zkSwap - Scaling Decentralized Exchanges through Transaction Aggregation

Paul Etscheit

March 2021

0.1 Introduction

When being launched in 2015, Ethereum [8] set out to change the way we compute. A trustless, permissionless, and decentralized world computer was envisioned, set to open a new class of applications. The importance of running verifiable, Turing-complete code in a permissionless and trustless manner cannot be overstated and enables products and services not thought to be possible. However, the technical limitations have also become apparent quickly. Computations are expensive, theoretical transactions per second are low, and the overall throughput has been stagnant. While Eth2 gives a path towards scaling the network, it is expected to take years to complete.

The first major use case for Ethereum was tokenization. With the development of the ERC-20 standard, launching a token on the Ethereum blockchain was trivial. As tokens run as smart-contracts on the Ethereum blockchain, they are secured by its proof of work consensus, which, given Ethereums PoW hash rate, makes consensus attacks infeasible. Running on Ethereum blockchain is a significant benefit when looking to tokenize things, as network security can be assumed. While tokenizations are a step in the right direction, they do not come close to the initial vision. While the standardization enables simple integrations into exchanges and wallets, most tokens are isolated in their functionality and ecosystem and lack productive usage.

token all isolated, not working together... Not a lot of gas needed blabla

With all of these developments over the past couple of years, it seems we have now entered a new phase of smart-contract use-cases, namely Decentralized Finance (DeFi). While DeFi has many different products and functionalities, at its core aims to utilize tokenized assets in some productive form.

Lending and collateralized borrowing is possible with Aave [6], assets can be deposited into liquidity pools[2] to generate yields, flash-loans [6][2] enabled uncollateralized borrowing as long as the loan is repaid in the same transaction and assets can be traded in a non-custodial way with Uniswap [2] . It can be questioned how useful or necessary these protocols really are, but the core idea behind them is impressive. Rebuilding traditional financial products, running as non-custodial and permission-less smart-contracts, all based on the same standardizations, has the potential to reshape the way finance works. With these developments not looking to slow down, they are quickly overwhelming the Ethereum blockchain, pushing transaction costs [4] higher and higher.

Mention other swap protocols?

One of these new DeFi applications is Uniswap [2]. Uniswap is a crypto-asset exchange running as a collection of smart-contracts on the Ethereum blockchain, enabling non-custodial, trust-less, and permission-less trading of ERC-20 assets. Since its running on the Ethereum Blockchain, reducing the computational complexity of trade execution is essential for making it a viable product. In typical crypto-asset exchanges, trading is built around a central order book. Users can add buy or sell orders for a given trading pair, and a matching engine checks if these orders can be matched, executing the trade once they do. While running this on modern server infrastructure is feasible, running it on the blockchain is not. The demand for memory and processing power is too large, so a different approach must be taken. Uniswap solves this by applying the automated market maker (AMM) model, which will be explained in detail in sec. XX. By applying this model, Uniswap reduces the computational complexity to make this a viable business model. At least it was, when Uniswap launched.

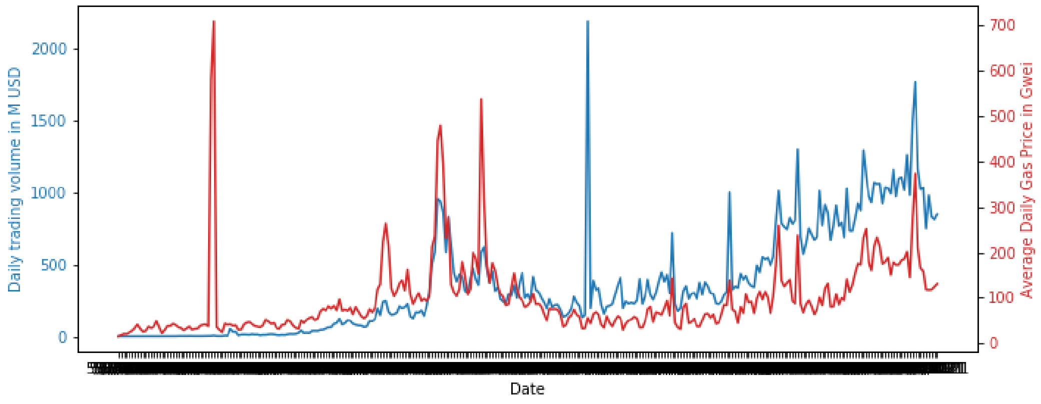

Figure 1: Combined daily Uniswap trading volume in million USD and average daily Ethereum gas price

The recent rise of Ethereums gas price [4] can also be attributed to the growing popularity of Uniswap. Currently, it is one of the most used smart-contract on the Ethereum blockchain, making up on average around 15% of gas [3] usage of a block at the time of writing. To date, it accrued over $280 million in transaction fees [5] and has settled over $100 billion in trading volume [1]. With the gas price having reached 500 gwei on a couple of occasions, a single Uniswap trade can cost upwards of $130. While it would be assumed that high gas prices cause a reduction in trading volume, the opposite is the case. As shown in F.1 there seems to be a strong correlation between daily trading volume on Uniswap and the average daily Ethereum gas price, so reducing gas consumption by Uniswap transactions should result in a reduced gas price for the entire network.

With longer-term scaling solutions in development but still years away, a shorter-term solution is needed. A couple of short-term scaling approaches have been proposed. While these do differ, they all aim to move transactional data to a layer-2[1] system, while ensuring correctness of that data in some way. One of the approaches is called zk-rollup, the focus of this work. Moving data to a layer-2 system can increase the number of transactions that fit into a block while also reducing transaction costs for the user, with is beneficial for Uniswap users and all other participants of the Ethereum network.

[1]A layer-2 system is a data storage that does not reside on the blockchain but has its state committed to it in some way

Current zk-rollup enabled applications running on the Ethereum mainnet, one of them being ZK Sync, are focused on reducing cost of Ether and ERC-20 transfers. Users deposit funds into its smart-contract, which results in the user's deposit being represented as balance in layer-2. When a user makes a transfer to another user, the involved balances get updated in layer-2, while the correctness of these updates is ensured via zkSNARK. It is important to note, that ZK Sync acts as a closed system, transfers only change balances in layer-2, while deposited funds in the smart-contract do not move. While this approach has significantly reduced costs of transfers, it only marks the first generation of potential(?) zk-rollup enabled apps.

not a great sentence

Aggregating Uniswap trades is an interesting application to explore the potential of zk-rollup technology. It combines the layer-2 storing and updating of balances already done by ZK Sync while opening the system to interact with other smart-contracts. When aggregating trades, we need to interact with the Uniswap contracts to execute the aggregated trade, then update the layer-2 balances according to the trade and verify everything via zkSNARK. It is the next step in exploring the potential of zk-rollups as a generalizable scaling solution, applicable to any kind of smart-contract.

0.2 Background

0.2.1 zk-rollup

Zk-rollup [7] is a layer-2 scaling approach first introduced to mass validate transfer of assets on the Ethereum blockchain in 2018. A user can deposit funds into a smart contract by providing a merkle path to its balance and adding funds to be deposited to the transaction. The smart-contract checks if its root can be recreated with the provided merkle path, updates the balance according to the funds in the transaction, rehashes the entire tree and updates its root to the resulting hash. An event is then emitted, containing the new balance, putting the data on-chain is a cheap way. The merkle tree, which is required for generating the merkle paths, containing the balances can be kept in sync by listing to these events.

To make a transfer, a user sends the receivers address, tranfer amount and its signature to the relayer as an http request. Once enough transfer requests have been received, the relayer checks if this transfer is covered by a users balance and if the signature is valid, and updates the balance of involved users accordingly. All of this is done in a zkSNARK program, which will return a proof object, the new balances and the new merkle root. These are then sent to the smart-contract. If the zkSNARK proof can be verified, we have proven that the new balances and the root are correct. We now emit the new balances as an event, thereby moving custody of transfered funds to the receiving users, who are now able to transfer or withdraw them. Withdrawing of funds follows the same logic as depositing, however instead of sending funds, a parameter is added containing the requested amount.

0.3 Design

The implementation of this system consist of a number of entites, that are required to aggregate and execute trades in a trust-less and non-custodial manner, while not having any data availability issues. As a first step, the smart-contracts used for depositing and withdrawing funds into the system will be explained. The second step will introduce the off-chain entity, that is tasked with aggregating, executing and verifying the trades.

Need to find a better word for data availablilty issues

0.3.1 System Overview

The system consists of two main entites that are required for it to function. The first entity to look at, is the on-chain entity, we call zkSwap. zkSwap is a smart-contract deployed on the Ethereum blockchain and has three main jobs, processing deposits and withdraws, aswell as verifying batched trades. As our goal is to not rely on external data-availability, while remaining trust-less and non-custodial, the zkSwap contract is the only entity that can change the state stored in layer-2, by emitting state updates via the event log. Since the event log can be extracted from the blockchain by clients, it is always accessible, while being signficantly cheaper then storing data as a variable in a smart-contract.

The second entity to look at is the aggregator. The aggregator consists numerous systems, both off-chain and on-chain, and is mainly tasked with receiving trade orders, batching and executing them, and then verifying them with the zkSwap contract. The aggregator stores a merkle-tree of users balances and keeps it in sync by listening for event emitted by the zkSwap contract.

0.3.2 Storing and Updating Balances

The first thing to understand, is the way balances are stored. Two main factors are dictating the ways balances are stored in the system. We want to make balance updates as cheap as possible, while not relying on any external data availability. To achive this, we store balances in a merkle tree in layer-2. Merkle trees are a suitable data structure, as the root represents the entire tree state in a highly compressed form, while proving a leafs inclusion in the tree can be done with O(log n). This is idea for our use-case. We can cheaply commit the merkle trees state to our smart-contract by storing the root, while verifying inclusion of a leaf is very efficient. To deposit funds, a user need to provide a valid merkle path to its current balance and the current balance object. The trader calls the deposit function, passing the merkle path and balance object as a parameter and adds the funds wishing to be deposited to the transaction.

First we check if the balance is included in the tree. To do this, we first hash the balance object with the sha256 algorithm. The balance object contains the following fields: ethBalance, tokenBalance, nonce and address.

When hashing the balance object, we extract the users address from the transaction. This ensures that the user is in control of the addresses private key which suffices as security check. Since the contract stores the merkle root, we can now hash the path and balance, checking if we can recreate the root. If the roots match, the trader is permitted to update its balance in the tree. We now update the balance object by adding the deposited amount to the corresponding field, increment the nonce, hash the updated balance object, and rehash the entire tree with the the new balance object. The resulting merkle root is set in the contract, which commits the updated state.

However, we still need to store the balances somewhere. As we don't want to rely on external data availability, while keeping storage costs as low as possible, we emit the new balance as an event. Writing data to the event log is a lot cheaper, compared to storage as a variable in a smart-contract. While the event log can't be accessed from a smart-contracts runtime, a client can query the event log and pass the data as a parameter. As the merkle trees state is committed in the contract with the merkle root, correctness can always be proven.

Withdrawing funds follows the same logic as deposits. Instead adding funds to the withdraw function, the user passes the amount to withdraw as a parameter. The merkle tree is rehashed, checking if the root is correct, it is ensured the withdrawAmount ¡= balance, the balance and nonce are updated, creating the new root. We emit the new balance and send the funds to the user.

Deposits and withdraws are complete on-chain operations by design. All data needed to withdraw funds are public, except for the private key. This ensures funds can always be withdrawn, as long as the user controlls its private key.

0.3.3 zkSwap Contract

Since we're looking to aggregate Uniswap trades, the obvious functionality required is for users deposit and withdraw funds, which will then in turn be used for trading. Users can deposit or withdraw funds by sending the corresponding transactions to the zkSwap contract, which will then

As mentioned before, the zkSwap contract is the only state-changing entity in this system. All state-changing operations are checked and then committed by this contract. This might be counter intuitive at first, as the goal is to move as much data as possible off-chain. It is important to remember though, that one main goal is not to rely on any external data availibility. This can only be achived by storing state on-chain, as compressed as possible.

At the center of storing state is our system, is the root of the balance merkle tree, stored in the zkSwap contract.

Since we're looking to aggregate Uniswap trades, the obvious functionality required is for users deposit and withdraw funds, which will then in turn be used for trading. As we're looking to move as much data as possible to layer-2.

Bibliography

[1] Uniswap cummulative volume, https://duneanalytics.com/projects/uniswap

[2] Adams, H., Zinsmeister, N., Robinson, D.: Uniswap v2 core. URl: https://uniswap.org/whitepaper.pdf (2020)

[3] Ethereum gas guzzlers, https://ethgasstation.info/gasguzzlers.php

[4] Ethereum gas price, https://etherscan.io/chart/gasprice

[5] Uniswap total fees used, https://etherscan.io/address/0x7a250d5630b4cf539739df2c5dacb4

[6] Kulechov, S.: The aave protocol v2 (Dec 2020), https://medium.com/aave/the-aave-protocol-v2-f06f299cee04

[7] Vbuterin: On-chain scaling to potentially 500 tx/sec through mass tx validation (Sep 2018), https://ethresear.ch/t/on-chain-scaling-to-potentially-500-tx-sec-through-mass-tx

[8] Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**(2014), 1–32 (2014)