Aleph: Account Abstraction over UTxO

SpectrumLabs
info@spectrumlabs.fi

Abstract

UTxO model is known for its determinism which implies that all effects of a transaction are known in advance. For applications that deal with shared on-chain resources such as AMM pools or CLOB this property implies certain design limitations that, if addressed in a naive way result in significant overhead. A more efficient approach to these limitations was previously introduced in Spectrum Bloom paper, in this work we describe the implementation of that concept referred to as Autonomous Accounts, or more known as Account Abstraction over UTxO on Cardano blockchain.

1. Structure

First we revisit the problem and the solution proposed in Spectrum Bloom paper, then proceed to technical details of implementation on top of Cardano.

2. Introduction

eUTxO pioneers faced a challenge when first tried to port protocols such as Uniswap: while EVM implementations allowed its users to transact with liquidity pools directly from clients, on eUTxO such scenario was extremely impractical. The root of the problem lays in the nature of eUTxO, unlike "Account" model it requires that all inputs of a transaction are deterministic. Therefore, direct transaction with shared, atomic on-chain resources (e.g. liquidity pools) would result in race conditions. A classical approach to the aforementioned issue is to synchronize user access to a shared resource via on-chain orders which are then picked up, ordered and executed by off-chain agents shortly after. On-chain order is encoded into a UTxO carrying some input value (e.g. some amount of base asset in case of limit sell order) and guarded with a validator script that ensures that the order is executed at a fair price provided by concrete liquidity pool at the time of actual order execution.

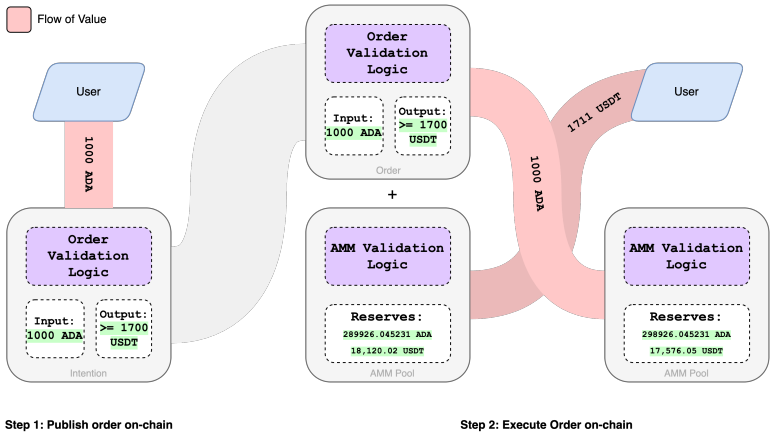

Figure 1: A diagram that shows how users interact with aggregated liquidity pools via orders. At first step a client publishes an order moving some of his funds into it, at the next step an off-chain operator matches this order with a proper pool and executes the exchange.

Classical approach is inefficient. On-chain orders require an extra transaction. As a result user has to cover fees for both order publishing transaction and execution transaction. Additionally, moving or canelling orders requires a separate transaction as well what draws bad user experience.

3. Account Abstraction

The proposed solution is to create a virtual transaction system on top of UTxO that would only require off-chain specification (aka Intentions) of the desired outputs without too many on-chain details. In order for intentions to work we need an on-chain entities (aka Accounts) capable of interpreting them in a safe way and release funds when needed.

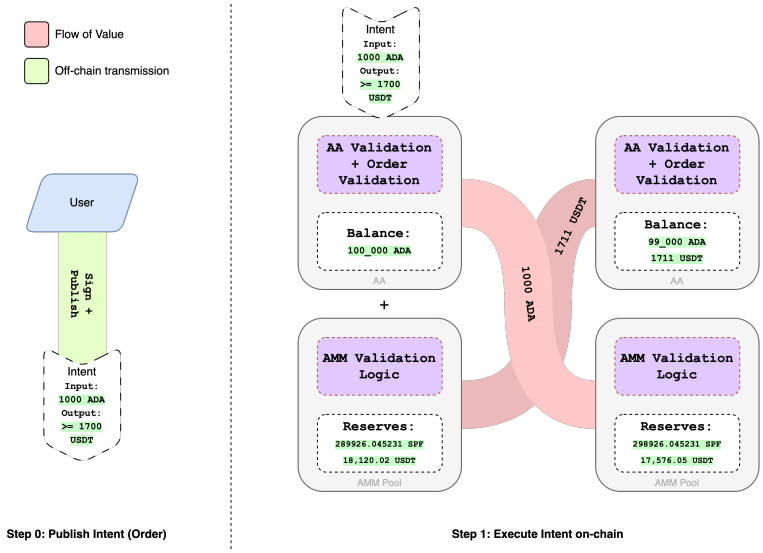

Figure 2: Account Abstraction. Intents are similar to on-chain orders, but live off-chain and create nearly zero overhead

3.1. Aleph

Aleph is designed with the following principles in mind:

- Security. Gurantees provided by the base layer (Cardano) must not be compromised
- Composability. Account should be able to inteact with arbitrary applications
- Efficiency. Account design should allow for transacting with many accounts in one on-chain transaction

3.1.1. Account

We model account as an on-chain entity encoded into UTxO. Account UTxO holds some non-zero balance of assets, equipped with a state (Table 1) and guarded by a simple script that has two paths of execution:

- Direct spending. User can spend account UTxO directly by signing transaction with a key corresponding to “cold_cred”
- Delegate to a witness. Leak control to a witness script listed in “allowlist”

Field	Type	Description
magic	`ByteArray`	Used to recognize accounts
allowlist	`List<ScriptHash>`	Set of allowed delegatees
nonce	`List<Nonce>`	N independent monotonically increasing counters used to protect the account from replay attacks
hot_cred	`(VerificationKey, Option<VerificationKey>)`	Credentials used to authorize intents. Composed of a main credential and an optional co-credential
cold_cred	`VerificationKeyHash`	Credential for direct spending
store	`ByteArray`	Local store of the account, used by dapps to save intermediate state. Represented as a root hash of Merkle Patricia Tree

Table 1: Account state. All fields except `magic` are mutable.

3.1.2. Witness

A witness script does all the magic that makes account abstraction appealing. Although implementations of overall witness may vary, all of them must perform the following validations:

- For all involved accounts check that all fields of the state are preserved except “nonce” and “store”
- For all involved accounts validate authorization of applied intents
- For all involved accounts validate intent spcific rules. This part is implementation specific, e.g in case of witness that models limit orders the validations would apply to exchange rate, trader fees etc.

Scripts that do not perform all the neccessary validation must not be added to “allowlist”. The design choice of placing critical validations into witnesses sacrifices convenience for developers in favor of efficiency: having all validations in one witness allows for batch validation in one pass (see Figure 3).

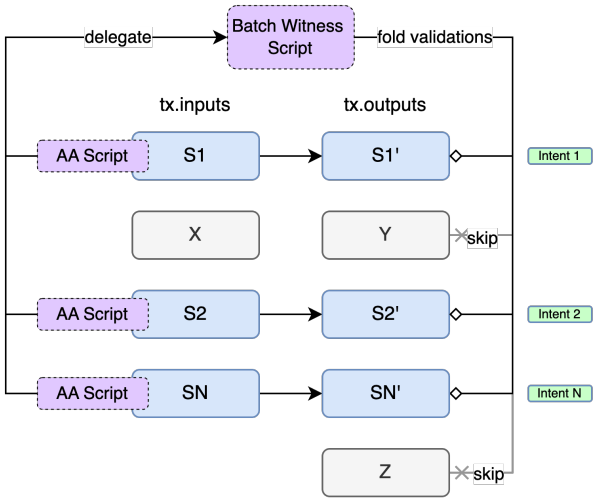

Figure 3: Batch witness. Transitions of all accounts involved are validated in one pass

3.1.3. Intent

Intent structure is an implementation detail in Aleph and each dapp integrating with accounts can have its own structure assuming their witness implementation knows how to work with that type of intent. Some fields are mandatory, though: “TargetNonce” - is a maximum value of account state “nonce” at the given “NonceIndex”. Although account nonce system allows for parallelization of a limited factor N (in practice $1 \leq N \leq 5$), some applications may dismiss nonces completely to sacrifice on-chain gurantees in favor of maximum parallelization and cost efficiency.

$$\begin{aligned} \text{TargetNonce} &= \text{NonceIndex} \times \text{Nonce} \\ \text{Intent} &= \text{IntentParams} \times \text{TargetNonce} \\ \text{AuthorisedIntent} &= \text{Intent} \times \sigma \end{aligned}$$

Table 2: Intent structure. User authorizes the intent by signing its contents and attaching the proof σ. IntentParams – parameters of the order, e.g. quote and base asset, price, etc., Nonce – monotonically increasing counter to prevent replay attacks.