Fusotao Greenbook(Draft)

v0.2.3

https://www.fusotao.org

UINB Tech

2021

1 Introduction

Fusotao is a set of network protocols which are composed of either a rule of data consistency or some certain constraints of data modification including not only a transfer constraint but also a matching verification rule. The matching verification sub-protocol, a.k.a Proof of Matching, is the main difference from other blockchains.

2 Matching System

Before start introducing Fusotao Protocol, let's take a look at the matching system and consider why a matcher's outputs should be verified.
Matching system is a trading platform that allows users to price orders. The core component of a matching system is a data structure named orderbook which stores all orders that are according to the price and time, and a placed order must follow the sequence to trade if price meets, while the owner of an order would be ignored. Usually, in order to trade on a matching system, users may hand over their account ownership so that the matcher can mutate the accounts, in another word, trading on matchers should reply on human trust.

Any data modifications should obey the rules below:

Rule 1 *Mutable data should not be shared, and shared data should be immutable.*

Rule 2 *There should be only one associated mutator once data shared.*

Compared to the transfer transaction, the matcher, as a mutator, must modify the orderbook's state for each order rather than just update the states of sender and receiver. Therefore, a matching system is a strict serial system which can be represented by the following procedure:

$$E_i + Orderbook_{i-1} \Rightarrow Orderbook_i + R_i$$

Consider an order placed. Let x be a price, y be an amount, (x_i, y_i) be the i_{th} maker by the order of price and time.

$$mb_i = -y_i$$

$$tb = \sum_i y_i$$

$$mq_i = -x_i \times y_i$$

$$tq = \sum_i x_i \times y_i$$

where

- $matches(x, x_i)$ is $true$

Blockchain is another typical serial system whose key rule is determining the order of all accepted incoming transactions. E.g. the Bitcoin network uses PoW algorithm and the Longest Chain Rule to ensure the global unique sequence of transactions. It seems natural to build a matching system on-chain just like a plain transfer function did. However, due to the limitation of block capacity and high latency, it is not straightforward to do it so.
Ideally, to implement a matching system on-chain, an order must occupy 73 bytes at least:

$$ID + Owner + Price + Unfilled + Direction = 73bytes$$

where

- $Price$ and $Unfilled$ are 128-bits fixed number
- ID indicates the order
- $Owner$ is the pubkey of user
- $Direction$ is 1-bit direction

Imagine there are tens of thousands of orders in a single orderbook, it is not acceptable to store all orders in the blockchain state machine, but only keeping some essential data to validate the matching results is possible. In the next section, we will introduce how to validate the matching results without holding the entire orderbook.

3 Global States

Sparse Merkle Tree is a full binary hash tree with fixed-depth which can be used for checking if there is a node belongs to a certain tree. A node of Sparse Merkle Tree can be represented by the following expressions:

$$\psi_x = \lambda(\psi_{xL}, \psi_{xR}),\ height \neq 0$$

$$\psi_x = v,\ height = 0$$

where

- λ is the hash function, e.g. sha256
- x is the key of a node calculated by $\lambda(value) >> height$, e.g. $0x00...a82e$

The capacity of a Sparse Merkle Tree depends on the hash function of the tree, e.g. a Sparse Merkle Tree using Sha256 has 2^{256} leaf nodes with height=256(root excluded). Given a data v, we can simply verify whether it belongs to a certain tree by calculating $height - 1$ times hash function:

```
let h = hash(v)
from 0 to 255:
    do h = hash(h, sibling_of_h)
return h == root
```

It is quite simple for validators, but it is not good for provers. Storing all 2^{257} (intermediate nodes included) nodes is unpractical for any storage system. Considering the distribution of a real Sparse Merkle Tree, most of the leaf nodes are empty, so are the intermediate nodes, that's why we call it sparse. In an empty plain Sparse Merkle Tree, a node at height h has a certain value:

$$\psi_h = \lambda^h(\phi, \phi)$$

To avoid pre-calculate hash for each height, we can redefine the hash function like below:

$$\lambda(\phi, \phi) = \phi$$

$$\lambda(\alpha, \phi) = \lambda(\phi, \alpha)$$

For data updates, once a leaf node is inserted, the nodes along the path would be updated until root. Since we have the first optimization, we can infer that there will be a mount of redundant nodes along the path which can be omitted. E.g. when inserting a single node n with key=0xff..ff, value=v into an empty Sparse Merkle Tree, the parent node of n whose key is 0x7f..ff would be value=v either, so are the rest of the nodes along the path. Thus, we can define the structure of nodes to store the entire tree into an available storage:

$$\Psi_k = (k, l, r, \lambda(\Psi_{k>>l}, \Psi_{k>>r}))$$

Back to the matching procedure, since we've done an optimized Sparse Merkle Tree so can encode the orderbook into it and only store the root hash on chain and leave the entire tree to matchers. Once a matcher executes the i_{th} event off-chain and

proves it by submitting some digests at $i-1_{th}$, a validator can simply verify the results like above.
Fusotao defines 4 types of key as the leaf keys of Sparse Merkle Tree:

$$orderbook(symbol) = \lambda(1, symbol)$$

$$account(currency) = \lambda(0, currency)$$

$$bestprice(symbol) = \lambda(2, symbol)$$

$$orderpage(symbol, price) = \lambda(3, symbol, price)$$

where

- $currency$ is a 32-bits unsigned number represented currency
- $symbol$ is consisted of $base$ currency and $quote$ currency
- $price$ is the index of orderpages sorted from best to worst

A matcher has a specific global state at the i_{th} event which can be verified by the leaf values:

$$orderbook_s = ask_s << 128 + bid_s$$

$$account_j = available_j << 128 + freezed_j$$

$$bestprice_s = askprice_s << 128 + bidprice_s$$

$$orderpage_{s,p} = vol$$

where

- s represents the $symbol$
- j represents the j_{th} account of current matching results
- p represents the $price$
- all numerics are 128-bits represented unsigned fixed number with $scale$=18

For simplicity, we can define $a \nabla b$ as $a << 128 + b$.

4 Proof of Matching

Given a user-signed event E_i and some merkle paths $P_{(i-1,j)}$ at $i-1$, a validator can verify it using a matching procedure with well-known root hash S_{i-1} stored on-chain:

$$\sum_{j=0}^{n} belongs(P_{(i-1,j)}, S_{i-1}) = n$$
$$P_{(i-1,j)} + E_i \Rightarrow S_i + P_{(i,j)}$$
$$\sum_{j=0}^{n} belongs(P_{(i,j)}, S_i) = n$$

A matcher must maintain a Sparse Merkle Tree and update it every time after the event is applied and generate an associated proof which is composed of the origin command and some merkle leaves including value before and after the i_{th} event executed.

Let (x, y) be an ask-limit order's price and amount of symbol (b/q) as the i_{th} event, (x_j, y_j) be the j_{th} maker exists at $i-1$. Then the proof generated by the matcher should be (Fee excluded):

$$account(b)_{(i,j)} = account(b)_{(i-1,j)} \nabla y_j$$
$$account(q)_{(i,j)} = account(q)_{(i-1,j)} - x_j \times y_j$$
$$account(b)_i = account(b)_{i-1} - (\sum_j y_j \nabla 0) + (y - \sum_j y_j)$$
$$account(q)_i = account(q)_{i-1} \nabla \sum_j x_j \times y_j$$
$$\sum_p^x orderpage(b,q)_{(i,p)} = \sum_p^x orderpage(b,q)_{(i-1,p)} + y - \sum_j y_j$$
$$orderbook(b,q)_i = orderbook(b,q)_{i-1} + (y - \sum_j y_j) \nabla 0 - \sum_j y_j$$

Once verified, the accounts can be mutated following the decoded account leaves and update the root hash stored on-chain to S_i.