DUO

Technical Whitepaper

FinBook Pte. Ltd. (token@finbook.co)

Version 0.3.0
25 Jul 2018

1. Introduction

In this paper, we lay out the detail design and implementation for the DUO structure described in our *Economic Whitepaper* and *Academic Whitepaper*.

2. Design and Structure

2.1 System and Platforms

The system consists of three major parts: smart contract that runs on Ethereum ("ETH") blockchain, price feed, and event triggers that run on cloud servers and web user interface.

Four smart contracts are deployed onto ETH blockchain, namely, DUO Network Token contract, Token A contract, Token B contract, and Beethoven contract. The first three contracts are standard ERC20 tokens, whereas the last one is the main contract that implements the DUO structure. Users can interact with these contracts directly or through our web app.

Three sets of price feed are setup independently on AWS, Azure, and GCP, running the same price fetching and aggregation algorithm. Beethoven then finds the consensus price from the three sources. Users can review the data sent to Beethoven by checking the blockchain directly or through our web app.

Triggers subscribe to the blockchain for events emitted by Beethoven, specifically **StartPreReset** and **StartReset** events. Once these events are emitted, the triggers will send new transaction to the blockchain to invoke relevant functions to progress the state transition.

2.2 State Transition

2.2.1 States

Inception: genesis state. No transit back to this state once it transits out. This state can only transit to **Trading** state.

Trading: normal state. the system is expected to be in this state most of the time. All functionality works in this state. This state can only transit to **PreReset** state.

PreReset: waiting state for the network to sync transactions before reset is started. Most of the smart contract functionalities are disabled in this state and effectively lock the transfer of A and B tokens. This state can transit to one of the three reset states below.

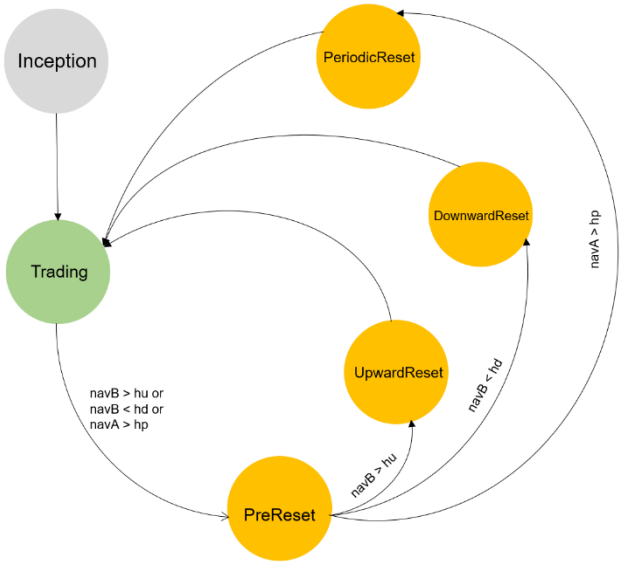
UpwardReset: upward reset is carried out in this state. This state can only transit to **Trading** state.

DownwardReset: downward reset is carried out in this state. This state can only transit to **Trading** state.

PeriodicReset: periodic reset is carried out in this state. This state can only transit to **Trading** state.

2.2.2 Transition

Below is the state transition diagram. When contract is in **Trading** state, it will receive ETH/USD price fix from a Price Feed process. Contract will transit to **PreReset** state when Net Asset Value ("NAV") of A or B hits predefined limits. PreReset trigger process will trigger **PreReset** state into one of the three reset states, **UpwardReset**, **PeriodicReset**, or **DownwardReset**. Reset trigger process will trigger contract transit from reset state to **Trading**.



2.3 Price Feed

2.3.1 Trade Fetch

Subscribe for trade information from exchange API of ETH/USD in Bitfinex, Kraken, CoinBase/GDAX, and Gemini. Persist id, price, volume and timestamp into database.

2.3.2 Price Aggregation

Traded prices are aggregated based on following algorithm:

1) For each exchange:

a. split 1 hour into 12 intervals of 5 minutes long

b. for the last 5-minute interval in the hour, take the volume based median price as the fix for this hour

c. if there is no trade in this 5-minute interval, use the previous interval that has trade

d. if there is no trade in the past hour, omit this exchange for overall fixing

2) For overall fixing:

a. average all valid exchange fixing for the hour weighted by the total trade volume of last 5 minutes, subject to weighting cap of 35% for 1st, 30% for 2nd, 25% for 3rd, 20% for 4th and 15% for 5th.

b. if none of the exchanges have valid fixing, use previous fix

2.3.3 Price Commit

Three price commit process are running on different platforms (AWS, Azure and GCP) to commit hourly ETH price to Beethoven. The purpose is to minimize the risk of the price commit system being hacked by attackers. If two servers are taken over by attackers and the third server still function as normal, it is still not possible for attackers to manipulate the price feed.

Price commits within cool down period from last commit are rejected/ignored.

All commits from the same sender as the first price are rejected if they are within cool down period from the first price.

The first price arrived is accepted immediately if the price is similar (5% difference) to the previous accepted price. Otherwise, smart contract will wait for the second price.

When the second price arrives beyond the cool down period after first price, the second price is accepted immediately if second sender is same as first sender, or first price is accepted if second sender is not the same as first sender.

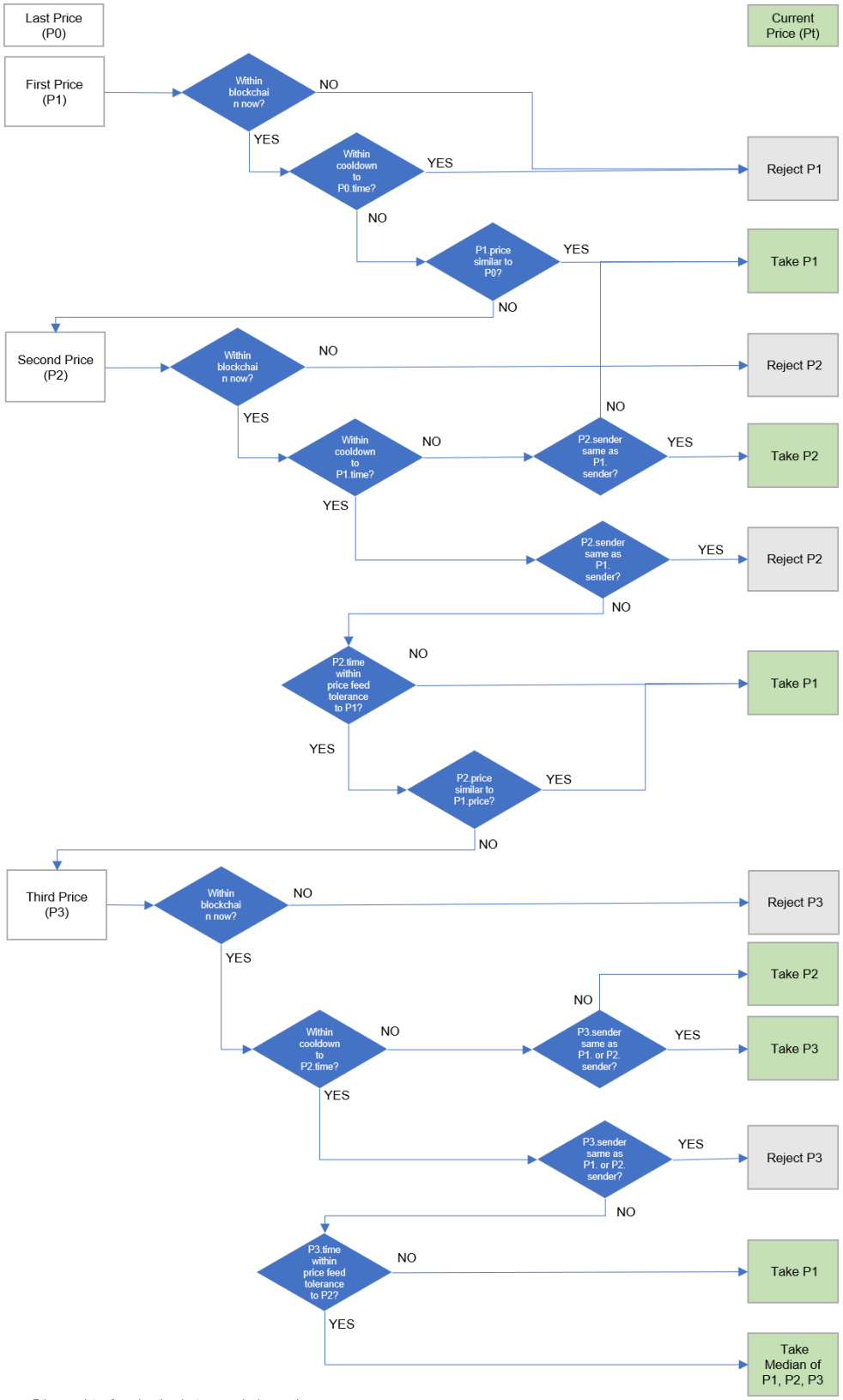
When the second price arrives within the cool down period after the first price and the second sender is different from the first sender, contract will compare the price arriving time between the two prices. First price is accepted if the second price arrives too late. If the second price's arriving time is within tolerance, it will further check the price difference. If the difference is within tolerance, the first price is accepted. If not, contract will wait for the third price.

All commits from the first or second sender are rejected if they are within cool down period from the first price.

When the third price arrives beyond the cool down period from the first price, it will further check the price sender. If the price sender is different from previous senders, the second price is accepted. If the price sender is same as any of the first two senders, the third price is accepted.

If the third price arrives within cool down period and sent by third sender, it will further check the arriving time difference to the first price. If the difference is too big, the first price will be accepted. If the difference is within tolerance, the median of the three prices are accepted.

Below is a graph showing the price selection logic.



Price consists of sender, time instance and price number
Two prices similar in both time instance and price number are considered similar prices
Cooldown period is set close to price update period

2.4 Resets

In *Academic Whitepaper*, users will receive ETH after any of the three resets as payout. It is assumed that users prefer to stay within the system, and thus convert ETH back to Token A and B, incurring a conversion fee. In our implementation, Beethoven just pays out the corresponding Token A and B to users after each reset, WITHOUT charging users any fee. The conversion logic is slightly different for the three resets to provide convenience for users.

Upward Reset: Token A holder will receive all the payout in Token A, while Token B holder will receive more Token B than Token A (instead of the 1:1 ratio). Effectively Token B holder exchanges part of their new Token A with Token A holder for their new Token B. Overall amount ratio of token A and B in the system remains 1:1 (Note that in current setting, i.e. A: B as 1:1, the new Token A from Token B holders can always cover the new Token B from Token A holders, after an upward reset). In this way, Token A holders can remain purely holding Token A after upward reset and do not require any rebalancing transactions.

Downward Reset and Periodic Reset: both Token A and Token B holder receives new Token A and B in 1:1 ratio, same as standard creation, but without conversion fee.

2.5 User Address and Balance Threshold

In theory, every user address that has any Token A or B (or both) balance should be processed in a Reset event. Thus, for every Creation and Transfer, the user address is recorded in an array to be used in the Reset process. When Reset happens, Beethoven loops through this array and adjusts Token A and Token B balance of each address. It costs about 20,000 gas to process one user address and given the current gas limit of 8 million gas for one message, we can roughly process 400 addresses in one transaction.

There can be a lot of addresses with negligible balances of Token A and B for various reasons, such as redemption or transfer to other accounts. Due to concerns of temporal and economical inefficiency, we have set a minimum balance of Token A and B for an address to be qualified for Reset processing. The threshold is set as **0.01** for both Token A and Token B, for the following reasons:

1) 0.01 Token A or Token B is expected to be worth roughly 0.01 USD, which is lower than the gas consumption of any ERC-20 operation (20,000 gas at 2 GWei with ETH at 500 USD equals 0.02 USD). We can safely assume such accounts have been abandoned for all practical purposes;

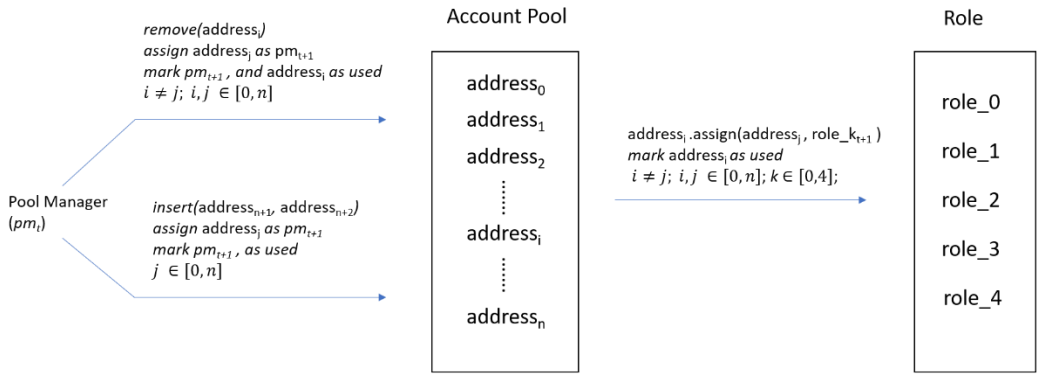
2) if someone created a huge number of addresses with Token B balance below 0.01 to intentionally avoid Reset event, it would not be profitable because gas consumption to create such addresses would cost more than these Token B's worth;

3) even if the addresses described in 2) are created and escape the balance adjustment of a downward reset, it is not economical for the address owner to withdraw these tokens, because the gas consumption would cost more than the worth of these Token B, even after downward reset.

Despite of the mathematical imperfection in the total supply of Token A and B, we skip these addresses so Beethoven can process Reset event in a much more practical and efficient manner without any real impact to the Token economy. Beethoven checks if **both** Token A and B balances of an address are below the threshold in every Redemption or Transfer. If so, the address is removed from the array of user addresses to be processed in a Reset event.

2.6 Administration Roles

There are six admin roles in our contract admin system. They are price feed 1 (pf1), price feed 2 (pf2), price feed 3 (pf3), fee collector (fc), operator (opt) and pool manager (pm). Price feed role can send price information to Beethoven. There are 3 addresses in this role at any given time, one for each cloud platform the service runs. fc is the one that collects all transaction fees incurred during creation and redemption. Operator in this role can update certain parameters in Beethoven. To safeguard Beethoven, we designed the admin system in a way that no single account can alter the contract and benefit from it. The admin system is described as below.



A list of ETH addresses in Beethoven can be assigned to different administration roles. There are minimum of 3 addresses in the pool at any given time. Any address can only assume one role at a given time. Once an address is assigned a role, it is removed from the address pool and marked as used so that it cannot be added back to the pool again.

In contract inception, 6 candidate accounts and one pool manager account are created. Pool manager can add two new account candidates into the pool. Prior to addition, the pool manager account is set randomly from the account pool. At the same time, new pool manager is removed from the pool. Pool manager is not allowed to add used accounts. The two accounts added must be different. Pool manager can also remove one account each time from the pool. Similarly, the pool manager is replaced randomly before candidate removal.

Any account in the pool can be an assignor. To assign roles, an assignee will be randomly select from the pool except the assignor itself. Therefore, even when one assignor's private key is exposed, the private key owner cannot benefit from assigning other accounts. Only when two accounts' private keys are owned by one person, there could be a possibility that the new role is assigned to one of that owner's addresses. However, since the assigning process is random, the probability is considerably low. This probability is further reduced if the pool size is large. After new role assignment, both the assignor address and the new role address are removed from the pool list.

Similar to price commit process, there is also an admin cool down window. The cool down window is set as one day. It means any admin operation can only be executed one day after last admin operation.

2.7 Security Analysis

2.7.1 Smart Contract

SafeMath: overflow and underflow are checked throughout the contract for all numerical calculations except for looping indices. Order of math operations are also checked to ensure intermediate results do not exceed the limit of 256-bit integer.

Function Restriction: all important functions in Beethoven are restricted by the caller and the state Beethoven is in. This prevents unwanted function invocations.

Events: every time a system parameter or state is changed, a related event is emitted so that everyone on the blockchain can monitor the change made to Beethoven.

2.7.2 Price Feed

Aggregation: timeliness and relevance take precedence over smoothness and manipulation resistance. Unlike futures settlement price in financial markets, this price feed does not always trigger a reset event. Futures market have bigger size than the underlying due to leverage effect, encouraging participants to manipulate the underlying market to impact future contracts settlement. On the contrary, the combined Token A and Token B could never be bigger than the underlying ETH market. Therefore, there is no incentive to manipulate the underlying ETH market to impact reset prices.

Redundancy: three different cloud server providers are used.

Consensus: rigorous algorithm to find the consensus price fixing from three price feeds.

2.7.3 Admin System Random Number Generation

In our admin system, after each admin operation, new admin account needs to be set randomly. However, the Ethereum EVM does not provide opcode for random number generation. The naive way of generating random number is using *block.timestamp*. However, *block.timestamp* can be arbitrarily altered by miner. A safer way is to use previous block's block hash because previous hash cannot be altered by current block miner. However, previous block hash is publicly known although the block interval is only 15 seconds. We need something that keeps varying and cannot be altered. A smarter idea is to use our user list. The user list length keeps changing and the user address cannot be changed arbitrarily. We combine block hash (from blockchain) and user list (from our product) to generate a random number.

The actual implementation is as following,

1) Take previous block's block hash and convert the hash into a number, a
2) If user list length, n is less than or equal to 255, a as the random number
3) If n is larger than 255, calculate $b = a \% n$, and convert the b-th user's address, *addr(b)* to number as the random number generated

3. Smart Contracts

3.1 DUO Network Token Contract

Standard ERC20 Token with 18 decimal places.

3.1.1 Functions

balanceOf: returns the balance of token for a given address

allowance: returns the allowance of token for a given spender authorized by an owner

transfer: transfer token to a given address

transferFrom: transfer authorized token to a given address

approve: authorize a given address to transfer token

totalSupply: returns the total number of token

3.1.2 Events

Transfer: emit after a transfer is done

Approval: emit after an allowance is approved

3.2 Token A Contract

ERC20 Token wrapper with 18 decimal places. The ERC20 functions listed in 3.1.1 are redirected to Beethoven, which implements the actual logic for Token A. Events listed in 3.1.2 works as standard.

3.3 Token B Contract

ERC20 Token wrapper with 18 decimal places. The ERC20 functions listed from 3.1.1 are redirected to Beethoven, which implements the actual logic for Token B. Events listed in 3.1.2 works as standard.

3.4 Beethoven (Custodian) Contract

Main contract implementing the DUO structure.

3.4.1 Price Feed only functions

startContract: start the contract by giving it an initial price and move state to Trading. Callable only in **Inception** state. Emit **StartTrading** event and **AcceptPrice** event.

commitPrice: accept price from external price feed and find consensus price among them. Callable only in **Trading** state. Emit **CommitPrice** event if a price is staged but not accepted. Emit **AcceptPrice** event if a price is accepted. Also checks if reset is required given the newly accepted price and emit **StartPreReset** event in such cases.

3.4.2 General user functions

create: create Token A and B by sending ETH to the contract and emit **Create** event and **TotalSupply** event. Callable only in **Trading** state. User can choose to pay conversion fee in ETH or DUO Network token.

redeem: merge Token A and B back to ETH and withdraw ETH to caller's address. Emit **Redeem** event and **TotalSupply** event. Callable only in **Trading** state. User can choose to pay conversion fee in ETH or DUO Network token.

calculateNav: calculate NAV for A and B with given prices based on current system parameters.

startPreReset: prepare the contract for reset. If the waiting period has reached predefined level, move the system to one of the three reset states and emit **StartReset** event and **TotalSupply** event in such cases. Otherwise a new **StartPreReset** event would be emitted to continue the wait. Callable only in **PreReset** state.

startReset: execute the reset for all users based on the gas available for this transaction. If gas is not enough for remaining users, **StartReset** event is emitted so that the next transaction can continue the reset process. If all users are processed, **StartTrading** event is emitted. Callable only in **UpwardReset**, **DownwardReset** or **PeriodicReset** state.

getSystemAddresses: return the contract address of token A and B and the current address for the 6 roles: operator, fee collector, price feed 1, price feed 2, price feed 3 and pool manager.

getSystemStates: return the current value for a list of system states or parameters. Please refer to the contract for the list.

getSystemPrices: return the value, timestamp and address of the two staging prices, last reset price and latest accepted price.

3.4.3 Token A & B functions

balanceOf: returns the balance of token A or B for a given address.

allowance: returns the allowance of token A or B for a given spender authorized by an owner.

transfer: transfer token A or B to a given address. Callable only in **Trading** state.

transferFrom: transfer authorized token A or B to a given address. Callable only in **Trading** state.

approve: authorize a given address to transfer token A or B.

totalSupply: returns the total number of token A or B.

3.4.4 Admin only functions

collectFee: withdraw ETH fee from the contract and emit **CollectFee** event. Callable only by fee collector in **Trading** State. Fees paid in DUO Network tokens are not

collectable and will remain in the contract address forever. These DUO Network tokens are effectively **burnt** as there is no way to transfer them out of the contract address.

setValue: set system parameter value based on a predefined index and emit **SetValue** event. Please refer to the contract for the list. Callable only by operator during update window.

addAddress: add two new addresses into the pool and assign a new address to be the pool manager. Emit **AddAddress** event. Callable only by pool manager during update window.

removeAddress: remove one address from the pool and assign a new address to be the pool manager. Emit **RemoveAddress** event. Callable only by pool manager during update window.

updateAddress: assign a new address to be the target role and remove the calling address from the pool. Emit **UpdateAddress** event. Callable only by addresses in the address pool during update window.

3.4.5 State Events

StartTrading: emit after state is changed to **Trading**.

StartPreReset: emit after state is changed to **PreReset** or during the waiting period before the state is changed to an actual reset state.

StartReset: emit after state is changed to **UpwardReset**, **DownwardReset**, or **PeriodicReset** or while resets are being processed before the state is changed to **Trading**.

CommitPrice: emit after a price is staged.

AcceptPrice: emit after a staged price is accepted.

Create: emit after a creation is performed.

Redeem: emit after a redemption is performed.

TotalSupply: emit after total supply for token A and B is changed

3.4.6 Admin Events

AddAddress: emit after address are added to the pool.

UpdateAddress: emit after an address from the pool is allocated to a role.

RemoveAddress: emit after an address is removed from the pool.

SetValue: emit after a system parameter is updated.

CollectFee: emit after ETH received for fee is collected.

3.4.7 Token Events

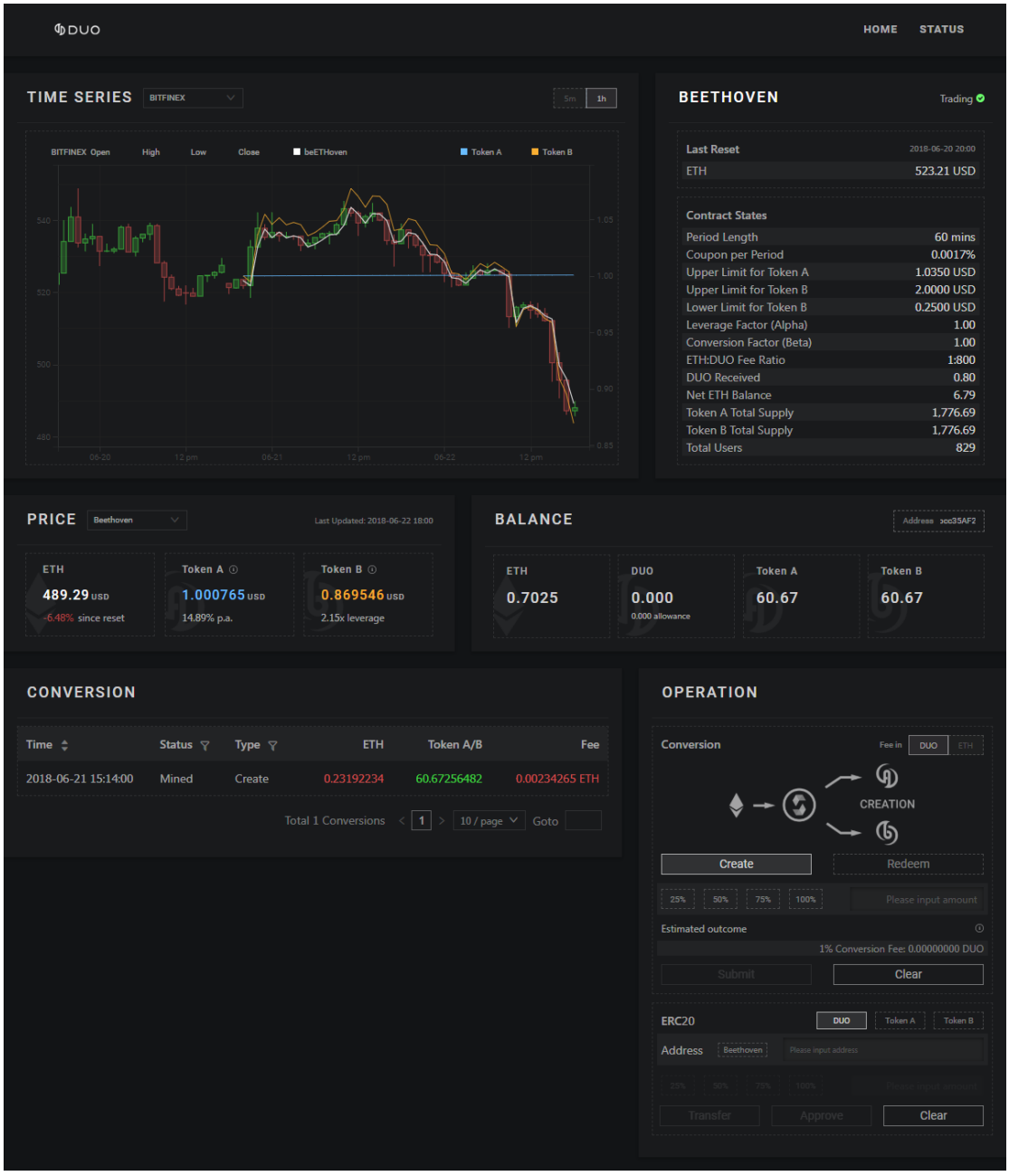
Transfer: emit after a transfer for either A or B is done

Approval: emit after an allowance for either A or B is approved

3.4.8 Gas Consumption

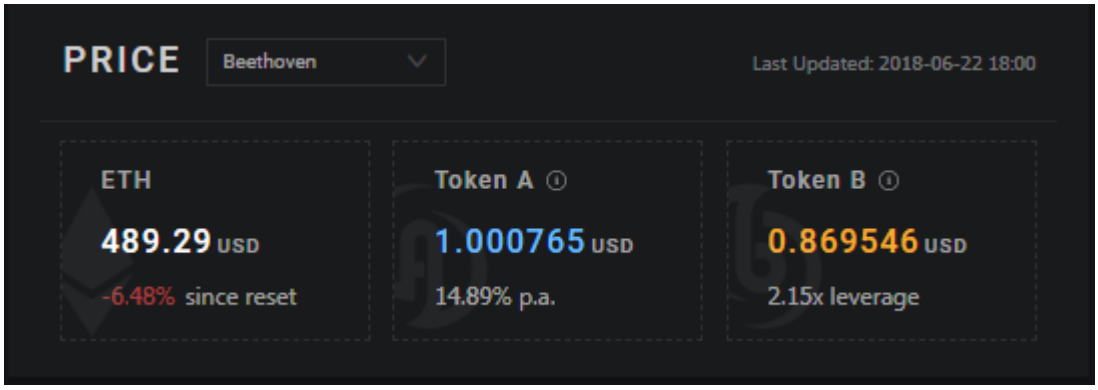
Method	Recommended gas
first time to create	160,000
create	90,000
redeem	90,000
approve DUO	50,000
transfer A/B	100,000
transfer A/B to new user	120,000
transfer DUO	60,000

4. Web User Interface

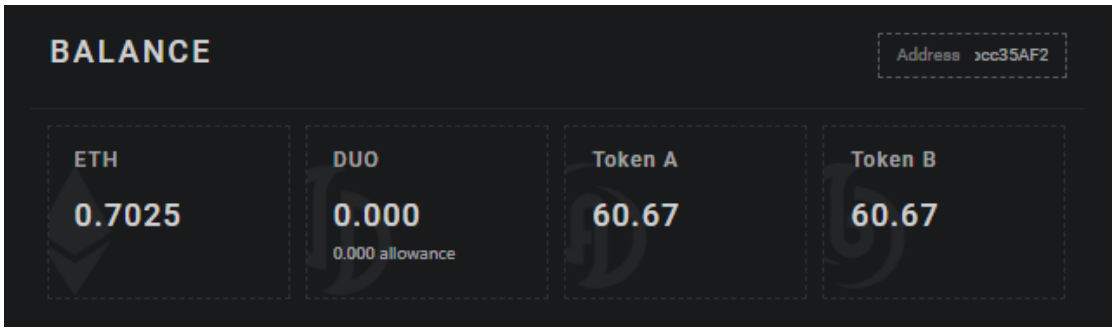


4.1 Historical Prices and Fixing

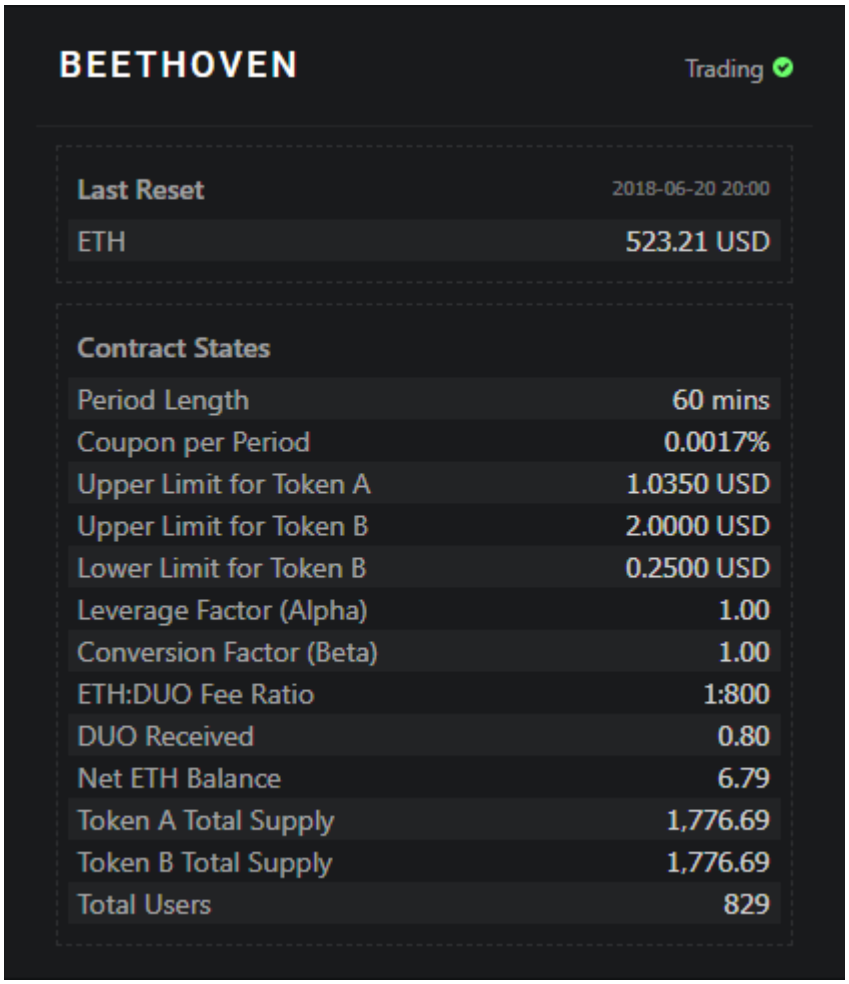
4.2 Current Price



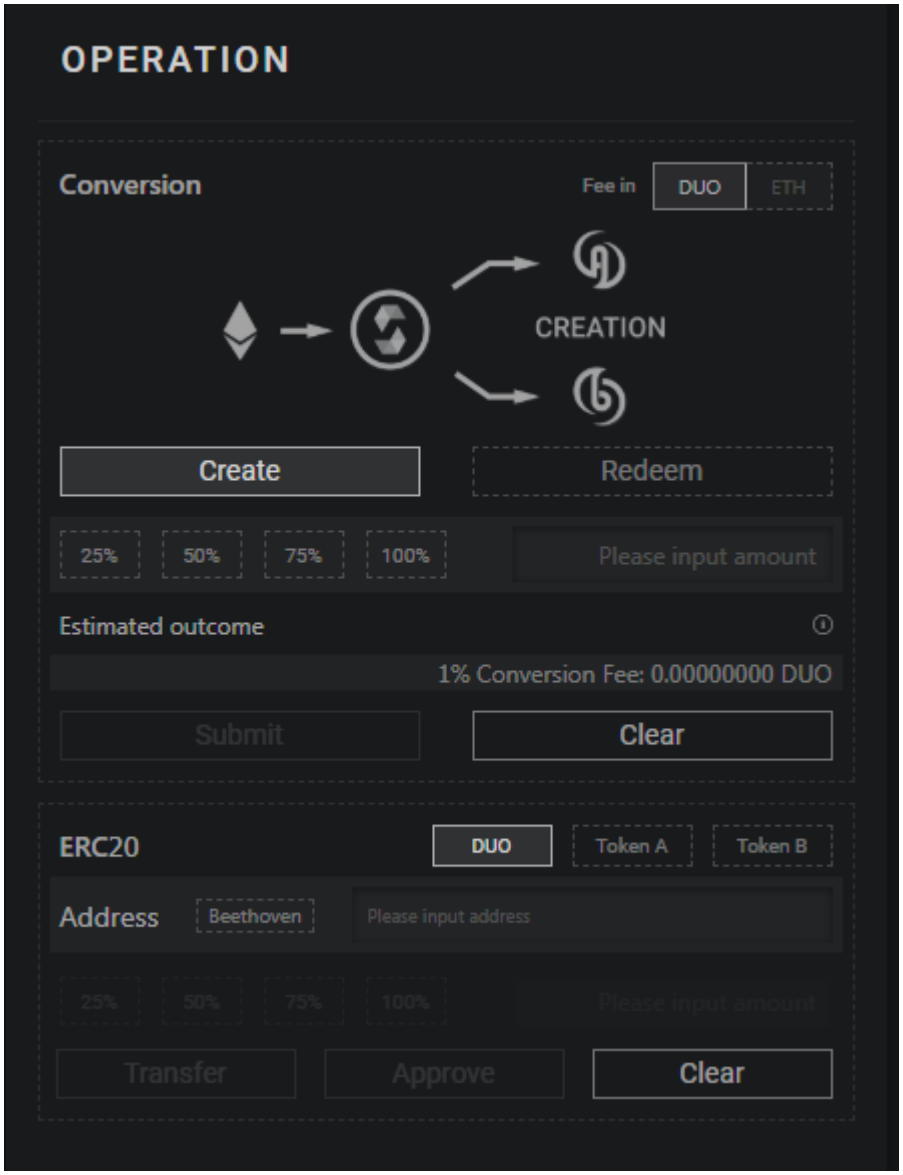
4.3 Current Balance



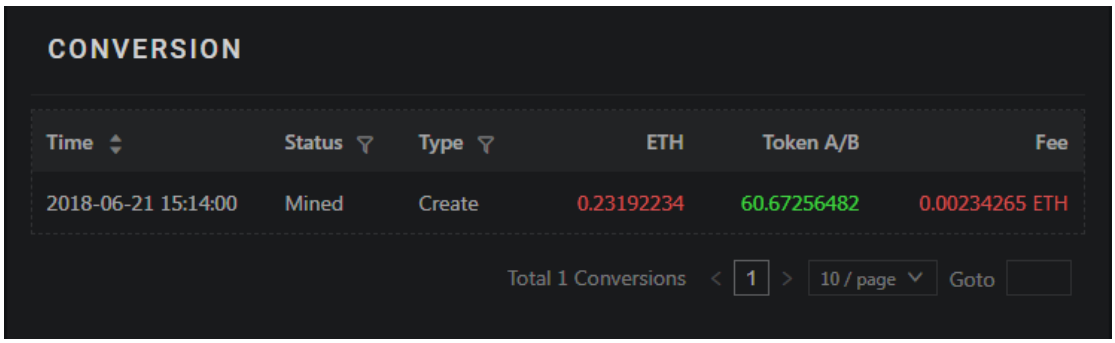
4.3 System Information and Status



4.4 Creation, Redemption and ERC20 Operations



4.4 Recent Conversion History



4.5 System Process Status

PROCESS STATUS

Process	Updated	Price	Volume	Block
CHAIN_AWS_PRIVATE	Just Now			7723568
EVENT_AWS_PRIVATE_STARTPRERESET	Just Now			
EVENT_AWS_PRIVATE_STARTRESET	Just Now			
EVENT_AWS_PUBLIC_OTHERS	8 Minutes Ago			
EVENT_AZURE_PRIVATE_STARTPRERESET	Just Now			
EVENT_AZURE_PRIVATE_STARTRESET	Just Now			
EVENT_GCP_PRIVATE_STARTPRERESET	Just Now			
EVENT_GCP_PRIVATE_STARTRESET	Just Now			
FEED_AWS_PRIVATE	Just Now			
FEED_AZURE_PRIVATE	Just Now			
FEED_GCP_PRIVATE	Just Now			
HOURLY_AWS_PUBLIC	Just Now			
MINUTELY_AWS_PUBLIC	Just Now			
PRICE_AWS_PRIVATE_BITFINEX	Just Now	516.06	2.29919131	
PRICE_AWS_PRIVATE_GDAX	Just Now	516.24	3.27131666	
PRICE_AWS_PRIVATE_GEMINI	2 Minutes Ago	516.81	0.683413	
PRICE_AWS_PRIVATE_KRAKEN	Just Now	514.97	2.09151245	
PRICE_AWS_PUBLIC_BITFINEX	Just Now	516.06	2.29919131	
PRICE_AWS_PUBLIC_GDAX	Just Now	516.24	3.27131666	
PRICE_AWS_PUBLIC_GEMINI	2 Minutes Ago	516.81	0.683413	
PRICE_AWS_PUBLIC_KRAKEN	Just Now	515	3.5080059	
PRICE_AZURE_PRIVATE_BITFINEX	Just Now	516.06	2.29919131	
PRICE_AZURE_PRIVATE_GDAX	Just Now	516.24	3.27131666	
PRICE_AZURE_PRIVATE_GEMINI	2 Minutes Ago	516.81	0.683413	
PRICE_AZURE_PRIVATE_KRAKEN	Just Now	515	3.5080059	
PRICE_GCP_PRIVATE_BITFINEX	Just Now	516.06	2.29919131	
PRICE_GCP_PRIVATE_GDAX	Just Now	516.24	3.27131666	
PRICE_GCP_PRIVATE_GEMINI	2 Minutes Ago	516.81	0.683413	
PRICE_GCP_PRIVATE_KRAKEN	Just Now	514.97	2.09151245	