0x: A protocol for decentralized exchange on the Ethereum blockchain

Will Warren, Amir Bandeali

Release Candidate v1

Abstract

We describe a protocol to facilitate low-friction peer-to-peer exchange of ERC20 tokens on the Ethereum blockchain. Data formatting conventions and a shared smart contract infrastructure (shared protocol layer) are proposed to establish a global liquidity pool for Ethereum applications that incorporate exchange functionality. Orders are generic data packets that are signed by the order originator and broadcast over arbitrary channels off of the blockchain...

Introduction

Peer-to-peer exchange > centralized exchange. Privacy, security, non-custodial, global.

Hacks of Mt. Gox, Shapeshift and Bitfinex have demonstrated that pooling user funds in a central location inherently creates systemic risks. Despite security measures such as multi-signature wallets and air-gapped cold storage systems, pooled user funds remain vulnerable to negligent or rogue employees as well as social engineering attacks. Peer-to-peer exchange eliminates the systemic risks associated with centralized exchange services by allowing users to transact directly - without a middleman - and by placing the burden of security onto individual users rather than onto a single custodian.

Exchange as a shared protocol layer > exchange as a proprietary application. Shared liquidity pool, standard APIs, cross-compatibility.

Numerous dapps incorporate exchange functionality. Since there are no standards in place, each dapp must implement exchange functionality from scratch. Dapp users must trust that each implementation is free of errors. Dapp users must go through a unique configuration process each time they use a new dapp that incorporates exchange functionality.

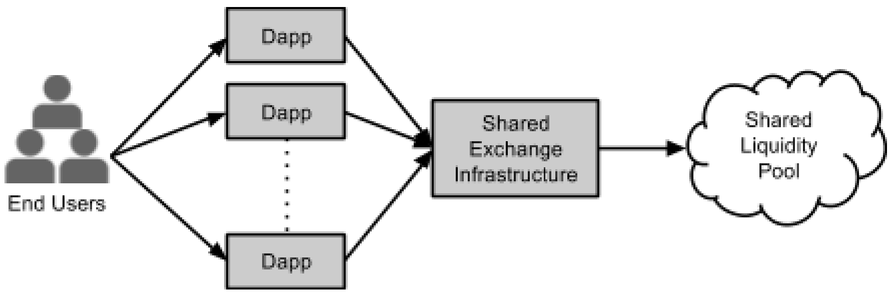

Figure 1: ...

Protocols should be decoupled from applications and standardized

Augur is an example of an application composed of multiple protocols. A decentralized oracle protocol (REP) and an exchange protocol (proprietary) are used for a prediction markets application. Observe that

- the two protocols are independent of one another, and
- each protocol could be used for a variety of applications besides prediction markets.

Shared protocol layers = public infrastructure, building blocks

Why is decoupling/standardization not the current trend?

It is easier to sell an application than a protocol. Applications form the top layer of the blockchain application stack; the layer that is exposed to the end-user. Protocols *have* use cases, applications *are* use cases. End-users want completed products, not building blocks.

Applications don't need a native token to function (their underlying protocols might). Tokens for highly specific applications often feel like a proxy for equity or a tool used to drive speculation. The purpose of a protocol token is to align incentives across a large network of nodes.

Fred draft
Blockchains have been revolutionary by allowing anyone to own and transfer an asset on an open network without the need for a trusted third party. Now that there are hundreds of blockchain-based assets, the need to exchange these assets has accelerated. With the advent of smart contracts, it is possible to trustlessly exchange any two blockchain assets without the need for a trusted third party. We propose a generic decentralized exchange protocol to accomplish this.

Decentralized exchange is an important progression from the ecosystem of centralized exchanges for a few key reasons:

- The decentralized exchange protocol is native to the blockchain, allowing any decentralized application to plug in directly.
- Unlike centralized exchanges, a decentralized exchange is open and impartial, making it much more likely to be widely adopted. This incentivizes a global liquidity pool rather than fragmented liquidity pools across centralized exchanges.
- Security guarantees are stronger. There is no longer a central party which can be hacked or run away with customer funds.
- The cost of exchange naturally approaches 0 over time by eliminating the need for a middleman.

In the long run, open standards tend to win over closed ones, and more assets are being tokenized on the blockchain each month. We will see blockchain applications which are likely to require the use of many different tokens. As a result, an open standard for exchange is critical to supporting this open economy.

Existing Work

Decentralized exchanges implemented with Ethereum smart contracts have failed to generate significant volume due to inefficiencies in their design that impose high friction costs on market makers. In particular, these implementations place their order books[1] **on** the blockchain[1-4], requiring market makers to spend gas each time they post, modify or cancel an order. While the cost of a single transaction is small, frequently modifying orders in response to evolving market conditions is prohibitively expensive. In addition to imposing high costs on market makers, maintaining an on-chain order book results in transactions that consume network bandwidth and bloat the blockchain without necessarily resulting in value transfer.

Automated market maker (AMM) smart contracts are proposed[5] as an alternative to the on-chain order book. The AMM smart contract replaces the order book with a price-adjustment model in which an asset's spot price deterministically responds to market forces and market participants on either side of the market trade with the AMM rather than with each other. Benefits of the AMM include availability (it is always available to act as a counterparty, though the spot price it offers may be worse than what one could get from a more traditional exchange) and ease-of-integration with external smart contracts that need to execute market orders. The deterministic nature of price-adjustment models make them "insensitive to market liquidity, meaning that trades cause prices to move the same amount in both thick and thin markets"[6]. AMMs impose artificial constraints on the supply curve. If the price-adjustment model is too sensitive, even small trades will produce large fluctuations in the spot price. If the price-adjustment model is not sensitive enough, the AMM's bankroll will quickly be depleted by arbitrageurs.

State channels are proposed as a means of scaling the Ethereum blockchain and reducing costs for a variety of applications - including exchange - by moving transactions off of the blockchain. Participants in a state channel pass cryptographically signed messages back and forth, accumulating intermediate state changes without publishing them to the canonical chain until the channel is closed. State channels are ideal for "bar tab" applications where numerous intermediate state changes may be accumulated off-chain before being settled by a single on-chain transaction (i.e. day trading, poker, turn-based games). Conversely, the friction costs associated with opening and closing a state channel make them inefficient for one-time transactions... Not flexible - can't buy a few dapp coins and immediately turn around and begin using the dapp.

If one of the participants leaves the channel or attempts to cheat, the other participant may publish the most recent message they received from the offender to the canonical chain. Channel participants must always be online to challenge a dishonest counterparty and are therefore vulnerable to DDOS attacks. Sacrifices security for **scalability** (speed and throughput).

...

Moving the order book off of the blockchain significantly reduces costs for all market participants, market makers in particular. However, this approach creates a vulnerability to spam attacks: since the funds underlying an order are not locked within an on-chain order book or within a state channel, they may be moved out from under an open order. There is no financial commitment tied to fulfilling an order. EtherDelta[7]. Proprietary, not standardized. Vulnerable to spam.

[1] An order book is used to publicly record the interest of buyers and sellers in a particular financial instrument. Each entry includes a reference to the interested party, the number of shares and the price that the buyer or seller are bidding/asking for the particular security.

Specification

Figure 2 presents the general sequence of steps used for off-chain order relay and on-chain settlement. For now, we ignore a few mechanisms that will become important later.

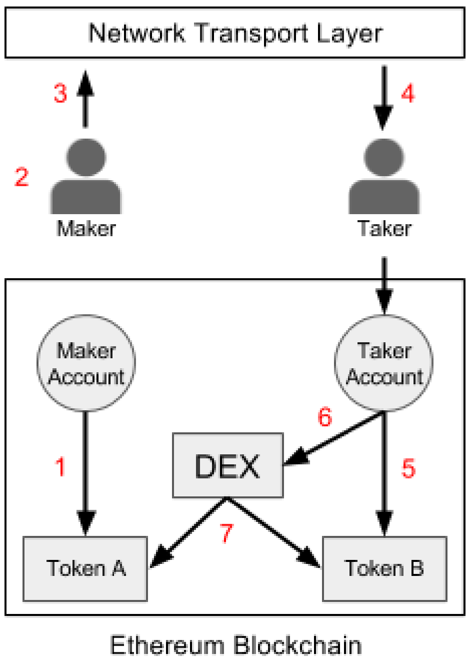

Figure 2

1. Maker approves the DEX contract to access their balance of Token A[2].
2. Maker creates an order to exchange Token A for Token B, specifying a desired exchange rate, expiration time (beyond which the order cannot be filled), and signs the order with their private key.
3. Maker broadcasts the order over any arbitrary communication medium.
4. Taker intercepts the order and decides that they would like to fill it.
5. Taker approves the DEX contract to access their balance of Token B.
6. Taker submits the maker's signed order to the DEX contract.
7. The DEX contract authenticates maker's signature, verifies that the order has not expired, verifies that the order has not already been filled, then transfers tokens between the two parties at the specified exchange rate.

[2] See ERC20 Token in Appendix. Approve once, execute an unlimited number of trades thereafter.

Message Format

Each order is a data packet containing order parameters and an associated signature. Order parameters are concatenated and hashed to 32 bytes via the Keccak SHA3 function. The order originator signs the order hash with their private key to produce an ECDSA signature.

Point-to-point

Point-to-point orders allow two parties to directly exchange tokens between each other using just about any communication medium they prefer to relay messages. The packet of data that makes up the order is a few hundred bytes of hex that may be sent through email, a Facebook message, whisper or any similar service. The order can only be filled by the "taker" address specified within the message, rendering the order useless for eavesdroppers or outside parties.

Name	**Data Type**	**Description**
version	address	Address of the Exchange smart contract. This address will change each time the protocol is updated.
maker	address	Address originating the order.
taker	address	Address permitted to fill the order.
tokenA	address	Address of an ERC20 Token contract offered by maker.
tokenB	address	Address of an ERC20 Token contract requested by maker.
valueA	uint256	Total units of tokenA offered by maker.
valueB	uint256	Total units of tokenB requested by maker.
expiration	uint256	Time at which the order expires (seconds since unix epoch).
v	uint8	ECDSA signature of the above arguments.
r	bytes32	
s	bytes32	

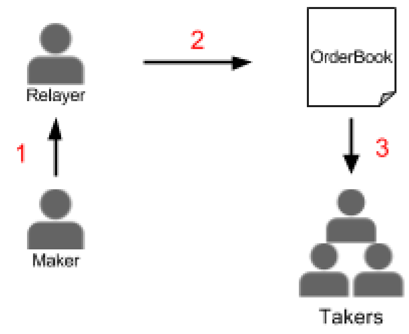

Figure 3: Relayers host and maintain a public order book in exchange for transaction fees.

Broadcast

For liquid markets to emerge, there must be public locations where buyers and sellers may post orders that are subsequently aggregated into order books i.e. exchanges. Building and operating an exchange is costly and the protocol we have described so far does not provide an incentive for someone to take on such an expense. Broadcast orders include two changes to the point-to-point message format in order to support public exchange services:

- Remove *taker* ⇒ broadcast orders can be filled by anyone.
- Add feeA*,* feeB*,* and *feeRecipient* ⇒ specifies transaction fee values (in protocol tokens) and the address that will receive the fees.

Name	**Data Type**	**Description**
version	address	Address of the Exchange smart contract.
maker	address	Address originating the order.
tokenA	address	Address of an ERC20 Token contract offered by maker.
tokenB	address	Address of an ERC20 Token contract requested by maker.
valueA	uint256	Total units of tokenA offered by maker.
valueB	uint256	Total units of tokenB requested by maker.
expiration	uint256	Time at which the order expires (seconds since epoch).
feeRecipient	address	Address of relayer. Receives transaction fees.
feeA	uint256	Total units of protocol token maker pays to feeRecipient.
feeB	uint256	Total units of protocol token taker pays to feeRecipient.
v	uint8	ECDSA signature of the above arguments.
r	bytes32	
s	bytes32	

While it may seem odd that the maker is specifying the transaction fees, keep in mind that relayers ultimately have control over which orders get posted. Therefore, if the maker wants their order to be posted to a specific order book, they must first ask the Relayer to specify the required feeA, *feeB,* and *feeRecipient* (this interaction occurs before the maker signs the order).

It is also important to recognize that the *feeRecipient* address can point to any arbitrary smart contract. This means that complex incentive structures can be "plugged into" the exchange protocol. For example, a *feeRecipient* contract could be designed to:

- Implement spam prevention
- split transaction fees between multiple relayers ⇒ a single order can be placed on multiple order books (imagine placing an order and having it show up on both GDAX and Poloniex).
- distribute transaction fees across a swarm of nodes according to the level of contribution each node makes in propagating the order book within a censorship-resistant p2p network[3].

[3] Development of a relay protocol that supports a low-latency, distributed order book is being considered for the next phase of this project.

Smart Contract

Written in the Solidity programming language. Contains two relatively simple functions: *fill* and *cancel*. The entire contract is approximately 100 lines of code. Costs ~90k gas to fill an order.

Signature Authentication

The exchange smart contract is able to authenticate the order originator's signature using the ecrecover function, which takes a hash and a signature of the hash as arguments and returns the public key that produced the signature. If the public key returned by ecrecover is equal to the 'maker' address, the signature is authentic.

ecrecover(hash, signature(hash)) => public key
If (public key != maker) throw;

Fills & Partial Fills

The exchange smart contract stores a reference to each previously filled order to prevent a single order from being filled multiple times. These references are stored within a mapping; a data structure that, in this case, maps a 32 byte chunk of data to a 256 bit unsigned integer. Passing the parameters associated with an order into the Keccak SHA3 function produces a unique 32 byte hash that may be used to uniquely identify that order (the odds of a hash collision, finding two different orders with an identical hash, are practically zero). Each time an order is filled, the mapping stores the order hash and the cumulative value filled.

A taker may partially fill an order by specifying an additional argument when calling the Exchange smart contract's *fill* function. Multiple partial fills may be executed on a single order, so long as the sum of the partial fills does not exceed the total value of the order.

Name	**Data Type**	**Description**
valueFill	uint256	Total units of tokenA filled by taker (valueFill <= valueA)

Expiration Time

An order's expiration time is specified by the order originator at the time the order is signed. The expiration time is an unsigned integer value that represents the absolute number of seconds since the unix epoch. This value cannot be changed once it has been signed.

Time within the Ethereum virtual machine is given by block timestamps that are set each time a new block is mined. Therefore, the expiration status of an order does not depend upon the time at which a taker broadcasts their intention to fill an order, instead it depends upon the time at which the fill function is being executed in the EVM by a miner. A miner cannot set the block timestamp of the current block to be earlier than the timestamp of the previous block.

Cancelling Orders

An unfilled and unexpired order may be cancelled by the order originator via the Exchange smart contract's cancel function. The cancel function maps an order's hash to its associated maximum value (valueA), preventing subsequent fills. Cancelling an order costs gas and, therefore, the cancel function is intended to serve as a fallback mechanism. Typically, market makers are expected to avoid on-chain transactions by setting their order expiration times to match the frequency with which they intend to update their orders.

One issue with this approach is that it can create situations where a maker attempts to cancel their order at roughly the same time a Taker is attempting to fill that same order. One of the two parties' transactions will fail, wasting gas, depending upon the order in which the two transactions are mined. Uncertainty regarding the order in which transactions are mined could lead to unexpected and undesirable outcomes at times. Uncertainty could increase if the Ethereum blockchain were to experience a significant backlog of pending transactions.

Protocol Token

Economic incentives drive behavior. Protocol tokens may be used to coordinate the behavior of large networks of nodes. If structured correctly, protocol tokens create incentives that drive each node to behave in a way that benefits the rest of the network. Poorly structured protocol tokens drive behaviors that are orthogonal to the interests of the network such as hoarding tokens strictly for the purposes of price speculation and without contributing to the network in other ways. Augur punishes token holders that do not contribute to the network by redistributing a portion of their tokens to active contributors…

The exchange protocol we describe fundamentally exists to facilitate

Decentralized Governance

Continuous Integration

The protocol will inevitably require updates to the underlying smart contracts. Such changes could potentially disrupt the network or create critical security risks for users (in the worst case, attackers could gain access to users' tokens). Decentralized governance via protocol token voting (or vetoing) can provide the oversight needed to safely roll out new contracts.

Changes to the protocol that invalidate existing orders or that require users to opt-in are analogous to hard forks. Hard forks risk fragmenting the network, decreasing the utility of the protocol's network effects. Protocol updates in which the network experiences no down-time and where existing orders remain valid are analogous to soft forks. Ideally, protocol updates can be structured to minimize disruption to the network.

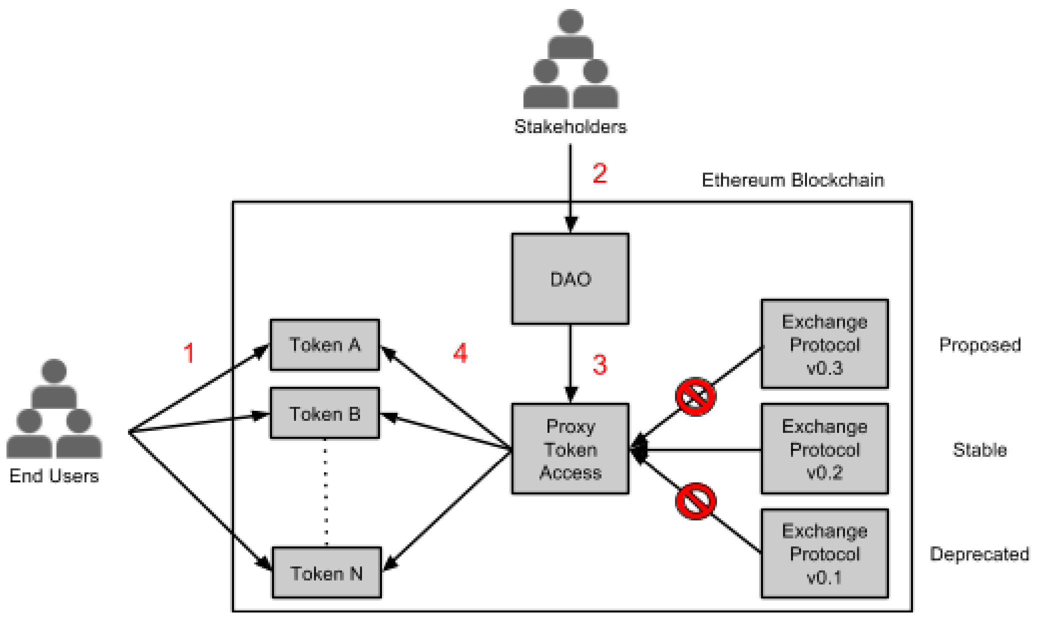

Figure 4: Protocol updates may be deployed without disrupting the network through a combination of contract abstraction and decentralized governance. (1) end users provide a Proxy contract with access to their tokens for trading (2) stakeholders propose and elect protocol improvements (implemented within entirely new smart contracts) via a DAO (3) the DAO authorizes the elected smart contract to access the Proxy (adds the new contract to the Proxy contract's whitelist), and eventually removes authorization from deprecated contracts (4) exchange smart contract accesses user tokens through the Proxy.

Token Registry

Order data consist of hexadecimal strings that are not necessarily human readable, making it difficult to visually confirm that an order will do what it is advertised to do. The Token Registry[4] contract stores a list of ERC20 tokens with relevant information about each token: name, symbol, contract address, and the number of decimal places needed to represent a single token in its smallest units (must know this to determine exchange rates between tokens). The registry serves as an “official” on-chain reference that may be used by market participants to audit order data before executing a trade. With this tool, anyone can independently verify token addresses and exchange rate through the Token Registry (don’t have to trust questionable websites, can avoid scams).

Since the Token Registry will serve as trusted source of information, oversight will be required for modifications (adding/removing tokens, updating token addresses). Decentralized governance may be used to provide this oversight.

While use of the Token Registry for verifying the integrity of an order will be highly recommended, the exchange protocol itself will be ERC20 token agnostic, allowing for maximum flexibility.

[4] https://github.com/ethereum/EIPs/issues/22

Summary

(Outline)

Solution to current lack of liquidity:

- Cheap to use, low friction off-chain orderbook
- Standardized set of smart contracts and APIs
- Allows for same order to be broadcast among different communication channels with different fee specifications
- Incentivizes relayers to provide liquidity (relayers can specify themselves as a fee recipient over their own channels and set their own fees to 0)

Solution to poor usability:

- Universal login/trading wallet for users
- Exchanges compete to get same orders filled with better UIs, lower fees

Maximum flexibility:

- Exchanges/relayers set and keep 100% of fees
- Custom fee structures can be build by specifying smart contract as feeRecipient

Protocol token:

- Decentralized governance
- Fee currency
- Decoupling value of protocol/ether
- Align incentives of stakeholders with best interests of protocol
- Layer 2 compatability (state channels, spam prevention, relay protocol)

Appendix

ERC20 Token

ERC20 establishes a standard contract ABI for tokens on the Ethereum blockchain and has become the de facto representation for all types of digital assets. ERC20 tokens share the same contract interface, simplifying integration with external contracts.

Core ERC20 functions include:

- transfer(to, value)
- balanceOf(owner)
- approve(spender, value)
- allowance(owner, spender)
- transferFrom(from, to, value)

EIP101 includes a proposal to change ether to follow the ERC20 token standard. For now, ether may be imbued with ERC20 token functionality through a “wrapper” smart contract. For reference, see the Gnosis implementation.

Contract ABI

EIP50 proposes an extension to the contract ABI to support structs. This would allow the community to establish standard *Order* and *Signature* data structures, simplifying our contract interface and integrations with external contracts.

Deposit Contract

The use of a smart contract in which to deposit trading funds is optional. However, we feel that the use of a deposit contract will be very beneficial for the following reasons:

- A deposit contract reduces the cost of spam filtering for relayers. Since orders do not provide a guarantee of availability of funds, relayers must actively monitor user balances if they wish to only show valid orders. With or without a deposit contract, a trader can create spam by placing orders and subsequently moving funds to a different account. However, monitoring these changes becomes a much simpler task when a deposit contract is used. An event will be logged whenever a deposit or withdrawal from the contract is made, allowing relayers to easily track user “available balances” by simply updating balances whenever a “Deposit”, “Withdrawal”, or “Fill” event has been logged. The lack of a deposit contract complicates this process; relayers would have to need to review all ERC20 token “Transfer” and “Approval” events that are logged, determine if the event is relevant to any open orders, and then update the user’s available balance accordingly.
- The deposit contract will also reduce the chances of traders accidentally invalidating orders by transferring funds elsewhere, as it creates a distinction between an account’s regular balance and trading funds. We envision the deposit contract becoming a universal “trading wallet” attached to each Ethereum account.
- After the DAO hack, the community has been understandably cautious of pooling large amounts of funds in a single smart contract.

(https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability/) The deposit contract will have very limited functionality, reducing potential attack vectors. In addition, all contracts will undergo rigorous security audits, both by the community and hired security firms.

Cross Orderbook Sharing

- orderHash = keccak256(version, maker, tokenA, tokenB, valueA, valueB, expiration)
- signature = sign(maker, keccak256(orderHash, feeRecipient, feeA, feeB))
- Exchange verifies signature with fee parameters before adding to orderbook
- maker, tokenA, tokenB, valueA, valueB, expiration, feeRecipient, feeA, feeB submitted to smart contract “fill” function
- Smart contract verifies signature, order executed and feeRecipient receives fee
- This allows for the same order to be broadcast among multiple communication channels with different fee specifications. Exchanges could potentially have the same orders and each compete to get them filled.

Expanding on this idea, we can allow relayers to lower an order’s fees and rebroadcast it (allowing relayers to compete for the same orders with fees). This would require slight changes in the exchange smart contract.

- Relayer determines new feeA and feeB that are lower than the original fee values.
- Relayer broadcasts order with added parameters feeA2, feeB2, sign(relayer, signature, feeA2, feeB2)
- When order is filled, exchange contract verifies both signatures and that the new fee values are < the original fee values, then applies the new fees.
- Advantages: This makes it such that market makers no longer have to worry about fee price volatility. Relayers are incentivized to keep fees competitive in order to get orders filled over their own communication channel and receive the fees.
- Disadvantages: Once relayers lower fees, they cannot raise fees (unless the smart contract supported multiple relayer signatures). This also bloats and further complicates the standard order parameters. This scheme is also incompatible with orders where smart contracts are specified as the feeRecipient, as contracts cannot sign arbitrary messages.

Protocol Variants

Protocol Token Function	**Pros**	**Cons**	**Other Comments**
Case A: -No protocol token	-Most flexibility/least friction -Lowest gas usage for traders -Correct incentives can potentially be created with ETH only (as in Swarm)	-"Hard fork" updates only without structured consensus	-Dapps can build custom incentive structures in all cases (by making smart contract fee recipient) -Cases A and B benefit most from sharing information among dapps (to prevent spam)
Case B: -Fee currency -Future relay protocol incentive -DAO	-Network effect -Decentralized governance -Standardized fees across dapps -Fee value determined by use of protocol -Decouples value of protocol and value of ETH -Provides correct incentives for relayers in future relay protocol -Best case for decentralized relay protocol -Can require stake in layer 2 solution for spam prevention -Arguably has the most modular incentives (traders have to worry about trading only, relayers have to do some work to earn fees)	-Volatility in price of fee -Potential conflicts with other dapp protocol tokens -Easier to create spam than it is to prevent spam -Slightly higher gas usage for traders (need to buy protocol tokens first)	-Not necessarily that expensive to prevent spam regardless -Fees for Gnosis currently denominated in proprietary Gnosis token, may not be compatible with fees denominated in exchange protocol token (as one example) -In cases B and C: When orders are filled, liquidity is taken out of the system (fewer protocol tokens are for sale). Relayers have incentive to sell back protocol tokens at a fair price to support market liquidity. Value of tokens should be correlated to velocity of token use -Volatility in price of fees is annoying, but not necessarily problematic. If the price of the protocol token goes up and a trader's previous orders now have too high of a fee, they can cancel/replace the order with a lower fee. This will cost gas, but it also means the trader made money from the price of the token increasing
Case C: -Fee currency -Future relay protocol incentive -DAO -Stake per order (spam prevention)	-Network effect -Decentralized governance -Standardized fees across dapps -Fee value determined by use of protocol -Decouples value of protocol and value of ETH -Provides correct incentives for relayers in future relay protocol -Easier to prevent spam than it is to create spam	-Most friction/worst usability -Volatility in price of fee -Volatility in price of stake -Potential conflicts with other dapp protocol tokens -Highest gas usage for traders (need to buy protocol tokens first, stake tokens, potentially adjust stake amount)	-Case C shifts the burden of spam prevention from relayers to traders

References

[1] Maker Market,
[2] EtherOpt,
[3] Augur,
[ ] IDEX,
[4] Intrinsically tradable tokens,
[5] Euler,
[6] Othman, Abraham, David M. Pennock, Daniel M. Reeves, and Tuomas Sandholm. "A Practical Liquidity-Sensitive Automated Market Maker." *ACM Transactions on Economics and Computation* 1.3 (2013): 1-25. Print.
[7] EtherDelta,
[ ] Fred Ehrsam, App Coins and the dawn of the Decentralized Business Model
[ ] Fred Ehrsam, How to Raise Money on a Blockchain with a Token