Project TXA: Decentralized System for Cross-Chain, Peer-to-Peer Settlement of Digital Assets

Jae Yang
jae@txa.app

Arseniy Klempner
arseniy@txa.app

S. Andrew Lanham
andrew.lanham@tacen.com

May 12, 2023

Abstract

Project TXA is a protocol for peer-to-peer settlement of self-custodied digital assets. It allows traders to enjoy self-custody of funds enabled by decentralized smart contracts while trading on trust-minimised orderbooks. A cross-chain network of validators is incentivized to witness on-chain transactions and off-chain trade data, earning fees for relaying the resulting obligations to traders upon request. This enables a novel method of settlement where peers clear funds directly between each other through self-custody smart contracts. The paper elaborates on the architectural considerations of such a system in its integration with a digital assets exchange, listing, or auction system. The paper then discusses the design and key elements of the TXA Decentralized Settlement Layer and its novel method of settlement and clearance. Finally, the paper describes the morphology of TXA tokens through an in-depth exploration of how different types of closed-loop token ecosystems can interact together to incentivize all players in the ecosystem.

I. Introduction

Financial exchanges provide a market solution to parties which are in possession of an asset and wish to exchange it for a quantity of a second asset. The exchange design task is to provide low-latency (efficient), liquid, and fair exchange services between assets. Exchanges typically operate by keeping track of trader balances and orders and facilitating trades when desired quantities and prices match. This requires an order management and trade settlement system to allocate assets to trader accounts. Implementation of exchange services differ significantly, and are tailored to the assets in question. In this work we focus on exchange design for cryptocurrency assets.

Each solution to the exchange design problem may emphasize some requirements at the expense of others. For example, centralized exchanges typically excel at providing low-latency order matching between traders. However, centralized exchanges require users to cede custodial control over their assets. This requires extra vigilance, as it exposes users to at least two major custodial vulnerabilities: (1) centralized control of fund custody and (2) centralization of fund transfer. Both characteristics may engender perverse incentives within the operator of the exchange. Additionally, centralized exchange architecture creates a security theater where constant and exceptional vigilance (e.g. in the form of costly and byzantine audits) is required to confirm that funds are secure. In short, centralized exchanges trade security and transparency for speed and efficiency.

More recently, blockchain-based exchange solutions have emerged thanks to the technology's ability to provide efficient mechanisms for trustless consensus. Decentralized exchanges (DEXes)

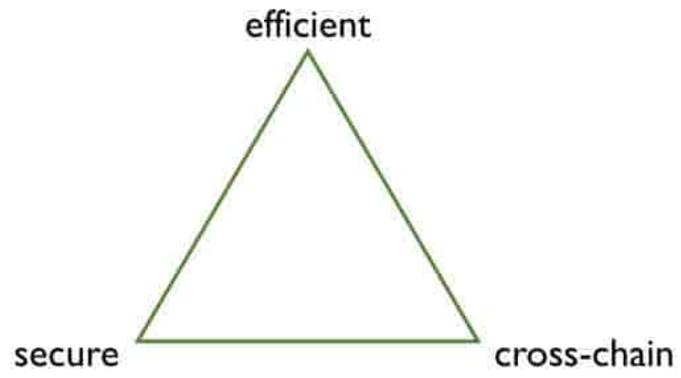

Figure 1: *Visual depiction of the trio of competing requirements in the exchange design problem. Highly secure, non-custodial systems sacrifice speed for security. Cross-chain operations can also incur excessive latency while exposing traders to cross-chain vulnerabilities.*

improve security and transparency by preserving trader custody of assets. However, they are limited by the underlying blockchain's blocktime for updating the ownership of an asset from one trader to another. Architectural constraints severely limit the ability to run a fully on-chain orderbook with competitively low latency. Furthermore, any benefits of security and transparency are contingent on proper implementation of smart contracts. Even if a contract is free of bugs, it may still grant privileged addresses the ability to call admin/upgradability functions, which can be misused to take control of funds or break determinism. Lastly, blockchain-based assets may originate from inherently different networks, for example, different blockchains, and a solution must handle exchange between them.

The above considerations motivate a trio of competing requirements that reflect important tradeoffs in exchange design (Figure 1). The first requirement is *efficient* operation, which encompasses competitive latency and infrastructure requirements as well as sufficient liquidity. The second requirement is *security*: this is primarily captured by the requirement that the solution be *non-custodial*, i.e. assets are maintained by exchange users rather than exchange operators. However, the security requirement also addresses a need for transparency of operation and of architecture. The third important requirement for a cryptocurrency exchange is the *cross-chain compatibility*, reflecting the ever-increasing diverse origins of this family of assets.

We propose the Project TXA **Decentralized Settlement Layer** (DSL) and **Cross-chain Settlement Protocol** (CSP) as a solution that successfully navigates these diverse requirements. Project TXA simultaneously overcomes the speed limitations of decentralized exchanges while improving on the security and transparency of centralized exchanges. It is designed with the ability for cross-chain monitoring and settlement without bridging requirements. Finally, it is developed as a protocol-level solution to the exchange design problem to create transparency and openness on both the architectural and operational level.

i. Core Design Objectives

We present the protocol at multiple levels of abstraction, starting from a high-level application layer and proceeding down to a payment signaling layer (Figure 2). The core features of the Project TXA protocol may be summarized as follows, presented roughly in order of the relevant layer of

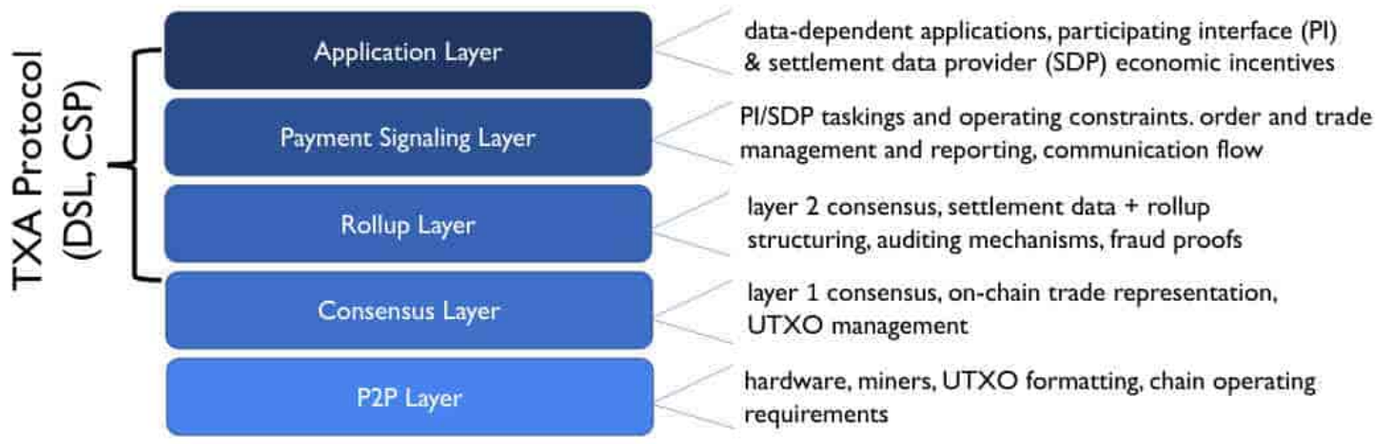

Figure 2: *Conceptual hierarchy of the layers of abstraction used by the Project TXA protocol in standardizing information handling for payment signaling and exchannge applications. We suggest that Project TXA encompasses protocols structuring data exchange through the rollup layer, the payment signaling layer, and part of the application layer (in terms of defining the incentive structure for applications built atop TXA.*

abstraction:

1. **Consistent rules**: The proposed system must have consistent rules and data across multiple chains. Rule consistency means that the way the settlement is handled must be same across multiple blockchains which the DSL spans. Since traders expect settlement rules to remain consistent from the time they enter the system to when they exit, smart contracts must have stable rule-sets that do not change between different versioned releases of the entire platform code.

2. **Deterministic trade execution and settlement**: Given an order of submission from participating traders, the handling of order matching and settlement should be deterministic. Constraining the system to provide deterministic results for a given set of input requires trade and settlement to be processed with causal relationships in mind. Imposing such a requirement also allows for events to be fully re-playable and yield the same result. Settlement thus needs causal relationships with one another, and the execution environment where such settlement calculation is done must be validated.

3. **Settlement network for independently operated markets**: The DSL is designed to accommodate any trading venue which conforms to the TXA protocol requirements. The protocol outsources settlement data provision from a PI-provided settlement structure, which incurs transparency and custody risk, to community-operated settlement data processing by Settlement Data Providers. Thus, the debit and credit relationships of market participants are determined independently of the PI.

4. **Cross-chain asset settlement**: The proposed DSL attempts to resolve the cross-chain settlement problem by instituting a simple collateral model which describes explicit state of money based on the life cycle of each trade, clearance, and settlement. Since cross-chain settlement is not purely atomic in nature, the associated settlement risk must be controlled.

5. **Value-at-risk for traders**: Since the proposed TXA DSL will have the responsibility of providing the settlement data, classical issues of collusion and time-based attacks need to be considered in assessing risk premiums and value at risk for traders seeking to settle using TXA DSL.

6. **Self-custody of collateral assets**: The proposed DSL aims to reduce systemic risks arising from relinquishing control and centralizing custody. An effective means to reduce perverse incentives is to restrict the actions that trading venue operators can perform beyond matching, recording, and reporting on debit-credit relationships. One can further reduce potential breaches of trust by removing the source of tension and temptation for the operators. Thus, the conflict of interest between principal and agent can be reduced further.

7. **Architectural simplicity for participating exchanges**: In existing centralized and decentralized exchanges, the exchange operators design the entire process path from deposit of collateral, trade processing, to clearance and settlement. The Decentralized Settlement Layer being proposed aims to reduce architectural complexity for the operators of the exchanges, thus lowering the barrier to entry for many potential digital assets exchange operators.

8. **Decentralized storage of orders and trade data**: One of the chief aims of Project TXA is to decentralize the data storage requirement for orders and trades so that secure storage of orders and trades do not impose high costs and increased complexity for the exchanges operating on top of the proposed Decentralized Settlement Layer (DSL). Secure storage is handled by SDPs, and protocol requirements are defined for consistent and uniform data validation.

Paper Outline

We continue by providing a review of related work in Section II. In Section III we present a high-level overview of the protocol with respect to the design objectives. Section

II. Motivation and Related Work

Permissionless blockchains introduce the opportunity for trustless and verifiable asset exchange. Current blockchain-based decentralized exchange solutions face significant performance bottlenecks, which has limited their potential for adoption to date. The primary bottleneck arises from the imposition that each order and transaction be verified via inclusion in a confirmed network block. This creates a latency in trade confirmation that is proportional to the block confirmation time. Lowering confirmation times can directly conflict with the security requirements of the network. The emergence of rollups and sidechain technology has improved confirmation latencies considerably, but not enough to justify an entirely decentralized orderbook architecture. For example, the underlying permissionless blockchains of Bitcoin and Ethereum can only process dozens of orders per second. With Optimism rollups in Ethereum, this increases to a few hundred milliseconds per order. But this pales in comparison to a typical trading exchange such as the NASDAQ, which can handle orders with a latency in the tens of microseconds.

The confirmation latency limitations in DEXs have motivated hybrid solutions that preserve trustless and verifiable asset exchange but that outsource trade management to centralized networks for the reduced latency. This general model has been broadly referred to as a payment channel network (PCN) [1, 2]. The PCN allows transaction processing to occur off the main network in a protocol-defined manner. In some cases, the processing occurs directly between

the transacting parties, while other protocols introduce third parties to mediate the off-chain transaction. Protocols that broadly adopt this model in the context of asset exchange applications include TEX [3], Gluon [4] and Project TXA.

Within the scope of PCN-based exchanges, there are a variety of new tradeoffs encountered. The foremost concern is the continued prevention of fraud. Other important concerns include information availability and proper incentives for network custodians. These considerations have led to implementations that emphasize the tradeoffs differently. A prominent example is the commit chain architecture, which supports user-custodied exchange with payment verification outsourced to a network of centralized but trustless *operators* [2]. Operators are tasked with maintaining balance of payments. The commit chain architecture uses an *account-based* data representation system, in contrast to networks which manage the UTXOs directly (*UTXO-based*). UTXO models preserve peer-to-peer settlement, while accounts-based models commit account balances directly. In practice, these representations approach the scalability problem differently.

More generally, a successful exchange should independently address the following areas of concern in a manner that does not catastrophically impact any other area:

- **Custody:** A user should not be forced to cede control of their assets to the exchange operator for management of the trades. Exchange operators may have perverse incentives.
- **Liquidity:** The exchange must be able to facilitate trades at fair price.
- **Latency:** Order and trade execution must occur on timescales requested by users. This is impacted by multiple factors, including illiquidity, insufficient infrastructure, computational overhead of the operations, or limitations of the underlying technology.
- **Latency Arbitrage:** Latency arbitrage occurs when information about user trades is broadcast and then exploited by other traders before it is executed, constituting a form of front-running and leading to perverse trader incentives [5, 6].
- **Infrastructure**: Exchange operation is degraded or a service outage is experienced as a result of high demand or insufficient supporting infrastructure. This is exacerbated by periods of market volatility, news, high-volume trading strategies, etc. A detailed model may be required for the exchange operators to understand the likelihood of a surge in trading and to allocate an appropriate amount of infrastructure.
- **Tick Size**: Order queuing creates adversarial races to optimize one's position in the ordering with low latency.
- **Private Information**: some traders have an asymmetric information advantage, creating opportunities to exploit uninformed participants. This could include an exchange operator illicitly profiting off privileged order information.
- **Transparency**: The exchange may not be able to accurately honor the listed prices due to possibly undisclosed implementation inefficiencies. More generally the exchange may not be able to verify that it operates in the manner that it claims to.
- **Uptime Requirement**: Traders may need to be online for settlements to be serviced.
- **Regulation Risk**: The technology driving exchange operation may be unapproved by regulators. This can prevent the exchange from providing consistent service.
- **Decentralized Incentive Risk**: When introducing novel centralized or decentralized systems of actors in an exchange architecture, there is a possibility for unforseen perverse incentives to emerge.

The underlying design principles of various exchanges can be seen as varied attempts to prioritize solutions to these inefficiencies, often some subset of at the expense of the rest. Our presentation of the TXA protocol is motivated to demonstrate how it addresses each of these exchange implementation concerns.

III. Peer-to-Peer Trading and Settlement

Project TXA is composed of a Decentralized Settlement Layer and a Cross-chain Settlement Protocol. Any exchange that operates on top of the TXA DSL must conform to the CSP. For simplicity, exchanges conforming to CSP will be called Participating Interfaces (). A PI is tasked with operating a trust-minimized orderbook and interfacing with a DSL. The DSL is designed in a manner that combines the performance and efficiency of a centralized service while minimizing trust requirements from participating traders.

The TXA DSL provides guarantees to all settlement participants that a trader's funds are immobilized and will only move as a result of smart contract logic based on trader-initiated settlement, SDP-reported obligations, and exchange-signed data. However, this on its own provides no guarantee to traders that a participating exchange does not manipulate trade data or improperly match orders. Nor does it prevent SDPs from colluding and reporting data that was not actually signed by the exchange. Without these guarantees, PIs and SDPs can manipulate settlements and steal trader's funds. To mitigate this, the TXA smart contracts must enforce constraints on all settlement data generated by PIs and reported by SDPs.

The most trustless way to enforce these constraints would be to simply implement the entire logic for both order-matching and settlement in on-chain smart contracts. However, this severely limits order and trade throughput and imposes high costs for traders due to transaction/gas fees. Instead, we rely on a combination of cryptographic verification and economic incentives whereby auditors explicitly detect and punish violations of constraints while implicitly allowing correct state updates to pass through. This allows us to delegate most of the storage and computation costs of settling trades to off-chain entities. Additional data is only ever brought on-chain to prove to the smart contracts that a state update is invalid.

In order to protect such a system from Sybil attacks, PIs and SDPs must stake collateral in Collateral Custody Contracts in order to operate. These smart contracts implement the logic for when and how much collateral must be locked, slashed, or released during the settlement process. In concert with the ACCs they also handle payment of fees from traders to PEs and SDPs upon successful completion of each settlement. The TXA Cross-chain Settlement Protocol, presented in more detail below, prevents the manipulation of settlement data through a cryptographically and economically enforced system of incentives.

We outline the role of each element of the TXA ecosystem, visualized in Figure 3.

Trader

The trader is an end-user. The trader collateralizes assets through the PI via self-custodied escrow contracts, called the Asset Custody Contracts (ACC). The trader is required to collateralize digital assets at a ratio of only slightly above 1. The risk premium involved in trading with each ACC is a function of multiple risk factors, such as systemic risk, individual settlement risks, underlying asset viability, and so on. The trader may unlock funds with a settlement request, and funds are released once their validity is confirmed in the DSL.

Upon initial trader deposit, assets in the ACC become immobilized. The trader is then granted a corresponding balance on the centralized venue to make or take liquidity on the orderbook. The trader begins interacting with the PI through the orderbook, triggering the following tasks and responsibilities in the PI: (1) verification of asset collateralization, (2) orderbook management, and (3) trader internal balance-of-payments updates, (4) trade reporting to the peer-to-peer coordinated settlement network.

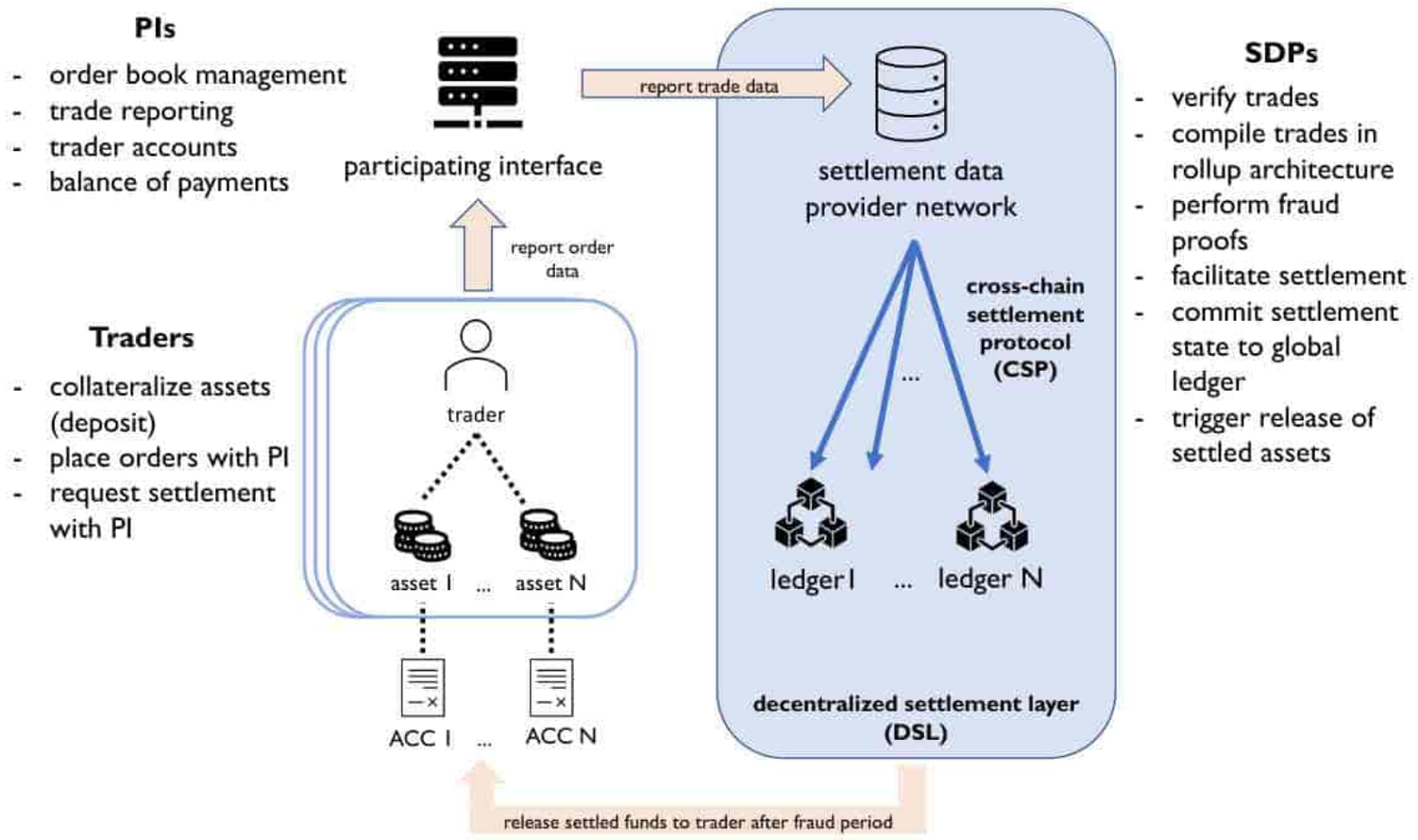

Figure 3: *A high-level diagram of the Project TXA system. Traders interact with PIs for orderbook management. PIs report trade activity to the peer-to-peer SDP network, which verifies trades and enables settlement and release of settled funds to the trader.*

Participating Interface

The participating interface's core responsibility with respect to the protocol is to process incoming deposit and settlement events, to generate new trade events arising from its order matching, and to submit these events to the ledger processing engine for verification by the SDPs. Two or more orders are considered executed once they are included in a trade signed by the PI, and once the PI emits the trade for witnessing by a peer-to-peer network of Settlement Data Providers (SDP). SDPs record both initial deposits to the Asset Custody Contracts and monitor all trades emitted by the PI. This allows them to maintain a database of obligations between counterparties involved in trades. At any point, an SDP servicing a PI may detect an opportunity to earn fees by providing trade and obligation data for a trader settlement request. This allows a trader to request assets owed as a result of trading on the PI, and that the assets are accurately accounted for.

PIs must not spend a trader's balance without the trader's explicit authorization via a signed, standard order. They must also prevent traders from spending more than what was originally deposited or credited as a result of trading activity. All trades emitted by the exchange must reference two orders, each signed by a trader. The smart contracts validate that the orders were signed by the same traders that had sufficient balances of the assets in question and that the parameters of each order (such as price or size) properly match with each other and with the resulting balances of the trade.

Settlement Data Provider

SDPs conduct settlements in a peer-to-peer manner, with a PI providing no service or data at the time of settlement. The PI and SDP operators are both subject to incentive and penalty systems based on verification of net obligations by other nodes in the network. This avoids the issue of placing too much trust in a single SDP. Importantly, the settlement model allows for cross-chain

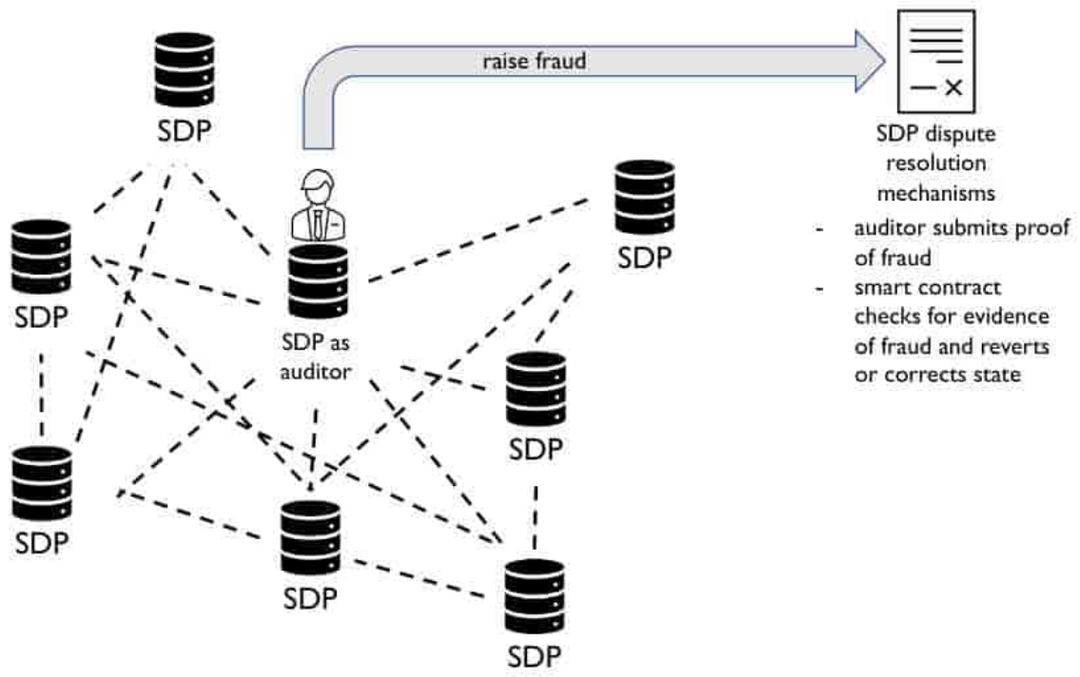

Figure 4: *The SDP network is comprised of a peer-to-peer network of SDP nodes. These nodes may act as data providers or optionally as auditors of other SDPs. SDPs acting as auditors may report fraud with proof, triggering a dispute resolution mechanism and possible reversion of the network state.*

settlement (depositing an asset into an ACC on one chain and trading it for an asset on an ACC on another chain) through duplication of trade data across each chain and RPC-based bridging of fraud detection.

For each settlement, SDPs must report trade data that was signed by the exchange. Trades reported must be in the correct range and sequence. The smart contracts validate that each trade was signed by the same participating exchange for which settlement was requested and that the identifier of each trade is within the correct range of identifiers.

When a trader initiates settlement of an asset through the ACC, it informs the network of SDPs and the PI by emitting a smart contract event. The PI signals to the SDPs that it has acknowledged the request and marked the corresponding ledger entries as settled. At this point, both the trader's assets in the ACC and the trader's balance on the participating exchange are immobilized. By including the acknowledgement in the same linear sequence as trades, the PIs allow the SDPs to use a snapshot of balances at a discrete moment in the sequence of events to calculate the resulting obligations (analogous to a time-tick). After obligation data is reported and a quorum is reached among SDPs, the trader is able to request owed funds from peers and withdraw the asset. A dispute period must pass before the participating SDPs are able to withdraw any provided collateral.

IV. Data Representation in the Rollup Layer

We now outline the relevant data structures for enforcing the protocol operating constraints, introducing appropriate notation where relevant. To enforce constraints on the data emitted by the participating exchanges and reported by SDPs, the CSP defines a standard way to represent orders, trades, and balances of a deposited asset. Recall that Project TXA is implemented with an unspent transaction output (UTXO)-based data representation. UTXO data structures mediate

interaction with the DSL and structure PI and SDP protocol participation. The state of the balances in a participating exchange on the DSL is represented as a set of UTXOs in a directed acyclic graph (DAG) structure and also as a hash-linked list of trades.

We introduce basic notation for UTXO-mediated transactions. We use $[n]$ for natural number n to abbreviate the set $\{0, \ldots, n-1\}$ where relevant. Protocol **end-users**, alternatively referred to as *traders* or *transactors* are indexed by a natural number, so we have $u \in [N]$ where N is some total number of possible users. Each user is assigned a public and a private key. For user u we define the functions returning the public and private key as $H_{\text{pu}}(u)$ and $H_{\text{pr}}(u)$. Suppose there are M choices of assets in the asset universe. Then we have asset set $\mathcal{A} = [M]$ and assets $a \in [M]$. The descendants of a node v are those nodes w for which a directed path exists.

The UTXO model is best visualized as a time-dependent *transaction graph*. For discretized time interval k we define the graph $\mathcal{G}[k] = (\mathcal{V}[k], \mathcal{E}[k])$ – the graph whose vertices are a set of transactions $v \in \mathcal{V}$ and set of edges $e \in \mathcal{E}$. Edges are represented as tuples comprised of a public key k, an asset $a \in \mathcal{A}$, and a spend amount $v \in \mathbb{R}_+$, where $\mathbb{R}_+$ signifies the positive-signed real numbers. Example: $e = (H_{\text{pu}}(u_0), a_1, 0.1)$ indicates that user u_0 spent 0.1 of asset a_1, which is represented as an edge. If edges are directed, as in the example of Figure 5, the arrows depict the chain of causality, i.e. the evolution of previous UTXOs to new UTXOs. The spend amount associated with an edge is $A : \mathcal{E} \mapsto \mathbb{R}_+$, e.g. $A(e) = 0.1$. The asset associated with an edge is $\rho : \mathcal{E} \mapsto [M]$, e.g. $\rho(e) = a_1$.

A *vertex*, or a *node* $v \in \mathcal{V}[k]$ encodes transactions as transformations of edges. A vertex $v \in \mathcal{V}[k]$ is, therefore, a tuple comprised of a set of incoming edges $E_{\text{input}}(v) \in \mathcal{E}[k]$ and a set of outgoing edges $E_{\text{output}}(v) \in \Omega_{\text{pu}} \times \mathcal{A} \times \mathbb{R}_+$. Each epoch, the new transactions create a new set of edges which form a part of the new set $\mathcal{E}[k+1]$. Deposit vertices are those which have a deposit edge as an outgoing edge. These are a special type of vertex for which $E_{\text{input}}(v) = \emptyset$, i.e. they have no incoming edge so the input set is empty. Other than deposit vertices, a vertex can be interpreted as joining two or more edges, i.e. public keys and amounts.

The set of descendants of a vertex can be used to verify necessary relationships for balanced transactions. We define the function $P : \mathcal{V} \times \mathcal{V} \mapsto \{0, 1\}$ as the directed path function, where $P(v, w) = 1$ if a directed path exists from vertex v to vertex w. If $P(v, w) = 1$, then the vertex v is the *ancestor* of w and w the *descendant* of v. The graphical characterization of UTXO relationships invites diagrammatic depictions like the example graph depicted in Figure 5.

The set of edges $\mathcal{U}[k] \subseteq \mathcal{E}[k]$ which are not connected to a vertex at both ends forms the set of **UTXO edges**, or **UTXO**s. A *new transaction* of asset a in amount x for user u at time $k+1$ is captured by a vertex v whose input set $E_{\text{input}}(v)$ contains that user's UTXO $(H_{\text{pu}}(u), a, x) \in \mathcal{U}[k]$. The vertex defines a set of output UTXOs resulting from the transaction, captured in $E_{\text{output}}(v)$. New transactions are logged as new vertices in the graph $\mathcal{V}[k+1]$. A UTXO may additionally represent an input to an asset custody contract, where it serves as a *deposit*. A deposit operation is a *new transaction*, represented as a vertex with no incoming UTXO edges and one outgoing UTXO edge $d \in \mathcal{U}[k+1]$. We formally distinguish deposits as the subset of UTXOs $\mathcal{D} \subseteq \mathcal{U}$ arising from deposit actions. An initial deposit amount by user u in asset a is captured by the function $D(u, a)$.

Deposit

A deposit $D(u)$ represents a balance of a digital asset that was deposited into the ACC, linked to user u. Upon deposit of a digital asset into the ACC, the contract emits an event. When the PI detects this event, it creates a UTXO for the amount deposited and credits it to the depositor's balance on the PI. The PI must then broadcast a signed acknowledgement of the deposit, which is validated by SDPs. When the depositor places a trade, the PI references the deposit UTXO as a

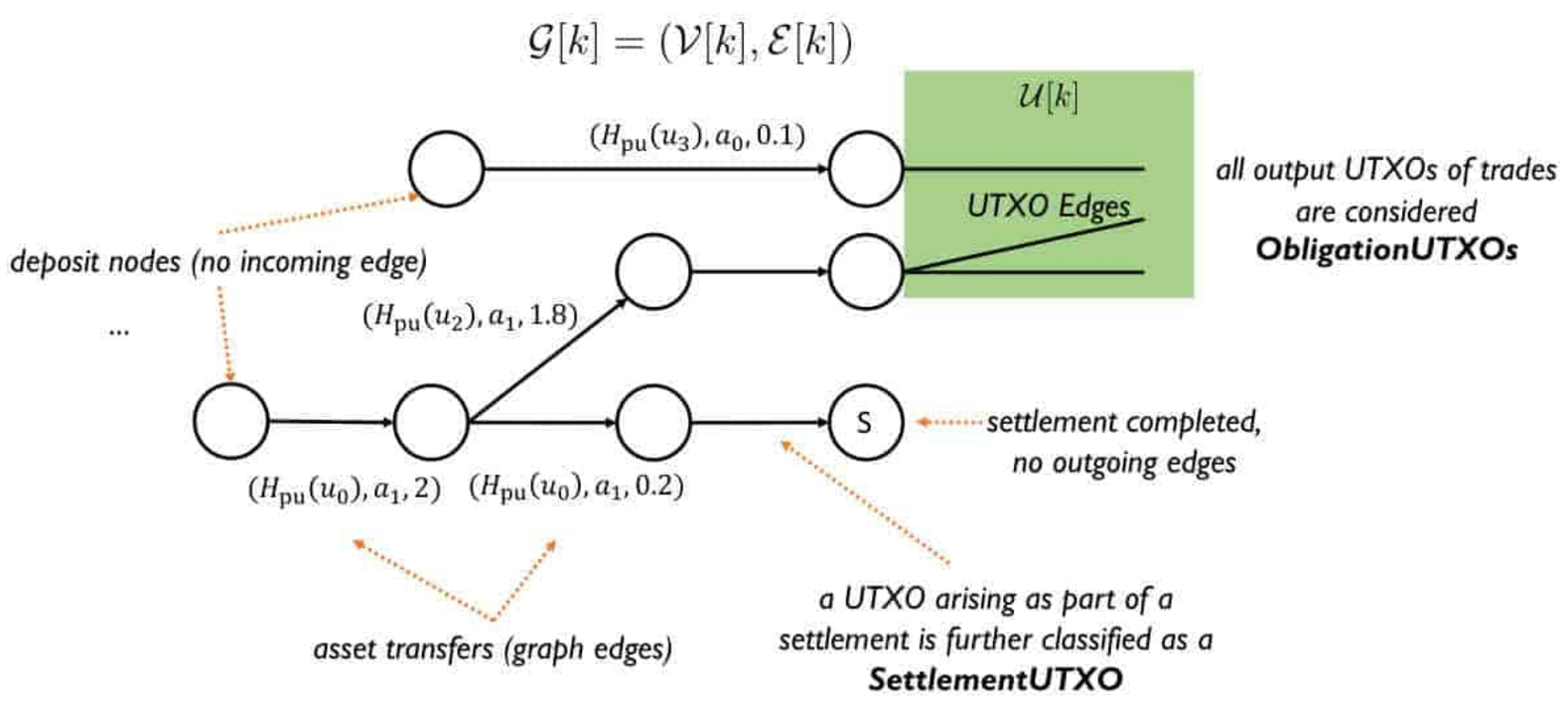

Figure 5: *A diagrammatic depiction of a UTXO graph state at an arbitrary time interval. Circular nodes represent the vertices in the vertex set $\mathcal{V}[k]$, while the set of edges forms the set $\mathcal{E}[k]$. Edge labels are explicitly represented for some of the edges, but each edge is characterized by such a tuple.*

spent input when proposing an update to the ledger. All balances on the PI must therefore be traced to deposit UTXOs. This yields an invariant: the sum of the outgoing edges of a deposit UTXO's descendants that do not themselves have descendants must be equal to the original amount of the deposit UTXO. Since the outgoing edges of a childless UTXO vertex are the set of UTXO edges, we may say that

$$\sum_{w:P(v,w)=1} \sum_{e \in E_{\text{output}}(w) \cap (\mathcal{U} \cup \mathcal{S})} A(e) = \sum_{f \in E_{\text{output}}(d)} A(f), \quad \forall d \in \mathcal{D} \tag{1}$$

Order

Orders are handled on the participating interface and are processed independently of the sidechain. This allows for relative freedom in the data representation of orders. Nevertheless, there are some minimal representation constraints that must be satisfied by orders. A user *balance* is a function $B : [N] \times \mathcal{A} \mapsto \mathbb{R}$ representing the quantity of an asset that a user possesses. For example, user u's balance of asset a is $B(u, a)$. Then an order must specify an asset pair and an amount, where the amount does not exceed the balance. The order must be associated with a user via a signature generated using the private key $H_{\text{pr}}(u)$. Orders must be further specified as buy or sell types, as well as the order execution type (e.g. limit, market, stop). Identical orders are those which are placed for the same asset at the same price, and must be distinguished by the time of placement and serviced in order of arrival.

Trade

A trade is generated and signed by a PI when two or more user-initiated orders are matched. A trade carries a partial representation in the UTXO layer, where its effect can be understood as a transformation of a set of input UTXOs into a set of output UTXOs. This is captured by the inclusion of a new vertex in the UTXO graph representation which encodes the transformation. At the level of the PI, trades are further associated with the users that signed the matched orders and their balances. The order matching condition requires that the sum of the amounts in the orders for each asset must match, given exchange ratio p_1 for asset 2 in terms of asset 1. This is extended to an equivalent condition on UTXOs, where the user input UTXO amounts must also match. Supposing a vertex v captures a trade between assets a_1 and a_2, we require that

$$\sum_{\substack{e \in V_{\text{input}(v)} \\ \rho(e)=a_1}} p_1 A(e) = \sum_{\substack{e \in V_{\text{input}(v)} \\ \rho(e)=a_2}} A(e)\,. \tag{2}$$

Input UTXOs for a trade must be either deposit UTXOs or obligation UTXOs without children.

Just as with orders, the PI is responsible for linearly sequencing all trades according to the time they were serviced, granting each an incrementing identifier according to time of placement. In addition to proper exchange-sequencing, the trades must be hash-linked, meaning each trade must include the hash of the previous trade in the sequence. Each trade must include two orders signed by a trader, as well as the price and size of the base asset. The parameters of the two orders must match with each other, as well as the price and size specified by the trade.

Each trade generates a minimum of four UTXOs:

- one or more output UTXOs credited to maker
- one or more output UTXOs credited to taker
- two or more output UTXOs credited to PI and SDP for fee payment

The sum of the amounts of output UTXOs must match the sum of the amounts of the input UTXOs, per asset in the trade, i.e., given a vertex v representing a trade,

$$\sum_{\substack{e \in E_{\text{input}}(v) \\ \rho(e)=a}} A(e) = \sum_{\substack{e \in E_{\text{output}}(v) \\ \rho(e)=a}} A(e) \quad \forall a \in \mathcal{A}\,. \tag{3}$$

Obligation UTXO

As soon as a deposit is used as an input in a trade, the deposit UTXO no longer represents the latest balance on the exchange, and is split into one or more new UTXOs. The new UTXOs are considered children of the input UTXO. All output UTXOs of trades are considered Obligation UTXOs until they become part of a settlement.

Settled UTXO

When a trader requests settlement of an asset, the PI must identify all obligation UTXOs that meet the following criteria:

- The address requesting settlement belongs to the trader to whom the UTXO is credited
- The asset represented by the UTXO is the same asset being settled
- The UTXO has no children

The settlement consumes those UTXOs as inputs and generates vertices with no output edges as outputs. The sum of the UTXOs connecting to these settlement vertices represents the total amount that the trader will be able to request from counterparties and withdraw through the ACC. Let S be a settlement node, then the total amount requestable r is defined as $r = \sum_{e \in S_{\text{input}}} A(e)$. Each Settled UTXO thus represents a portion of an immobilized deposit that can now be transferred. When reporting Settled UTXOs, the sum of the collateral staked by SDPs participating in the settlement must be greater than the sum of the amounts of the UTXOs.

i. Rollup State Structure

Deposits, Trades, and Settlements proposed by the PI along with all input and output UTXOs are packaged into blocks. The headers of each block are stored on-chain. The UTXOs are aggregated in a Merkle tree structure for storage, following typical formulations of such structures [7, 8]. Let $\mathcal{M}$ designate a general Merkle tree structure. Because the Merkle tree state may have a dependence on the settlement period, divided arbitrarily, we may choose to reflect this dependence, e.g. $\mathcal{M}[k]$ at discretized time k. Each block header for a block of events serves as the root hash of a Merkle tree. Thus, the state that the funds have reached on-chain is represented by a sequence of events that can be compared to blocks in a typical roll-up design. Each settlement stores hashes akin to block headers.

Accumulators are a type of Merkle tree that are used significantly throughout the TXA protocol. For example, the *trades accumulator* can be defined as $\mathcal{M}_{\text{trade}}$. Its leaves are the trade UTXOs specified in Section IV. The settlement accumulator is denoted $\mathcal{M}_{\text{settle}}$. It stores all obligation UTXOs that are relevant to a given settlement block. The rollup contract must provide functions that can be called by auditors to prove that a proposed settlement contains state that is valid. The Merkle tree rollup provides a structure designed around such verification actions.

i.1 Data Storage

Each data type must be stored on-chain and off-chain. The protocol specifies storage mechanisms for key data structures, highlighted below.

- Deposits
  - **On-chain**, every deposit UTXO is first generated from a transaction to the Asset Custody Contract. The UTXO resulting from the deposit is hashed and used as a key in a mapping to store the deposited amount.
  - **Off-chain**, each deposit is stored as an uncompressed UTXO.
- Trades
  - **On-chain**, each trade must reference a minimum of four UTXOs. A minimal-UTXO trade will require two UTXOs for the two orders used to define the trade. Another two 2 are input UTXOs (1 for each asset in the trade). Every trade stores the hash of the previous trade in the sequence. For every settlement, the latest trade hash is reported. Every trade occurring in the settlement period is packaged into a Merkle tree. The root of the tree is reported, while the final trade message is reported unhashed.
  - **Off-chain**, every trade is stored uncompressed until enough time passes such that it meets the condition for being pruned.
- Obligation UTXOs:

  - **On-chain**, for every update to the state of obligations, all new output UTXOs generated as a result of trading are packaged into a Merkle tree $\mathcal{M}_{\text{trade}}$. The root of the tree is reported.
  - **Off-chain**, every obligation UTXO is stored uncompressed until enough time passes such that it meets the condition for being pruned.
- Settlement UTXOs
  - **On-chain**, for every settlement, all obligation UTXOs that will be frozen as a result of settlement are packaged into $\mathcal{M}_{\text{settle}}$. The root of the tree is reported. Once a settlement is finalized, the UTXOs must be used to update on-chain balances by being written as obligations, thus affecting the amount of an asset one trader can claim from another trader within the Asset Custody Contract.
  - **Off-chain**, Every uncompressed UTXO has a flag indicating whether or not it has been included in a settlement.

i.2 Rollup State Invalidation

A series of assertions are included as a pre-defined set which applies to the proposed state. Most of these assertions involve referencing state that was committed to the contract in the form of deposits or previous blocks. The assertions are defined to accept accumulator roots as input, and to return a confirmation bit asserting validity (1) or invalidity (0). Assertion functions can be further categorized into: (a): membership assertions (answering questions of the form “is this leaf a member of either this block or a previous block?") and (b): invariance assertions (function answers questions of the form “does the tree have this invariance property?").

ii. Fraud Proofs

For a smart contract to enforce protocol operating constraints on relevant data structures, it must additionally verify that the state of balances after a settlement is the result of a valid transition from the state of balances before a settlement based on deposits, trades, and previous settlements. Validating every trade and calculating updated balances after each trade is too computationally slow and expensive to perform on-chain. Leveraging a rollup-style architecture allows the SDPs to package many trades and changes to balances into a single on-chain state update. Instead of writing every single trade to the chain, the SDPs construct a Merkle tree using a set of trades and report the single root hash of that tree.

In order to verify the validity of the state change, each event is subject to a fraud period during which an auditor can submit proof that the smart contract can use to determine if the Merkle root(s) included in the update are invalid. If an auditor submits proof and it passes all checks by the smart contract, then the state update is either reverted or corrected (depending on the kind of fraud) and assets posted as collateral by the SDPs which proposed the update are penalized. Some of the slashed funds must be rewarded to the auditor who reported the fraud. A similar mechanism is used for reporting fraud committed by a participating interface. Participating interfaces must also provide collateral to operate on the TXA DSL. If fraud is reported, the collateral is slashed and the DSL stops supporting settlements for that exchange up to the trade where fraud was detected.

ii.1 Collateral Requirements

By requiring PEs and SDPs to stake collateral in order to operate on the DSL, TXA can use positive and negative incentives to maintain proper behavior of participants.

SDP

In order to report settlement data, an SDP must stake at least two tokens:

- Settlement Asset: same asset that users being serviced request for settlement. Total staked by SDPs must be greater than total amount reported as settled from any given settlement event.
- TXA: Total staked by SDPs must be greater than 10% of the total settlement asset amount, with a flat minimum set by governance.

Both tokens are locked when the SDP reports data for a settlement. In the locked state, the collateral cannot be queued for withdraw or used to qualify for a different settlement. If an auditor detects fraud in the proposed settlement data and reports it to the smart contract, both tokens will be slashed and used to compensate any traders, reward the auditor, and optionally pay a community fund. If the settlement completes without fraud, the SDPs earn fees proportional to the amount settled (and thus to the amount staked). Fees are paid in the settled asset.

PI

The TXA smart contracts do not allow any deposits for trading on a PI until a significant amount of collateral is provided by the PI. The amount of collateral is determined by governance, and used to punish the PI if any violated constraints are detected in events signed by the PI. Any aggrieved parties are compensated with the slashed funds, with anything remaining going to a community fund.

V. Consensus and Dispute Resolution

In this section we formally detail fraud resolution mechanisms required for protocol constraint enforcement. We express all sub-protocols in the language of the formalized data structures presented in Section IV. Algorithmic descriptions of relevant fraud proofs can be found in Appendix B.

i. SDP Fraud

The most likely auditor to report fraud by an SDP reporting for a settlement would be another SDP, since any SDP with the latest trade data from the PI and the latest events from each blockchain can quickly calculate the correct settlement data and compare the resulting Merkle roots. Fraud reporting requires identification of a specific instance of fraud in the settlement and a report to to the smart contract.

Fraud committed by SDPs is mostly limited to:

- Incorrectly ordering trades in the Merkle tree
- Omitting a trade that should be included
- Including an invalid trade (doesn't have signature of correct PI)

If there is fraud related to incorrect order-matching by the PI, SDPs are expected to report it outside of the settlement process as it supersedes cases of SDP fraud. Ideally, the incentive for an SDP to report PI fraud should be much higher than an SDP earning fees on a settlement with fraudulent data. If an SDP reports it anyways, it will be slashed when an auditor reports the PI fraud.

ii. PI Fraud

The types of fraud committed by a participating interface are more complex, and mostly arise from either a violation of UTXO invariants, or from failing to sequence on-chain events in time-ordered fashion.

PI fraud modalities may be summarized as:

- Trade has no input UTXOs
- Trade input UTXOs have already been spent
- Sum of amounts of output UTXOs do not match sum of input UTXOs
- Deposit, Trade, or Settlement sequencing doesn't match expected ordering or trade sequence number has been already used

The PI is therefore required to maintain proper classification, accounting, and time-sequencing of all new on-chain events. *State updates* are the set of events which cause an update to the ledger state. It is crucial that the PI performs state updates correctly, as errors can create conditions for the SDPs to report fraud and halt support for the PI. Proper sequencing of state updates is partially enforced by the normal operation of the smart contract with additional event-handling oversight provided by the PI and confirmed by SDPs. Proper cross-chain sequencing is a related topic that we discuss in this section. At a high level, the PI, throughout the course of its general event processing tasks, *submits* proposed state updates to the network in a standardized form for consideration by SDPs. The state update is then subjected to validation checks by the SDPs that uphold honest operation of the network.

The possible classes of state updates are four-fold: the `DepositAcknowledgement` indicates a user-initiated deposit event at the PI, the `Trade` event indicates that orders have been matched and assets marked as exchanged, the `SettlementAcknowledgement` is generated from a user-initiated request for settlement, and the `SettingsAcknowledgement` is generated as part of the PI-onboarding process on the network. One of these four classifications is stored as the event `typeIdentifier` of a state update message. As mentioned, the PI must sequence these state updates in a linear, time-ordered fashion for each chain.

The SDP network monitors for irregularities in trade reporting. The UTXO model presented in Section V serves as a useful framework for expressing violations of trade conditions. Considering a trade, represented as a vertex v, in context of the first three PI fraud modalities presented in this subsection. A reported trade with no input UTXOs is equivalent to the condition $V_{\text{input}}(v) = \emptyset$. A reported trade with one or more UTXOs already spent indicates that $\exists e \in V_{\text{input}}(v), v_2 \in \mathcal{S}$ such that $e \in V_{\text{input}}(v_2)$. A mismatch between the sum of the input and the sum of the output UTXOs is captured as the condition of Equation (3).

ii.1 State Update Message Structure

PI state update proposals are generated according to a protocol-defined message format and reporting process. First, every state update proposal consists of the following:

- `uint8 typeIdentifier`: this specifies the type of state update, previously mentioned as either `DepositAcknowledgement`, `SettlementAcknowledgement`, `Trade`, or `SettingsAcknowledgement`
- `uint256 id` a sequential identifier of the message in the participating interface's ledger
- `address participatingInterface`: the address of the participating interface for which the message applies
- `bytes structData`: data that deserializes to the struct specified by the `typeIdentifier`
- `bytes32 previousUpdateHash`: the hash of the previously signed state update

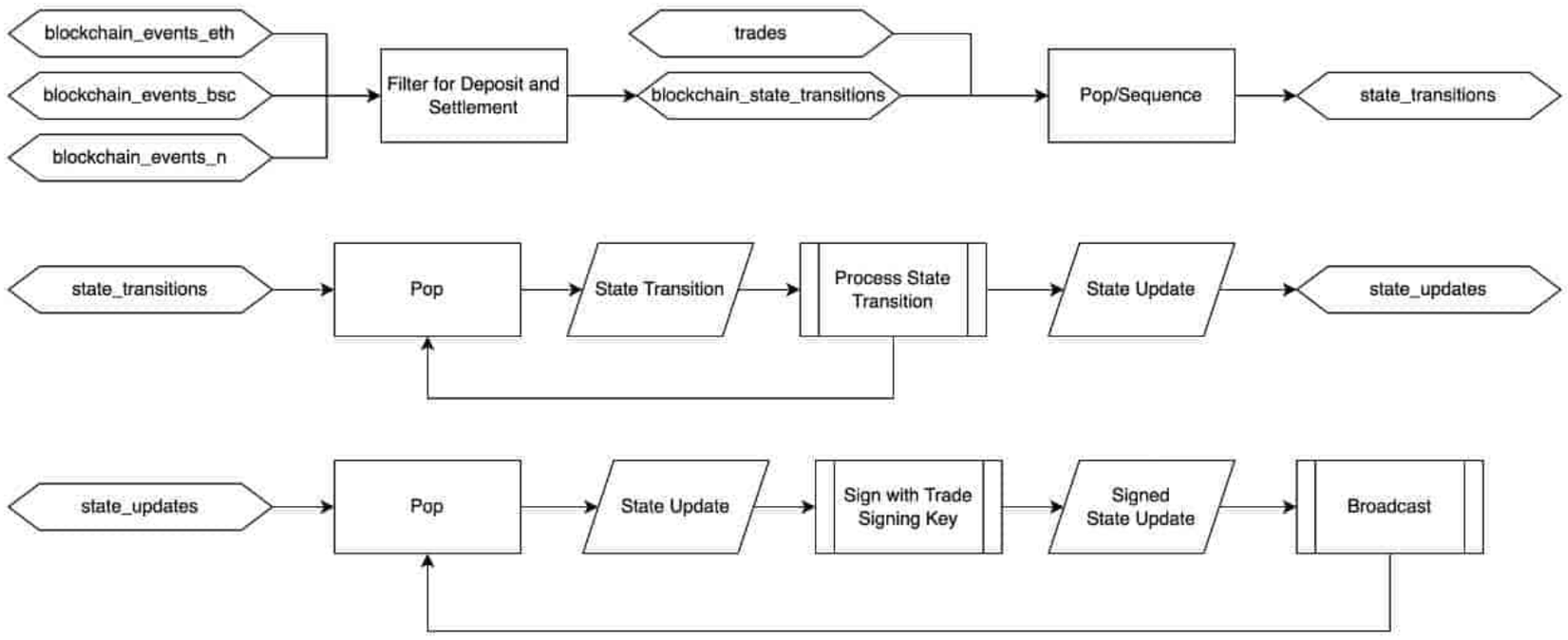

Figure 6: *The dynamics of a PI state update process.*

- `Signature sig`: the signature of everything above, signed with the Participating Interface trade signing key

ii.2 State Update Proposal Dynamics

The PI-coordinated state update process can be broken into the events depicted in Figure 6. All incoming chain events are filtered to retrieve deposits and settlements. These constitute the blockchain state transitions. The stream of trades, which originates with the PI order-matching process rather than on-chain, is merged into the event stream and all are enqueued in sequential order. The resulting stream is the set of state transitions. The PI processes state transitions by structuring them in the protocol-compliant message format, after which they become state updates. The state updates are finally signed by the PI and broadcast to the SDP network for verification.

ii.3 Event Sequencing

We first summarize the contributions provided by the smart contract itself toward maintaining honest PI sequencing. The contract associated with each chain assigns a `chainSequenceID` to each event. This represents a contractually-enforced canonical ordering that can serve as an oracle to verify honest PI sequencing. Indeed, `chainSequenceID` arises as an invariant in the DepositUTXO fraud checking process, described in Appendix B and enforced by SDPs during settlement. In a similar manner to deposits, the PI settlement request processing must uphold honest time-sequencing on the PI layer. To this end, the contract also generates a `settlementRequestID` functioning as another sequencing label that can be used to verify honest PI sequencing. This procedure has an obvious benefit – honest PIs can simply consult the smart contract sequencing data to enforce honest sequencing.

The sequencing IDs originating from the smart contract aid the deposit and settlement verification mechanisms implemented by the SDPs. Trade events, on the other hand, are subjected to a different series of fraud check mechanisms within the SDP network. Specific fraud mechanisms are covered in Appendix B as part of algorithms `validateInputUTXOs` and `validateOutputUTXOs`. Finally, `SettingsAcknowledgement` updates are verified in a custom manner as they are PI-specific onboarding requirements. Specifically, onboarding includes setting the public address of the update signing key, the fixed fee percentage paid to the PI, the fixed fee percentage paid to the

SDP and other protocol-relevant parameters. The very first signed state update emitted by the PI includes this data, which must match an on-chain record. Please note that all classes of events are sequenced in a single queue per chain.

While the above specifies verification mechanisms for PI-generated state updates, the mechanisms described so far generally treat each chain individually, leveraging the fact that each one is serviced by an associated smart contract. However, PI responsibilities require maintenance of state update proposals across all supported chains, in adherance to the CSP. This implies additional responsibilities for maintaining proper event sequencing across different chains simultaneously. In particular, a PI may opt to report state updates on one chain preferentially, disrupting cross-chain synchronization. A major incentive serving to curb this behavior is that many trading pairs provided by a PI are naturally cross-chain in the sense that the inputs used to generate a trade involve assets from different chains. Since trades are the fee generation mechanism for PIs, PI's are generally incentivized to report all incoming chain activities impartially. This incentive provides another assurance of cross-chain synchronization arising from the CSP.

Deposit and settlement state updates may occur relatively asynchronously relative to order processing. The protocol, in other words, can tolerate a slight lag in the state update processing relative to order processing, since settlement is less time-sensitive than order processing. On the other hand, a constant lag that's too large can cause a processing bottleneck, where an ever increasing queue of orders and trades across parallel markets is routed through a slow, serial settlement process. This potential bottleneck is avoided by the PI's infrastructural responsibility of keeping the relative lag times short between off-chain order processing and state update proposals. Queues are expected to be cleared during periods of relative inactivity on the exchange (e.g. low-volume sessions).

iii. Data Availability Challenge

Since all trade data for a settlement is hashed into Merkle roots, the smart contract has no knowledge of the contents of the actual trades until an auditor provides proof that certain data belongs in the tree represented by one of the roots. While the auditors themselves should know the correct unhashed data, they may not have the unhashed data reported by a fraudulent SDP. Without the unhashed data, the auditor cannot prove to the smart contract that fraud has occurred.

To prevent lack of data availability from catching a fraudulent settlement, the smart contracts must enforce a data availability challenge. The data availability challenge allows an auditor to stake some TXA and request that a reporting SDP provide an unhashed trade with that belongs in a specific position in the Merkle tree. This is possible because we require all trades to be ordered by their sequence number in the Merkle tree. Auditors can repeat this process until finding a leaf that is different via a binary search. If reporting SDPs fail to provide proofs in a timely manner, it is assumed that the missing data would reveal the fraud and the settlement is reverted. If the reporting SDPs comply to all requests, then the auditor should be able to detect the fraudulent leaf within a reasonable period.

VI. TXA Tokens

i. Morphology of TXA Tokens

The TXA token ecosystem consists of four tokens: TXA, TXA.B, TXA.L, and TXA.D

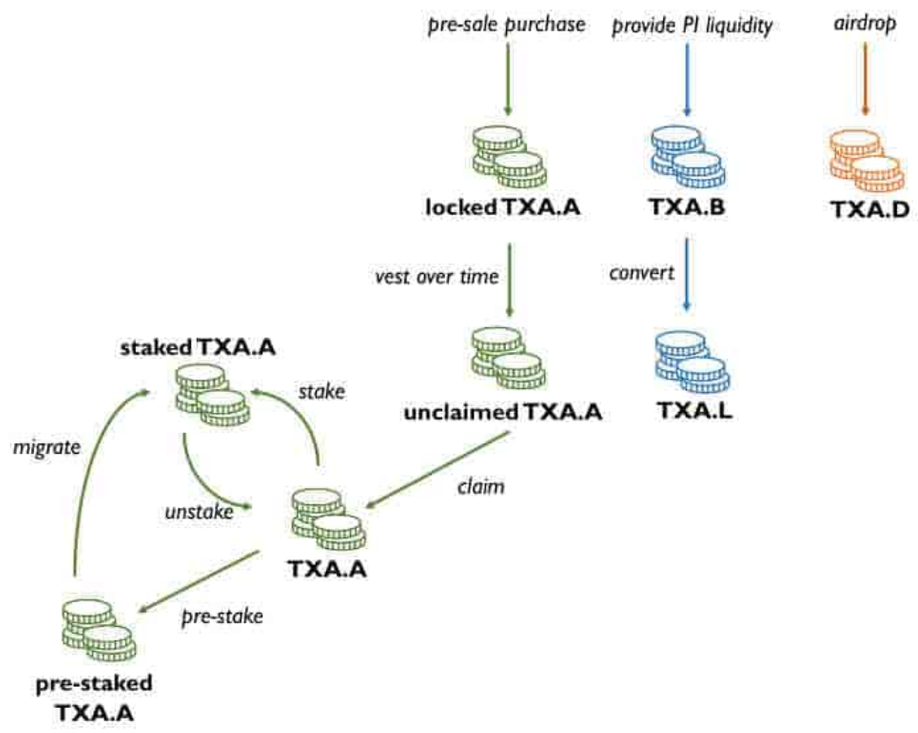

Figure 7: *visual depiction of the TXA token morphology and relations between tokens.*

i.1 TXA Token

TXA Token is an ERC777 token minted to act as an ecosystem token that can be used for:

- Fee replacement
- Governance
- Staking
- Settlement Risk Management
- System Reputation Management

The TXA token has been deployed on the Ethereum mainnet at address here [1].

i.2 TXA.B

TXA.B Token is minted for traders who participate and provide liquidity for participating interfaces. It is not transferable between addresses, nor is it intended for listing in secondary markets.

i.3 TXA.L

TXA.L Token is minted when TXA.B is destroyed by a holder of TXA.B. It is not transferable between addresses, nor is it intended for listing in secondary markets.

TXA.L Token is used for:

- Fee Replacement
- Governance
- Staking
- Settlement Risk Management
- System Reputation Management

[1] The TXA token has additionally undergone an audit, the results of which can be viewed in this GitLab repository: `https://gitlab.com/ProjectTXA-audit/txa-token`

i.4 TXA.D

TXA.D Token is airdropped to pre-staking participants as well as to addresses with locked and unclaimed TXA tokens at the time of the airdrop snapshot.

TXA.D Token is used for:

- Governance

Appendices

A. Technical Specifications

In this appendix section we present a detailed technical description of the TXA protocol data types and requirements.

i. TXA CSP Data Types

This section describes the data types that must be standardized across participants in the DSL.

All data types are defined in Solidity, as this is the format they must take to be properly validated on-chain. Services written in other languages need to ensure proper serialization/deserialization.

i.1 Deposits

```
struct Deposit {
  address trader;
  address asset;
  address participatingInterface;
  uint256 amount;
  uint256 depositId;
    uint256 chainId;
}

struct DepositUTXO {
  Deposit deposit;
  bytes32 depositHash;
}
```

i.2 Products

```
struct Asset {
  uint64 assetId;
  uint64 networkType;
  uint64 chainId;
  uint64 extra;
}

struct Product {
  uint256 assetA;
  uint256 assetB;
}
```

```

mapping(bytes32 => Product) products;
```

i.3 Orders

The order type is used to trace an executed trade to a signed order (original payload submitted by trader) with matching parameters.

For complex orders that resolve to limit orders, some will have a predetermined range of prices for which the contracts can verify a proper execution price.

For some complex orders, the price cannot be known ahead of time and will be set as 0.

Absolute minimum necessary to represent a limit order:

```
struct Order {
  Product p;
  boolean buyOrSell;
  uint256 size;
  uint256 price;
  address trader;
  uint64 traderId;
  uint64 participatingInterface;
  uint8 v;
  bytes32 r;
  bytes32 s;
}
```

When a Participating Interface receives an order and moves it through its system, it will apppend additional data to the original signed payload. When the order is included in a trade, it will have this additional data.

For example, the participating interface must assign a timestamp to each order received.

This will also be used when handling complex and contingent orders.

In all cases, an order MUST include the original payload signed by the trader.

i.4 Trades

The two orders included in a trade must have attributes that allow one to prove that the parameters of the trade is compatible with each order.

Participating Interface must designate input UTXOs used to fill each side of the trade. Participating Interface must generate output UTXOs.

First input UTXOs in list MUST fill side represented by `Order a` Last input UTXOs in list MUST fill side represented by `Order b`

```
struct TradeParams {
  // Sequence ID
  uint256 tradeId;
  // Participating Interface ID
  uint64 participatingInterface;
  // Trade Parameters
  Order a;
  uint256 timestampA;
  Order b;
  uint256 timestampB;
  Product p;
  uint256 size;
  uint256 price;
}

```

```
struct UnsignedTrade {
  TradeParams params;
  // Input and Output UTXOs
  bytes32[] inputUTXOs;
  bytes32[] outputUTXOs;
}

struct Trade {
    UnsignedTrade trade;
    // PI Signature
  uint8 v;
  bytes32 r;
  bytes32 s;
  // Root of tree of all UTXO outputs after appending this trade's outputs
  bytes32 stateRoot;
}

struct TradeSide {
  uint256 amount;
  address asset;
  address trader;
}

struct ValidateTradeResult {
  bool valid;
  // Below only included if valid is false
  FraudInfo fraudInfo;
}
```

i.5 ObligationUTXO

When a deposit is used as an input UTXO in a trade, it generates obligation UTXOs that represent the new state of the balance. Each obligation must reference the deposit UTXO from which it originates. Obligations do not require signatures from PI, as the hash of each output obligation is included in each signed Trade.

```
struct ObligationUTXO {
  address trader;
  uint256 amount;
    uint256 parentTradeId
  bytes32 parentUtxo;
  // May be optional, as it can be derived using parent
  bytes32 depositUtxo;
  // May be optional, as it can be derived using deposit
  address asset;
  address participatingInterface;
}
```

i.6 SettledUTXO

When a trader requests settlement, the PI must query for all unspent deposits and obligations, marking them as "spent" by creating a child UTXO which can no longer be used as an input.

```
struct SettledUTXO {
  uint256 settlementId;
  bytes32 parentUtxo;
  // May be optional, as it can be derived using parent
  address trader;
```

```
  uint256 amount;
  bytes32 depositUtxo;
  // May be optional, as it can be derived using deposit
  address asset;
  address participatingInterface;
}
```

i.7 Settlement Request

```
struct SettlementRequest {

    address trader;
  address asset;
    address participatingInterface;
  uint256 chainSequenceId;
    uint256 chainId;
}
```

i.8 Settlement Block

When SDPs report for settlement, they must include a state commitment that contains all `StateUpdate` messages signed by the PI for the range of IDs specified in the settlement.

```
struct SettlementBlock {
  // accumulator of all trades included in the block
  bytes32 tradeRoot;
  //
  // Note that the data rolled up in the commitments below
  // is also contained in the tradeRoot. There may be
  //
  // accumulator of all outputs created by trades in this block
  bytes32 outputsRoot;
  // accumulator of all outputs spent by trades in this block
  bytes32 spentOutputsRoot;
  // accumulator of all outputs settled in this block
  bytes32 settledOutputsRoot;
  // accumulator of state root after each trade
  // stateRoot = hash(tradeRoot,outputsRoot,spentOutputsRoot,settledOutputsRoot)
  bytes32 stateRootAccumulator;
}
```

i.9 Settlement Report

```
struct SettlementReport {
  SettlementBlock proposedBlock;
  SettlementAcknowledgement acknowledgement;
  StateRootAccumulatorMessage stateRootMessage;
  uint256 settlementId;
  bytes32[] stateRoots;
}

struct StateRootAccumulatorMessage {
  // accumulator of all trades included in the block
  bytes32 tradeRoot;
  // accumulator of all outputs created by trades in this block
  bytes32 outputsRoot;
```

```
  // accumulator of all outputs spent by trades in this block
  bytes32 spentOutputsRoot;
  // accumulator of all outputs settled in this block
  bytes32 settledOutputsRoot;
  // id of trade which results in the above accumulators
  uint256 tradeIndex;
  // PI Signature of hash of above
  Signature piSig;
}
```

i.10 State Updates

A `StateUpdate` is a message signed by the Participating Interface. It represents an update to the ledger. The update must be valid (doesn't break constraints imposed by the ledger) and must reference a signed message from each trader involved.

All `StateUpdate` messages must be linearily sequenced by the Participating Interface.

Note that the `Trade` data type defined above is also considered a `StateUpdate`, as each trade must be the result of matching two trader-signed orders.

```
struct StateUpdate {
  uint256 id;
}
```

i.11 DepositAcknowledgement

A `DepositAcknowledgement` is a `StateUpdate` that updates the ledger by creating a new output UTXO symmetric to the digital asset deposited by a trader in the `AssetCustody` smart contract.

```
struct DepositAcknowledgement {
  // Sequence ID
  uint256 tradeID;
  bytes32 depositUTXOHash;
  // Participating Interface ID
  address participatingInterface;
  // Participating Interface Signature
  uint8 v;
  bytes32 r;
  bytes32 s;
}
```

i.12 Settlement Acknowledgement

A `SettlementAcknowledgement` is a `StateUpdate`. When a trader requests settlement of an asset, the PI must detect the smart contract event and take any action necessary to ensure that no further trades will occur that could affect that trader's balance of the asset. For example, a PI operating a limit order book should ensure that there are no active orders for products which include the requested asset.

Once a PI has confirmed that the trader's balance is now static, it signs and emits a message that's included in the same sequence as trades.

```
struct SettlementParams {
  SettlementRequest settlementRequest;
  // Input UTXOs
  bytes32[] inputUTXOs;
}
```

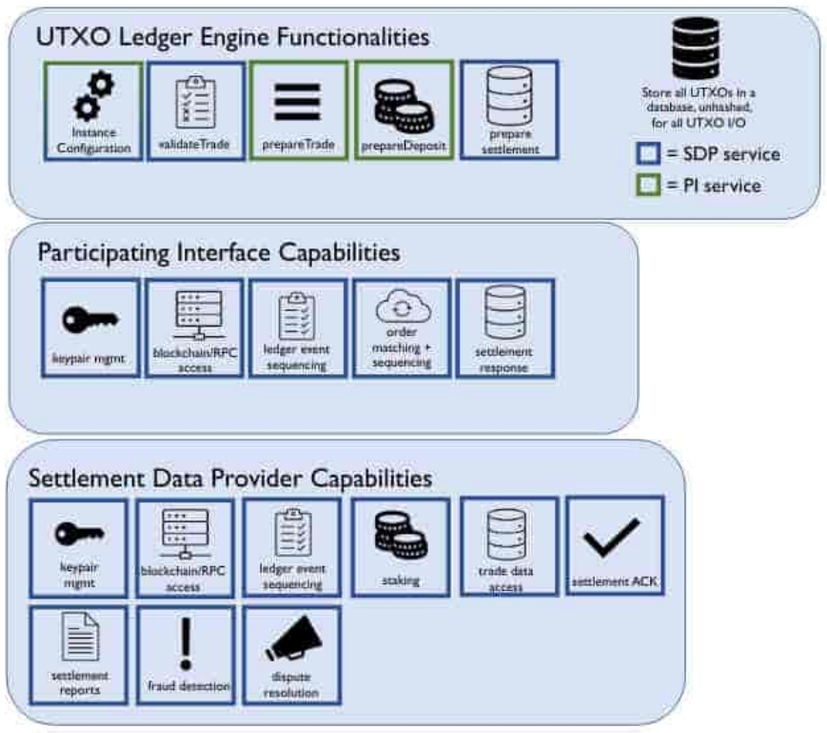

Figure 8: *Core functionalities of the UTXO ledger engine. Core capabilities of the PI and SDP.*

```
6
7  struct SettlementAcknowledgement {
8    // Sequence ID
9    uint256 tradeID;
10   // Settlement Params
11   SettlementParams settlementParams;
12   // Participating Interface Signature
13   uint8 v;
14   bytes32 r;
15   bytes32 s;
16 }
```

ii. UTXO Ledger Engine

The UTXO Ledger Engine defines the logic for maintaining an immutable history of all deposits, trades, and settlements occurring on a Participating Interface. The latest state of the engine can be used to derive the balances of all traders with assets deposited on the PI.

Both PIs and SDPs must run an instance of the engine. SDPs must validate that UTXOs included as inputs and generated as outputs in all trades emitted by the PI are the result of correctly running the UTXO engine. Thus, the core logic of the engine must be deterministic, so that given the same order of deposits and trades, the PI and all servicing SDPs reach the same ledger state.

ii.1 Configuration

•MUST require user to specify whether the engine is being run by a PI or an SDP

ii.2 SDP Operation

•MUST provide an endpoint for the SDP to send a `Trade` and call the `validateTrade` function, responding with the result

ii.3 PI Operation

•MUST provide an endpoint for the PI to send a `TradeParams` and call the `prepareTrade` function, responding with the result

•MUST provide an endpoint for the PI to send a `Deposit` and call the `prepareDeposit` function, responding with the result

•MUST provide and endpoint for the PI to send a `SettlementRequest` and call `prepareSettlement`

ii.4 Ledger Database

•MUST store all UTXOs in a database, unhashed

•MUST support querying UTXOs by any of their attributes

iii. UTXO Ledger Functions

Note that every function defined below MUST be atomic.

iii.1 validateTrade

Given `Trade`:

•extract `TradeParams` and pass to `determineTradeUTXOs`

•using the result of `determineTradeUTXOs`:

•hash each input UTXO for side A

•hash each input UTXO for side B

•hash each output UTXO for side A

•hash each output UTXO for side B

•extract `inputUTXOs` and `outputUTXOs` from `Trade`

•if hashes of input and output UTXOs generated match UTXO hashes in `Trade`, return `true` to indicate a valid trade

•if UTXOs don't match, return `false` along with an `UnsignedTrade` with the correct input and output UTXOs

iii.2 prepareTrade

Given `TradeParams` :

•call `determineTradeUTXOs` with `TradeParams`

•using the result of `determineTradeUTXOs`:

•hash each input UTXO for side A

•hash each input UTXO for side B

•hash each output UTXO for side A

•hash each output UTXO for side B

•using `TradeParams` and the input and output UTXO hashes create an `UnsignedTrade`

•append output UTXOs to the ledger

•return the `UnsignedTrade`

iii.3 prepareSettlement

Given `SettlementRequest`:

- query for all unspent UTXOs with the same `asset` and `trader` to use as inputs
- for all UTXOs above, generate a `SettledUTXO` to use as output
- hash all input and output UTXOs
- use `SettlementRequest` and UTXO hashes to create `SettlementParams`
- return the `SettlementParams`

iii.4 prepareDeposit

Given `Deposit`:

- call `hashDeposit`
- using `Deposit` and the result of `hashDeposit`, create a `DepositUTXO`
- append the `DepositUTXO` to the ledger
- return the hash of the deposit

iii.5 determineTradeUTXOs

Given `TradeParams`:

- determine `TradeSide` for trader that submitted `Order a` and call `determineInputUTXOs`
- determine `TradeSide` for trader that submitted `Order b` and call `determineInputUTXOs`
- use `TradeSide` and input UTXOs determined from `Order a` , and trader of `Order b` to call `generateOutputUTXOs`
- use `TradeSide` and input UTXOs determined from `Order b` , and trader of `Order a` to call `determineOutputUTXOs`
- return all input and output UTXOs

iii.6 determineInputUTXOs

Given `TradeSide`:

- query for all unspent UTXOs with same `asset` and `trader` as specified in the `TradeSide`, sorted descending by UTXO hash
- iterate through UTXOs from the query result and add to a list until the sum of the amounts of the selected UTXOs is greater than or equal to the `amount` in the `TradeSide`
- return the list of selected UTXOs

iii.7 generateOutputUTXOs

Given `TradeSide` , list of input UTXOs, and `address counterParty`:

- iterate through input UTXOs and generate symmetric output UTXOs
- if the sum of the amounts in input UTXOs is greater than the `amount` in the `TradeSide`, then the final input UTXO will generate a second output UTXO granting the remainder to the `counterParty`

iv. Participating Interface Specifications

This document specifies the requirements for the software a Participating Interface must run in order to earn service fees from traders on the TXA DSL. This describes the minimum requirements to comply with the TXA CSP, and does not include details for how to run an orderbook.

iv.1 Cryptographic Key Management

Operating a PI requires a minimum of two ECC keypairs capable of signing transactions on most EVM-based networks. One keypair, called the `adminKey` is used for registering the PI in the DSL, staking TXA and collateral, and registering the PI `tradeSigningKey`, which is used by the PI to sign trade data.

iv.2 Requirements

- PI SHALL generate an ECDSA private key for use as an `tradeSigningKey`
  - PI MUST store an ECDSA private key for use as an `tradeSigningKey`
  - PI MUST prompt the operator for the public address of the `adminKey` and generate a signature using the `tradeSigningKey`
  - PI SHOULD display instructions for including the generated signature in a transaction to configure the PI administration contract

iv.3 Blockchain RPC Access

A PI needs to read data from and submit transactions to DSL smart contracts on every blockchain that it supports.

iv.4 Requirements

•PI MUST establish connections to RPC endpoints for each blockchain it supports
•PI SHOULD allow operator to set redunduncy endpoints incase of no connectivity

iv.5 Blockchain Indexing

Transactions to the DSL will include data or emit events that the PI needs to process.

iv.6 Requirements

•PI MUST receive a sequenced stream of parsed blockchain blocks containing data from the DSL smart contracts for each blockchain it supports.

iv.7 Detecting Deposit

Upon detecting that a trader's transaction to deposit an asset in the DSL for trading on the PI has been mined and passed enough confirmations, the PI:

- MUST call `prepareDeposit` on the UTXO Engine with the `Deposit` as input
- MUST sign and broadcast a `DepositAcknowledgement` message created from the `DepositUTXO` generated above
- SHOULD send the trader a receipt of the acknowledgement

iv.8 Order Matching

- Upon matching two orders, the PI must generate corresponding `TradeParams`
  - Upon matching an order and generating a `TradeParams`, PI MUST call `prepareTrade` on the UTXO Engine with the `TradeParams` as input

  - Upon receiving an `UnsignedTrade` from the UTXO Engine, PI MUST sign the trade with the trade signing key to generate a signed `Trade`
  - PI MUST update the state root with the signed trade
  - PI MUST forward the `Trade` to Streamer for SDP broadcast
  - PI MUST forward the `Trade` to Streamer for trader broadcast

iv.9 Sequencing

•PI MUST sequence all `StateUpdate` messages (deposits, trades, settlements) via an incremental integer

iv.10 Responding to Settlement

Upon detecting that a trader requested to settle an asset, the PI:

- MUST call `prepareSettlement` on the UTXO Engine with the `SettlementRequest` as input
- MUST sign and broadcast a `SettlementAcknowledgement` message created from the `SettlementParams` generated above
- SHOULD send the trader a receipt of the acknowledgement

v. Settlement Data Provider Specification

v.1 Blockchain RPC Access

In order to fully service a Participating Interface, an SDP needs to read data from and submit transactions to DSL smart contracts on every blockchain that the Participating Interface supports.

- SDP MUST establish connections to RPC endpoints for each blockchain that it services
- SDP SHOULD allow operator to set redunduncy endpoints in case of no connectivity

v.2 Blockchain Indexing

Transactions to the DSL will include data or emit events that the SDP needs to process in order to properly service a PI.

- SDP MUST receive a sequenced stream of parsed blockchain blocks containing data from the DSL smart contracts for each blockchain supported by each PI serviced by this SDP.

v.3 Cryptographic Key Management

Operating an SDP requires a minimum of two ECDSA keypairs capable of signing transactions on most EVM-based networks.

One keypair, called the `stakerKey` is used for staking TXA and collateral assets on the DSL and registering the SDP `nodeKey`, which is used by the SDP node to report settlement data to the DSL. A relationship between the two keys must be established in the `CollateralCustody` contract for the SDP to be able to report.

- SDP SHALL generate an ECDSA keypair for use as an `nodeKey`
  - SDP MUST store an ECDSA private key for use as an `nodeKey`

  - SDP MUST prompt the operator for the public address of the `stakerKey` and generate a signature using the `nodeKey`
  - SDP SHOULD display instructions for submitting a transaction to the `CollateralCustody` smart contract using the `stakerKey` that includes the signature from the `nodeKey`

v.4 Asset Staking

In order to participate in settlement, an SDP operator must be registered with a staking account that has enough assets available to meet the collateral requirements of a settlement. To determine eligibility, the SDP queries the `CollateralCustody` contract.

- SDP MUST check that a `nodeKey` is present
  - SDP MUST submit an `eth_call` RPC request to check if the `nodeKey` is registered to a `stakerKey` in the `CollateralCustody` contract
  - SDP SHOULD display instructions for submitting a transaction to the `CollateralCustody` smart contract using the `stakerKey` to deposit an asset for use as collateral

v.5 Blockchain Event Handling

v.6 Deposit

•SDP MUST forward the `Deposit` to the `processDeposit` function of the UTXO Ledger Engine

v.7 Trade Data Subscription

In order to provide settlement services for a PI, the SDP needs to listen for all trade data emitted by that PI.

- SDP MUST listen for WebSocket events emitted by a PI
- SDP MUST request any missing trades from the PI

v.8 Trade Data Handling

Upon any event emitted by the PI, the SDP needs to validate the trade and update its internal records of trader balances.

v.9 Trade

- SDP MUST store the `Trade` in a database
  - SDP MUST validate that the address recovered from the signature of the trade matches the address of the PI `tradeKey` that generated this message
  - SDP MUST validate that the address recovered from the signature of each order included in the trade matches the party of the trade
  - SDP MUST pass the trade to the `validateTrade` function of the UTXO Ledger Engine

v.10 Settlement Acknowledgement

- SDP MUST store the `SettlementAcknowledgement` in a database
  - SDP MUST validate that the address recovered from the signature of the trade matches the address of the PI `tradeKey` that generated this message

v.11 Calculating Settlement Reports

A settlement report is generated relative to a specified participating interface address, trader address, chain ID, and asset address:

- SDP MUST query for all unspent `ObligationUTXOs` with the above parameters

v.12 Detecting Fraud

Fraud detection protocols are detailed in Appendix B. The requirements specified in those algorithms are used to determine whether or not an SDP or a PI have committed fraud. The fraud detection process also aggregates the data necessary to submit proof of fraud to the DSL smart contracts.

Assuming full data availability, an auditor only needs to provide inclusion proofs in order to demonstrate that fraud occurred. It should not be necessary to submit an exclusion proof in order to demonstrate fraud.

B. Fraud Detection Algorithms

We specify the algorithmic fraud detection algorithms that can be employed by auditors to verify honest participation in the protocol. The inclusion proofs specified in these algorithms are designed to be sufficient to demonstrate fraud. The fraud engine runs a trade through a series of functions which compare the trade data against the current state of the UTXO database. The functions should be called in a logical order, such that checks for which fraud is easily detectable which would cause later checks to also fail are called first. The functions may be called in parallel, as the smart contracts must accept any type of fraud that can be proven.

i. Signature Fraud

The `validate TradeSignature` algorithm (Algorithm 1) detects whether the StateUpdate provided by the PI uses a signature where the recovered address does not match the address in the Trade. It: (1) recovers the address from the signature of each Order, (2) checks if recovered address matches the address trader in the Order.

The `validateSettlementSignature` algorithm (Algorithm 2) detects whether the StateUpdate provided by the PI uses a signature where the recovered address does not match the address in the SettlementRequest. It (1) recovers the address from the signature, (2) checks if recovered address matches the address trader in the SettlementRequest

Algorithm 1 validateTradeSignature

```
for Order in Trade do
    addr ← recoverOrderAddress(Order.signature)
    assert(addr == Order.trader)
end for
```

Algorithm 2 validateSettlementSignature

```
addr ← recoverSettlementAddress(SettlementRequest.signature)
assert(addr == SettlementRequest.trader)
```

ii. Order Fraud

Order fraud occurs when the parameters of an Order included in a Trade by the PI does not match the parameters of the trade. This is detected by `validateOrderParameters` (Algorithm 3). Given a trade, the algorithm (1) verifies that `participatingInterface` in the `TradeParams` matches the one specified in each `Order`, (2) verifies that `Product p` in the `TradeParams` matches the one specified in each `Order`, (3) verify that the size in the `TradeParams` is less than or equal to the size specified in each `Order`, (4) verifies that the price in the `TradeParams` matches the conditions specified in each `Order`, (5) verify that one `Order` sets `buyOrSell` as true, while the other specifies it as false

Algorithm 3 validateOrderParameters

```
for Order in Trade do
    assert(TradeParams.participatingInterface == Order.participatingInterface)
    assert(Product TradeParams.p == Product Order.p)
    assert(TradeParams.size ≤ Order.size)
    assert(TradeParams.price == Order.price)
end for
assert(AtLeastOne(Order.buyOrSell == BUY) and AtLeastOne(Order.buyOrSell ==
SELL))
```

iii. UTXO Fraud

The `validateInputUTXOs` function (Algorithm 6) protects against UTXO fraud propagated through InputUTXOs. It (1) verifies that every input UTXO exists in the UTXO ledger. Function ExistsIn-UTXOLedger will validate that chain Id of the asset matches chain ID of the settlement contracts. It will additionally hash the UTXO and reveal if it either does not exist in the on-chain ledger, or if it exists but the asset does not match, (2) asserts that `ObligationUTXO` exists in a settled root in the ledger, or within the same proposed root but as an output of a trade earlier in the sequence by performing appropriate inclusion proofs, (3) verifies that every input UTXO is unspent (has no children in the ledger), (4) verifies that the spender of every input UTXO is either the signer of `Order` a or `Order` b, (5) verifies that every input UTXO where the spender is the signer of `Order` a represents the base asset, (6) verify that every input UTXO where the spender is the signer of `Order` b represents the counter asset, (7) verifies that the sum of each input UTXO of the base asset is greater than or equal to the size of the trade (8) verifies that the sum of each input UTXO of the counter asset is greater than or equal to the size of the trade multiplied by the price.

The validateOutputUTXO function protects against UTXO fraud propagated through outputUTXOs. It (1) verifies that for every input UTXO, there are either 3 or 4 output UTXOs, (2) verifies that every output UTXO is an `ObligationUTXO` or a `FeeUTXO` by checking whether the first `uint8` of the signed data matches type identifier for `ObligationUTXO` or `FeeUTXO`, (3) verifies that every output UTXO is a child of an input UTXO by iterating through the input UTXOs and verifying that none of the inputs match the parent of the output, (4) verifying that the asset of every output UTXO matches the asset of its input UTXO, (5) verifying that the sum of the children of each input UTXO matches the amount of the input UTXO

Algorithm 4 validateDepositUTXO

Input: `DepositUTXO`

```
Deposit ← DepositUTXO.Deposit
assert(DepositUTXO.DepositHash == H(Deposit.trader, Deposit.asset, Deposit.amount))
assert(Deposit.asset == SettlementContractAssetType())
assert(existsOnLedger(DepositUTXO))
```

Algorithm 5 validateObligationUTXO

Input: `ObligationUTXO`, `Trade`

```
assert(InclusionPfInSettledRoot(inputUTXO) or InclusionPfInPrevLeaf(inputUTXO))
for outputUTXO in Trade.outputUTXOs do
    assert(outputUTXO ≠ ObligationUTXO)
end for
```

Algorithm 6 validateInputUTXOs

Input: `Trade`

```
TradeParams ← Trade.trade.params
InputUTXOAssetSum ← 0
InputUTXOCtrAssetSum ← 0
OrderATraderID ← TradeParams.OrderA.trader
OrderBTraderID ← TradeParams.OrderB.trader
for inputUTXO in Trade.trade.inputUTXOs do
    assert(ExistsInUTXOLedger(inputUTXO))
    if type(inputUTXO) == DepositUTXO) then
        validateDepositUTXO(inputUTXO)
    end if
    if type(inputUTXO) == obligationUTXO then
        validateObligationUTXO(inputUTXO)
    end if
    assert(isUnspent(inputUTXO))
    UTXOTraderID ← inputUTXO.trader
    assert(UTXOTraderID == OrderATraderID or UTXOTraderID == OrderBTraderID)
    if UTXOTraderID == OrderATraderID then
        assert(inputUTXO.asset == TradeParams.OrderA.p.AssetA)
        InputUTXOAssetSum ← InputUTXOAssetSum + inputUTXO.amount
    else
        assert(inputUTXO.asset == TradeParams.OrderB.p.AssetA)
        InputUTXOCtrAssetSum ← InputUTXOCtrAssetSum + inputUTXO.amount
    end if
end for
assert(InputUTXORunningSum ≥ TradeParams.size)
assert(InputUTXOCtrAssetSum ≥ TradeParams.size * TradeParams.price)
```

Algorithm 7 validateOutputUTXOs

Input: `Trade`

```
TradeParams ← Trade.trade.params
assert(Data that hashes to the trade, including input and output UTXOs)
for inputUTXO in Trade.trade.inputUTXOs do
    AssociatedOutputUTXOs ← 0
    SumOfinputUTXOChildren ← 0
    for outputUTXO in Trade.trade.outputUTXOs do
        assert(Type(outputUTXO) == ObligationUTXO or Type(outputUTXO) == FeeUTXO)
        if IsChildOfInputUTXO(outputUTXO) then
            AssociatedOutputUTXO ← AssociatedOutputUTXO + 1
            SumOfinputUTXOChildren ← SumOfinputUTXOChildren + outputUTXO.amount
            assert(outputUTXO.asset == inputUTXO.asset)
        end if
    end for
    assert(SumOfinputUTXOChildren == inputUTXO.amount)
    assert(AssociatedOutputUTXO == 3 or AssociatedOutputUTXO == 4)
end for
```

iv. Error Fraud

Error fraud occurs as a result of incorrectly running the UTXO Engine software. The `ValidateUTXOOrder` algorithm verifies that input and output UTXOs are correctly sorted in ascending order by the numerical value represented by their hashes.

Algorithm 8 validateUTXOOrder

Input: `UTXO_1, UTXO_1_Index, UTXO_2, UTXO_2_Index`

```
assert(UTXO_2_Index > UTXO_1_Index)
assert(H(UTXO_1) < H(UTXO_2))
```

C. Reference Exchange Design

This section covers the exchange and settlement subsystems in a reference design for the DSL architecture.

Participant Exchanges may be composed of the following subsystems:

i. Order Intake

Participating Interface relies on witnessing trader-led activities in a time-ordered fashion.

Constraints on PI:

- Simultaneous order acceptance at API gateways
- Disparate VM time sync
- Trader privacy

Desirable Properties:

- Order acceptance fairness - first-come first-serve

- Order persistence
- Time-ordered
- Payload validity check
- Trade origin verification
- Microsecond timestamp

ii. Order Matching

Constraints:

- Sparse orderbook with clustering
- Single threaded process per symbol
- Comparable orderbook updates to equity exchanges (less than 100 micro-sec)
- Full replayability of all orders and trades

Desirable Properties:

- Rapid order addition and modification
- Optimize for order cancellation (90% of orders cancel)
- O(1) search time for each price level
- O(1) time to access each order
- Fast, iterable collection at each price level, ordered by time
- Determinism when historical orders are replayed

iii. Exchange Public API

Exchange Public API is a way for external applications using REST to interact with the exchange platform. Its responsibilities include:

- Accept signed orders from traders
- Show per-asset balance information given an Asset Custody Contract
- Retrieve and present historical orderbook entry values for client-side initialization
- Provide WebSocket hook for streaming orderbook data
- Provide dApp hook for interacting with escrow contracts

iv. Un-ordered Message Subsystem

Un-ordered Message Subsystem, composed of Message Ingestion and Message Queue, is responsible for rapidly queuing various messages from Exchange Public API. Un-ordered Message Subsystem takes in messages as quickly as possible from as many API service hosts as possible. Its responsibilities are:

- Store in semi-permanent basis all incoming messages
- Normalize all incoming messages with common encoding schema
- Integrate with real-time logging facility
- Provide streaming data to Ordered Message Subsystem

v. Ordered Message Subsystem

Ordered Message Subsystem, also composed of Message Ingestion and Message Queue, is responsible for taking in raw un-ordered message stream and do its best to reorder them using time and items bucketing. Time and items bucketing is a fixed size buffer that expires. Its responsibilities are:

- Process un-ordered messages based on receipt time
- Store in semi-permanent basis all time-ordered (processed) messages
- Integrate with real-time logging facility
- Provide streaming data to rest of the Tacen infrastructure, such as Order Matching Engine
- Handles both standing orders as well as new incoming orders

vi. Historical Data Subsystem

Historical Data Subsystem stores and provides historical data for internal and public API consumption. Its responsibilities are:

- Store bulk stream data into database
- Interface with database and provide convenient API service for consumers

vii. Ordered Message Streamer

Ordered Message Streamer takes in streaming data from Ordered Message Subsystem intended for both public and internal consumption and sends out to all WebSocket subscribers. Its responsibilities are:

- Send time-ordered, processed messages to WebSocket topic subscribers

viii. Orderbook and Order Matching Engine

Orderbook and Order Matching Engine maintains orderbook information. Orderbook is responsible for:

- Receive time-ordered new and updated orders
- Enter new order entries into orderbook with price and time priority
- Update order entry in an orderbook with time priority
- Match incoming and updated orders against entries in orderbook
- Enqueue matched orders into Ordered Message queue for Trade Processor

ix. Trade Processor

Trade Processor takes in matched orders from OME and handles escrow balance updates. Its responsibilities are:

- Receive time-ordered matched orders
- Update per-escrow balance
- Enqueue executed trades into queue in time-ordered manner

x. Public Blockchain Interface

Public Blockchain Interface components of Participant Exchange is responsible for the following:

- Check for full-collateralization of the trades
- Listen for completion of all events required for start of settlement
- Handle multi-party coordinated settlements

The smart contracts are symmetric in nature, meaning that across multiple blockchains, their behavior is same for the same class of smart contracts.

xi. Trader Asset Custody Contract

Collateralize the trades by assigning expiration to each asset class.

Asset Custody Contract is responsible for the following:

- Accepts transfer of both tokens and native asset
- Record and recall its obligations to counter-parties
- Acts as an identity when submitting orders
- Remits available native and token assets to its beneficiary when requested

xii. Settlement Coordinator

Settlement Coordinator acts as a task coordinator for the settlement process. Its responsibilities are as follows:

- Keep canonical list of Settlement Data Providers
- Accept request to initiate settlement from a Asset Custody Contract
- Obtain settlement obligation information from a quorum of SDPs
- Contact Settlement Coordinator to initiate coordinated settlement on counter-product's blockchain
- Relay transfer instructions to escrow contracts
- Finalize a settlement

xii.1 Settlement Data Provider

Settlement Data Provider provides settlement obligation information to Settlement Coordination Contract when requested. SDP controlled wallet addresses are enrolled into Settlement Coordination Contract RBAC. SDP responsibilities are as follows:

- Provide settlement obligation data to Settlement Coordination Contract
- Listen to exchange updates via WebSocket and record into local persistent storage
- Listen for completion of all events required for settlements
- Calculate asset balances on ongoing basis

D. TXA.D Airdrop Details

In order to bootstrap a DAO for the TXA DSL, Project TXA will distribute a governance token, TXA.D, through an airdrop. Full details of the airdrop can be found here.

References

[1] N. Papadis and L. Tassiulas, "Blockchain-based payment channel networks: challenges and recent advances," *IEEE Access*, vol. 8, pp. 227 596–227 609, 2020.

[2] R. Khalil, A. Zamyatin, G. Felley, P. Moreno-Sanchez, and A. Gervais, "Commit-chains: Secure, scalable off-chain payments," *Cryptology ePrint Archive*, 2018.

[3] R. Khalil, A. Gervais, and G. Felley, "Tex-a securely scalable trustless exchange," *Cryptology ePrint Archive*, 2019.

[4] B. Rao, "Gluon layer2," 2020.

[5] E. Budish, P. Cramton, and J. Shim, "The high-frequency trading arms race: Frequent batch auctions as a market design response," *The Quarterly Journal of Economics*, vol. 130, no. 4, pp. 1547–1621, 2015.

[6] E. Budish, R. S. Lee, and J. J. Shim, "A theory of stock exchange competition and innovation: Will the market fix the market?" National Bureau of Economic Research, Tech. Rep., 2019.

[7] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Advances in Cryptology—CRYPTO'87: Proceedings 7*. Springer, 1988, pp. 369–378.

[8] E. Mykletun, M. Narasimha, and G. Tsudik, "Providing authentication and integrity in outsourced databases using merkle hash trees," *UCI-SCONCE Technical Report*, 2003.