eidoo

EIDOO HYBRID EXCHANGE TECHNICAL PAPER

Version 1.0

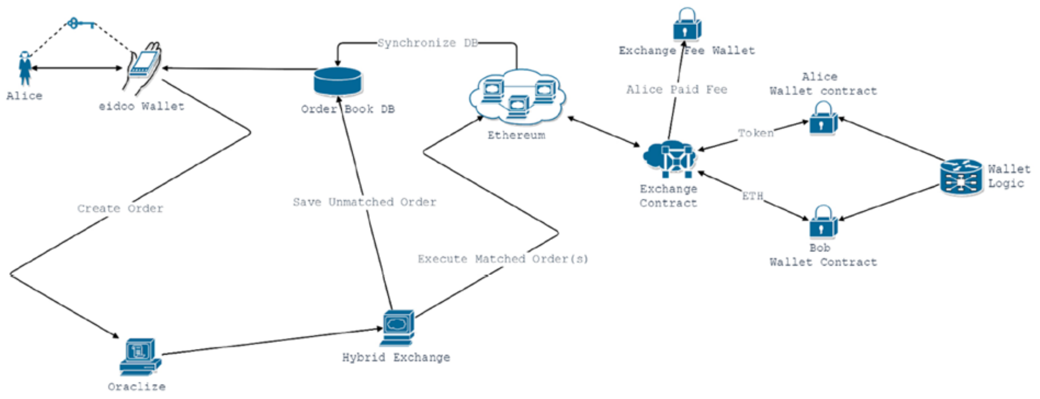
Overview

The Eidoo hybrid exchange is an architecture utilizing a centralized server for matching user-signed orders, with final verification and settlement occurring on-chain via a smart contract system on Ethereum.

Users will be creating signed orders via a front-end UI, by way of the Eidoo wallet, which then proxies the signed order to a long-running Oraclize off-chain computation instance, logging all the requests and backing them with proofs, to ensure the centralized order book server is matching the orders on-chain in a provably honest manner, with its standard matching method of first-in, first-out order matching with the aim being to mitigate any front-running.

Once verified and logged in the Oraclize instance, an API call is made to the central order book server, for submission of the order. The order is verified for validity first, then is checked if it matches any existing orders in the book, and gets matched as a taker if it does and the 2 matching orders broadcast for on-chain settlement.

Any unmatched amount from the order is appended to the order book.



Design Rationale

During the design process, there were a number of tenets outlined and followed to create this hybrid exchange, with the goals of it having the potential to be at least as efficient as a centralized exchange, while not having all the risks associated with a centralized entity, and instead enjoying as many of the trustless benefits a decentralized exchange does.

- Assurance that user capital stays safe and retain control of capital movement.
    - Each user's capital is deposited and held within a personal segregated wallet contract, which they have ultimate control over
    - In case of exchange updates or wallet logic updates, the user must manually optin.

- Presume an adversarial environment amongst exchange participants, including that eidoo may become an adversary in the system.
    - Trading logic occurs on-chain, and the segregated wallets themselves also verify the orders internally before allowing trade execution to proceed.

- Encourage market liquidity
    - Centralized order book facilitates no-fee posting of orders.
    - Makers are currently not charged any fees, takers pays fees.
    - Trade pairs will exclusively be between Ether and ERC-20 tokens, to avoid market pair oversaturation.

- Future-proofing the smart contract architecture.
    - Supports all current ERC-20 tokens and any future standard ERC-20 tokens.
    - Many parts of the logic are upgradeable by eidoo, which will allow for improvements or fixes to the platform and also support of any future token standards that become widely accepted.

- Intuitive and functional trading platform.
    - Trading will be available via the eidoo wallet app

- Maintain a scalable and distributed backend for the order book and matching.
    - Node.js as language of choice, due to vast and most mature web3 and ethereum library support, and non-blocking asynchronous nature.
    - Initial building block of the backend was tailoring it towards running in a multicluster environment.
    - RethinkDB as database stack thanks to native compatibility with Node.js, FOSS, real-time support, and sharding capability.

- Ensure proper and accurate trade engine and avoid fallacious on-chain implementations as generally seen by other EVM-based exchanges.
    - Rigorous development and testing of a solution to overcome the EVM limitation of no floating-point support, which tends to result in rounding errors and loss of

revenues for parties involved. The formulated trading engine that resulted is one that does not sacrifice accuracy and does not run into any rounding errors or remainder arising from a purely integer-based system, with either completely filled orders or partial fill orders.

In summary, the objective of following these principles is for critical pieces such as capital movement and exchange of tokens to be decentralized and trustless, and delegating as many other parts of the architecture on outer centralized layers with the intention of avoiding the various pitfalls associated with what are current decentralized exchanges, including high latency, usage costs, and low volume, whilst operating in a provably honest manner.

Design Rationale

The on-chain architecture is made up of 2 major subsets; the exchange and the segregated wallets. The exchange runs as a monolithic contract with a number of dependencies, including the wallet, a safe math library for mathematical operations to avoid overflow or underflow issues, a utility contract that handles error logging via events, and ERC-20 interface contract to facilitate their trading. It acts as the trade broker between user's wallets, having pull permissions to atomically swap a token and Ether pair if each order passes verification from the wallets.

The wallets consist of 3 parts; the base wallet which acts mainly as the interface the user will use to administer their trade funds with while having minimal cost and effort to deploy or upgrade as the logic is called in a delegated manner, the wallet logic which each of the base wallets will depend on and use to run their logic, and the wallet connector which enables the upgradeability feature and for user's to be notified when new logic is available and disable any obsolete or bugged logic from partaking in the exchange system.

Exchange Contract

The goal of the monolithic design is to allow for greater readability and auditability of the underlying exchange code, thereby providing the chance for a greater technical audience to ensure correctness and fairness within the code and the absence of any underhanded designs available through exploit and hidden away using convoluted contract writing by multiple hierarchies and imports.

The trade execution logic consists of a number of mechanics being leveraged to provide the capability of off-chain order creation and on-chain verification and execution.

The off-chain mechanic used is the eth_sign method provided via JSON-RPC in the standard Ethereum clients. As the signatures are sent back on-chain along with the matching order properties, the exchange contract will verify that the matched order relayer is whitelisted as the order book server address, which for the moment will be exclusively held by eidoo. The reasoning behind only using this single account is that it requires orders to only be invalidated off-chain from the order book, thereby not costing user's for on-chain transactions to cancel. A future iteration may find it useful to implement some on-chain safeguard or even time decay in case of larger orders, as there is a trust element with eidoo here that it won't potentially play these orders out regardless of the off-chain cancellation.

For verification it makes use of Solidity's ecrecover function, to recover the associated address from the elliptic curve signature, or in the case of an invalid signature, will return 0, in which case the execution is halted and an error returned. The order parameters are hashed on-chain and then passed manually to the ecrecover, rather than a hash provided ahead of time, as any parameters could then be passed with a valid hash, but by ensuring the parameters are hashed on-chain, order mutability can be mitigated.

A salt is included in the parameters to ensure the possibility of creating multiple orders that may otherwise be identical.

Once the matched order signatures have passed verification, the trade execution is sent to the respective wallets to be validated, and once confirmed by each party's wallet, the balances are atomically swapped. Upon execution, the order fill states are updated, to avoid replays and ensure only the amount specified gets swapped over the lifetime of the order even in the event of multiple partial fills.

The exchange takes advantage of multiple matching orders from the same taking order by running them through the batched order execution function, which both provides benefits in terms of gas savings, and quicker on-chain settling by minimizing effects of network congestion as it will not be posting several separate transactions, and instead batch their execution into 1.

Wallet

To begin with, the purpose behind the wallet is to allow for isolated wallets containing user funds and being administered by the user, with the choice of eidoo-approved logic it may use as well as direct choice of exchange that has pull permissions for order execution.

This provides the user with the opportunity to ensure the exchange contract is a static and safe contract along with the underlying logic the wallet is to use, providing ample opportunity for a vigilant user to keep their funds safe and move them out if needed.

The portion of the wallet that is interacted with is the base wallet; essentially an interface contract with barebones code that is persistent and holds relevant storage information. The rationale behind this is to define a set of core features that are needed to facilitate appropriate wallet interaction with the exchange, while giving it the flexibility for the underlying logic to be upgraded and improved upon, as the logic portion is resident within wallet logic contracts.

The base wallet proxies most of its method calls to a single instance of the currently chosen wallet logic contract within the base wallet. In addition to providing greater flexibility, it also saves users significant gas costs upon initial deployment, and especially in the longrun, when it comes to updates, it not only saves them the gas costs of subsequent upgrades, but also from having to go through the inconvenient steps of withdrawing funds from an old wallet, creating a new wallet, and then depositing to said new wallet; instead users are able to reduce these steps, into a single transaction, indicating the specific version of the accepted logic they wish to opt into, that are tracked by the wallet connector.

The afore mentioned connector is a persistent contract that bridges each of the segregated wallets with the versions of wallet logic currently accepted by the eidoo hybrid exchange. The owner of the contract will be an eidoo controlled ETH account which has the possibility to add and remove accepted logic.

Wallet users will have the choice to opt-in to whatever is considered the latest version as specified in the connector. Those running older versions that have been deprecated, may be unable to execute or post trades, and find their previous orders removed, until they optin to the latest logic. It would be recommended for users to wait for some third-party audit or verification of the new logic before opting-in. In the case of some malicious logic being forced, thanks to the opt-in mechanic, the user's wallet should still be running using safe logic, thereby being able to withdraw their funds away in such a case.

The wallets also partake in the validation process for execution of an order with the current exchange iteration. The first part of the validation ensures that the exchange doing the pull request is in fact whitelisted within the wallet. Thereafter the wallet ensures it has a sufficient balance to cover the trade from its side, along with enough EDO tokens to pay any applicable trading fees. It then provides approval to the exchange to appropriate the needed relevant amounts for the trade.

Off-Chain Backend Architecture

The tech stack that the order book backend runs on consists of:

- Core: Node.js 8+
    - Logging: bunyan
    - Testing: Mocha/Chai

- API: restify

- DB: RethinkDB

The various parts are built using a number of independent components varying from a component that handles communication with Ethereum clients via web3, another which handles the database operations, along with a separate API component.

Development of the backend was targeted towards it running on a multi-cluster environment, and has the capability to utilize as many cores as may be available on the server it runs on. Wherever possible, ES7 asynchronicity was utilized, to both avoid callback hell, provide better readability and maintainability, and running non-blocking code.

These factors aid in the possibility of running the backend in a distributed fashion, due to this setup and RethinkDB sharding. With these principles the order book API achieved a throughput of 50 orders/second being created; when running on 3 threads, on a singlecontained 4 core server, which was running Parity and RethinkDB as well, which during the throughput tests are hogging the majority of CPU cycles and time.

That already surpasses the maximum Ethereum transactions per second of 15, on just a single mid-end server, but is expected to achieve even greater throughput in a properly distributed architecture, where rather than everything being contained on one server, multiple parts are spread across load-balanced servers. This same instance was able to handle thousands of read requests a second.

Order Book Server

To submit orders to the exchange, an authenticated api endpoint is made available which will receive requests from Oraclize's long-running computation instance, with the purpose being third-party verification and authenticity to ensure no front-running or other underhanded practices are happening against users.

The orders are normally expected to have been prepared and sent by the eidoo wallet/ exchange app, containing the order parameters including token or Ether being offered, token or Ether wanted in exchange, and the accompanying signature. This portion is expected to be done client-side, as the signing makes use of the user's private key that controls the exchange wallet, and signs the requested order parameters.

Once the order book api receives such a request, it does a number of sanity checks and parsing on the order request. Thereafter, it verifies the accompanied signature, and ensures the order creator, has a sufficient balance for the order to execute, also taking into account existing orders in the book and the EDO token fee charged. If these checks are passed, the server attempts to match the order with any other orders posted in the book. If such a match is made, they will be batched together, and broadcast on-chain via the exchange relayer address, to the exchange.

Any orders that were matched, have the amounts included in the order locked until on-chain settlement, after which they'll be confirmed as being filled, or in the case of some unexpected issue with the transaction, the orders will be reopened for taking. In regards to the original taking order, any amounts that are not matched, are posted in the order book which runs on RethinkDB, for a future order to take.

Users may also cancel any submitted orders from the order book. In this case they will be signing the order Id and a cancellation confirmation string. Cancellation of orders thereby works akin to creation of orders, and holds the benefit of no on-chain costs associated with it, while still providing relative security in that to cancel the order, one needs the private key of the account associated with the order.

Once the order is cancelled, it is flagged to be ignored by the order matcher, and is another reason why the relayer is currently a single permissioned address held by eidoo, as non-permissioned relaying would require an on-chain mechanism for cancellation of the orders. One pitfall is the trust needed in eidoo to not replace these orders sometime in the future.

This can be mitigated currently, if a user feels there is a risk of this happening, and they've cancelled a high value order in the past, they could make it impossible to replay by simply creating a new wallet using a different account, and transferring all funds to the new account. Another workaround is an on-chain decay/expiry option for orders, where orders are not tradable after a certain declared timestamp, or simply allowing on-chain order invalidation through a future wallet implementation.

There is also a watchdog process running in a separate thread, watching for any pertinent on-chain events happening with the exchange and wallets associated with it. In addition it watches any pending transactions from the relayer address. This allows it to work as a bridge synchronizing the off-chain and on-chain contexts. In the case a user withdraws funds away from their wallet, but still has orders listed in the order book, the watchdog reacts and cancels those orders which are no longer valid or able to be executed upon.

It is also its duty to release any amounts that were locked in the order book, but where the associated on-chain transaction failed, potentially due to network congestion, and it being removed from the network's transaction pool. It also has a configurable parameter, for assuming blockchain finality, to avoid issues with blockchain reorganizations, allowing it to adapt to changing network conditions.