Solidity Decentralized Orderbook v.1.0

July 14, 2021

Team Members: val314159 (Joel Ward) annebot (Anne Ahola Ward)

https://gitcoin.co/issue/SovrynBTC/sovrython-2021-bounties/1/100025809

https://github.com/val314159/decentralized-orderbook.git

Description

A Solidity-based stand-alone smart contract that maintains an order book between two tokens.

Both Market and Limit orders are supported.

Motivation

For the Sovryn hackathon, as a basis to use for a market.

Features

Decentralized: no centralized clearinghouse to freeze up, go bankrupt, or get hacked.

Accessible: Anyone can trade with no ID or KYC requirements.

Secure: As a smart contract, no one can hijack the operation of the internals.

Uses any ERC20 token pairs, so you could use something like the SOV (sovergeng) vs USDC (US Dollar coin)

- The thing you're trading is the INSTRUMENT.
- The thing you're trading against is the CURRENCY.

Terminology

Instrument: The thing you're trading is the INSTRUMENT. (it's literally a financial instrument)

Currency: The thing you're trading against is the CURRENCY. (yes all tokens may be currencies, but what you're naming the prices against)

Bid: a BUY order

Ask: a SELL order

Side: Either BID or ASK, depending on whether it's a buy or a sell order

Price: How much currency the order is denoted in

Size: the size of the order, denoted in the instrument

Maker: The order that is being filled against, and was already sitting in the book.

Taker: The order currently being filled against the taker. The way the matching engine works, pending orders are attempted to be filled immediately before putting them in the book. This has the confusing ramification of an order that is partially filled as a taker and partially a maker.

Matching engine: the part of the system that matches up pending orders with orders in the book so they can be filled.

Fill: actually performing the orders against each other and either adjusting them or removing them from the book.

Data Structures

The fundamental problem to be solved here is the fact that Solidity has a dearth of traditional data structures,

The contract brings together several well-known data structures in computer science:

- An Extended Red-black tree (of units) for keeping track of prices.
    - Since it's implemented with a map, testing for existence is practically free.

        - It's a balanced binary tree, so it's efficient
        - Red-black trees require less writes than traditional balanced trees, so it's a match for Solidity's built-in capabilities
        - Just searching the tree from a client is extremely easy
        - Additional operations were added to the Red-Black tree:
            - findLE(x): find numbers less than or equal to x
            - findGE(x): find numbers greater than or equal to x

- A map of prices (uints) -> Order Queues: each price level gets its own Queue or orders (for matching)

- Order Queues: FIFOs that keep track of Orders.
    - They're actually implemented using mappings, not arrays
    - They also keep track of the total volume, so we don't have to compute it if we need it.
    - There's actually also a map of ids -> order in each order queue for efficiency

- Orders: a struct that keeps track of all the things we need to keep track of and fulfill orders.

- Instrument Balances: keeps track of how many instruments each user has in the exchange

- Currency Balances: keeps track of how much currency each user has in the exchange

- Instrument Holds: keeps track of how many instruments are actively for sale (per user), therefore on hold, and can't be taken out of the market.

- Currency Holds: keeps track of how much currency each user has on hold because of active Bids. and therefore can't be taken out of the market.

External API

Functions:

bestPrice(side): get the best price for either BID or ASK. (for BIDs it will be the highest price, for ASKs it will be the lowest price). 0 means there aren't any orders.

createLimitOrder(side, size, price): create a limit order at a set price, if the book would end up crossed, start matching with the best price until it isn't.

createMarketOrder(side, size): create an order that will get immediately fulfilled no matter

getOrderID(side, price, address): get the ID by hashing all the info together, useful for looking up or cancelling orders.

cancelOrderId(side, price, id): find the order and cancel it. Only limit orders can be cancelled, as market orders are immediately filled and discarded if the book's empty.

Events:

In solidity, Events do much of the heavy lifting for the UI events. Here are the

CreateOrder(Order newOrder) - emitted whenever a new order is received.
A price of 0 means it's a market order.

FillOrder(uint size, Order maker, Order taker) - emitted whenever an order is filled.
Note that an order may get partially filled, this will result in multiple FillOrder events.
Whenever the size of an order becomes zero, this order is deleted. (having the client keep track of this is preferable to have a separate DeleteOrder event)

CancelOrder(orderId)

Future directions

- Eliminate front-running with something like sunken submarine calls.
- Expanding order types (stop-loss, kill-or-fill, etc)
- Making the matching engine pluggable.
- Supporting multiple instruments (supporting multiple currencies is probably best handled by just having more than one exchange)