Chapter 2: Deep Dive into OpenBook V1 on Solana

Introduction

Welcome to the second chapter of this series where we explore the universe of web3 in Rust, particularly focusing on the Solana blockchain. In this chapter, we will dive deeply into the inner workings and mechanics of OpenBook v1. If you have been following along, you may have noticed that I am working on a crate called `openbook`, which allows you to interact with any OpenBook v1 and v2 markets.

To start, let's explore some key terminologies and definitions, starting with the concept of a central limit order book (CLOB).

1. Central Limit Order Book (CLOB)

A central limit order book (CLOB) is a transparent system used by trading platforms to aggregate and match buy and sell orders. Prices and sizes can be used to represent an order book. Let's illustrate this with an example:

```
Size(Bid)  Price  Size(Ask)
            6      30
   20       5
```

In this setup, **asks** represent the prices at which people are willing to sell assets (such as WSOL or USDC), while **bids** represent the prices at which people are willing to buy. If people are placing bids at a price of 5, they are not willing to buy at higher prices, e.g., 6 or above. For instance, if there are 20 bids at price 5 and 30 asks at price 6, the best purchase price available in this order book is 5, and the best selling price is 6.

This is how price discovery is conducted via a central limit order book. The most significant challenge of implementing a CLOB on Solana or any web3 platform is that prices are very dynamic and change rapidly due to high trades volume in the market. For example, on blockchains with high gas costs, each action (such as placing a bid or canceling an order) requires gas fees, which can be detrimental. When the gap between bids and asks is narrow, order execution fees are lower, which is desirable compared to a wider gap.

Let's consider how CLOBs function in trading environments, such as OpenBook market, to understand their importance and challenges. A decentralized exchange (DEX) like OpenBook on the Solana blockchain uses a CLOB to enable peer-to-peer trading. Users can place their bids and asks directly on the blockchain, and the CLOB handles the matching process. OpenBook leverages Solana's high throughput and low latency to provide a seamless trading experience. By using a CLOB, OpenBook ensures that orders are matched fairly and transparently, promoting trust among traders. The decentralized nature of OpenBook's order

book means that no single entity controls the order matching process, enhancing security and reducing the risk of manipulation.

Size(Bid)	Price	Size(Ask)
500	25.0	300
400	24.5	200
600	24.0	100

In this example, the highest bid is for 500 tokens at `$25.0`, and the lowest ask is for 100 tokens at `$24.0`, illustrating the active trading levels on the DEX.

Implementing a CLOB on a blockchain like Solana offers numerous advantages, including transparency, security, and efficiency. However, it also comes with challenges such as handling high-frequency trades and ensuring low transaction costs. This example highlight the versatility and importance of CLOBs in various trading environments.

2. OpenBook V1 Accounts

OpenBook V1 requires several accounts to build this central limit order book. The essential accounts are listed as follows:

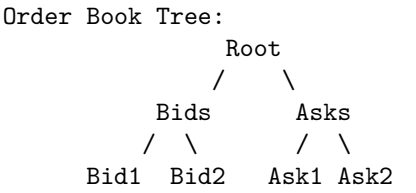
```rust
pub struct MarketState {
    // ...snip...
    pub own_address: [u64; 4],
    pub coin_mint: [u64; 4],
    pub pc_mint: [u64; 4],
    pub coin_vault: [u64; 4],
    pub pc_vault: [u64; 4],
    pub req_q: [u64; 4],
    pub event_q: [u64; 4],
    pub bids: [u64; 4],
    pub asks: [u64; 4],
    // ...snip...
}
```

Reference: openbook-dex/program

These are the main accounts representing a market, consisting of public keys in a slice format `[u64; 4]`. To understand these public keys, let's consider a `JLP/USDC` market (as seen on OpenSerum). In this case, the `coin_mint` or `base_mint` would be JLP, and the `pc_mint` or `quote_mint` would be USDC.

Now, let's examine each of these public keys closely.

2.1 Market Bids/Asks Bids and asks in a market are represented as order book trees. In these trees, nodes correspond to various orders, categorized into bids and asks based on whether they are buy or sell orders. This structure is crucial for organizing and searching through orders efficiently.

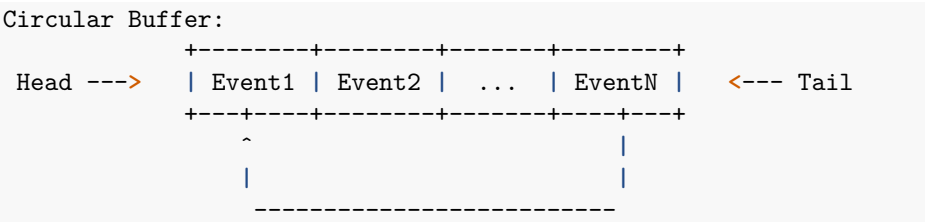


In this diagram, the root node splits into bids and asks, with each side further divided into individual bid and ask orders. This hierarchical structure allows for quick searching and matching of orders.

In traditional stock markets, the order book structure is used to manage and display bids and asks. For instance, if there are multiple bids at different prices for a stock, the order book tree helps in organizing these bids in a way that the highest bid is easily accessible for matching against the lowest ask.

On decentralized exchanges such as OpenBook, the order book also follows a hierarchical structure. This ensures that even in a decentralized setting, orders are matched efficiently and transparently. For instance, if a trader places a bid for 10 ETH at `$3,500` and another trader places an ask for 5 ETH at `$3,600`, the order book tree helps in matching these orders based on price and quantity.

2.2 Market Event/Request Queues Event and request queues are implemented as circular buffers (FIFO - First In, First Out). These queues handle the sequential processing of events and requests, ensuring that orders are executed in the order they were received. This is essential for maintaining the integrity and orderliness of the trading process.

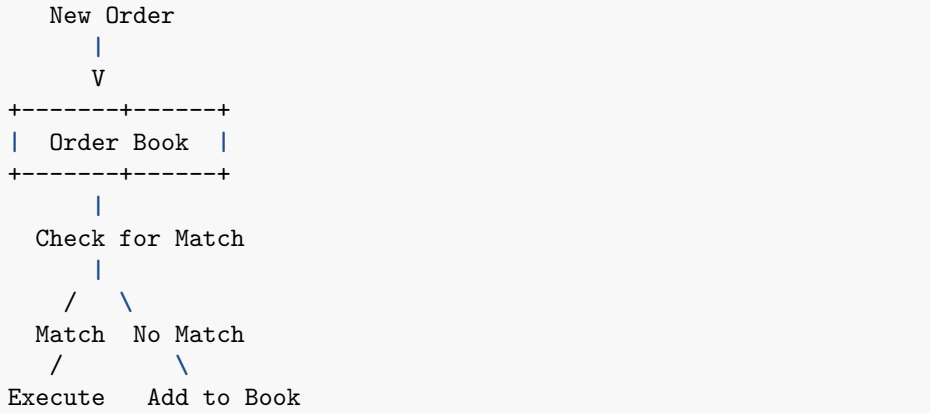


In this circular buffer, events are processed in the sequence they arrive, ensuring a fair and orderly execution of trades.

In high-frequency trading environments, event queues are crucial for processing large volumes of trades at high speeds. Platforms like **Virtu Financial** use sophisticated event queues to handle the influx of orders and ensure that each trade is executed in a timely manner.

On blockchains, transaction queues work similarly. For instance, in Ethereum, transactions are placed in a queue (the mempool) and processed in order. This ensures that transactions are executed fairly and according to the sequence they were submitted.

2.3 Placing Orders To place an order on OpenBook V1, three important parameters are required: **side (bid/ask)**, **price**, and **amount**. The order is then compared against existing orders in the order book to see if it can be matched. If a match is found, the order is executed; otherwise, it is added to the order book.



In this diagram, a new order is submitted to the order book, where it is either matched with an existing order or added to the book.

In stock trading, placing an order involves specifying the number of shares, the price, and whether it's a buy or sell order. For instance, if you place an order to buy 100 shares of Apple at `$210` each, the order book checks for existing sell orders at or below `$210`. If a match is found, the trade is executed; otherwise, the order remains in the book.

On OpenBook, placing an order to buy 50 WSOL at $150 involves checking the order book for matching sell orders. If a match is found, the trade occurs instantly; otherwise, the buy order waits in the order book.

```
New Order: Buy WSOL/USDC @ $150
      |
      V
+-------+------+
|  Order Book  |
+-------+------+
      |
  Check for Match
      |
    /   \
  Match  No Match
   /        \
Execute   Add to Book
```

Placing orders in OpenBook V1 is a critical process that ensures efficient trading. By checking for matches and executing trades or adding new orders to the book, the system maintains a dynamic and fluid market environment.

2.4 Open Orders Account When placing a new order on the OpenBook market, tokens must be transferred to OpenBook, such as USDC. Some tokens transition from a locked state to a freed state. The quantities of locked and free tokens change depending on the side of the order (bid or ask).

```rust
pub struct OpenOrdersAccount {
    // ...snip...
    pub locked_quantity: u64,
    pub free_quantity: u64,
    // ...snip...
}
```

```
Token States:
  +---------+           +---------+
  | Locked  |   ---->   |  Freed  |
  +---------+           +---------+
```

This diagram illustrates the transition of tokens from a locked state to a freed state during the order placement process.

In stock trading, when an order is placed, funds are held in a pending state until the trade is executed. Once executed, the funds are either used to purchase stocks or released back to the trader's account if the order is canceled.

In DeFi platforms like **Compound**, when placing a collateralized loan order, the collateral tokens are locked until the loan is either taken or the order is canceled. This ensures the security of the loan process.

2.5 Slab The slab object comprises a header and nodes, with two types of nodes: **inner nodes** and **leaf nodes**.

```
struct InnerNode {
    // ...snip...
    key: u128,
    children: [u32; 2],
    // ...snip...
}
```

Reference: openbook-dex/program

```
pub struct LeafNode {
    // ...snip...
    client_order_id: u64,
    key: u128,
    owner: [u64; 4],
    quantity: u64,
    // ...snip...
}
```

Reference: openbook-dex/program

This structure helps in efficiently managing and searching the order book.

```
Slab Structure:
    Header
     / \
  Inner  Leaf
    / \
 Leaf Leaf
```

Placing a new order involves comparing nodes and adding new leaf and inner nodes. The **root** becomes an inner node, and two leaf nodes are sorted.

In databases, slab-like structures are used for indexing. For instance, a **B-tree** in a database helps in efficiently searching and managing records. Similarly, the slab structure in an order book helps in quick order matching and retrieval.

Filesystems use hierarchical structures to manage files and directories. The slab structure in an order book is analogous to this, with inner nodes representing directories and leaf nodes representing files (orders).

In memory management, slab allocation involves dividing memory into slabs to efficiently manage free and used memory blocks. Similarly, the slab structure in an order book helps in efficiently managing orders.

2.6 Order ID Each LeafNode (order) contains a unique u64 order ID. This ID is essential for tracking and managing orders within the system.

```
pub struct LeafNode {
    // ...snip...
    pub client_order_id: u64,
```

```
    // ...snip...
}
```

On stock exchanges, each order is assigned a unique identifier. This helps in tracking the order through its lifecycle, from placement to execution or cancellation.

In ticketing systems, each ticket (issue or request) is assigned a unique ID. This helps in managing and tracking the ticket through its resolution process, analogous to order IDs in trading platforms.

In the Openbook v1 DEX, orders are handled with a focus on efficiency and clarity in their identification and management. The key aspect of this system involves the generation and management of order IDs, which ensures that bids and asks are sorted correctly and processed efficiently.

The following is the main function responsible for generating a new order ID:

```
impl RequestQueue<'_> {
    fn gen_order_id(&mut self, limit_price: u64, side: Side) -> u128 {
        let seq_num = self.gen_seq_num();
        let upper = (limit_price as u128) << 64;
        let lower = match side {
            Side::Bid => !seq_num,
            Side::Ask => seq_num,
        };
        upper | (lower as u128)
    }

    fn gen_seq_num(&mut self) -> u64 {
        let seq_num = self.header.next_seq_num;
        self.header.next_seq_num += 1;
        seq_num
    }
}
```

Reference: openbook-dex/program

The `gen_seq_num` function generates a new sequence number, which increments by 1 each time it's invoked. This sequence number is integral to creating a unique order ID. In the `gen_order_id` function, order IDs for asks are created by directly using the sequence number combined with the price through a bitwise OR operation. This method ensures that ask order IDs are sorted from low to high. The process for bid IDs involves flipping the bits of the sequence number and then combining it with the price using a bitwise OR operation. This bit manipulation ensures that bid order IDs are sorted from high to low.

To illustrate, let's consider a sequence number represented in binary as `111001`. Flipping the bits results in `000110`. The order ID is then generated by combining

this flipped sequence number with the price. This clever use of bitwise operations facilitates efficient searching and management of the order tree, making OpenBook a robust and efficient DEX.

To illustrate how this system works, consider the following examples:

1. **Generating an Ask Order ID**:
   - **Limit Price**: 50000
   - **Initial Sequence Number**: 111001 (binary: `1100100001001001111101101101111110101101111110`)
   - **Resulting Order ID**:
     - `upper` = `50000 << 64` = `92233720368547758080000`
     - `lower` = `111001` (binary: `11001000010010011111011011011111101011011111001`)
     - Final Order ID: `9223372036854775809110001`
2. **Generating a Bid Order ID**:
   - **Limit Price**: 50000
   - **Initial Sequence Number**: 111001 (binary: `1100100001001001111101101101111110101101111110`)
   - **Resulting Order ID**:
     - `upper` = `50000 << 64` = `92233720368547758080000`
     - `lower` = `!111001` (binary NOT: `11111111111111100100111001100110`)
     - Final Order ID: `9223372036854818756562940`

Let's consider a clear visual representation of how order IDs are generated for ask and bid orders by shifting the limit price, manipulating the sequence number, and combining them through bitwise operations:

```
1. Initial Variables
   +---------------------+------+
   | Limit Price (5)     |  101 |
   | Sequence Number (3) |  011 |
   +---------------------+------+
                         |
                         v
2. Shift Limit Price Left by 5 Bits
   +---+---+---+---+---+---+---+---+
   | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
   +---+---+---+---+---+---+---+---+
                         |
                         v
3. Bitwise NOT, bits flip on Sequence Number
   +--------------------------+---------------+
   | Original Sequence Number |  ... 0011     |
   | Bitwise NOT              |  ... 1100     |
   +--------------------------+---------------+
                         |
                         v
4. Combine Limit Price with Bitwise NOT of Sequence Number
   +---+---+---+---+---+---+---+---+---+---+
   | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
```

```
   +---+---+---+---+---+---+---+---+---+---+
   Order ID: [1][0][1][0][0][1][1][0][0]
```

The `gen_order_id` function, along with related mechanisms in OpenBook v1, provides a robust framework for order ID generation and management. By leveraging bitwise operations and a well-structured approach to sequence numbers, this system ensures efficient and reliable order processing. This methodology not only supports high throughput but also maintains the integrity and accuracy of the order book, essential for a decentralized exchange's successful operation.

3. OpenBook Crate

The following code snippet utilizes the **`openbook`** crate to fetch information about the current market using a given market ID, and the trader wallet information on this market.

```rust
use openbook::commitment_config::CommitmentConfig;
use openbook::v1::ob_client::OBClient;

let commitment = CommitmentConfig::confirmed();

let market_id = "8BnEgHoWFysVcuFFX7QztDmzuH8r5ZFvyP3sYwn1XTh6".parse().unwrap();
let mut ob_client = OBClient::new(commitment, market_id, true, 1000).await?;

ob_client
```

```
OB_V1_Client {
    owner: 4mPiqgbyyo7kFHA5yeWgUPaA4U2joGYvN8pLTwpuKBn8
    rpc_client: RpcClient { commitment: CommitmentConfig { commitment: Confirmed } }
    quote_ata: 5g1Dko9ERCqgA6uqZwkhSjH9AmaenWW5aYmmaxj4WvNR
    base_ata: FkK4ysFDHWG3K6T44GyqX4i3WcHzZUWqtSH7Erwfodju
    open_orders: OpenOrders {
        oo_key: 9A5UszFmPAN1Ss2xMPrzNEwxKxcL69w6NXW1zZuYRha3
        min_ask: 149000
        max_bid: 147500
        open_asks: []
        open_bids: []
        bids_address: 5jWUncPNBMZJ3sTHKmMLszypVkoRK6bfEQMQUHweeQnh
        asks_address: EaXdHx7x3mdGA38j5RSmKYSXMzAFzzUXCLNBEDXDn1d5
        open_asks_prices: []
        open_bids_prices: []
        base_total: 0.0
        quote_total: 0.0
    }

    market_info: Market {
        program_id: srmqPvymJeFKQ4zGQed1GFppgkRHL9kaELCbyksJtPX
```

```
        market_address: 8BnEgHoWFysVcuFFX7QztDmzuH8r5ZFvyP3sYwn1XTh6
        coin_decimals: 9
        pc_decimals: 6
        coin_lot_size: 1000000
        account_flags: 3
        pc_lot_size: 1
        quote_mint: EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v
        base_mint: So11111111111111111111111111111111111111112
        coin_vault: CKxTHwM9fPMRRvZmFnFoqKNd9pQR21c5Aq9bh5h9oghX
        pc_vault: 6A5NHCj1yF6urc9wZNe6Bcjj4LVszQNj5DwAWG97yzMu
        vault_signer_key: CTz5UMLQm2SRWHzQnU62Pi4yJqbNGjgRBHqqp6oDHfF7
        event_queue: 8CvwxZ9Db6XbLD46NZwwmVDZZRDy7eydFcAGkXKh9axa
        request_queue: CPjXDcggXckEq9e4QeXUieVJBpUNpLEmpihLpg5vWjGF
        bids_address: 5jWUncPNBMZJ3sTHKmMLszypVkoRK6bfEQMQUHweeQnh
        asks_address: EaXdHx7x3mdGA38j5RSmKYSXMzAFzzUXCLNBEDXDn1d5
        events_authority: 11111111111111111111111111111111
    }

}
```

You can verify the market information on **openserum.io**.

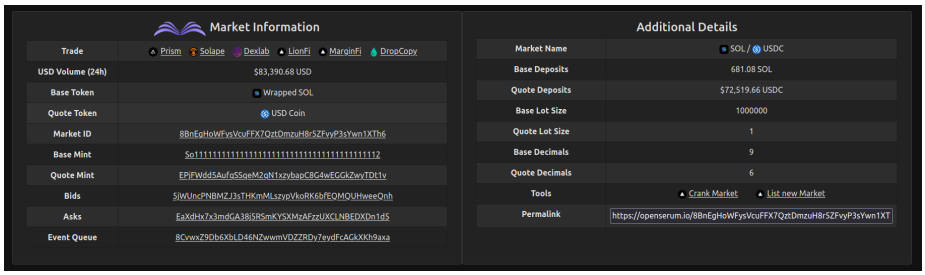


Figure 1: openserum

Conclusion

In summary, OpenBook V1 on Solana provides a robust framework for implementing a central limit order book, leveraging efficient data structures and mechanisms to ensure fair and efficient trading. The combination of market bids/asks, event/request queues, order placement, open orders accounts, slabs, and unique order IDs ensures that the trading system is transparent, secure, and scalable. We've barely scratched the surface here. In the upcoming chapter, we'll dive deeper into various key aspects of OpenBook V1.