YOSEMITE

On/Off-Chain Hybrid Exchange System

Technical White Paper

Bezalel Lim / bezalel@yosemiteX.com   Patrick O'Grady / patrick.ogrady@yosemiteX.com

Anthony Di Franco   Joe Park   Brandon Griffin
Ethan Kim   Eugene Chung   Eric Hwang   Elizabeth Reichert

31 July 2017

Abstract

The YOSEMITE On/Off-Chain Hybrid Exchange technology is designed to implement a highly performant, secure, fully transparent and user-friendly exchange system for trading Asset-Share tokens built upon the Ethereum blockchain, IPFS peer-to-peer storage and traditional server side technology. Our hybrid system, an on/off-chain solution, overcomes the critical drawbacks of a blockchain-based decentralized exchange system while still adopting the efficient and performant processing capability of a centralized exchange system. With the hybrid model, users do not directly handle a blockchain transaction fee (gas fees). Users can trade Asset-Share tokens using an easily purchasable fiat-pegged stable coin, Digital USD (dUSD) instead of Ether (Ethereum's native currency), which is overly difficult to obtain for most users and is also highly volatile. dUSD is an Ethereum-based token designed for YOSEMITE but can be utilized on other decentralized applications. Our downloadable multi-platform client application takes full advantage of our hybrid architecture while providing a user-friendly interface for our exchange system with secure blockchain account management (implemented in a sandboxed environment). In this paper, the Ethereum public blockchain is used as a target blockchain, but YOSEMITE Hybrid technology can be implemented on any other public blockchains having tokenization support(e.g. ERC20).

Contents

1 The YOSEMITE On/Off-Chain Hybrid Exchange System

1.1 Problems of Current Crypto Exchange Systems

1.1.1 Centralized Exchanges

Currently, popular crypto exchange services (Coinbase, Poloniex, Kraken, Bitfinex, …) are highly centralized systems. The market transactions of crypto assets (BTC, ETH, LTC, ETC, XRP, GNT, …) on these centralized systems are processed by the application server and all relevant data are held in centralized databases. For such systems, only the deposit/withdrawal transactions of a crypto asset involve interaction with the public blockchain. Buy/sell order-books and trading events are stored in database records on a centralized exchange server system for which only the exchange operator has full read/write access.

This poses a problem for the crypto community. Regardless of exchange policy, centralized exchanges cannot guarantee that trading data or transaction execution is fully transparent to their users. There is always a possibility that data can be modified/manipulated by the service operator or by an unauthorized malicious actor. There are many reported incidents of hacking, malfunctioning systems, and suspected manipulation. Nonetheless, most trading volume for crypto assets remains on centralized exchanges simply because centralized server systems are currently the only solution efficient and performant enough to handle the huge amounts of throughput necessary for a well-functioning exchange.

1.1.2 Decentralized Exchanges

With the advent of smart contract technology (Ethereum), fully decentralized crypto exchanges like OasisDEX(Maker), EtherDelta, EtherEx, and VariabL have been (or are being) developed. Exchange functionalities can be implemented by using smart contracts on the blockchain itself. Through decentralized architecture, all trading actions (making/taking/cancelling orders, etc.) can be handled exclusively on the blockchain, making all trading transactions transparent and secure without a centralized server system.

Decentralized exchange systems are ideal but there are currently critical problems that make them a less attractive option when compared to centralized architecture. Firstly, users have to pay blockchain transaction fees (gas fees, in ETH) to execute trading actions. This is true even in the cases of making open orders which are not immediately traded, and cancelling unfilled open orders which needlessly consumes gas fees.[1] This added cost can significantly reduce market liquidity as traders pay costs simply to fill orders and are punished again when canceling them. Even worse, blockchain transaction delay (limited by block confirmation time) is measured in tens of seconds on the Ethereum network. This means that users have to wait long periods for

[1] Currently, the gas cost on the Ethereum network ranges between \$0.3 ~ \$0.8 per transaction.

resolution of each trading action, even if the network is not congested. This is not tenable for an efficiently functioning market where transaction executions should be measured in terms of milliseconds, not tens of seconds. In times of high volatility or volume, a market with such long execution granularity would struggle at best.

Additionally, for an ideal setup of a blockchain application, users must use a specialized web browser (like Ethereum Mist) with a blockchain node running on the user's local machine to synchronize full blockchain data. Syncing can cause long wait times (often several hours) and consume large portions of a machine's local storage. Even after initial syncing a user usually has to wait several minutes to synchronize incremental blockchain data. Moreover, decentralized exchanges also have the disadvantage of storing order-book and trading data on-chain. With large activity, and through prolonged usage, this will bloat the blockchain data size duplicated on every full blockchain node. Crypto-currencies (like ETH) are used as a trading currency on decentralized exchanges. Only users who already have these crypto-currencies can use decentralized exchange services. This means that users must go through the burdensome process of buying these currencies on external crypto-exchanges before trading. In addition, crypto prices are very volatile, making traded asset prices unintentionally volatile, since buy/sell order prices already on the order book denominated in volatile crypto-currencies (ETH) can fluctuate drastically in terms of dollar prices.

With these current technological limitations, fully decentralized exchange systems are too inefficient and expensive to be a viable option for a commercially successful product. In fact, the YOSEMITE development team has already built and tested a fully decentralized exchange PoC (Proof of Concept) for our asset token trading system using only the Ethereum blockchain and IPFS peer-to-peer distributed storage with serverless architecture.[2]

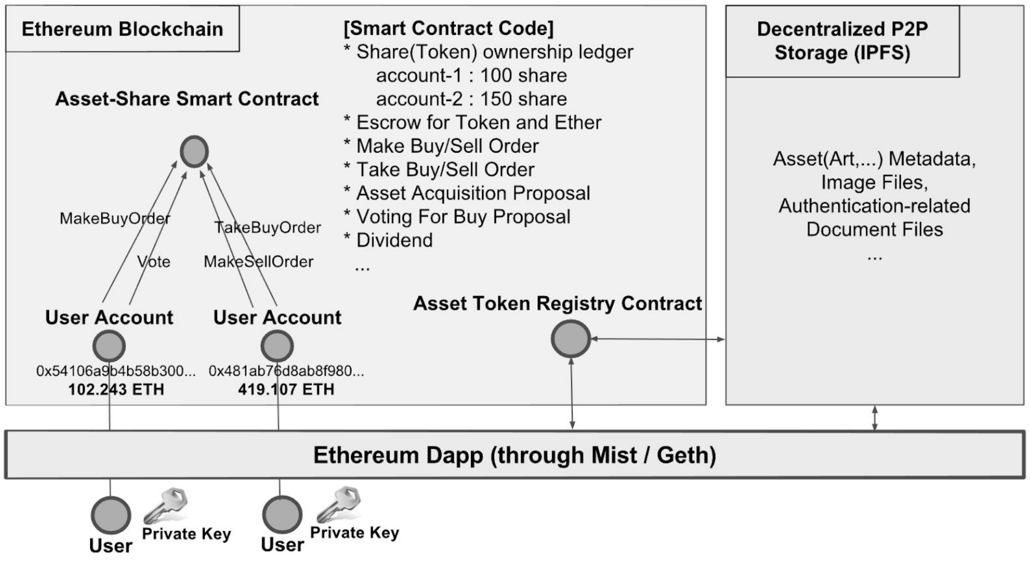

Ethereum Blockchain
[Smart Contract Code]
* Share(Token) ownership ledger
account-1 : 100 share
account-2 : 150 share
* Escrow for Token and Ether
* Make Buy/Sell Order
* Take Buy/Sell Order
* Asset Acquisition Proposal
* Voting For Buy Proposal
* Dividend
...
Asset-Share Smart Contract
MakeBuyOrder
Vote
TakeBuyOrder
MakeSellOrder
User Account
User Account
0x54106a9b4b58b300...
102.243 ETH
0x481ab76d8ab8f980...
419.107 ETH
Asset Token Registry Contract
Decentralized P2P Storage (IPFS)
Asset(Art,...) Metadata, Image Files, Authentication-related Document Files
...
Ethereum Dapp (through Mist / Geth)
User
Private Key
User
Private Key

Figure 1 - System Architecture of 1st Generation Fully Decentralized Asset Token Exchange System PoC, 2016

[2] This PoC was completed in Dec. 2016, and can be found at http://exdappdev.artstockx.com by running the Ethereum Mist Browser on the Ropsten test-net.

1.2 Introduction to the **YOSEMITE On/Off-Chain Hybrid** System

- A hybrid exchange system with on/off-chain solutions and Digital USD (dUSD, a fiat-pegged crypto token as a stable trading currency)

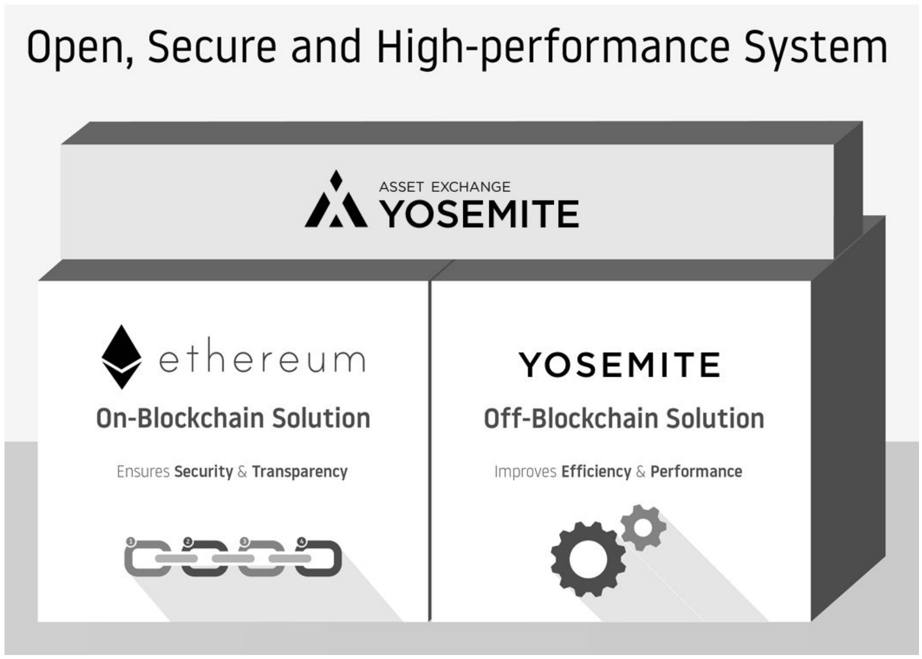

Open, Secure and High-performance System
ASSET EXCHANGE
YOSEMITE
ethereum
On-Blockchain Solution
Ensures Security & Transparency
YOSEMITE
Off-Blockchain Solution
Improves Efficiency & Performance

Figure 2 - Hybrid Exchange System Architecture with On/Off Chain Systems

The YOSEMITE Hybrid system is designed to overcome the critical drawbacks of decentralized exchange systems by adopting the efficient and performant transaction-processing capability of centralized exchange systems while at the same time retaining the key blockchain benefits of transparency and security. This is done by combining emerging decentralized system technologies (blockchain / peer-to-peer distributed storage) with widely used secure and performant traditional server side technology.

To achieve this, the balances (ledger) of Asset-Share and dUSD[3] tokens are securely recorded on the Ethereum blockchain while trading transactions are processed on exchange servers transparently and securely. Since trading occurs off-chain, exchange users do not need to pay blockchain transaction fees (gas fees) for each trading action and these trading transactions can be processed in terms of milliseconds. Transparency and security of off-chain transactions are ensured by requiring that any trading data generated by users and the exchange server is cryptographically signed using an Ethereum account. The results of these signed transactions are then published to immutable and secure public peer-to-peer storage (IPFS). This crypto signing and publishing of all trading data implies that any third party can fully audit the history of transactions generated by both users and the exchange system itself. Additionally, with dUSD,

[3] We will discuss how dUSD tokens are issued and redeemed in section 3.

our fiat-pegged stable trading currency token, users can trade Asset-Share tokens easily without the high-volatility issue of using crypto-currency (ETH) as trading currency.

The YOSEMITE exchange system is a specialized software platform for global trading of Asset-Share tokens created for each asset (real estate, art pieces, ...) listed through the YOSEMITE service. However, the core technology behind the hybrid-style exchange system can be applied to general crypto-currency / stock exchange systems.

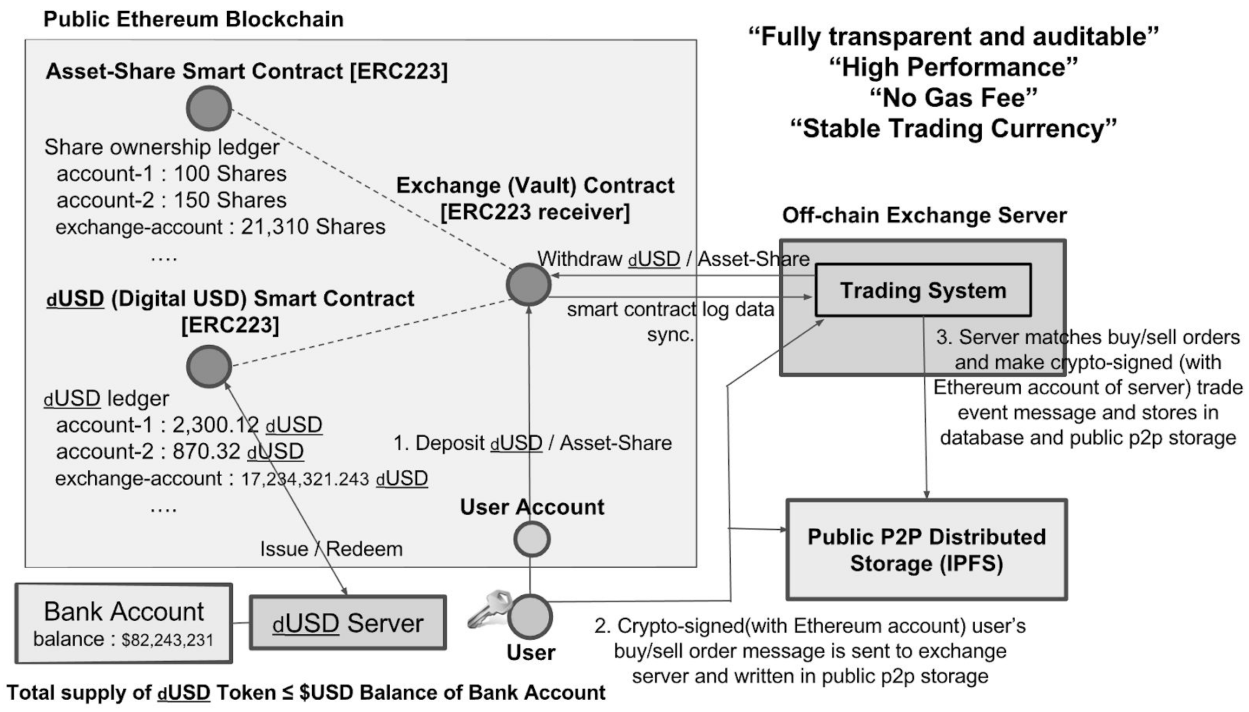
Our development team has already developed the alpha test version of our hybrid-style exchange system using Ethereum and IPFS.[4] We open the alpha test system with this whitepaper. The alpha test version is developed based on the Ethereum blockchain, but the hybrid technology can be implemented on any other blockchains having tokenization support(e.g. ERC20).

	Decentralized Exchange **[EtherDelta, YOSEMITE PoC, ...]**	**Centralized** Exchange **[Coinbase, Poloniex, Kraken, ...]**	**Hybrid-style** Exchange **[YOSEMITE Hybrid Alpha]**
Transaction Performance	Very Slow Very Low Throughput	High Speed High Throughput	**High Speed** **High Throughput**
Blockchain Fee	User pays Gas Fee (Ether)	No Gas Fee	**No Gas Fee**
Data Transparency / Auditability	Fully transparent / auditable, but **bloats the blockchain**	**not transparent**, data could be manipulated, **Not auditable**	**All trading data is transparent and auditable**
User Account	Blockchain Account (created on client-side)	**Server Generated User Account**	**Blockchain Account (created on client-side)**
Fiat Stable Coin	-	**Fiat Money, USDT**[5]	**Fiat-Pegged Token (dUSD)**
User Interface	Difficult (only blockchain experts can use)	Easy	**Easy**
Trading Volume	Very Low Volume	Most Crypto Trading	**-**

Table 1 - Exchange System Comparison

[4] The alpha version can be tested at http://alpha.yosemitex.com by running Chrome web-browser + MetaMask plugin on the Ethereum Rinkeby test-net.
[5] USDT is a fiat backed crypto-token built on the Bitcoin and Omni protocols - https://tether.to/wp-content/uploads/2016/06/TetherWhitePaper.pdf.

1.3 **YOSEMITE On/Off-Chain Hybrid System** Architecture Overview



Public Ethereum Blockchain
Asset-Share Smart Contract [ERC223]
Share ownership ledger
account-1 : 100 Shares
account-2 : 150 Shares
exchange-account : 21,310 Shares
....
Exchange (Vault) Contract [ERC223 receiver]
dUSD (Digital USD) Smart Contract [ERC223]
dUSD ledger
account-1 : 2,300.12 dUSD
account-2 : 870.32 dUSD
exchange-account : 17,234,321.243 dUSD
....
Withdraw dUSD / Asset-Share
smart contract log data sync.
1. Deposit dUSD / Asset-Share
User Account
Issue / Redeem
"Fully transparent and auditable"
"High Performance"
"No Gas Fee"
"Stable Trading Currency"
Off-chain Exchange Server
Trading System
3. Server matches buy/sell orders and make crypto-signed (with Ethereum account of server) trade event message and stores in database and public p2p storage
Public P2P Distributed Storage (IPFS)
Bank Account
balance : $82,243,231
dUSD Server
User
2. Crypto-signed(with Ethereum account) user's buy/sell order message is sent to exchange server and written in public p2p storage
Total supply of dUSD Token ≤ $USD Balance of Bank Account

Figure 3 - YOSEMITE On/Off-Chain Hybrid System Architecture Diagram

The YOSEMITE Hybrid system uses Ethereum blockchain accounts themselves as exchange user accounts[6], and these are generated independently on the client side without any server intervention. This is in contrast to the traditional centralized exchange systems in which users need to sign-up / sign-in to the server system where the user account data and user credentials are created and stored in a server side database which is managed and controlled by the exchange service operator. Hence, the user's exchange account is fully controlled by the user rather than a centralized server.

In the YOSEMITE Hybrid Exchange system there are three main types of smart contracts: i) Asset-Share smart contracts which are created for each listed asset (e.g. real estate, art piece, ...), ii) the dUSD token smart contract for managing the trading currency, and iii) the Exchange (Vault) smart contract for the exchange system.

All trading messages which indicate a user's trading action (Buy-Order, Sell-Order, Cancel-Order, ...) are cryptographically signed by the private key of that user's Ethereum account on the client side. Anyone, including the exchange server, can verify these cryptographic signatures to make sure that a specific trading message is made by the owner of a specific Ethereum account.

[6] Ethereum accounts are public/private key pairs generated using secp256k1 elliptic curve cryptography.

User-signed trading messages are sent to the exchange server and at the same time published to public IPFS peer-to-peer storage. Once on IPFS the immutability of published data is ensured by IPFS's content addressable storage system using a Merkle DAG data structure. The exchange server gathers users' buy/sell orders and then matches any validated buy/sell orders creating a server generated order-matching (trade) transaction on a first-come, first-serve basis. All order-matching transactions are also cryptographically signed with the private key of an exchange-controlled account registered on the Exchange Vault smart contract. These server-signed 'Trade' messages are published to IPFS public storage. In this way all exchange transactions can be fully audited and verified by any external party.

During this process the exchange server manages its own server-side database to provide high speed data services like real-time order books and stock price-charts to the client side application. Because buy/sell orders and order-matching (trade) transactions are not directly executed on the Ethereum blockchain but are instead processed on a high-speed exchange server securely and transparently, no gas fee for the Ethereum blockchain is needed for user trading actions, and transactions can be processed at a high rate comparable to traditional centralized exchange services.

IPFS peer-to-peer storage is used as a public immutable database of exchange transaction records similar to the way blockchain transactions are stored on the blockchain. Even in the catastrophic event of an exchange server crash, the whole history of transaction data can be restored from IPFS storage and the Ethereum blockchain with a mathematical guarantee that no modification to the pre-crash exchange transaction data occurred.

Only deposit and withdrawal transactions of dUSD and Asset-Share tokens to/from the exchange smart contract are actually executed directly on the Ethereum blockchain. All other transactions such as buy/sell/cancel orders and order-matching (trade) are securely saved on IPFS storage and used as proof data for the account balance settlement process upon a user's withdrawal request for dUSD or Asset-Share tokens. After account balance settlement verification and confirmation occur between the exchange server and the user, an actual Ethereum blockchain transaction is executed transferring dUSD/Asset-Share tokens from the Exchange Vault smart contract to the user's Ethereum account along with the confirmed account balance settlement proof data.

1.4 Ethereum Smart Contracts for Asset-Share Token / Asset Token Registry / dUSD Token / Exchange Vault

For each asset listed on YOSEMITE exchange, an Asset-Share Token Ethereum smart contract instance is deployed with functions for the Asset-Share token account balance ledger, token transfer of Asset-Share tokens (used for deposit/withdrawal to exchange), initial share subscription and shareholder voting.

The addresses of the Asset-Share token smart contracts and the crypto hash addresses pointing to the immutable asset metadata files (e.g., text, images, PDF files of authentication-related documents, ...) on IPFS are registered on the Asset Token Registry Ethereum smart contract. Only registered Asset-Share tokens can be deposited and traded on YOSEMITE exchange.

The dUSD token Ethereum smart contract includes functions for the dUSD token account balance ledger, transfer of dUSD tokens (used for deposit/withdrawal to exchange) and issuing/redeeming dUSD.

The Exchange Vault smart contract receives dUSD tokens and Asset-Share tokens from exchange user accounts as deposits for trading, and has a withdrawal function which receives as parameters proof hash of the exchange account balance settlement and the user's signature. dUSD tokens and Asset-Share tokens being traded through the exchange server are held in the Exchange Vault contract and are unlocked when the withdrawal settlement process between the user and exchange system is successfully completed.

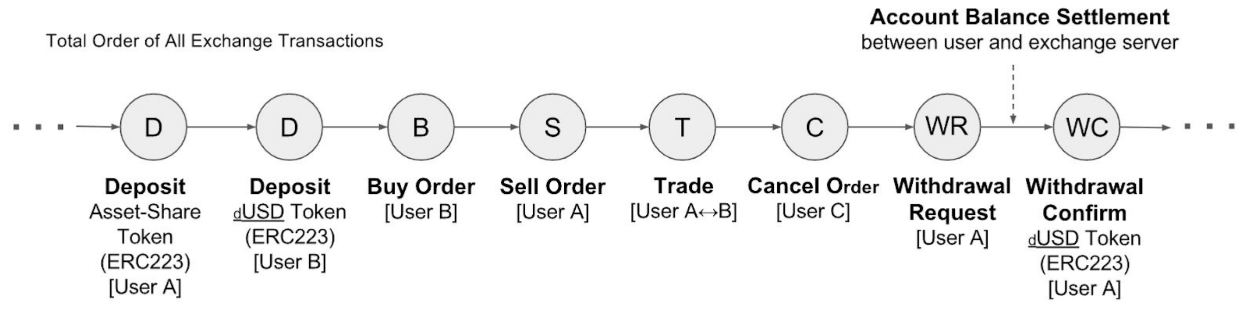
Both the Asset-Share token smart contracts and the dUSD token smart contract are *ERC20*[7] and *ERC223*[8] compatible. *ERC223* is adopted to process the 'deposit-to-exchange' transactions of Asset-Share/dUSD tokens efficiently with just one blockchain transaction (*transfer*) whereas usually two transactions (*approve/transferFrom*) are needed in *ERC20*. With this, the Exchange Vault is an *ERC223* receiver smart contract (having a *tokenFallback* function). The Exchange Vault smart contract generates Deposit/Withdrawal Ethereum blockchain log events so that any other system including the exchange server can securely capture all token transfer transactions to/from the Exchange Vault smart contract.

A top-priority design goal of the YOSEMITE Hybrid Exchange System is that average users having no ETH should be able to use the Ethereum blockchain-based YOSEMITE service seamlessly. For any Ethereum blockchain transaction, gas fees (ETH) should be paid by the transaction sender, including deposit-to-exchange transactions. To prevent users from having to pay this network fee, the Asset-Share/dUSD token smart contracts have implemented a 'transferDelegated' function. This function is called by the exchange server with a user's crypto signature asserting that the user has delegated the Ethereum transaction (a token transfer/deposit to the Exchange Vault contract) to the exchange server on behalf of the user. In this manner, the exchange server pays the network transaction fees instead of the user, but only on the user's permission. In such cases, the server will charge the user a little portion of dUSD from the token amount being deposited or dUSD already available on the user's exchange account as a deposit transaction fee. This feature is possible because YOSEMITE provides the dUSD stable token system. With this, average users can use our Ethereum-based exchange service seamlessly without having to purchasing any ETH from an external crypto exchange service, and dUSD can be used to pay the blockchain gas fee.

[7] *ERC20* token standard - https://github.com/ethereum/eips/issues/20.
[8] *ERC223* is still in active discussion before final standardization. See here EIP - *ERC223* token standard - https://github.com/ethereum/EIPs/issues/223. The community commonly refers to *ERC223* as *ERC23*.

1.5 Data Architecture of the YOSEMITE On/Off-Chain Hybrid System

1.5.1 Exchange Flow Data Structures


Total Order of All Exchange Transactions
Account Balance Settlement
between user and exchange server
D
D
B
S
T
C
WR
WC
Deposit Asset-Share Token (ERC223) [User A]
Deposit dUSD Token (ERC223) [User B]
Buy Order [User B]
Sell Order [User A]
Trade [User A↔B]
Cancel Order [User C]
Withdrawal Request [User A]
Withdrawal Confirm dUSD Token (ERC223) [User A]

* Deposit / Withdrawal-Confirm → Ethereum on-chain transactions
* Buy/Sell/Cancel Order, Trade, Withdrawal-Request/Confirm
→ Ethereum account(user/server) signed / IPFS-published off-chain transactions

Figure 4 - Total order of exchange transactions on-chain and off-chain

All of the transactions in the exchange service should be serially ordered. The exchange system can be viewed as a state-machine whose state is the set of exchange account balances (tradable balances and in-trade/escrowed balances) for dUSD/Asset-Share tokens and open (unfilled) buy/sell orders. Each transaction makes a state-transition from the current exchange system state to the next.

$$\delta \ : \ S_n \times T \rightarrow S_{n+1} \quad \textit{where S : exchange state, T : input transaction, } \delta \textit{ : state transition function}$$

A total order of all exchange transactions is transparently and securely managed on the public Ethereum blockchain and in IPFS storage through the exchange server. A complete list of transaction types processed on the YOSEMITE exchange system are 'Buy-Order', 'Sell-Order', 'Trade-Buy', 'Trade-Sell', 'Cancel-Buy-Order', 'Cancel-Sell-Order', 'Deposit', 'Withdrawal-Request', 'Cancel-Withdrawal-Request', and 'Withdrawal-Confirm'.

- json keys of transaction messages

xa : exchange Ethereum smart contract address (Exchange Vault) identifying the exchange server
tt : transaction type ('OB' : Order Buy, 'OS' : Order Sell, 'TB' : Trade-Buy, 'TS' : Trade-Sell, 'CB' : Cancel-Buy, 'CS' : Cancel-Sell, 'WR' : Withdrawal-Request, 'CWR' : Cancel-Withdrawal-Request, 'WC' : Withdrawal-Confirm)
ea : user's Ethereum account address
sy : Asset-Share token symbol (id for an Asset-Share type) or 'dUSD'
am : Asset-Share token amount or dUSD amount (6 decimals big integer, 1 dUSD = 1000000, 30.5 dUSD = 30500000)
pr : dUSD price for 1 Asset-Share (6 decimals big integer)
mfr : maker fee rate (unit in percentage(0-100) multiplied by 100 (0(0%)-10000(100%), ex: 102 = 1.02%, 10 = 0.1%)
tfr : taker fee rate (same unit as 'mfr')

ts : client timestamp at which transaction message is made
si : crypto signature signed by the user's Ethereum account for transaction message (stringified json object)
mea : Ethereum account address of the maker in a Trade transaction
tea : Ethereum account address of the taker in a Trade transaction
bI : buy-order id in a Trade transaction
sI : sell-order id in a Trade transaction
mf : maker fee dUSD amount in a Trade transaction (6 decimals big integer)
tf : taker fee dUSD amount in a Trade transaction (6 decimals big integer)
tsS : server timestamp when the transaction message is made on exchange server
siS : crypto signature signed by the exchange server's Ethereum account for a transaction message
txF : transaction fee dUSD amount in a Withdrawal-Request transaction (6 decimals big integer)
wrI : withdrawal-request id in a Cancel-Withdrawal-Request or Withdrawal-Confirm transaction
wsI : withdrawal-settlement-data file hash id

- **Buy-Order** : An exchange user makes a buy-order message (a stringified json object) and cryptographically signs this message. The signed buy-order message is sent to the exchange server and published to IPFS storage.

$$signedBuyOrderMsg = buyOrderMsg + (\text{“}si\text{”} \rightarrow ECDSA.Sign(PK_U, buyOrderMsg))$$

where PK_U is the private key of the user account, ECDSA.Sign is signing function of the elliptic curve digital signature algorithm, buyOrderMsg/signedBuyOrderMsg are tightly packed (no whitespace) stringified json objects

$$buyOrderId = MerkleRootHash(signedBuyOrderMsg) = IPFS\ file\ address$$

where MerkleRootHash is a base58 encoded root hash of merkle tree/dag of file data on IPFS [9]

$$signedBuyOrderMsg = IPFS.Get(buyOrderId)$$

where IPFS.Get is a file data retrieving function using the merkle root hash address of a file on IPFS

$$UserEthAddress = ECDSA.Recover(buyOrderMsg, signedBuyOrderMsg(\text{"}si\text{"}))$$

where ECDSA.Recover is public key recovering function of the elliptic curve digital signature algorithm, buyOrderMsg = signedBuyOrderMsg -"si"

[buy-order message examples]

buyOrderMsg = { "xa" : "0x33e50109...119cba0a088", "tt" : "OB", "ea" : "0x38909c7...d7b25e0590" , "sy" : "AS_PC_GN", "am" : "30", "pr" : "30500000", "mfr" : 10, "tfr" : 20, "ts" : 1500882556820 }

signedBuyOrderMsg = { "xa" : "0x33e50109...119cba0a088", "tt" : "OB", "ea" : "0x38909c7...d7b25e0590" , "sy" : "AS_PC_GN", "am" : "30", "pr" : "30500000", "mfr" : 10, "tfr" : 20, "ts" : 1500882556820, **"si" : "0xdfef1901548ec804ecfa...72a5c8909d7961c1c"** }

buyOrderId = "QmQjxtDWvHVMVp...9Y3r5g5QP2nX6Kev"
signedBuyOrderMsg = *IPFS.Get*("QmQjxtDWvHVMVp...9Y3r5g5QP2nX6Kev")

"0x38909c7...d7b25e0590" = *ECDSA.Recover*(*buyOrderMsg*,"0xdfef1901548ec804ecfa...72a5c8909d7961c1c")

[9] For merkle tree explanation , see https://en.wikipedia.org/wiki/Merkle_tree.

- **Sell-Order** : An exchange user makes a sell-order message and cryptographically signs this message. The signed sell-order message is sent to the exchange server and published to IPFS storage.

$$signedSellOrderMsg = sellOrderMsg + (\text{“}si\text{”} \rightarrow ECDSA.Sign(PK_U, sellOrderMsg))$$

$$sellOrderId = MerkleRootHash(signedSellOrderMsg) = IPFS\ file\ address$$
$$signedSellOrderMsg = IPFS.Get(sellOrderId)$$

$$UserEthAddress = ECDSA.Recover(sellOrderMsg, signedSellOrderMsg(\text{"}si\text{"}))$$

[sell-order message examples]

sellOrderMsg = { "xa" : "0x33e50109...119cba0a088", "tt" : "OS", "ea":"0x38909c7...d7b25e0590", "sy" : "AS_PC_GN", "am" : "80", "pr" : "28750000", "mfr" : 10, "tfr" : 20, "ts" : 1500882742634 }

signedSellOrderMsg = { "xa" : "0x33e50109...119cba0a088", "tt" : "OS", "ea":"0x38909c7...d7b25e0590", "sy" : "AS_PC_GN", "am" : "80", "pr" : "28750000", "mfr" : 10, "tfr" : 20, "ts" : 1500882742634, **"si" : "0x4c45a4c457a5d12645...ccc1ef8147dfe5ddb1b"** }

sellOrderId = “QmeUwJTsG...vJJonet9hSfT”
signedSellOrderMsg = *IPFS.Get*(“QmeUwJTsG...vJJonet9hSfT”)

“0x38909c7...d7b25e0590” = *ECDSA.Recover*(*sellOrderMsg*,”0x4c45a4c457a5d12645...ccc1ef8147dfe5ddb1b”)

- **Trade-Buy / Trade-Sell** : The exchange server matches buy-order and sell-order messages submitted by users, then makes trade-buy or trade-sell messages accordingly. A trade-buy message is made when a taker buys tokens from a sell-order maker. A trade-sell message is made when a taker sells tokens to a buy-order maker. The exchange server cryptographically signs the trade message and publishes the signed trade transaction message to IPFS storage.

$$signedTradeMsg = tradeMsg + (\text{“}siS\text{”} \rightarrow ECDSA.Sign(PK_{EX}, tradeMsg))$$
where PK_{EX} is private key of exchange server Ethereum account

$$tradeId = MerkleRootHash(signedTradeMsg) = IPFS\ file\ address$$
$$signedTradeMsg = IPFS.Get(tradeId)$$

$$ExchangeServerEthAddress = ECDSA.Recover(tradeMsg, signedTradeMsg(\text{"}siS\text{"}))$$

[trade-buy message examples]

tradeMsg = { "xa" : "0x33e50109...119cba0a088", "tt" : "TB", "mea" : "0xccd78c364ca...9c3252cb4", "tea" : "0x38909c7...d7b25e0590", "bl" : "QmQevjMCw66E...9uGyBnh17p", "sl" : "QmPYyQsve...1osuWYJQ", "sy" : "AS_PC_GN", "am" : "15", "pr" : "31800000", "mf" : "477000", "tf" : "954000", "tsS" : 1500882596513 }

signedTradeMsg = { "xa" : "0x33e50109...119cba0a088", "tt" : "TB", "mea" : "0xccd78c364ca...9c3252cb4", "tea" : "0x38909c7...d7b25e0590", "bl" : "QmQevjMCw66E...9uGyBnh17p", "sl" : "QmPYyQsve...1osuWYJQ", "sy" : "AS_PC_GN", "am" : "15", "pr" : "31800000", "mf" : "477000", "tf" : "954000", "tsS" : 1500882596513, **"siS" : "0x61f56e248cae719bd...aa3177248a01ac4a1c"** }

tradeId = “QmUQmgiRD...sSwzq8S4qw”
signedTradeMsg = *IPFS.Get*(“QmUQmgiRD...sSwzq8S4qw”)

“0x5677e2388...ee9dcaa6” = *ECDSA.Recover*(*tradeMsg*,”0x61f56e248cae719bd...aa3177248a01ac4a1c”)

- **Cancel-Buy-Order / Cancel-Sell-Order** : An exchange user can cancel his/her own open(unfilled) buy/sell order by creating and cryptographically signing a cancel-buy/sell-order message. The signed message is sent to the exchange server and published to IPFS storage.

$$signedCancelOrderMsg = cancelOrderMsg + (\text{“}si\text{”} \rightarrow ECDSA.Sign(PK_U, cancelOrderMsg))$$

$$cancelOrderId = MerkleRootHash(signedCancelOrderMsg) = IPFS\ file\ address$$
$$signedCancelOrderMsg = IPFS.Get(cancelOrderId)$$

$$UserEthAddress = ECDSA.Recover(cancelOrderMsg, signedCancelOrderMsg(\text{"}si\text{"}))$$

[cancel-buy-order message examples]
cancelOrderMsg = { "xa" : "0x33e50109...119cba0a088", "tt" : "CB", "ea" : "0x38909c7...d7b25e0590", "sy" : "AS_PC_GN", "bl" : "QmP48eKQ...YAroA846N", "ts" : 1500882760269 }

signedCancelOrderMsg = { "xa" : "0x33e50109...119cba0a088", "tt" : "CB", "ea" : "0x38909c7...d7b25e0590", "sy" : "AS_PC_GN", "bl" : "QmP48eKQ...YAroA846N", "ts" : 1500882760269, **"si":"0xb19dafeaf91...a9a85a8a91c"** }

cancelOrderId = “QmSSEDAc...LZt9oEcRLNF”
signedCancelOrderMsg = IPFS.Get(“QmSSEDAc...LZt9oEcRLNF”)

“0x38909c7...d7b25e0590” = *ECDSA.Recover*(*cancelOrderMsg*,”0xb19dafeaf91...a9a85a8a91c”)

- **Deposit** : An exchange user transfers ownership of the user’s dUSD/Asset-Share tokens to the Exchange Vault smart contract by executing an Ethereum blockchain transaction on the dUSD/Asset-Share smart contract. A delegated ‘Deposit’ blockchain transaction can be made by the exchange server on behalf of the user using the ‘transferDelegated’ function of dUSD/Asset-Share contract, thereby consuming the server’s ETH but charging a transaction fee in dUSD to the user. For each token transfer, the Exchange Vault smart contract generates a ‘Deposit’ Ethereum log event so the exchange server can monitor all deposit transactions. The Ethereum transaction hash is sufficient for data security and transparency of a deposit because an Ethereum transaction cannot occur without the user’s crypto-signature and because all blockchain transaction records are immutable public data. The transaction receipt containing ‘Deposit’ log data can be retrieved through the standard Ethereum API interface. Since everything is executed and recorded on the blockchain, additional signing and IPFS publishing for deposit messages are not necessary.

$$Ethereum.getTransactionReceipt(depositEthereumTransactionHash)$$

[deposit transaction hash / Deposit Exchange Vault smart contract log example]

{ "address" : "0x33e50109...119cba0a088", "blockNumber" : 591404, **"transactionHash" : "0xa83808838c798a9...37a92a6ec4d75ec2"**, "blockHash" : "0x4080637f189e90b...61e07677d033957d", **"event" : "Deposit"**, "args" : { "_from" : "0x38909c7...d7b25e0590", "_symbol" : "DUSD", "_value" : "3000000000000000000000" }, … }

- **Withdrawal-Request** : An exchange user can request a withdrawal of dUSD/Asset-Share tokens from the Exchange Vault smart contract to the user's Ethereum account on the public blockchain. The user specifies the amount of dUSD, or amount of specific Asset-Share tokens, to be withdrawn as well as the withdrawal transaction fee amount acceptable (calculated in dUSD) by the exchange server. After a withdrawal request is accepted by exchange server, the user's exchange account is locked and an account balance settlement procedure between the user and the exchange server is executed. The user's withdrawal-request message is signed and stored in IPFS similarly to other transaction messages.

$$signedWithdrawalReqMsg = withdrawalReqMsg + (\text{“}si\text{”} \rightarrow ECDSA.Sign(PK_U, withdrawalReqMsg))$$

$$withdrawalReqId = MerkleRootHash(signedWithdrawalReqMsg) = IPFS\ file\ address$$
$$signedWithdrawalReqMsg = IPFS.Get(withdrawalReqId)$$

$$UserEthAddress = ECDSA.Recover(withdrawalReqMsg, signedWithdrawalReqMsg(\text{"}si\text{"}))$$

[withdrawal-request message examples]

withdrawalReqMsg = { "xa" : "0x33e50109...119cba0a088", "tt" : "WR", "ea" : "0x38909c7...d7b25e0590", "sy" : "dUSD", "am" : "5000000000", "txF" : "500000", "ts" : 1500882917498 }

signedWithdrawalReqMsg = { "xa" : "0x33e50109...119cba0a088", "tt" : "WR", "ea" : "0x38909c7...d7b25e0590", "sy" : "dUSD", "am" : "5000000000", "txF" : "500000", "ts" : 1500882917498, **"si" : "0xc0cfe019db...52a7d11b"**
}

withdrawalReqId = "QmPim5DrZ9Zo...q3PyyDvLmd3"
signedWithdrawalReqMsg = *IPFS.Get*("QmPim5DrZ9Zo...q3PyyDvLmd3")

"0x38909c7...d7b25e0590" = *ECDSA.Recover*(*withdrawalReqMsg*,"0xc0cfe019db...52a7d11b")

- **Cancel-Withdrawal-Request** : An exchange user can cancel a withdrawal-request before the account balance settlement procedure has completed. If the user cancels, the user's exchange account is unlocked. The user's cancel-withdrawal-request message is signed and stored in IPFS similarly to other transaction messages.

$$signedCancelWdReqMsg = cancelWdReqMsg + (\text{“}si\text{”} \rightarrow ECDSA.Sign(PK_U, cancelWdReqMsg))$$

$$cancelWdReqId = MerkleRootHash(signedCancelWdReqMsg) = IPFS\ file\ address$$
$$signedCancelWdReqMsg = IPFS.Get(cancelWdReqId)$$

$$UserEthAddress = ECDSA.Recover(cancelWdReqMsg, signedCancelWdReqMsg(\text{"}si\text{"}))$$

[withdrawal-request message examples]

cancelWdReqMsg = { "xa" : "0x33e50109...119cba0a088", "tt" : "CWR", "ea" : "0x38909c7...d7b25e0590", "wrl" : "QmPim5DrZ9Zo...q3PyyDvLmd3", "ts" : 1501040519725 }

signedCancelWdReqMsg = { "xa" : "0x33e50109...119cba0a088", "tt" : "CWR", "ea" : "0x38909c7...d7b25e0590", "wrl" : "QmPim5DrZ9Zo...q3PyyDvLmd3", "ts" : 1501040519725, **"si" : "0x0880eff7f3a00c...c5233d79831c"** }

cancelWdReqId = "QmPHji7WQHW...UsGBMbKu3hv6F9"
signedCancelWdReqMsg = IPFS.Get("QmPHji7WQHW...UsGBMbKu3hv6F9")

"0x38909c7...d7b25e0590" = *ECDSA.Recover*(*cancelWdReqMsg*,"0x0880eff7f3a00c...c5233d79831c")

- **Withdrawal** : After a user's withdrawal-request has been submitted before the actual Ethereum blockchain transaction for dUSD/Asset-Share token withdrawal, an **Account Balance Settlement** procedure is performed between the exchange server and the user to agree and confirm the user's current token balances on the exchange. Upon a user's withdrawal-request, the exchange server proposes a settlement data file stored on IPFS to the user. The settlement data contains:

  - withdrawal request data
    * token type (dUSD/Asset-Share token symbol), amount, transaction fee, withdrawal request id
  - current account balances in exchange
    * tradable / in-trade (escrowed for open orders) dUSD balances
    * tradable / in-trade (escrowed for open orders) Asset-Share token balances for each Asset-Share token the user owns
  - the user's current open (unfilled) buy/sell orders in the exchange
  - all of the user's exchange/blockchain transactions since last account balance settlement
  - previous account balances at the last account balance settlement
  - the last account balance settlement data IPFS file address and previous Withdrawal Ethereum blockchain transaction hash
  - the exchange server's crypto signature for the current settlement data

  On the server side, settlement data is first verified before sending it to the client. On the client side, by using the predefined account balance verification algorithm/procedure published by YOSEMITE which can be re-implemented by any 3rd party, current account balance values can be computed exactly from the received settlement data. Though settlement data is assembled and provided by the exchange server, all information in the settlement data can be validated via cryptographic proof on the Ethereum blockchain and IPFS storage without any reference to the exchange server. All exchange transaction data is chained together cryptographically through crypto-hash and crypto-signatures. Thus, the exchange server cannot manipulate settlement data to forge the user's account balance records.

  Starting from the previous account balance values that have already been validated and settled, the client side application (or any 3rd party verifier) can compute the exact current account balance values by sequentially applying the account balance state transition function for each transaction related to the user's account on the settlement data.

$$AB_i = \delta(AB_{i-1} , T_i)$$

where T_i : i-th transaction related to the user, AB_i : user's account balance state after applying transaction T_i , δ : account balance state transition function

$$\text{given input } \{ AB_p , T_{p+1} , T_{p+2} , \ldots , T_{p+n} \},$$
$$AB_c = \delta(\delta(\ldots \delta(AB_p , T_{p+1}), \ldots), T_{p+n})$$

where AB_p : account balance state at previous settlement, AB_c : current account balance state on withdrawal request , n : number of transactions related to the user since the previous settlement until the current settlement

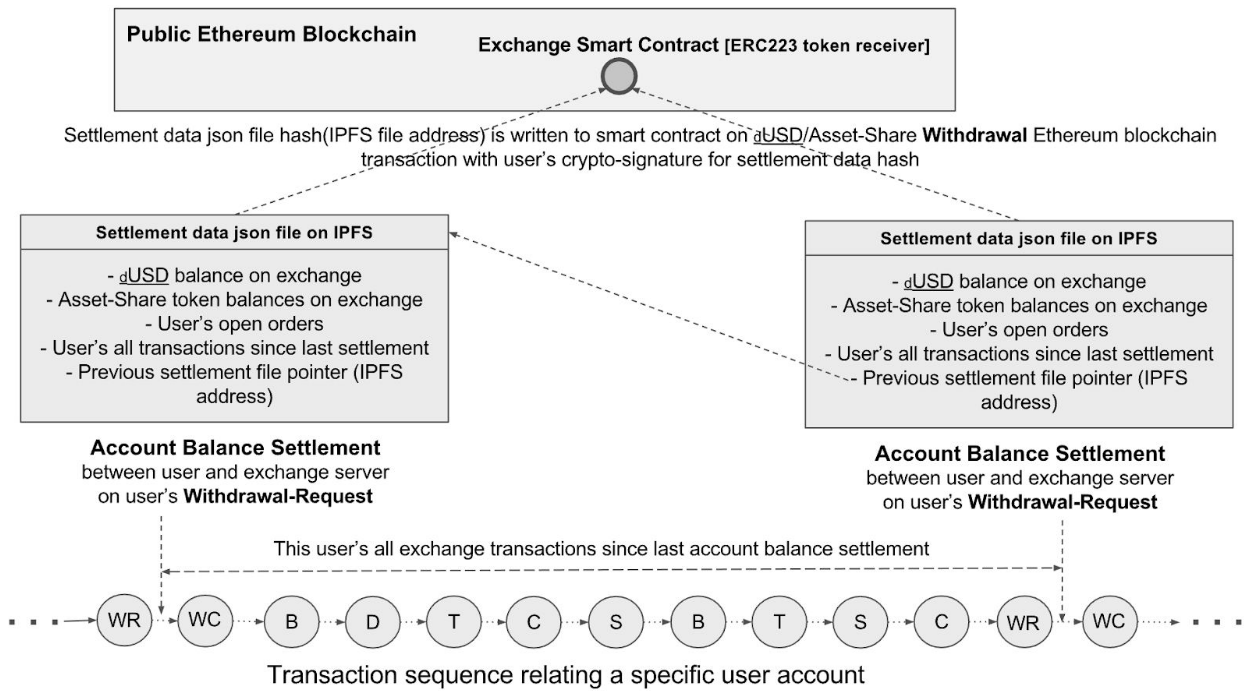

Public Ethereum Blockchain
Exchange Smart Contract [ERC223 token receiver]
Settlement data json file hash(IPFS file address) is written to smart contract on dUSD/Asset-Share Withdrawal Ethereum blockchain transaction with user's crypto-signature for settlement data hash
Settlement data json file on IPFS
- dUSD balance on exchange
- Asset-Share token balances on exchange
- User's open orders
- User's all transactions since last settlement
- Previous settlement file pointer (IPFS address)
Account Balance Settlement between user and exchange server on user's Withdrawal-Request
Settlement data json file on IPFS
- dUSD balance on exchange
- Asset-Share token balances on exchange
- User's open orders
- User's all transactions since last settlement
- Previous settlement file pointer (IPFS address)
Account Balance Settlement between user and exchange server on user's Withdrawal-Request
This user's all exchange transactions since last account balance settlement
WR
WC
B
D
T
C
S
B
T
S
C
WR
WC
Transaction sequence relating a specific user account

Figure 5 - Account Balance Settlement procedure before a Withdrawal Blockchain Transaction

If the user successfully verifies the settlement data and agrees on the final account balances, then the user makes a withdrawal-confirm message including their crypto-signature for the settlement data and submits the message to the exchange server. Only after the user also verifies and confirms the settlement data, will an actual 'Withdrawal' blockchain transaction be executed on the Exchange Vault Ethereum smart contract by the exchange server. The Exchange Vault smart contract code also checks the validity of the user's crypto signature for the withdrawal settlement data.

$$signedSettlementDataJson = settlementDataJson + (\text{“}siS\text{”} \rightarrow ECDSA.Sign(PK_{EX}, settlementDataJson))$$

$$settlementDataHash = MerkleRootHash(signedSettlementDataJson) = IPFS\ file\ address$$
$$signedSettlementDataJson = IPFS.Get(settlementDataHash)$$

$$ExchangeServerEthAddress = ECDSA.Recover(settlementDataJson, signedSettlementDataJson("siS"))$$

$$signedWithdrawalConfirmMsg = [\ \dots, "wsI" \rightarrow settlementDataHash, "si" \rightarrow ECDSA.Sign(PK_U, settlementDataHash)\ ]$$

$$withdrawalId = MerkleRootHash(signedWithdrawalConfirmMsg) = IPFS\ file\ address$$
$$signedWithdrawalConfirmMsg = IPFS.Get(withdrawalId)$$

$$UserEthAddress = ECDSA.Recover(settlementDataHash, signedCancelWdReqMsg("si"))$$
⇒ Exchange Vault smart contract code also verifies this on Withdrawal blockchain transaction

[settlement data json file example]

{ "xa":"0x33e50109...119cba0a088", "doc":"WS", "tsS":1501040540125, "ea":"0x38909c7...d7b25e0590",
"sy":"dUSD",
"am":"7000000000", "txF":"500000", "**wrl**":"QmReLCpLfk...6ySFSaTgR", "cTxS":486,
"**cAB**":{ "a":"3820893603300", "aE":"617266876000", "ATs":[{ "s":"AS_DH_LG", "t":"12300", "tE":"0",
"aE":"106065960000"}, { "s":"AS_PC_GN", "t":"3655", "tE":"0", "aE":"416428012000" }, …],
"**OOs**":[{ "Xt":"OB", "Xsy":"AS_DH_LG", "Xp":"35320000", "Xta":"3000",
"Oi":"QmSDB51B3h...SWVqFbCZ", "OmF":10, "OtF":20, "Ouf":"3000" }, { "Xt":"OB",
"Xsy":"AS_PC_GN", "Xp":"27200000", "Xta":"120", "Oi":"Qmdg683...YxEJmp1", "OmF":10, "OtF":20,
"Ouf":"120" }, …]},
"**TXs**":[{ "Xt":"W", "Xs":468, "Xsy":"dUSD", "Xts":1500858270444, "Xta":"2000000000",
"Wi":"QmS8FQpMo...Ln2hQGuYnc", "WsI":"QmYDDEQQk...MJt1JFgxG1k", "WtxF":"500000",
"Wts":1500858234097, "Wsi":"0xb26bdbb...e81041251b", "WtxH":"0xbc7ed959ace...2211e813e1a71b" },
{ "Xt":"D", "Xs":472, "Xsy":"dUSD", "Xts":1500882510417, "Xta":"2999800000", "Di":"0xa8380...d75ec2" },
…,
{ "Xt":"OB", "Xs":477, "Xsy":"AS_PC_GN", "Xts":1500882596513, "Xp":"31800000", "Xta":"15",
"Oi":"QmQevjM...9uGyBnh17p", "OmF":10, "OtF":20, "Ots":1500882593820,
"Osi":"0x6a3187662ac...e1866489d81c"},
{ "Xt":"T", "Xs":478, "Xsy":"AS_PC_GN", "Xts":1500882596513, "Xp":"31800000", "Xta":"15",
"Ti":"QmUQmgiR...Swzq8S4qw", "TbI":"QmQevjMCw...X9uGyBnh17p", "TsI":"QmPYyQsve...a1osuWYJQ",
"Tty":"B", "Tmt":"T", "Tf":"954000" }, ...],
"pTxS":466,
"**pAB**":{ "a":"3816836281300", "aE":"618511112000", "ATs":[{ "s":"AS_DH_LG", "t":"12300", "tE":"0",
"aE":"106065960000"}, { "s":"AS_PC_GN", "t":"3630", "tE":"90", "aE":"417672248000" }, …],
"**OOs**":[{ "Xt":"OB", "Xsy":"AS_DH_LG", "Xp":"35320000", "Xta":"3000", "Oi":"QmSDB51...VqFbCZ",
"OmF":10, "OtF":20, "Ouf":"3000" }, { "Xt":"OB", "Xsy":"AS_PC_GN", "Xp":"27200000", "Xta":"120",
"Oi":"Qmdg6837...bYxEJmp1", "OmF":10, "OtF":20, "Ouf":"120" }, ...] },
"**pSI**":"QmYDDEQQkCE...MJt1JFgxG1k",
"**pWtxH**":"0x082ead4bc29...69e49a24429"
"**siS**":"0xa7a2f2d65e42...2c0ea73f1b1b" }

(**cTxs**: current transaction sequence, **cAB**: current account balances, **a**: dUSD balance, **aE**: dUSD escrowed, **ATs**: Asset-Share tokens, **s**: symbol, **t**: token amount, **tE**: token escrowed, **OOs**: open orders, **TXs**: transactions, **Xt**: transaction type, **Xs**: transaction sequence, **Xts**: transaction server timestamp, **Xsy**: transaction token symbol, **Xp**: transaction price, **Xta**: transaction token amount, **Oi**: order id, **OmF**: order maker fee rate, **OtF**: order taker fee rate, **Ouf**: order unfilled amount, **Ots**: order client timestamp, **Osi**: order user signature, **Wi**: withdrawal id, **WsI**: withdrawal settlement id, **WtxF**, withdrawal transaction fee, **Wts**: withdrawal client timestamp, **Wsi**: withdrawal user signature, **WtxH**: withdrawal ethereum transaction hash, **Di**: deposit id (ethereum transaction hash), **Ti**: trade id, **TbI**: trande buy order id, **TsI**: trade sell order id, **Tty**: trade type, **Tmt**: trade maker or taker, **Tf**: transaction fee, **pTxS**: previous settlement transaction sequence, **pAB**: previous settlement account balances, **pSI** : previous settlement hash id, **pWtxH**: previous withdrawal ethereum transaction hash)

settlementDataHash = "QmZunhbJXVc...e9F9LaoqXvUtcy"

[withdrawal-request message examples]

withdrawalConfirmMsg = { "xa" : "0x33e50109...119cba0a088", "tt" : "WC", "ea" : "0x38909c7...d7b25e0590", "sy" : "dUSD", "am" : "7000000000", "txF" : "500000", "wrI" : "QmReLCpLfkZjV...p6ySFSaTgR", "wrTs" : 1501040537947, "wsI" : "QmZunhbJXVc...e9F9LaoqXvUtcy", "ts" : 1501058354631 }

signedWithdrawalConfirmMsg = { "xa" : "0x33e50109...119cba0a088", "tt" : "WC", "ea" : "0x38909c7...d7b25e0590", "sy" : "dUSD", "am" : "7000000000", "txF" : "500000", "wrI" : "QmReLCpLfkZjV...p6ySFSaTgR", "wrTs" : 1501040537947, "wsI" : "QmZunhbJXVc...e9F9LaoqXvUtcy", "ts" : 1501058354631, **"si" : "0xbf8596b...770869dad1c"** }

withdrawalId = "QmV17EEgtg...VMbRr67rFi8"
signedWithdrawalConfirmMsg = *IPFS.Get*("QmV17EEgtg...VMbRr67rFi8")

"0x38909c7...d7b25e0590" = *ECDSA.Recover*("QmZunhbJXVc...e9F9LaoqXvUtcy","0xbf8596b...770869dad1c")

1.5.2 Off-chain Blockchain-like Data Structures with Blockchain Anchoring

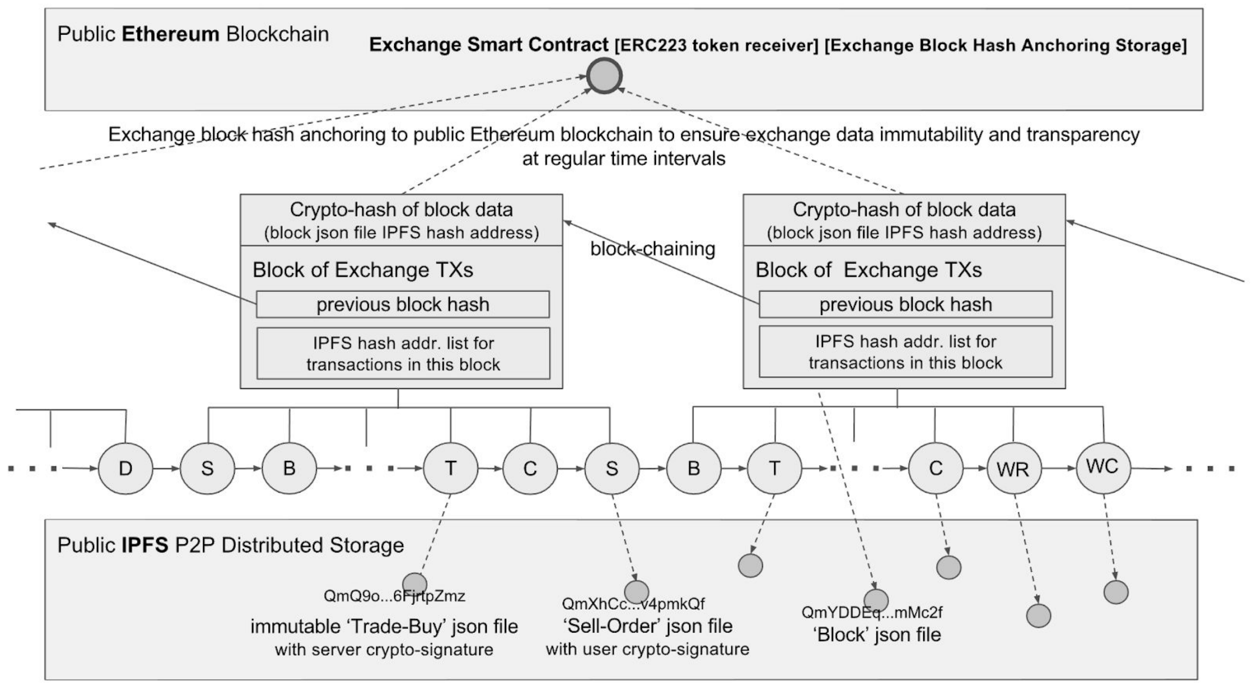

Public Ethereum Blockchain
Exchange Smart Contract [ERC223 token receiver] [Exchange Block Hash Anchoring Storage]
Exchange block hash anchoring to public Ethereum blockchain to ensure exchange data immutability and transparency at regular time intervals
Crypto-hash of block data (block json file IPFS hash address)
Block of Exchange TXs
previous block hash
IPFS hash addr. list for transactions in this block
block-chaining
D
S
B
T
C
S
B
T
C
WR
WC
Public IPFS P2P Distributed Storage
QmQ9o...6FjrtpZmz
immutable 'Trade-Buy' json file with server crypto-signature
QmXhCc...v4pmkQf
'Sell-Order' json file with user crypto-signature
QmYDDEq...mMc2f
'Block' json file

Figure 6 - Off-chain blockchain-like exchange data structure anchored to the public blockchain

To ensure immutability and transparency of all exchange data, all total-ordered exchange transactions are "block-chained" or "file-chained" through IPFS files linked to the each previous one by hash-links. Those files are published regularly via public Ethereum blockchain, so called blockchain anchoring.

Exchange transactions are grouped together at regular time intervals as a ‘Block json file’ stored on IPFS. A Block json file contains i) a sequence of transaction message hashes (IPFS file addresses) pointing to each transaction message file immutably stored on IPFS[10] and ii) the previous file’s IPFS address creating a chain of Blocks. Some IPFS addresses is anchored(or published) on the Exchange Vault smart contract regularly.

This structure is very similar to the one of Bitcoin and Ethereum. Once a Block json file’s hash is anchored to the public blockchain, all exchange data of the current Block along with all previous Blocks becomes provably immutable. With this structure, the total history of exchange transactions can be reconstructed from the public Ethereum blockchain data and public IPFS storage by any 3rd party without any reference to the exchange server.

[Block json file example]
```
{"prev":"QmZpsfnAch3FPQJKzqVjhsWSErzxLCQMniqSDkQqKyHevZ","txList":[{"ty":"OB","addr":"QmZ8cck7
ETbFNScbjKrmC7GnsZrAHLVvAtksHb7CP3HGsz"},{"ty":"OS","addr":"QmWNKUTtwy2sbq88WV5eJuN7zeUt56o
FgjkLU4EMjmFHB1"},{"ty":"T","addr":"QmRPFx9w3rCQJDS5o255Y3SjtZ7LZhjG3kGM8vGQ7hGv7z"},{"ty":"D","e
a":"0x99ca2a286099f813fa3bb970e34cdaf099337242","sy":"dUSD","am":100000000000,"di":"0xd0ae34bf4968c4
972d01fea2462ff2c519bb7fbeec4fd04af01a520bc439fda5"},{"ty":"D","ea":"0x053309e65b33b384f64a6dad7ecef6
750e07acfc","sy":"ASD","am":1000000000,"di":"90beff59-d60c-4a14-90ea-d253d67bb657"}, …]}
```
(**prev**: the IPFS address of the previous Block json file, **txList**: the list of total-ordered transactions)

1.6 Strictly Safer than Centralized Architecture

To understand why we place such faith in the hybrid architecture, it makes sense to consider the types of hacks that could be attempted on the centralized server. First, we must clarify that the hybrid architecture is not susceptible to the most prevalent attack on centralized architectures over the last few years, database destruction or exposure. Any data stored in our databases, if destroyed, could be easily regenerated following the immutable IPFS audit trail created during trading. Furthermore, our centralized server stores no private user information, so gaining access to our databases would not expose any data other than what is already stored publically in IPFS.

The only hacks of interest are restricted to undermining the execution of the centralized server. The simplest hack of this nature would pertain to crashing the centralized server. In this case, the state of the exchange could be reconstructed from the IPFS audit chain with minimal event. For the exchange server to be more secure and highly available, a failover system can be implemented where a standby secondary matching engine server follows in lockstep with the primary centralized matching engine server.

[10] With the exception that for deposits only the Ethereum transaction hash is written to the Block file.

A more advanced hack of this nature would pertain to compromising the centralized server to steal the private key of the exchange server's Ethereum account. In this case, the malicious hacker would be able to sign arbitrary withdrawal requests and assets could be stolen from the Exchange Vault contract as a hacker would possess the ability to sign invalid requests and commit them to the public blockchain. While this is a significant risk and something not applicable to a decentralized architecture, the probability of executing such an attack is outside the reasonable realm of possibility and a vulnerability of all centralized architectures. Practice would indicate this threat is accepted as "reasonable and preventable" by most security experts as all banks, credit card systems, and healthcare systems share the same vulnerability. However, the ability of the hybrid architecture to make database hacking irrelevant makes it strictly conceptually more secure than a traditional centralized architecture.

A Reasonable Compromise: Simple Tokens and Powerful Servers

In proposing this architecture, the YOSEMITE team expects to receive criticism for relying on an architecture that sacrifices consensus for speed and throughput. While this is common practice in industry for financial services companies, the cryptocurrency community holds crypto startups to a higher standard as they operate without any significant regulatory oversight and rogue crypto transactions cannot be reversed after a glitch unlike in the multistep public exchange settlement process used in the United States[11]. While the DAO hack forced an Ethereum hard fork[12], it is inadvisable to rely on the community to solve future problems by performing the same costly action. As discussed previously, implementing an exchange,that runs entirely on the public blockchain remedies these security issues but requires a reconstruction of market dynamics to accommodate significantly slower orders and non-zero cost orderflow. The YOSEMITE team does not believe this is an acceptable solution.

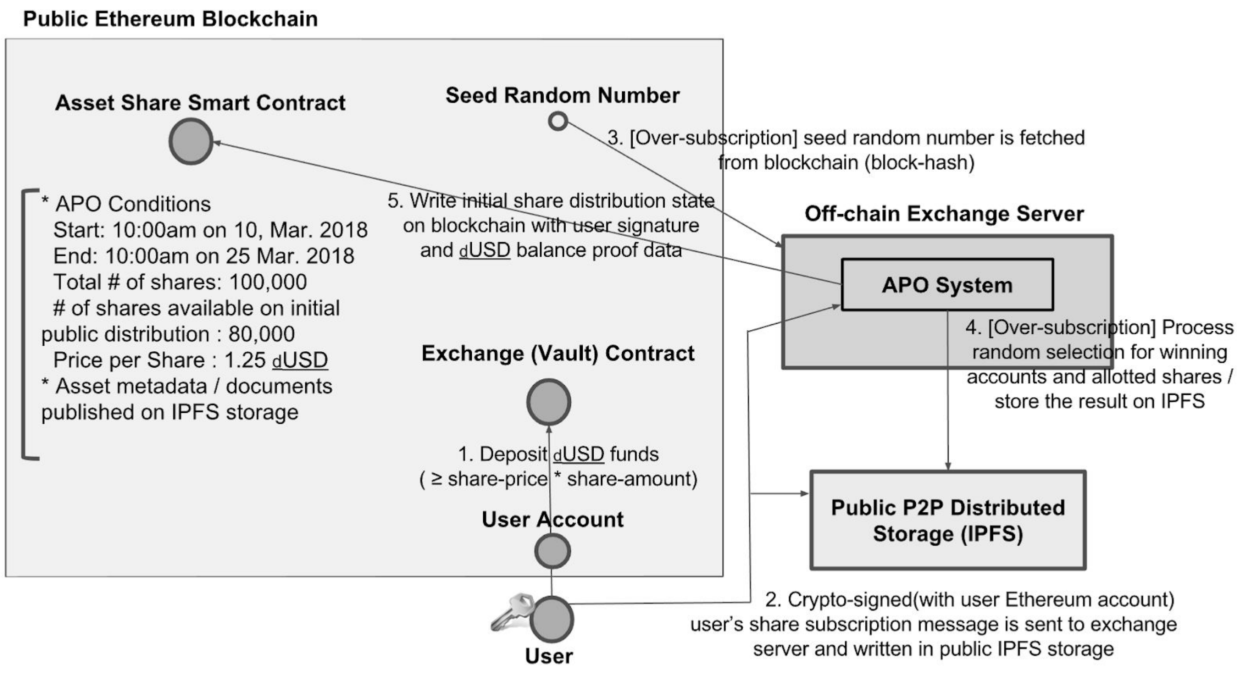
More generally, the complex nature of smart contracts has shown the security risk of building complex logic directly into the contract layer time and time again. As adoption of blockchain grows, the implications of smart contract exploits will only grow more severe. Our architecture allows the abstraction of all complicated logic to the server level meaning simple compliance with ERC20/ERC223 is all that is often required on the contract layer, when generalizing our architecture to other applications that benefit from the notion of a token.

The YOSEMITE team is currently researching additional solutions to augment the centralized exchange server in our architecture to prevent a rogue actor attack by distributing trust across many independent nodes to confirm withdrawal of assets from the exchange vault contract. However, most alternative solutions currently have a degree of security and usability that we believe to be inferior or equal to our hybrid architecture both from a technical perspective and an incentive perspective. Apart from suggesting the use of a pure delegated proof-of-stake

[11] *SPY Flash Crashes: NYSE Cancels $500 Million Worth Of Trades,* http://www.zerohedge.com/article/spy-flash-crashes-nyse-cancels-500-million-worth-trades
[12] *Ethereum Executes Blockchain Hard Fork to Return DAO Funds,* https://www.coindesk.com/ethereum-executes-blockchain-hard-fork-return-dao-investor-funds/

(dPOS) service that achieves higher throughput and lower latency for the off-chain transactions, a potential suggestion to our architecture could be requiring an additional signature on withdrawals from a delegated proof-of-stake system that provides low latency, high throughput, and decentralized consensus on correct exchange execution. However, major developers of this type of technology still warn of significant development hurdles in resiliency and security before they believe it to be production-ready. For example, a popular dPOS solution, EOS, states (as of 10/13): "This code is currently alpha-quality and under rapid development."[13] Tendermint, a common alternative to EOS, also states (as of 10/13): "*NOTE: This is alpha software.*"[14] During the interim, we believe significant utility can be derived using the tokenization function of the Ethereum blockchain, and our hybrid architecture offers the best solution to allow trading of said tokens until other consensus algorithms mature.

1.7 Initial Asset-Share Token Distribution System (for Asset Public Offerings - APOs)



Public Ethereum Blockchain
Asset Share Smart Contract
Seed Random Number
3. [Over-subscription] seed random number is fetched from blockchain (block-hash)
* APO Conditions
Start: 10:00am on 10, Mar. 2018
End: 10:00am on 25 Mar. 2018
Total # of shares: 100,000
# of shares available on initial public distribution : 80,000
Price per Share : 1.25 dUSD
* Asset metadata / documents published on IPFS storage
5. Write initial share distribution state on blockchain with user signature and dUSD balance proof data
Off-chain Exchange Server
APO System
4. [Over-subscription] Process random selection for winning accounts and allotted shares / store the result on IPFS
Exchange (Vault) Contract
1. Deposit dUSD funds (≥ share-price * share-amount)
User Account
Public P2P Distributed Storage (IPFS)
2. Crypto-signed(with user Ethereum account) user's share subscription message is sent to exchange server and written in public IPFS storage
User

[Figure.7] Initial Asset-Share Token Distribution System

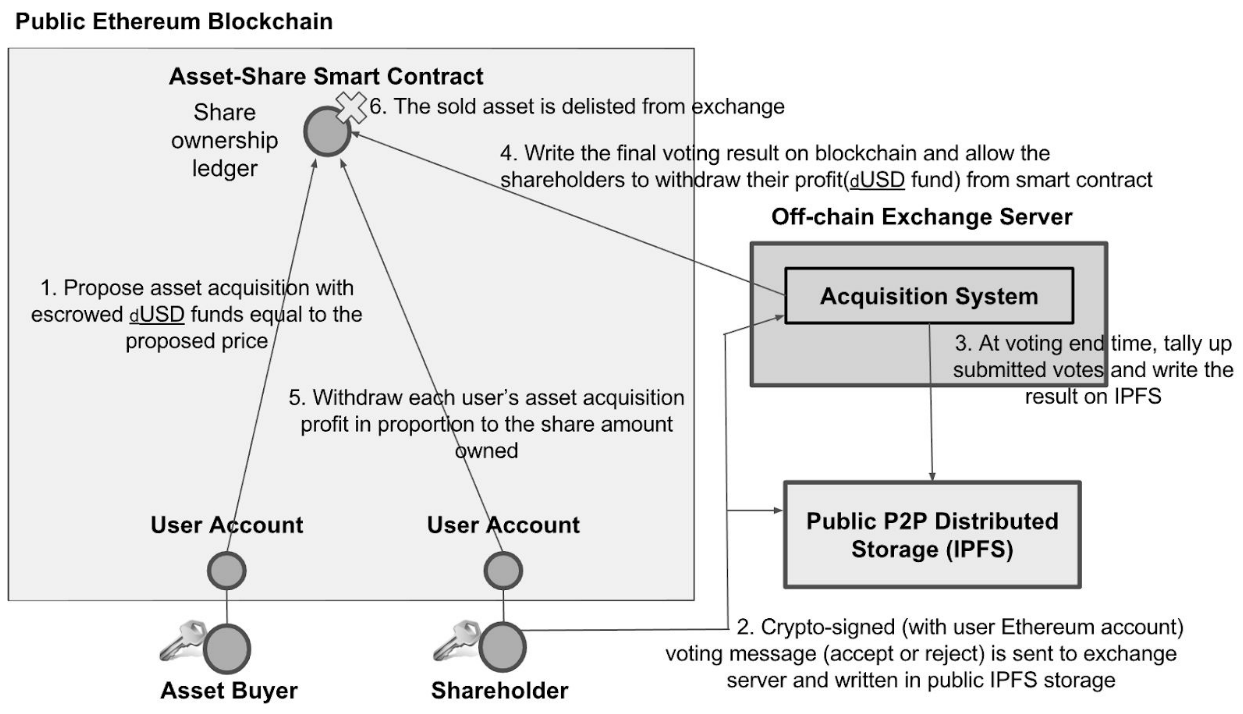
When an asset is being listed on YOSEMITE exchange[15], available Asset-Share tokens will be initially distributed to any user accounts that have sent a share subscription request message indicating the user wants to buy a certain amount of Asset-Share tokens and escrowing a corresponding amount of dUSD for the initial offering. If a user does not have enough dUSD in

[13] The EOS github repository, https://github.com/EOSIO/eos
[14] The Tendermint github repository, https://github.com/tendermint/tendermint
[15] This listing process is referred to as an Asset Public Offering or APO; see the YOSEMITE Asset Exchange Service Overview.

the exchange, that user needs to first deposit an appropriate amount of dUSD to the Exchange Vault smart contract. Each subscription request message is crypto-signed by a user's Ethereum account. Immediately after the end of the initial share subscription period, the exchange server executes transactions on the Asset-Share Ethereum smart contract to claim/write the finalized initial token distribution. These transactions include the user's signature and proof data for their current dUSD exchange balance published on IPFS storage. The accumulated dUSD funds are withdrawable by the asset listing account. As with the trading system, the end user does not need to have any ETH for gas fees to participate in initial offerings.

For the case of oversubscription[16], a fair and transparent random distribution of Asset-Share tokens among the subscribed user accounts is implemented. The seed random number is fetched from the public Ethereum blockchain (the block-hash of the Ethereum block data at subscription end time) and is used to generate a sequence of random numbers to decide deterministically the winning accounts and allotted amount of shares. The timestamp-ordered list of requested subscription messages signed by users, seed random number from the blockchain, the pseudo-random number generation method, and the final result of allotted shares among user accounts are all published to IPFS. The IPFS hash address for the file containing all relevant data is then written to the Asset-Share Ethereum smart contract, so the initial token distribution process is verifiable/auditable as provably-fair by any external party.

1.8 Asset Acquisition and Shareholder Voting System


Public Ethereum Blockchain
Asset-Share Smart Contract
Share ownership ledger
6. The sold asset is delisted from exchange
4. Write the final voting result on blockchain and allow the shareholders to withdraw their profit(dUSD fund) from smart contract
Off-chain Exchange Server
Acquisition System
1. Propose asset acquisition with escrowed dUSD funds equal to the proposed price
3. At voting end time, tally up submitted votes and write the result on IPFS
5. Withdraw each user's asset acquisition profit in proportion to the share amount owned
User Account
User Account
Public P2P Distributed Storage (IPFS)
2. Crypto-signed (with user Ethereum account) voting message (accept or reject) is sent to exchange server and written in public IPFS storage
Asset Buyer
Shareholder

[16] Oversubscription occurs when the amount of shares available for initial public distribution < requested share amount by subscribers.

Figure 8 - Asset Acquisition and Voting System

An asset (real estate, art, ...) listed on YOSEMITE exchange can be purchased in its entirety and subsequently delisted from YOSEMITE exchange.[17] To do this, the interested asset buyer proposes an asset acquisition by transferring dUSD funds as escrow to the Asset-Share Ethereum smart contract equal to the proposed acquisition price. A buyer's proposed bid price must be higher than the current total market cap of the asset being traded in exchange. If the exchange server perceives a valid acquisition proposal from the blockchain, the acquisition proposal event is announced to every shareholder through registered user email or the push notification system of the client application. Then every shareholder account can cast a vote to either accept or reject the proposed bid price within the voting period. Each shareholder makes a voting message crypto-signed with their Ethereum account. The signed messages are sent to the exchange server and published on IPFS storage as proof data of that user's vote. Before the voting period ends, any other prospective buyer can overbid the current proposed acquisition price with additional premium price. If an overbidding event occurs, the previous vote is cancelled and a new voting period is started for the new proposal. Just after the final voting period end time, the exchange server tallies up the voting results and makes them public on the Asset-Share token smart contract with the IPFS hash address for the file containing a list of links pointing to all of the submitted user-signed voting messages stored immutably on IPFS. If the proposed acquisition is accepted by a successful vote, every shareholder can withdraw a portion of the whole acquisition price in proportion to the each user's own Asset-Share token amount from the Asset-Share Ethereum smart contract. Finally, the sold asset is delisted from the exchange.

2 Digital USD - dUSD

2.1 dUSD Overview

The dUSD token is an Ethereum standard token compatible with *ERC20* and *ERC223*.

Acquiring Ether (ETH) can be a burdensome task, and transacting with ETH can introduce a significant amount of volatility risk. Therefore, rather than transacting directly in Ether, we have introduced a fiat-pegged token named Digital USD (dUSD) which will act as the primary trading currency on the YOSEMITE Hybrid token exchange platform.

Taking inspiration from the Tether model, we are bringing the same pegging process to Ethereum, where 1 dUSD is created only when users deposit 1 USD worth of assets and is destroyed when users redeem it for USD. Additionally, the dUSD system not only supports the purchase of dUSD with USD, but also ETH, which is sold for fiat in 3rd party crypto-exchanges at current market price. This one-to-one structure ensures the amount of dUSD in circulation is

[17] For a detailed description of the asset acquisition process, see the YOSEMITE Asset Exchange Service Overview.

less than or equal to the USD held in a reserve bank account of the dUSD system. USD balances held in the reserve bank account are regularly published and audited, and the total supply amount of dUSD tokens is public information on the Ethereum blockchain. This mechanism prevents the dUSD system from arbitrarily issuing or destroying dUSD tokens by holding it publicly accountable.

2.2 dUSD System Architecture

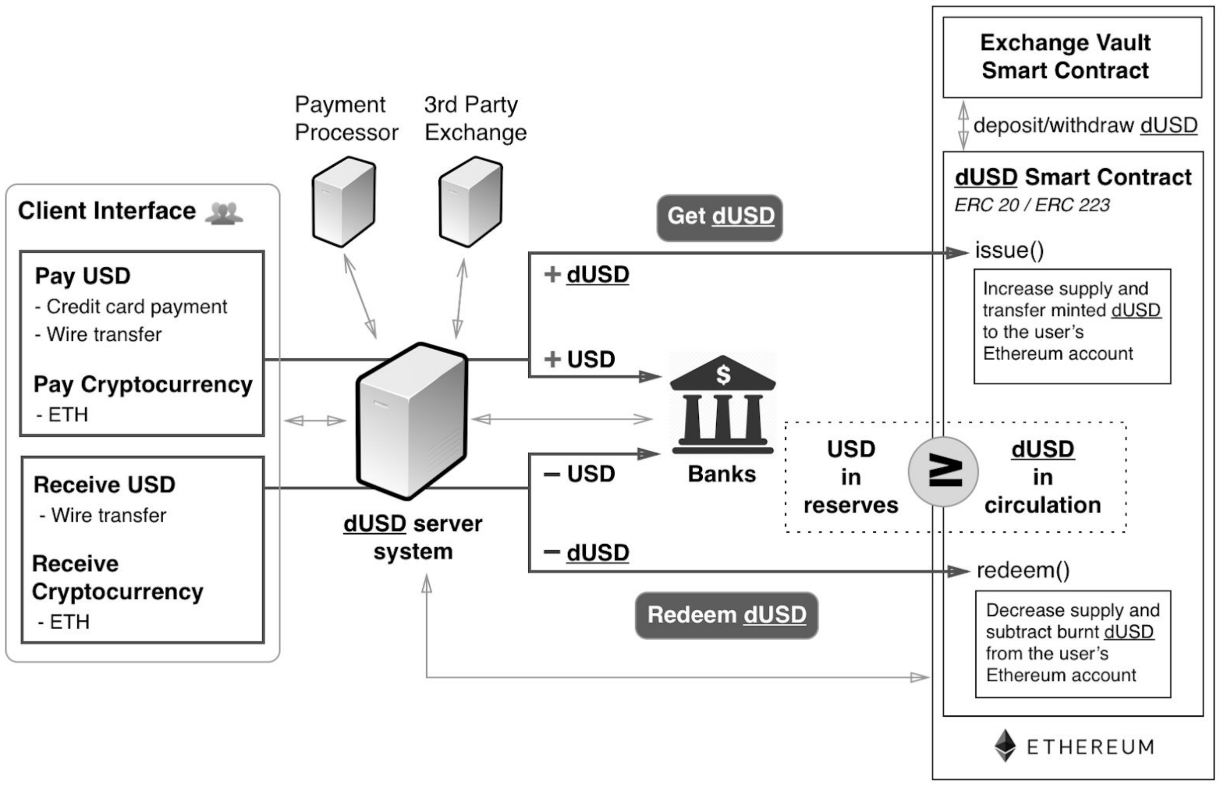


Figure 9 - dUSD action flow and system architecture

2.2.1 Issuing dUSD

- Issuance from USD via Credit Card/Wire Transfer

  When a user pays an amount of USD with a credit card or via wire transfer, an equal amount of dUSD tokens are issued to that user's Ethereum account. During this issuance, a payment transaction receipt from the external payment processor or financial institution is also provided. A hash address of the IPFS file containing this receipt data is recorded on the blockchain as proof data when the dUSD system executes an "issuance" Ethereum transaction.

- dUSD Issuance from ETH

When a user sends an amount of ETH to the dUSD token smart contract, the corresponding amount of dUSD is issued to that user's Ethereum account at the current ETH/USD market rate. In this process, the dUSD server system collects the average market ETH/USD price and available volume information from 3rd party crypto-exchanges in a transparent manner and posts this data to the user's client interface.

To issue dUSD with ETH, a user first makes a request to the server with the price and available volume data they would like to consume with their transaction. The server verifies the user's Ethereum address, price, and volume data of the request. After verification, the server cryptographically signs the request data and sends the crypto-signed data back to the user. The user then sends this server signed data with the appropriate amount of ETH to the dUSD token smart contract. Once the smart contract verifies all of the signed data (and accompanying ETH amount), dUSD is issued to the user's Ethereum address in the quoted amount. Any ETH collected from a dUSD issuance will be directly sold for USD at current market prices. The converted USD funds will be forwarded to the dUSD reserve bank account.

2.2.2 Redeeming dUSD

- Redeeming dUSD with USD via wire transfer

  Only users who have provided the proper bank account and KYC/AML compliance information can redeem dUSD for USD. When a user requests a dUSD redemption for USD, the redemption amount of dUSD is subtracted from the user's Ethereum account balance and destroyed on the dUSD token smart contract. Afterwards the corresponding amount of USD (minus any transaction fees) is sent to the user's provided bank account.

- Redeeming dUSD with ETH

  The dUSD server posts the price and available redemption volume data for ETH/dUSD based on real-time ETH/USD market data. A user interested in redeeming dUSD with ETH can make a crypto-signed request containing the price and volume values for redemption to the dUSD servers to make a "redeemDelegated" call on the user's behalf. After verification of the user's signed request message, the dUSD server executes the "redeemDelegated" function on the dUSD token smart contract with the user's signature. The contract will adjust the user's dUSD balance, and destroy the redeemed dUSD. With the "redeemDelegated" method, users do not need to pay Ethereum gas fees in ETH directly; instead the dUSD server pays the gas fees and charges the user a corresponding amount of dUSD which is taken from the redeeming amount. When this transaction is confirmed, the dUSD servers will directly sell an equal amount of USD for ETH on a partnering crypto-exchange, then the acquired amount of ETH is sent to the user's Ethereum account.

Alternatively, users who are already holding ETH can choose to make a direct smart contract transaction rather than using “redeemDelegated”. In such cases the flow will be very similar to “dUSD Issuance from ETH” flow in section 2.2.1.

2.2.3 Transferring dUSD

Since dUSD is compliant with *ERC20* and *ERC223,* it can be freely transferred to any Ethereum account (user or contract) in a safe manner. With the *ERC223* interface, the dUSD token provides "transfer to smart contract" in a safe and efficient manner. This includes deposits to the Exchange Vault smart contract in the YOSEMITE exchange system. Since the dUSD token is an *ERC* standardized token and a fiat stable coin, it can be used in any Ethereum based decentralized application (Dapp).

2.3 Proof of Reserves Process

The dUSD system’s reserve management process can be represented by the following equation:

$$dUSD_{RESERVE\text{-}BACKED} \leq USD_{BANK\text{-}RESERVE} + USD_{CRYPTO\text{-}EXCHANGE}$$

where $dUSD_{RESERVE\text{-}BACKED}$: the total amount of dUSD issued via wire transfer/credit card/ETH ,
$USD_{BANK\text{-}RESERVE}$: the total amount of USD in dUSD reserve bank account ,
$USD_{CRYPTO\text{-}EXCHANGE}$: the aggregate amount of USD held in dUSD crypto-exchange accounts

At any given time, the total Digital USD in circulation on the Ethereum blockchain is backed by an equivalent amount (or greater) of reserve US dollars in the dUSD bank account and crypto-exchange accounts (used in the issuing and redeeming of dUSD with ETH).

The total number of dUSD in circulation is contained in a publicly viewable storage variable on the dUSD token smart contract. This amount represents the current total supply of dUSD and is updated whenever the issue and redeem functions are executed on the dUSD token smart contract.

To ensure transparency in this reserve process, the above balances are published openly on a regular basis. Additionally, independent third party auditors will regularly verify, sign, and publish the underlying bank balance and statement of financial transactions for the reserve account holding funds. Similarly, third party auditors will perform the same auditing process on the crypto-exchange transaction information for all dUSD crypto-exchange accounts holding USD funds. The total supply of dUSD in circulation is always public information in the dUSD token smart contract on the blockchain.

For security, since part of the dUSD flow is centralized, it is extremely important to manage server credentials and smart contract interaction securely. In addition to best practice server side security, all Ethereum transactions which are triggered by the dUSD servers are subject to multisig authorization whereby multiple parties must agree before the transaction is executed. Multisig has proven to be one of the best modern tools for securing blockchain transactions.

2.4 Decentralized Issuance of dUSD with Asset-Share Tokens

Users can receive an issuance of dUSD by escrowing their Asset-Share tokens. At a later time, the user can redeem Asset-Share tokens by returning the corresponding amount of dUSD. The issued amount of dUSD is always less than the USD price of escrowed tokens by some margin. The exact rates and ratio of issuance will largely be determined by the historical performance of the Asset-Share token being escrowed. A very similar decentralized stable coin (bitUSD) implemented by BitShares[18] has proven to closely hold parity with USD. Stable coin issuance via Asset-Share tokens is beneficial to the YOSEMITE ecosystem because this style of issuance relies exclusively on crypto-assets held on the blockchain (in our case, Asset-Share tokens which have value backed by real world assets (real estate, art pieces, ...)) without any need for a centralized reserve of fiat funds. This makes the dUSD system more decentralized than fiat pegged currencies with centralized reserves.

$$\underline{dUSD}_{ASSET\text{-}SHARE\text{-}TOKEN\text{-}BACKED} < \sum_{i=1}^{n} (A_i \times P_i)$$

where $\underline{\mathbf{dUSD}}_{\mathbf{ASSET\text{-}SHARE\text{-}TOKEN\text{-}BACKED}}$: the total amount of dUSD issued via escrowing Asset-Share token ,
n : the number of Asset-Share escrows locked for dUSD issuance ,
$\mathbf{A_i}$: the amount of each Asset-Share token escrowed ,
$\mathbf{P_i}$: the current USD price (changing over time) of each Asset-Share token escrowed

The total amount of dUSD issued from escrowing Asset-Share tokens must always remain strictly less than the sum of the total value of all escrowed Asset-Share tokens at current prices in USD. To ensure this equation always holds, the dUSD system will execute a margin call style operation, liquidating Asset-Tokens into the market, at a price strictly higher than the total value of dUSD issued from escrowing Asset-Share tokens at the time.

$$\underline{dUSD}_{ASTB} = \sum \underline{dUSD}^{i}_{ASTB} = \sum (A_i \times P_i^E \times r_i) < \sum (A_i \times P_i^E \times m_i) \le \sum (A_i \times P_i)$$

if the token price drops below the maintenance margin, $P_i < P_i^E \times m_i$*, dUSD system automatically liquidates(sells) the escrowed Asset-Token* (A_i) *and the* $\underline{dUSD}^{i}_{ASTB}$ *is burned*

*where **ASTB** : ASSET-SHARE-TOKEN-BACKED ,* $\underline{\mathbf{dUSD}}^{\mathbf{i}}_{\mathbf{ASTB}}$ *: the amount of dUSD issued via each Asset-Share escrow,*
$\boldsymbol{P_i^E}$: the USD price of an Asset-Share when it is being escrowed ,
$\boldsymbol{r_i}$ *: the rate at which dUSD is issued based on the price* P_i^E *,* $\boldsymbol{m_i}$ *: the maintenance margin rate,* $\boldsymbol{0 < r_i < m_i < 1}$

The dUSD system provides a hybrid-style model supporting both reserve-backed and Asset-Share token-backed pegging methods.

$$\text{Total supply of } \underline{dUSD} = \underline{dUSD}_{RESERVE\text{-}BACKED} + \underline{dUSD}_{ASSET\text{-}SHARE\text{-}TOKEN\text{-}BACKED}$$

[18] BitShares white paper - http://www.the-blockchain.com/docs/BitShares%20A%20Peer-to-Peer%20Polymorphic%20Digital%20Asset%20Exchange.pdf.

The total supply of dUSD tokens in circulation is the sum of dUSD tokens backed by USD reserves and dUSD tokens backed by Asset-Share escrows.

3 Multi-Platform Clients of YOSEMITE Hybrid Exchange

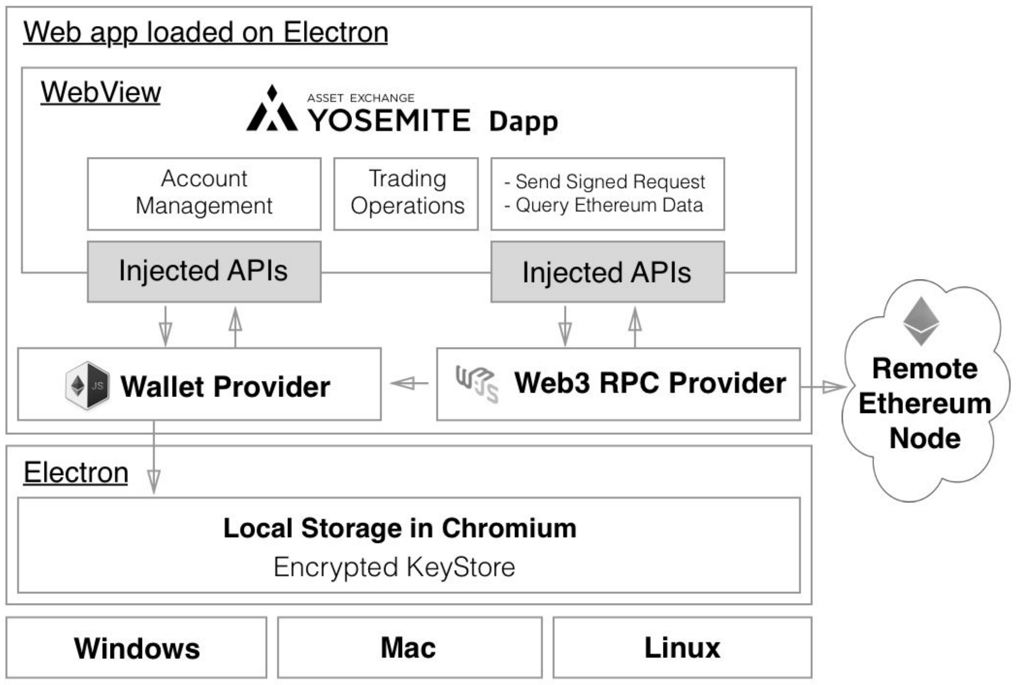


Figure 10 - Application Architecture

3.1 Current Ethereum Client Problems

Although using Ethereum offers enhanced security and simplifies trading implementation, introducing apps on Ethereum has always been hard for end-users. The two main barriers to good user experience are:

- downloading and syncing chain data before use
- using cryptographic keys with an Ethereum wallet

For mass adoption, we need to remove these hurdles so that average users can use Ethereum-based apps without having to understand the underlying technologies.

3.2 Solutions

3.2.1 Connecting to a Remote Ethereum Node

To provide the best user experience, we have decided to maintain and connect the YOSEMITE exchange Dapp to a remote Ethereum node so that users are able to use the YOSEMITE exchange services without running nodes of their own. Querying Ethereum data is done through a connected node by simulating existing normal web services, but storing user credentials and signing transactions will be handled locally for security reasons. Some have argued against this kind of remote node architecture since it could compromise security or lead to delays in network access since it provides a single point of failure. However, we can mitigate these problems by providing a cluster of Ethereum nodes maintained and secured specifically for YOSEMITE exchange users. Nonetheless, if users prefer, they may alternatively choose from a list of available public nodes.

3.2.2 API Injection In a Sandboxed Environment

The primary goal of the YOSEMITE exchange client is to provide an easy-to-use app within a secure environment. Currently, to use Dapps on Ethereum a user must download Mist/Parity or use the MetaMask extension with Chrome browser. Of course the YOSEMITE exchange Dapp would run on these web platforms with web3 support, but they require a general understanding of cryptographic keys and management of an Ethereum wallet. Aragon offers an Electron[19] powered client; however, it also integrates MetaMask inside an iframe for Ethereum-related tasks leaving the same account management issues.

Taking inspiration from Aragon's well-designed client architecture, but without integrating a full-fledged MetaMask, we have only exposed a minimum set of APIs which are necessary to interact with Ethereum, thereby allowing the Dapp UI developers to build their own account creation, account management, transaction signing, and transaction-sending UI.

As you can see in Figure 10, we added an Electron specific element <WebView> to host the Dapp as guest content. The hosting app injects web3 - Ethereum standard API and wallet APIs into the webview before the Dapp is loaded, thus the embedded content can create wallets and manage encrypted private keys within the hosting app context.

The default app loaded on Electron is the hosting app which is running on a renderer process. Unlike an iframe, the webview we added runs in a separate renderer process keeping the containing app safe from the embedded content. This architectural design allows for secure hosting of any Dapp while maximizing custom action flows for Ethereum integration. With this we can provide custom UX and security to the average YOSEMITE exchange user.

[19] Electron is a framework for creating native applications with web technologies.

In the future, we will open-source our client application project once it is ready for public contribution. An Electron package with this architecture of API injection could prove very useful to the Ethereum development community.

3.2.3 A Packaged Client Application

By simply downloading and installing a packaged desktop application, users are able to trade asset shares just like any other traditional web application. The client will run on Windows, Mac and Linux.

3.3 Light Client on Mobile

On our first iterations, we tried interfacing with Ethereum by porting geth with the “light client” mode enabled on both iOS and Android for mobile support. A popular Ethereum mobile client, *Status.im*’s open source project has aided greatly in successfully completing this task. However, the light client is still an experimental feature and needs some work before it is ready for production. Until the light client becomes production-deployable, we will take a similar approach to what we have done with the desktop client environment.