}(P)] = C$
  ensure IsSafePreCommit(C) ▷ Enforce safety rule
  $\mathsf{locked}^{\mathsf{swid}} \leftarrow C$ ▷ Record the pre-commit for future safety checks
  return Vote($\mathsf{Commit}(P)$) ▷ Success: return a signature meant to commit P

function HandleCommit(C^*, L_1, L_2) ▷ Handle a commit request
  verify $\mathsf{cert}[\mathsf{Commit}(P)] = C$
  match $\mathsf{Proposal}(\mathsf{swid}, k, V) = P$
  for $i \leftarrow 1..2$ **do**
    $(\mathsf{id}_i, n_i, \mathsf{pk}_i) \leftarrow (\mathsf{id}_i^{\mathsf{swid}}, n_i^{\mathsf{swid}}, \mathsf{pk}_i^{\mathsf{swid}})$ ▷ Set locals with information from swid or $\bot$
    if $V = \mathsf{Abort}$ and $L_i \neq \bot$ **then** ▷ Accept to unlock any account locked into swid, once aborted
      verify $\mathsf{cert}[R_i] = L_i$
      match $\mathsf{Lock}(\mathsf{id}, n, \mathsf{LockInto}(=\mathsf{swid}, =i, \mathsf{pk})) = R_i$
      $(\mathsf{id}_i, n_i, \mathsf{pk}_i) \leftarrow (\mathsf{id}, n, \mathsf{pk})$
  if $V = \mathsf{Confirm}$ and $\mathsf{swid} \in \mathsf{atomic\_swaps}$ **then**
    for $i \leftarrow 1..2$ **do**
      do asynchronously ▷ Cross-shard request to id_i
        ensure $\mathsf{next\_sequence}^{\mathsf{id}_i} = n_i$
        $(\mathsf{next\_sequence}^{\mathsf{id}_i}, \mathsf{pending}^{\mathsf{id}_i}) \leftarrow (n_i + 1, \bot)$ ▷ Unlock account id_i
        $\mathsf{pk}^{\mathsf{id}_i} \leftarrow \mathsf{pk}_{3-i}$ ▷ Set the public key to the new value
        $\mathsf{confirmed}^{\mathsf{id}_i} \leftarrow \mathsf{confirmed}^{\mathsf{id}_i} :: C^*$
  else
    for $i \leftarrow 1..2$ **do**
      if $\mathsf{pk}_i \neq \bot$ **then**
        do asynchronously ▷ Cross-shard request to id_i
          ensure $\mathsf{next\_sequence}^{\mathsf{id}_i} = n_i$
          $(\mathsf{next\_sequence}^{\mathsf{id}_i}, \mathsf{pending}^{\mathsf{id}_i}) \leftarrow (n_i + 1, \bot)$ ▷ Unlock account id_i
          $\mathsf{confirmed}^{\mathsf{id}_i} \leftarrow \mathsf{confirmed}^{\mathsf{id}_i} :: C^*$
  delete instance swid from atomic_swaps

Algorithm 3 Safety rules

function IsSafeProposal(P) ▷ Determine if it is safe to vote for pre-committing P
  let $\mathsf{Proposal}(\mathsf{swid}, k, V) = P$
  if $\mathsf{proposed}^{\mathsf{swid}} \neq \bot$ and $k \leq \mathsf{round}(\mathsf{proposed}^{\mathsf{swid}})$ **then**
    return **false**
  if $\mathsf{locked}^{\mathsf{swid}} \neq \bot$ and $\Big(k \leq \mathsf{round}(\mathsf{locked}^{\mathsf{swid}})$ or $V \neq \mathsf{decision}(\mathsf{locked}^{\mathsf{swid}})\Big)$ **then**
    return **false**
  return **true**

function IsSafePreCommit(C) ▷ Determine if it is safe to vote for committing C
  let $\mathsf{cert}[\mathsf{PreCommit}(P)] = C$
  let $\mathsf{Proposal}(\mathsf{swid}, k, V) = P$
  if $\mathsf{proposed}^{\mathsf{swid}} \neq \bot$ and $k < \mathsf{round}(\mathsf{proposed}^{\mathsf{swid}})$ **then**
    return **false**
  if $\mathsf{locked}^{\mathsf{swid}} \neq \bot$ and $k < \mathsf{round}(\mathsf{locked}^{\mathsf{swid}})$ **then**
    return **false**
  return **true**

Algorithm 4 Account service (unchanged from Zef)

function InitAccount(id, pk) ▷ Initialize a new account
  $\mathsf{pk}^{\mathsf{id}} \leftarrow \mathsf{pk}$
  $\mathsf{next\_sequence}^{\mathsf{id}} \leftarrow 0$
  $\mathsf{balance}^{\mathsf{id}} \leftarrow \mathsf{balance}^{\mathsf{id}}(\mathsf{init})$ ▷ Initial balance is 0 except for special accounts
  $\mathsf{confirmed}^{\mathsf{id}} \leftarrow [\,]$
  $\mathsf{received}^{\mathsf{id}} \leftarrow [\,]$

function HandleRequest($\mathsf{auth}_{\mathsf{pk}}[R]$) ▷ Handle an authenticated request from a client
  let $\mathsf{Execute}(\mathsf{id}, n, O) \mid \mathsf{Lock}(\mathsf{id}, n, O) = R$ ▷ Allow regular and locking operations
  ensure $\mathsf{pk}^{\mathsf{id}} \neq \bot$ ▷ Make sure the account is active
  verify that $\mathsf{auth}_{\mathsf{pk}}[R]$ is valid for $\mathsf{pk} = \mathsf{pk}^{\mathsf{id}}$ ▷ Check request authentication
  if $\mathsf{pending}^{\mathsf{id}} \neq R$ **then**
    ensure $\mathsf{pending}^{\mathsf{id}} = \bot$ and $\mathsf{next\_sequence}^{\mathsf{id}} = n$ ▷ Verify sequencing
    ensure ValidateOperation(id, n, O) $= R$ ▷ Validate the operation
    $\mathsf{pending}^{\mathsf{id}} \leftarrow R$ ▷ Lock the account on R
  return Vote(R) ▷ Success: return a signature of the request

function HandleConfirmation(C) ▷ Handle a certified request
  verify $\mathsf{cert}[R] = C$
  match $\mathsf{Execute}(\mathsf{id}, n, O) = R$ ▷ Allow regular operations only
  ensure $\mathsf{pk}^{\mathsf{id}} \neq \bot$ ▷ Make sure the account is active
  if $\mathsf{next\_sequence}^{\mathsf{id}} = n$ **then**
    run ExecuteOperation(id, O, C)
    $\mathsf{next\_sequence}^{\mathsf{id}} \leftarrow n + 1$ ▷ Update sequence number
    $\mathsf{pending}^{\mathsf{id}} \leftarrow \bot$ ▷ Make the account available again
    $\mathsf{confirmed}^{\mathsf{id}} \leftarrow \mathsf{confirmed}^{\mathsf{id}} :: C$ ▷ Append certificate to the log