AirSwap is an implementation of the Swap Protocol.

Swap: A Peer-to-Peer Protocol for Trading Ethereum Tokens

EN	CN	JP	KR	RU	UA

Michael Oved, Don Mosites

Published May 10, 2017

Abstract

We present a peer-to-peer methodology for trading ERC20 tokens on the Ethereum blockchain. First, we outline the limitations of blockchain order books and offer a strong alternative in peer-to-peer token trading: off-chain negotiation and on-chain settlement. We then describe a protocol through which parties are able to signal to others their intent to trade tokens. Once connected, counterparties freely communicate prices and transmit orders among themselves. During this process, parties may request prices from an independent third party oracle to verify accuracy. Finally, we present an Ethereum smart contract to fill orders on the Ethereum blockchain.

Table of Contents

Introduction

The number of digital assets on Ethereum over the past twelve months has increased aggressively as more and more use cases are implemented as smart contracts. It is our thesis that this trend will continue into the future; as such we believe this growth will augment the demand to swap into and out of assets as users move between use cases or rebalance their tokenized portfolios. Exchanges based on blockchain order books are not without inherent limitations, many of which can be mitigated by the design decisions outlined in this paper. We seek to provide an alternative to blockchain order books by specifying a set of protocols that unlock asset liquidity and free the Ethereum ecosystem to progress without such limitations.

Order Books

Order books offer a highly automatable way to match supply and demand of a given tradeable asset. Traditionally, these are centralized and are combined with order execution, which allows orders to be created, executed, and canceled at a central source of truth. In the spirit of decentralization, order books have been redesigned for blockchains. However, deploying an order book on a blockchain presents several constraints.

Blockchain order books do not scale. Executing code on a blockchain incurs a cost, so an automated order-cancel-order cycle quickly becomes expensive and defeats the strength of an order book as a high performance, automatable matching system. Indeed, if that matching algorithm is running on the blockchain, a party placing orders will incur an execution cost that increases substantially with the size of the order book.

Blockchain order books are public. Because the transaction to create an order on the blockchain is processed by miners, those miners are privy to an order before it’s posted to the book. This creates an opportunity for front-running that could materially affect the original order. Additionally, because the order is published publicly, the order price is the same for everyone, removing a supplier’s ability to tailor liquidity.

Blockchain order books are unfair. Physically distributed systems inherently suffer latency between their nodes. As miners are geographically distributed, sophisticated parties may be able to colocate, detect orders, and outperform blockchain latency, effectively acting on order information before other parties. This information asymmetry may very well dishearten less sophisticated parties from taking part in the ecosystem at all.

Peer-to-peer (P2P)

Alternatively, peer-to-peer trading enables individual parties to trade with each other directly. Most of the transactions we make day to day are peer-to-peer: buying coffee at a cafe, selling shoes on eBay, or buying cat food on Amazon. Because these are private transactions between people or businesses, each party knows and ultimately chooses with whom they transact.

Peer-to-peer trading scales. Orders are transmitted between individual parties and are “one and done” as opposed to orders on a public exchange with no guarantee to completely fill. This makes cancels on an order book a regular occurrence, whereas peer-to-peer orders are likely filled because they are provided to parties that have already expressed interest. Additionally, peer-to-peer supply and demand matching can be solved through lightweight peer discovery as opposed to expensive algorithmic matchmaking–regardless of whether on or off chain.

Peer-to-peer trading is private. Once two parties have found and chosen to trade with each other, no third parties are required to negotiate. The communication between these parties remains private for the duration of the negotiation, removing the opportunity for other parties to act on order request behavior. Only when the order is submitted to be filled will it become public knowledge.

Peer-to-peer trading is fair. Because orders are created and transmitted directly between two parties, no outside participants can have an advantage. As long as they are working with multiple independent parties, participants can get prices that are comparable to or better than what they would achieve on an exchange. Additionally, those pricing orders can do so aggressively without fear of being taken advantage of by automated, low-latency trading strategies.

The scalability, privacy, and fairness constraints imposed by blockchain order books have necessitated an alternative. Today’s Ethereum ecosystem needs an open peer-to-peer solution for asset exchange.

Introducing Swap

Swap is a protocol to facilitate a true peer-to-peer ecosystem for trading tokens on the Ethereum blockchain. The following is an extensible specification that supports efficient counterparty discovery and negotiations. These protocols are intended to become a foundation for the asset trading ecosystem and to accelerate Ethereum ecosystem growth. By publishing this paper and opening for discussion, we seek comments from ecosystem stakeholders with the aim to produce high-quality protocols to enable a wide variety of real-world applications.

Peer Protocol

With only a few messages passed between counterparties, trades can be negotiated quickly, fairly, and privately. For the purposes of this document, a Maker is the party that provides an order, and a Taker is the party that fills it. Because each party is a peer, any party can assume the role of Maker or Taker at any time. Tokens in the following specification are ERC20 compliant and any token that implements the standard can be traded using this protocol.

The core protocol is sequenced in the following diagram. The Maker and Taker perform trade negotiation off-chain. The Contract below is an Ethereum smart contract, which the Taker calls when ready to fill an order on the blockchain.

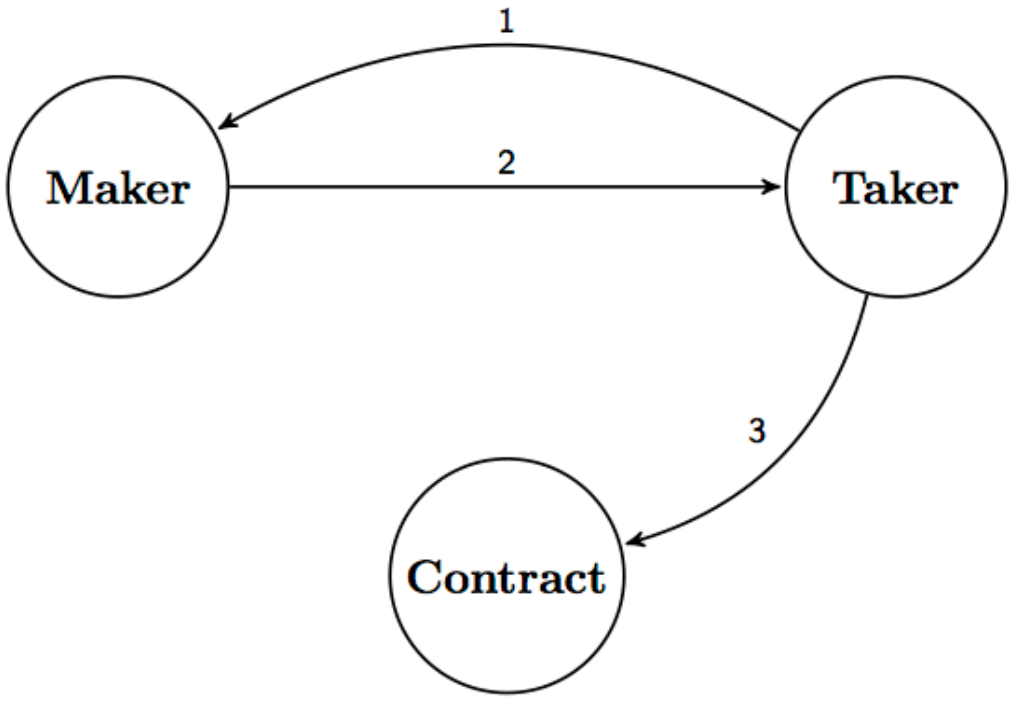

Figure 1: Request, provide, and fill an Order

1. Taker calls *getOrder* on the Maker.
2. Maker replies with an order.
3. Taker calls *fill([order])* on the Contract.

Order API

The following APIs are transport-agnostic remote procedure calls (RPC) used to communicate among peers and services. Examples use token tickers instead of addresses, but the actual calls require addresses of ERC20 compliant tokens. The call signatures below are for discussion purposes as further technical details are to be published in a separate document.

The Order API is off-chain and specifies asynchronous calls made between counterparties during trade negotiation. An implementor may choose to serve a request-provide cycle as a synchronous request-response. Because an order is signed by the Maker, the Taker is able to later submit it to the smart contract to be filled.

getOrder

getOrder(makerAmount, makerToken, takerToken, takerAddress)

Called by a Taker on a Maker, requesting an order to trade tokens.

```
Example: “I want to buy 10 GNO using BAT.”
getOrder(10, ‘GNO’, ‘BAT’, <takerAddress>)
```

provideOrder

provideOrder(makerAddress, makerAmount, makerToken, takerAddress, takerAmount, takerToken, expiration, nonce, signature)

Called by a Maker on a Taker, providing a signed order for execution.

```
Example: “I’ll sell you 10 GNO for 5 BAT.”
provideOrder(<makerAddress>, 10, ‘GNO’, <takerAddress>, 5, ‘BAT’, <expiration>,
<nonce>, <signature>)
```

Indexer Protocol

An Indexer is an off-chain service that aggregates and matches peers based on their intent to trade: whether prospective Makers and Takers wish to buy or sell tokens. Indexers are off-chain services that aggregate this “intent to trade” and help match peers based on intent to buy or sell specific tokens. Many prospective Makers can signal intent to trade, and when a Taker asks the Indexer to find suitable counterparties, there may be multiple results. Once the Taker has found a Maker with whom they would like to trade, they proceed to negotiate using the Peer Protocol above. Once agreement is reached between a Maker and Taker, the order is filled on the smart contract.

The interactions between a Maker, Taker, and Indexer are illustrated in the following diagram. The Maker, Taker, and Indexer all operate away from the blockchain and communicate by any preferred messaging medium.

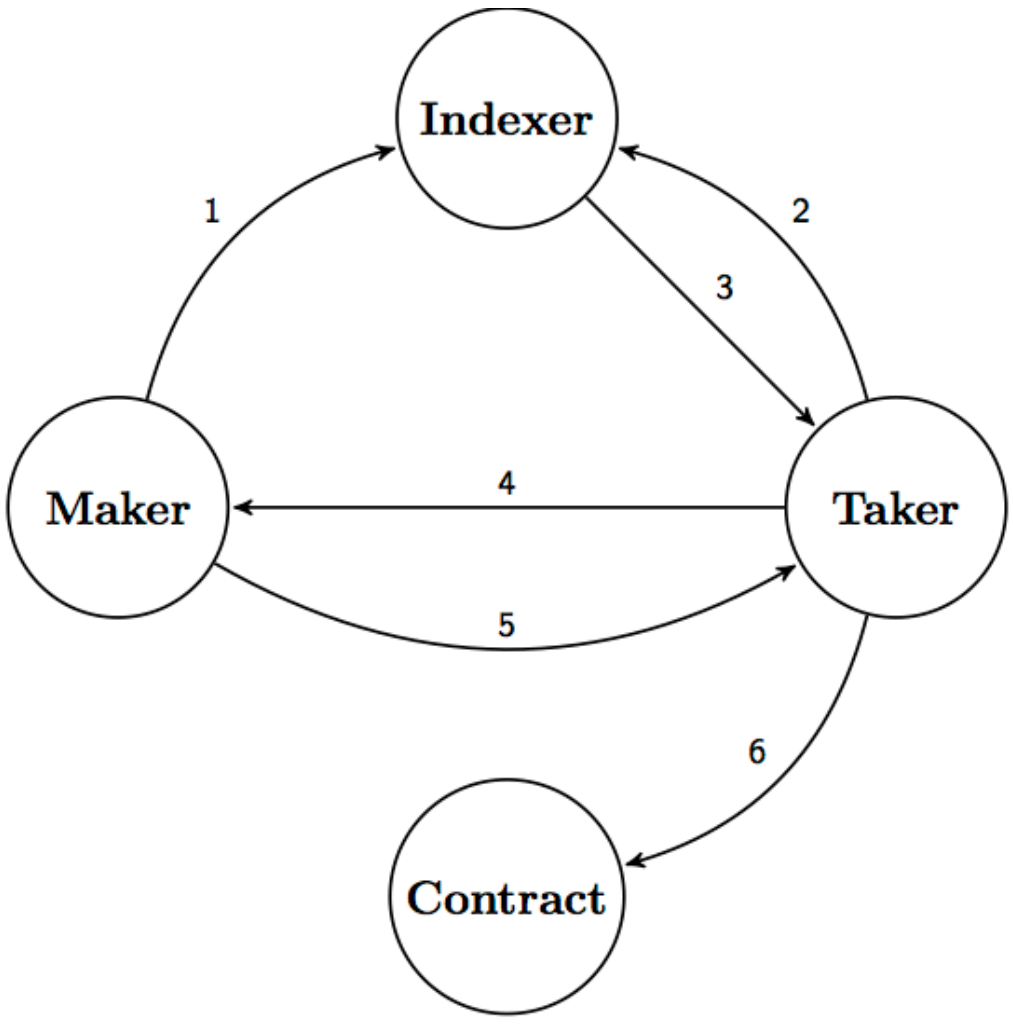

Figure 2: Find a counterparty and make a trade

1. Maker calls *addIntent* on the Indexer.
2. Taker calls *findIntent* on the Indexer.
3. Indexer calls *foundIntent([Maker])* on the Taker.
4. Taker calls *getOrder* on the Maker.
5. Maker replies with an order.
6. Taker calls *fill([order])* on the Contract.

The interaction between several Makers, a Taker, and an Indexer is illustrated in the following diagram. Each Maker independently announces their intent. The Taker asks to find Makers with specific intent, and the Indexer returns a list of Ethereum addresses and details.

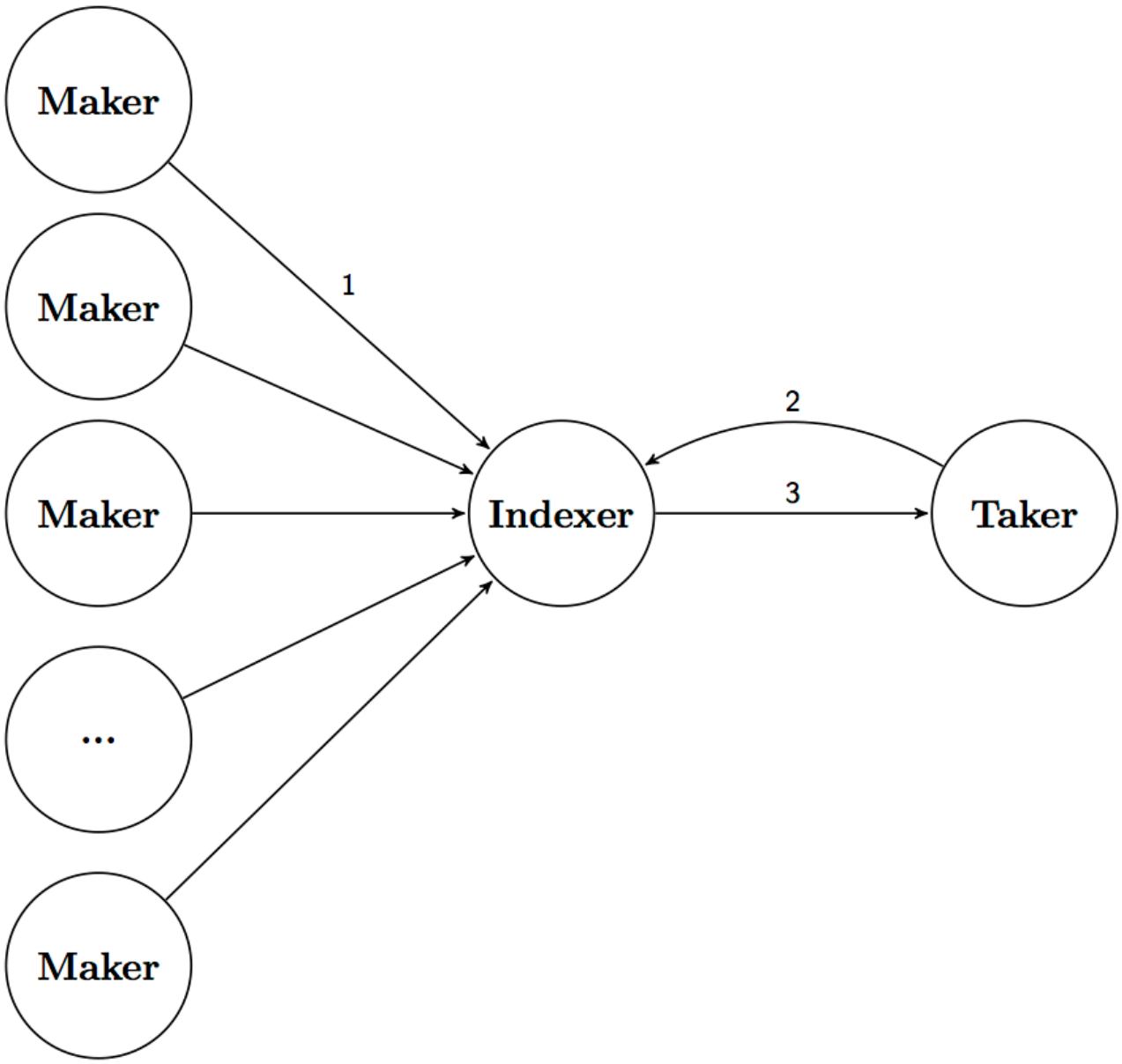

Figure 3: Makers call addIntent, a Taker calls findIntent on the Indexer

1. Several Makers call *addIntent* on the Indexer.
2. Taker calls *findIntent* on the Indexer.
3. Indexer calls *foundIntent([Maker])* on the Taker.

Once a Taker has found suitable Makers, they may use the Order API to request orders from each Maker to weigh them against each other. If the Taker has decided to fill a given order, they will make a fill call on the smart contract.

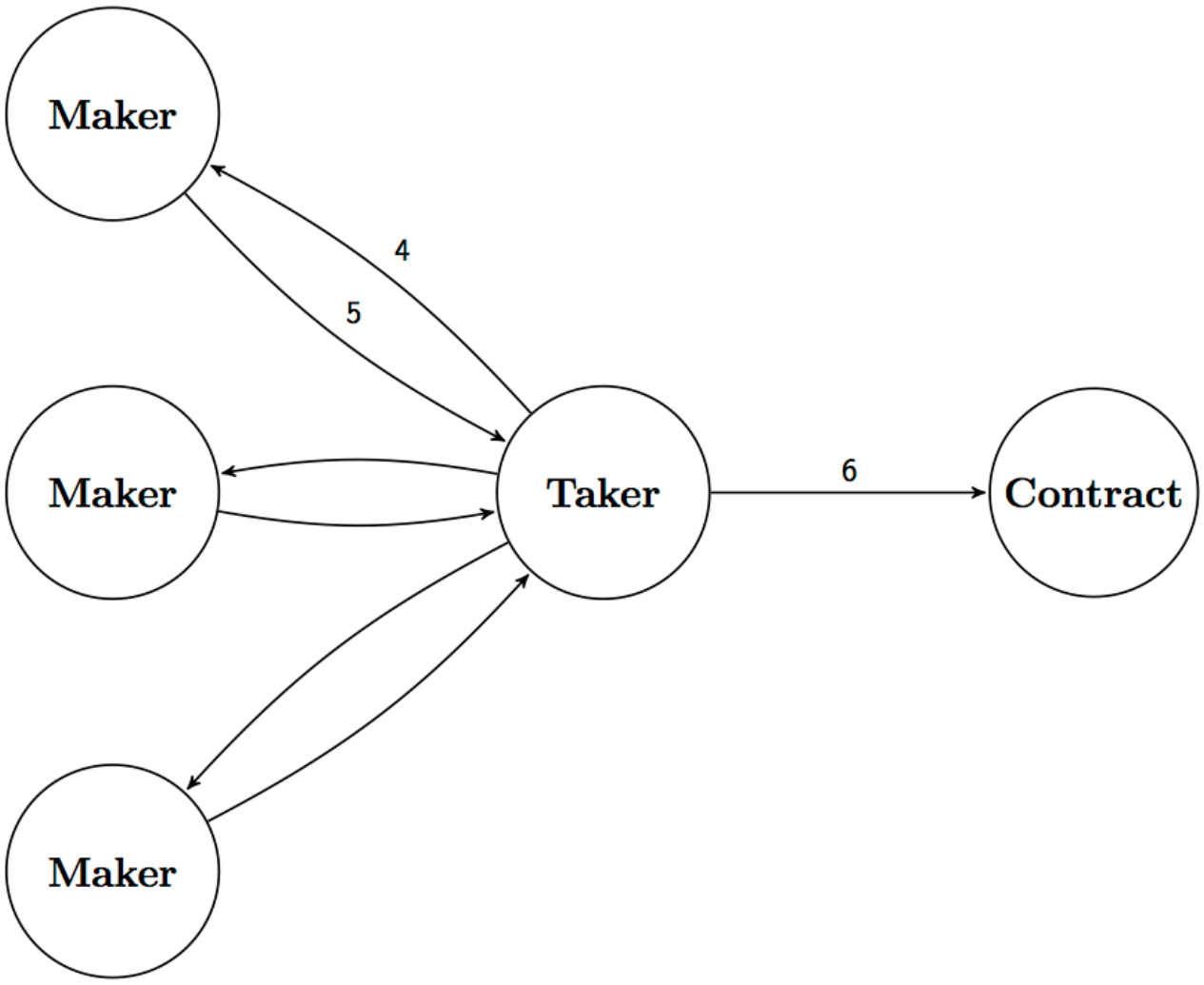

Figure 4: Taker calls getOrder on Makers, Taker calls fillOrder on Contract

1. Taker calls *getOrder* on several Makers.
2. Makers reply with orders.
3. Taker selects an order and calls *fill([order])* on the Contract.

Indexer API

The Indexer API manages intent to trade, which is signalled between peers. The following calls are made between peers and an Indexer.

addIntent

addIntent(makerToken, takerTokens)

Add an intent to buy or sell some amount of token.

```
Example: “I want to trade GNO for BAT.”
addIntent(‘GNO’, [‘BAT’])
```

removeIntent

removeIntent(makerToken, takerTokens)

Remove an intent to trade tokens.

```
Example: “I am no longer interested in trading GNO for BAT.”
removeIntent(‘GNO’, [‘BAT’])
```

getIntent

getIntent(makerAddress)

List active intent associated with an address.

```
Example: “List the tokens that [makerAddress] wants to trade.”
getIntent(<makerAddress>)
```

findIntent

findIntent(makerToken, takerToken)

Find someone willing to trade specific tokens.

```
Example: “Find someone trading GNO for BAT.”
findIntent(‘GNO’, ‘BAT’)
```

foundIntent

foundIntent(makerAddress, intentList)

The Indexer found someone with intent to trade.

```
Example: “Found someone selling 10 GNO for BAT.”
foundIntent(<makerAddress>, [{ makerAmount: 10, makerToken: 'GNO', takerTokens:
['BAT'] })
```

Oracle Protocol

An Oracle is an off-chain service that provides pricing information to Makers and Takers. When pricing an order prior to delivering it to a Taker, a Maker may ask the Oracle for what it considers a fair price suggestion. Likewise, having received an order, a Taker may ask the Oracle to check the price on the order to verify that it's fair. The Oracle provides this pricing information to help both the Maker and the Taker make more educated pricing decisions and to smooth the process of trade negotiation.

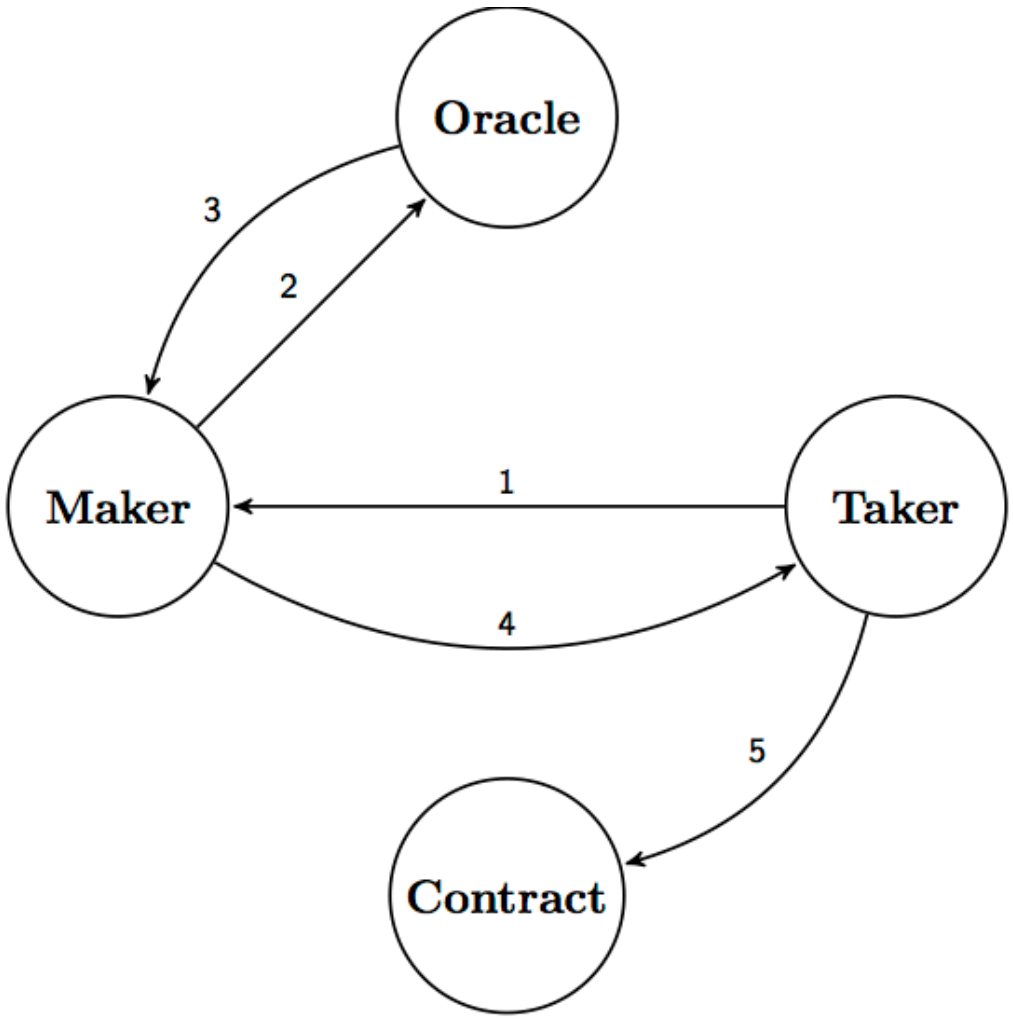

Figure 5: Maker querying Oracle before providing order

1. Taker calls *getOrder* on the Maker.
2. Maker calls *getPrice* on the Oracle.
3. Oracle returns a price to the Maker.
4. After analyzing price information, Maker replies with an order.
5. Taker calls *fill([order])* on the Contract.

A very similar interaction happens between Taker and Oracle when the Taker receives an order.

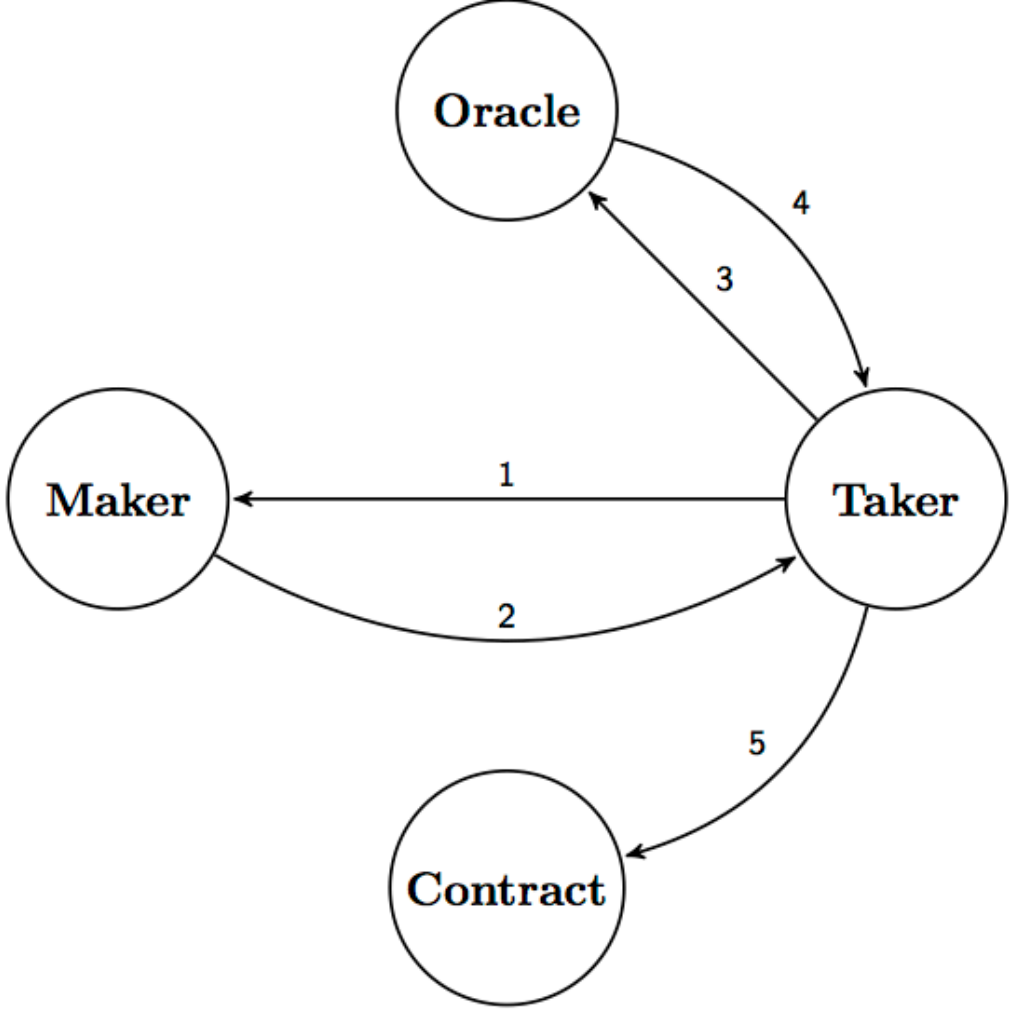

Figure 6: Taker querying Oracle before filling order

1. Taker calls *getOrder* on the Maker.
2. Maker replies with an order.
3. Taker calls *getPrice* on the Oracle.
4. Oracle returns a price to the Taker.
5. After analyzing price information, Taker calls *fill([order])* on the Contract.

Oracle API

The Oracle API is used by Makers and Takers to determine order prices. Prices are suggestions and are not executable.

getPrice

getPrice(makerToken, takerToken)

Called by a Taker or a Maker on an Oracle to get a price.

```
Example: “What is the current price of GNO for BAT?”
getPrice(‘GNO’, ‘BAT’)
```

providePrice

providePrice(makerToken, takerToken, price)

Called by an Oracle on a Maker or Taker to give a price.

```
Example: “The current price of GNO for BAT is 0.5.”
providePrice(‘GNO’, ‘BAT’, 0.5)
```

Smart Contract

An Ethereum smart contract to fill or cancel orders.

fill

fill(makerAddress, makerAmount, makerToken, takerAddress, takerAmount, takerToken, expiration, nonce, v, r, s)

An atomic swap of tokens called by a Taker. The contract ensures that the message sender matches taker and ensures that the time indicated in expiration has not passed. To fill orders, peers must have already called approve on the specified tokens to allow the contract to withdraw at least the specified amounts. For token transfers, the contract calls transferFrom on the respective tokens. At the successful completion of this function a Filled event is broadcast to the blockchain. The parameters `v`, `r`, and `s` constitute the maker signature.

```
Example: “I want to fill this order of 5 GNO for 10 BAT.”
fill(<makerAddress>, 5, 'GNO', <takerAddress>, 10, 'BAT', <expiration>,
<nonce>, <v>, <r>, <s>)
```

cancel

cancel(makerAddress, makerAmount, makerToken, takerAddress, takerAmount, takerToken, expiration, nonce, v, r, s)

A cancellation of an order that has already been communicated to a Taker but not yet filled. Called by the Maker of the order. Marks the order as already having been filled on the contract so a subsequent attempt to fill the order will fail. At the successful completion of this function a Canceled event is broadcast to the blockchain.

```
Example: “I want to cancel this order of 5 GNO for 10 BAT.”
cancel(<makerAddress>, 5, 'GNO', <takerAddress>, 10, 'BAT', <expiration>,
<nonce>, <v>, <r>, <s>)
```

Ether Orders

The smart contract supports trading ether (ETH) for tokens. If the order includes a null takerToken address (0x0) the smart contract will check the value of ether that was sent with the function call and transfer that on behalf of the Taker to the Maker.

Summary

The Swap protocol serves a growing demand for a decentralized asset exchange on the Ethereum network. Blockchain-based order books, while novel and certainly within the ethos of our ecosystem, have limitations that we believe ultimately make it difficult for them to compete with currently available centralized solutions. Swap provides a method that is both decentralized and unaffected by these limitations.

By implementing the protocol, participants gain access to liquidity in a scalable, private, and fair way, without sacrificing access to great pricing. The protocol and APIs are extensible and we encourage the community to build applications with us. We welcome feedback and look forward to pushing the Ethereum community forward with you.

For questions, comments, or feedback, please reach us at team@swap.tech.