zkSwap - Scaling Decentralized Exchanges through Transaction Aggregation

Paul Etscheit

March 2021

1 Introduction

When being launched in 2015, Ethereum [9] 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.

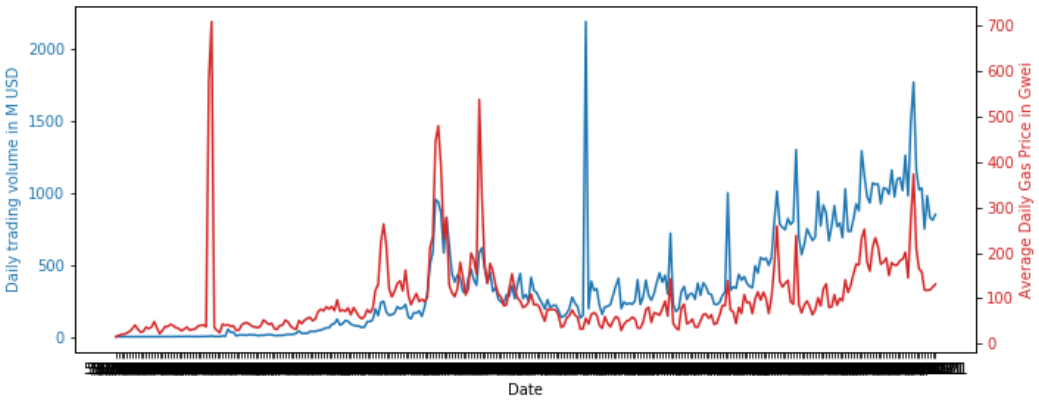

Figure 1: Combined daily Uniswap trading volume in million USD and average daily Ethereum gas price

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.

2 Background

2.1 zk-rollup

Zk-rollup [8] 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.

3 Design

The goal of the work is to explore if zk-rollups can be used to aggregate Uniswap trades in an effective manner. The prototype is able to aggregate trades for a single trading pair, Ether and an ERC-20 token of choice. 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. It holds users funds and exposes the on-chain functionality, namely deposits and withdrawls, to the user.

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, aggregating 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.

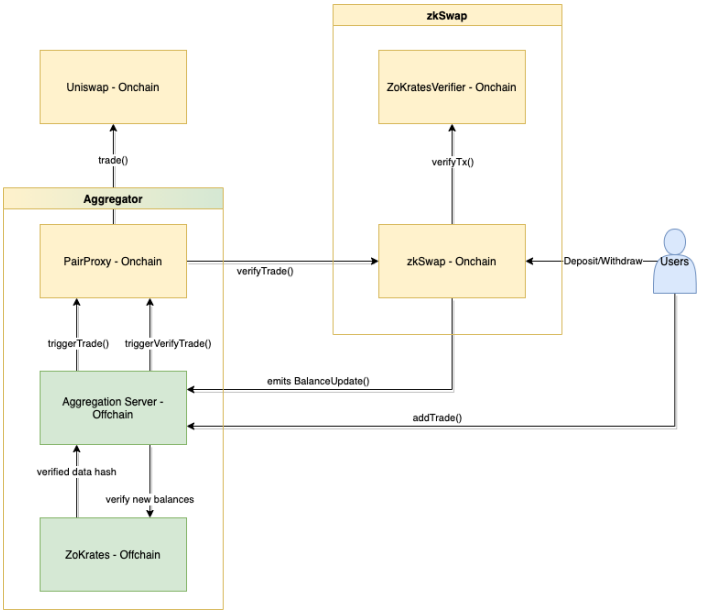

Figure 2: High level architecture of the system

3.1 zkSwap Contract

The zkSwap contract, is the core entity the user interacts with. Its a smart-contract, that from the users perspective, is mainly used for depositing and withdrawing funds in out system.

3.1.1 Storing and Updating Balances

Two main factors are dictating the way balances are stored in the system. It is important to understand the core technique used to store and update balances before we look at the different functions that trigger them. We want to make balance updates as cheap as possible, while not relying on any external data availability. Essentially, this means that we need to store the balances on-chain. Storing data on-chain is typically very expensive. It is important to make a distinction between storing data in a smart-contracts runtime and storing data in the event log. Both are on-chain, the event log is significantly cheaper though. While the event log is cheap to use and can be queried by any client, it is not accesible to a smart-contracts runtime. This solves the external data availablity problem. We can store data cheaply, without relying on other systems to stay online. A client can query the event log, gather the required data and pass it as parameters to the transaction. However, we now need a mechansism to ensure, the parameters passed are equal to the parameters found in the event log.

We can achive this, be using a merkle tree [7]. Merkle trees are a suitable data structure, as its 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 ideal for our use-case. Every balance is stored as a leaf in a merkle tree, running in layer-2. The merkle tree is built and kept in sync by subscribing to the 'BalanceUpdate' event emitted by our smart-contract. A client can query a balance from this tree, receiving the valid merkle path along with the data. Since the smart-contract stores the root of the balance merkle tree in its runtime, it can verify the correctness of the passed balance by hashing it along with the provided merkle path. If the resulting hash is equal to the stored root, the correctness of the balance is proven.Any changes to the balance data by the client will result in the hashes to mismatch, thereby invalidating the data. While the hashing of the balances and merkle path does create some fixed, overhead cost, the savings by using the event log far outstrip it. The hashing costs will be analysed in S. Results.

Should I explain on what security assumptions this is based

This fulfills the balance storage requirements, reducing costs, not relying on external data availabilty and ensuring the correctness of data. It must be noted, that this also causes the smart-contract to be the single source of 'truth'. Since all balances changes are committed by a new event being emitted and the root being updated, any balance updates must be done through the smart-contract.

3.1.2 Deposits and Withdraws

When using the system, a user first has to deposit funds. Since the entire idea of zk-rollup is to move funds to layer-2, the deposit function can be seen as a bridge that connects the mainnet and layer-2. When a user makes a deposit, the funds are represented as a balance object in layer-2, which in turn give custody to these funds. When moving funds in layer-2, we don't actually

move the funds residing in the smart-contract, but update the balance objects to represent the movement. Since a balance object gives a user custody of represented funds, it can always be redeemed, moving from layer-2 back to mainnet.

As described in S. 3.1.1 balance updates are secured by merkle inclusion proofs in the smart-contract. When depositing the user needs to provide a valid merkle path to its balance object, and the balance object itself. The balance object consists of four fields that are needed to represent the balance: ethAmount, tokenAmount, nonce and userAddress. As a first step, the balance object is hashed with the sha256 algorithm, the result being the leaf in the merkle tree. The leaf is now hashed with the merkle path, according to the standart merkle root hashing algorithm. If the resulting hash equals the stored balance root in the contract, we can be assured that the balance object is correct. We now add the value of the transaction, which is the deposit amount sent with the transaction, to the ethAmount, increment the nonce and hash the new balance. Since the balance object has the same position in the tree, the same merkle path is also valid for the new balance. The new leaf is hashed with the merkle path, and the resulting hash is set as the new balance root. As a last step we emit the 'BalanceUpdate' event, containing the new balance object, matching the balances with the newly set root.

Withdraws largely follow the same logic, there are small differences though. As we're requesting funds, instead of sending them, we pass a withdrawAmount as an additional parameter to the function call. We then check if the withdrawAmount ¡= ethAmount to make sure the withdraw is covered by the users balance. The inclusion proofs are the same, we then subtract the withdrawAmount from the balance, and update the balance root. As a last step we emit the 'BalanceUpdate' event with the new balance object, and send requested funds to the user as an on-chain transaction.

This however, is an incomplete explaination, as we're not checking if a user is permitted to deposit or withdraw funds. As balance objects are emitted as an event, anyone can access them and compute valid merkle paths for any balance. This would allow any user to withdraw any balance. To ensure a user is permitted to update a balance object, we need ensure the user controlls the private key belonging to the balance objects user address. Fortunetly we can ensure this by accessing the sender in transaction object. The Ethereum blockchain ensures a user is allowed to make a transaction by requiering the transaction to be signed with the private key of the senders address. If that signature is valid, it is proven that the user has access to the addresses private key and the transaction can be executed. Because of this, the transactions object sender can be trusted to be in controll of the corresponding private key. Instead of passing the users address as part of the balance object, the smart-contract uses the sender of the transaction object. This suffices as a security check.

It must also be noted, that ERC-20 deposits and withdraws behave a bit differently. As every ERC-20 token has its own smart-contract, representing

a users balance as a mapping, we need to transfer the assets in that contract. This requires different functions to handle ERC-20 deposits and withdraws, however implementing this is trivial and not worth explaining in the context of this work.

Figure 3: Pseudocode of the deposit and withdraw steps

3.2 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] Szydlo, M.: Merkle tree traversal in log space and time. In: International Conference on the Theory and Applications of Cryptographic Techniques. pp. 541–554. Springer (2004)

[8] 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

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