University of Zurich[UZH]

MASTER PROJECT – Communication Systems Group, Prof. Dr. Burkhard Stiller

Design and Implementation of a Decentralized Token Exchange Platform

Ankan Ghosh, Corey Bothwell, Saiteja Pottanigari
Zurich, Switzerland
Student ID: 19-763-085, 19-767-672, 19-763-093

Supervisor: Prof. Dr. Burkhard Stiller, Sina Rafati
Date of Submission: March 15, 2022

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Master Project
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

Some of the most prominent applications of decentralised finance are decentralised exchanges, which facilitate cryptographic token exchange without any trusted intermediaries. Existing implementations of decentralised exchanges generally fall into one of two categories: Automated Market Makers (AMM) or Non-Custodial Orderbooks (NCO). These approaches pioneered the decentralised exchange model but lack important functionalities and generally offer a degraded user experience (UX) in comparison to the most popular centralised exchange platforms.

We present a novel architecture for a decentralised exchange protocol that attempts to build towards the UX standard offered by leading centralised trading platforms and runs completely in the user's web browser. We construct a peer-to-peer network of traders that communicate directly with each other's browsers via WebRTC, a protocol for direct peer communication, without any intermediate server. Peers pass order information to one another via a series of signed messages, which are broadcast to the network through a gossip protocol implemented via PubSub channels. Orders are sent to the chain by a decentralised network of validator nodes, which are users that opt-in to cover the transaction fees required to commit orders to the chain. We use a Layer 2 Ethereum scaling solution in order to facilitate high order throughput, and we integrate a Graph Protocol Subgraph implementation in order to index smart contract events in our protocol.

We perform some initial load testing and describe the benefits of such an architecture while additionally presenting some exciting avenues for further development. Finally, we describe a novel technique for sending signed messages on behalf of a user without knowledge of their private key, using a technique we developed over the course of the project called *Delegated Signing*.

Acknowledgments

We would like to express our sincerest gratitude to our supervisor Dr. Sina Rafati for providing invaluable feedback and support during the course of our master's project. We would also like to thank Eder Scheid for assisting us with all of our system infrastructure requests. Lastly, we would like to thank Prof. Dr. Burkhard Stiller for the opportunity to realize our master's project with the Communication Systems Group at the University of Zurich.

Contents

Chapter 1

Introduction

1.1 Motivation

Centralised exchanges (CEX's), such as Coinbase[1], Kraken[2], and Binance[3] dominate cryptocurrency trading activity. The use-of-use, speed, and peace-of-mind offered by centralised exchanges allows an easy on-ramp and convenience for retail traders.

Where centralised exchanges bring convenience, they also present certain challenges for traders. Namely, centralised trading congregates activity into a single centralised point-of-failure, and exchanges must maintain custody of trader's assets on their behalf. Exchanges can suffer from this weakness in a multitude of ways, often manifested in disruptions, unavailabilities, or in the most severe cases, hacks and asset loss.

In contrast, decentralised exchanges (DEX) allow traders to exchange while maintaining complete custody over their own assets. At the time of this writing, decentralised exchange methods are dominated by two different approaches, the *Automated-Market-Maker* (AMM) and *Non-Custodial Orderbooks* (NCO). We investigate both of these approaches in more detail in later sections.

While decentralised exchange methodologies allow users to retain complete custody over their assets, they come with their own set of downsides, most notably high cost and shallow liquidity. We expect these issues to alleviate over time as blockchains continue to grow in popularity and methods to reduce network fees, such as Ethereum's transition to a Proof-of-Stake model, continue to evolve[24].

However the most salient drawback to DEX methods for the end user is that they fail to match the depth of user-experience and ease-of-use offered by centralised exchanges. Indeed some informal estimates pegged 2021 aggregate centralised trading activity at approximately USD 14 trillion, while decentralised methods accounting for a measly USD 67 billion in comparison[48].

[1] `https://www.coinbase.com/`
[2] `https://www.kraken.com/`
[3] `https://www.binance.com/`

This project attempts to build on and further develop the current ideas behind existing DEX methods towards a decentralised exchange protocol which closely emulates the functionality and user-experience of CEX's, yet still allows uses to maintain full custody over their assets.

We present a novel approach to decentralised token exchange that communicates order information via a peer-to-peer network running completely in the user's web browser. We additionally incorporate functionality that empowers users to easily create their own tokens allowing anyone to create assets on demand. In doing so we expand the functionality offered to the user beyond what CEX's can provide[4].

Finally, we elucidate an important & novel technique developed during the course of this project that allows blockchain applications to provide signing authority on behalf of users *without knowledge of their private key*. We term this technique "delegated signing" and consider it a novel and powerful tool for decentralised application developers.

1.2 Use Case

Our system incorporates two primary activities carried out by actors in the crypto-token markets, **fundraising** and **trading**. We look at the stakeholders inherent in each activity and outline how our system proposes to facilitate these use cases. We offer some context and background to how these are traditionally performed, and show how our system builds on this in a novel way. Thereafter we enumerate a few of the guiding principles we followed during the course of this project's development and then turn our attention to some technical background.

1.2.1 Fundraising

Cryptographic tokens can be used to represent shares of an entity or organisation which lends them well to fundraising. In the recent years the concept of an *Initial Coin Offering* (cf. Section 1.3.21) has become a well known method of utilizing a token sale to raise funds.

While the subject of increasingly more regulatory attention, barriers to entry are very low, which allows organisations to generate funds with relatively little red-tape when compared to traditional fundraising methods.

Our system builds on this concept and allows any user to create tokens, both fungible and non-fungible, with relative ease. We associate a canonical, immutable ID with each token set owner and associate metadata with each individual token.

[4]We note that CEX's are not prohibited from allowing their users from creating tokens by any *technical* reason. It is simple practicality that encourages them not to. Allowing user's to create their own assets and thereafter trade them would necessitate maintaining orderbook infrastructure and importantly liquidity for an arbitrary number of tokens.

In combination with our trading functionality, we provide users a complete end-to-end solution for creating and selling token shares to facilitate a fund-raising use case. Our approach is relevant to any entity that might wish to raise funds in a completely decentralised manner. Users that wish to invest in a third-party token offering can easily do so by simply adding the token information and submitting a buy order.

1.2.2 Trading

Our system facilitates trades by allowing users to send signed order messages to each other via a peer-to-peer gossip protocol. Orders are propagated throughout the peer-to-peer network without any centralised server. Once two orders are found that are a suitable market match, they can be submitted to the blockchain where they can be validated and executed against one another.

We allow users to elect to become special "validator" nodes in order to observe the orderbook and perform this matching procedure. Additionally, validators commit matched orders to the chain and cover the associated gas fees in exchange for a commission fee on the order amount.

Our concept thus innovates on traditional off-chain matching methods by essentially distributing responsibility for order submission to the validator nodes, which is a role open to anybody willing to pay the associated gas fees. Traders can easily drop-in and drop-out of the network and be sure in the knowledge that their assets remain in their own wallet.

1.2.3 Development Principles

We settled on a few main principles which we used to guide our development, namely that the system should:

1. Avoid any form of centralisation wherever possible
2. Never take control of user's assets
3. Prioritise ease of use and a simplified user experience as much as possible
4. Attempt to utilise the web platform, specifically the capabilities of the web browser, as much as possible
5. Be accessible by simply by visiting a website. The user should not need to download any software via the command-line, an installer, or otherwise

1.3 Technical Background

We now turn our attention to some important technical background and terminology, both regarding blockchains in general as well as specific techniques and implementations that our solution builds on top of[5].

1.3.1 Ethereum and the Ethereum Virtual Machine

Ethereum is a decentralised network protocol that facilitates the execution of arbitrary Turing-complete machine instructions in what are called ”smart contracts”. These smart contracts can represent any sort of arbitrary state transitions between participants in the network, and participants in this case can mean nodes in the network, or contracts themselves[16].

It is helpful to think of Ethereum of the ”infrastructure layer” runtime that empowers decentralised applications to be built on top of it.

Ether is the native currency token of the Ethereum network and is used as an independent asset as well as the medium of exchange to validate transactions in the network by paying what is called ”Gas”. Gas is basically a transaction fee paid in Ether to ”miners”, who validate transactions and further propagate the blockchain[16].

1.3.2 Smart Contracts

In the context of Ethereum, smart contracts are computer programs which run immutable and run deterministically. They are immutable because the only way a smart contract which is deployed to Ethereum can be changed is if a new instance of the contract is deployed. Furthermore, they are considered deterministic because they result in the same output given that they are run on the same state of the blockchain [3].

1.3.3 Solidity

Ethereum smart contracts are almost exclusively written in the Solidity programming language. Solidity is a programming language which has been specifically designed for Ethereum. While there are other programming languages which are available, Solidity has become the de facto standard on Ethereum. Along the likes of C++ and Java, Solidity is also an imperative programming language, meaning that it is used to write programs which consist of a set of procedures that together form the logic and flow of a program. It is sometimes hard to write code which executes exactly as intended in imperative languages as side effects are hard to predict and identify. This poses one of the greatest threats in smart contracts, as any unintended side effects of a smart contract could be found and exploited, which in almost all cases can lead to a potential monetary loss [3].

[5]Portions of this section were reproduced from one of the author’s own works, with the permission of all co-authors, see [13].

1.3.4 Ethereum Standards (EIP)

The Ethereum Protocol maintains a variety of standards and interfaces that regulate common application patterns, protocol development, and other development specifications[4].

EIP's can be additionally referred to via the name "ERC(Ethereum Request for Comments)", and we will use these interchangeably throughout this document. The semantic differences between these two terms are not relevant to this work.

ERC-20

ERC-20 facilitates the standard community interface that governs *fungible* tokens on the Ethereum blockchain. Fungible tokens are identical between each other[17].

ERC-20 is a very low level interface, designed to be general and support any type of fungible token. A token in this case can represent a typical coin, but also other things, like shares in a firm, or votes in a governance scheme.

ERC-721

In contrast to ERC-20, ERC-721 facilitates the usage of *non-fungible* tokens, which are tokens that represent an individual unique entity, not comparable with a generalised pool[21].

ERC-721 compatible tokens can represent unique claims on assets, pieces of digital artwork, or any other sort of unique asset. They can be thought of as "deeds" to borrow the phrase from the original specification[21].

ERC-1155

ERC-1155 can be thought of as a generalised abstraction "laid over" ERC-20 and ERC-721. ERC-1155 is officially known as the "Multi Token Standard" and facilitates the creation of an arbitrary amounts of fungible and non-fungible tokens, all under a single umbrella smart contract[46].

Our system supports ERC-1155, and therefore supports both fungible and non-fungible tokens.

EIP-712

EIP-712 is an Ethereum standard for specifying structured signed data[12]. One of the core tenets of the cryptographic fundamentals underlying the Ethereum blockchain is the ability of wallet owners to generate digital signatures via their private key (cf. Section

1.3.8). These digital signatures are used to authorise the transactions issued to the chain are valid, but they have other uses as well.

Numerous blockchain based applications can use digital signatures for other methods of verification, both on and off-chain[13]. In the early days of Ethereum, these signatures were performed only on simple bytestring data which posed usability issues for client-facing applications, as it was impossible for a user to understand the message they were signing without decoding the binary data.

EIP-712 improves on this situation by providing a mechanism for signing *structured* data stored in simple object records with defined data types. By allowing this, client-facing apps can present a more user friendly representation of the data to be signed.

1.3.5 JSON-RPC

JSON-RPC is a general protocol for defining Remote Procedure Calls encoded into JSON. In the context of Ethereum and the Ethereum Virtual Machine, clients communicate with the Ethereum blockchain by making JSON-RPC calls to an available blockchain node. The node then submits the transaction to the chain on the client's behalf.

1.3.6 Application Binary Interface (ABI)

An Application Binary Interface (ABI) is a general term describing how software systems can interact with an application executable at the binary level. In the context of Ethereum and the EVM, ABI's define an interface that client applications can use when making JSON-RPC calls to a deployed smart contract. Client applications use the ABI to correctly reference and call available functions defined in the deployed contract, along with any required arguments.

1.3.7 OpenZeppelin

OpenZeppelin is a software organisation delivering blockchain tooling, auditing, and code libraries[6].

OpenZeppelin libraries are a well known and trusted source which implement base implementations of popular Ethereum standards, like ERC-20 and ERC-721[50]. By utilising an OpenZeppelin solidity library, developers can be sure that their code uses secure, audited, and performant code at its base level, and that they only need to add the desired functionality on top of it.

Using OpenZeppelin libraries in this way speeds up development and helps developers write code with less bugs. We use OpenZeppelin libraries to build our smart contracts. See Chapter 4 for exact implementation details.

[6] `https://openzeppelin.com/`

1.3.8 ECDSA

The Elliptical Curve Digital Signature Algorithm is the digital signature algorithm used by the Ethereum Virtual Machine and serves to provide proof that owners of a wallet have authorised transactions originating from that wallet. (They also serve two further purposes, to establish non-repudiation and unmodifiability, see [3] for more details) Given a signature, a serialised transaction, and public key of the supposed originator of the transaction, the ECDSA allows for verification of a transaction, i.e. to say "Only the owner of the private key that generated this public key could have signed this transaction" [3].

1.3.9 Merkle Trees

Merkle Trees are a type of datastructure represented by a tree, where each node in the tree is represented by a hash of its own content, and the hashes of all of its child nodes. The hash of every node is then dependent not only on its own content hash, but also on the content hash of its children. Changing the children changes their hash, and the hash of all of their parents respectively. Thus, a root node hash can be said to represent a unique tree.

The technique was originally pioneered by Ralph Merkle as an efficient method for verifying variable length signature chains[38].

Merkle Trees lend data *content addressability* via its unique hash (including children), and thus find a wide variety of uses in diverse systems from blockchains to git. Our deployment solution uses Merkle Trees to provide our application in a decentralised manner without a central file server.

1.3.10 Diffie-Hellman Key Exchange

Diffie-Hellman Key Exchange is a method of public key cryptography which allows two parties to exchange secure messages with one another over an insecure channel[18]. Messages between each party can be encrypted via a shared secret, derived from a common base and each party's individual private key. The shared secret can be computed by each party without revealing their private decryption key. It is computationally infeasible for an eavesdropper on the communication channel to derive a private key of a participating party.

A visual overview is shown in Figure 1.1[7]. For more background see [18].

We use a modified version of Diffie-Hellman Key Exchage in order to safely store encrypted signer keys publicly on the Ethereum blockchain (cf. Section 3.4).

[7]Image courtesy of Wikipedia - `https://en.wikipedia.org/wiki/Diffie-Hellman_key_exchange`

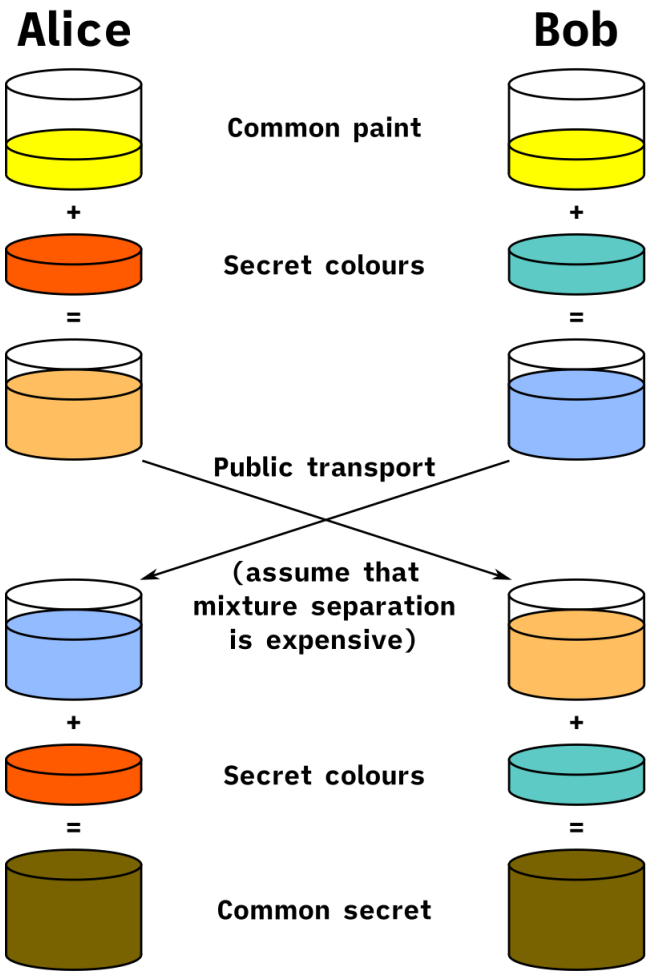

Figure 1.1: A visual interpretation of Diffie-Hellman Key Exchange.

1.3.11 Browser-based Signature Authorities

Browser based signature authorities are software extensions that serve as a signing authority and manage a user's private keys on their behalf. They run in a web browser as an extension and provide users very easy access to browser based blockchain based applications. Metamask is an example of a browser based signature authority[39].

Browser based signature authorities are important because they facilitate an easy interface for applications to request signatures from the user. Importantly, browser based signature authorities never reveal the private key to any application. This is an important security feature, but it also means that users need to manually approve every signature request.

1.3.12 WebRTC

WebRTC (Web Real Time Communication) is a communication standard that enables browser-to-browser communication without an intermediary server[28]. Browsers negotiate initial connection data via an intermediate relay server (called a STUN server) and then communicate directly thereafter. Our system uses WebRTC to facilitate peer-to-peer communication.

SDP

Session Description Protocol (SDP) is a standard for describing the content of a networked multimedia connection such as resolution, formats, codecs, encryption, etc, so that peers in a connection can understand the data being transferred between one another[40]. SDP is in essence the metadata describing the content of a connection.

ICE

Interactive Connectivity Establishment (ICE) is a technique for establishing a direct connect between peers in a networked system, especially considering the presence of Network Address Translation (NAT)[34]. It is most commonly found in VoIP and Peer-to-Peer systems.

NAT

Network Address Translation (NAT) is a technique adopted to expand the span of potential devices connected to the internet under IPv4[40]. Network Address Translation converts a singular public IP address (for instance, from a router) into individual private IP addresses for connected devices. By doing so, NAT obviates the need for each device to have its own unique public IP address. NAT can pose problems when attempting to dial peers in a peer-to-peer network, as the public address and port combination of the peer needs to be translated to the correct private IP address at the discretion of the router/translator.

STUN & TURN

Session Traversal Utilities for NAT (STUN) is a protocol for discovering peers behind NAT[40]. A special server known as a STUN server will help negotiate the connection and determine if peers can be reached behind their NAT implementation.

In some cases, it is not possible to directly connect to a peer due to what is known as Symmetric NAT, a form of NAT where peers can only accept connections from peers previously connected to. In a case where both peers are behind a Symmetric NAT, a direct connection cannot take place. Instead, peers route their communications through a relay server known as a TURN server (Traversal Using Relays around NAT), which relays the respective communication between the peers[40].

1.3.13 Web Workers

Web workers are a browser-based threading solution written in Javascript that allow applications to run isolated resource-intensive or long running code in a sandboxed background environment[41]. As Javascript is generally single threaded, web workers are designed to assist developers in running compute-heavy procedures isolated from the main application to prevent thread blocking. Web workers communicate with the main application process vie messages passed between the main application and the worker.

1.3.14 Publish/Subscribe Architecture (PubSub)

PubSub architecture is a generalised architectural pattern designed for dynamically delivering messages between groups of software entities. PubSub as a technique is useful because it provides a comfortable abstraction over a more direct messaging implementations.

Without a PubSub architecture, software implementations designed to pass messages between entities might manually route messages to interested peers/components[8] after querying for their location. Entities might need to query before every message delivery to determine if new entities are interested in the message. This can quickly entail a lot of complexity.

PubSub is designed to ease some of these issues by providing ”topics” or ”channels” in which entities register, or ”subscribe” themselves, as interested in. When a peer broadcasts on a specific topic, all subscribed peers are delivered the published message. Peers can dynamically drop in/out of channels at will, which lends the system flexibility. External implementations have a greatly simplified interface for sending and receiving messages. Figure 1.2 demonstrates an example of how a peer in a PubSub network might broadcast a message to all interested peers.

[8]PubSub is a generalised pattern, it can be applied to separate peers/hosts, packages, or even individual software components

Figure 1.2: An example of a PubSub network[35]. Here all peers are interested in a generic topic "X", shown by the light blue shading. When a peer publishes a message (shown in purple), all subscribed peers eventually receive the message. Recall that PubSub is a generalised pattern, so the specific implementation details are not relevant

1.3.15 Distributed Hash Tables

Hash tables provide a lookup service to data records or memory objects via an indexed key. Each key points to a location in memory where the desired data can be accessed. In a traditional setting, a hash table is maintained as a singular entity, often managed by an application library or a database.

It is also possible to distribute the hash table over multiple nodes. In this scenario, nodes take responsibility over a portion of the hash table and persist the associated data records for every key in their responsible range. Nodes also maintain metadata information about a subset of other nodes in the network. When a client requests content from the distributed hash table, they supply a key to an arbitrary node. If the requested node happens to be the responsible node for the requested key, the node returns the content immediately. Otherwise the node routes the request along to other peers most likely to have the requested key.

With clever routing techniques, client nodes can be guaranteed to receive the requested content in $O\log(n)$ time, that is, logarithmically in the size of the table. One of the most popular implementations of distributed hash tables is the well-known Kademlia DHT, popularised by Maymounkov & MaziÃ¨res at New York University[37].

1.3.16 Protocol Buffers

Protocol Buffers are a language neutral, platform-neutral specification for serializing structured data[27]. Protocol Buffers are language agnostic and encode data into a binary format. Their advantages include backward compatibility and a tiny size compared to more human-readable encoding schemes.

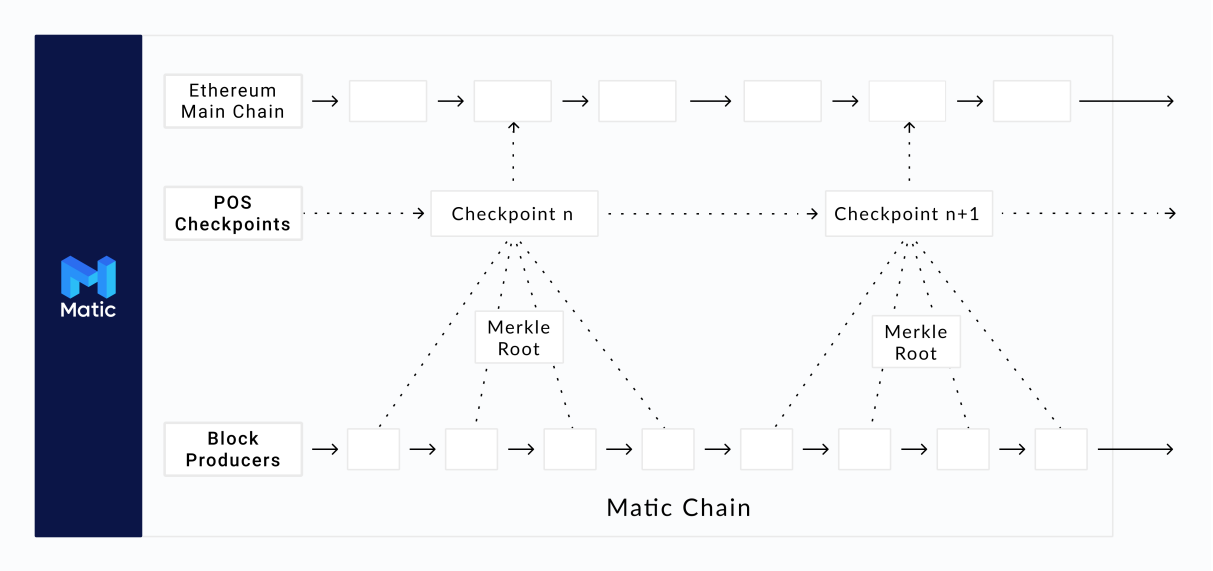

Figure 1.3: The polygon (formerly matic) scaling architecture

1.3.17 Layer 2 Solutions

Layer 2 is a general term for a set of scaling techniques for blockchains. Layer 2 protocols usually provide some sort of separate ledger ”off-chain” and then use a variety of techniques to commit the results in batches to the blockchain.

The off-chain ledgers can be on their own blockchain with relaxed decentralisation requirements, some sort of private state-channel, or a bundle of transactions secured by a zero-knowledge cryptographic proof. There exist other techniques. For a complete overview of various layer 2 scaling techniques, we direct the reader to [25].

Plasma Sidechains

Plasma sidechains are a specific layer-2 scaling technique. Plasma sidechains were conceptualised by Joseph Poon and Vitalik Buterin[44] and are concurrent child blockchains derived from a root blockchain. Transactions in the child blockchains can use relaxed consistency requirements and organise their specific sub-transactions into an aggregated overall transaction represented by a Merkle Tree. This overall transaction is then submitted to the chain in checkpoints, with cryptographic proofs allowing network participants to verify the validity of the submitted overall transaction.

Our layer-2 scaling solution, Polygon (formerly known as Matic), uses Plasma Sidechains in order to scale transaction throughput [43]. See Figure 1.3 for an overview.

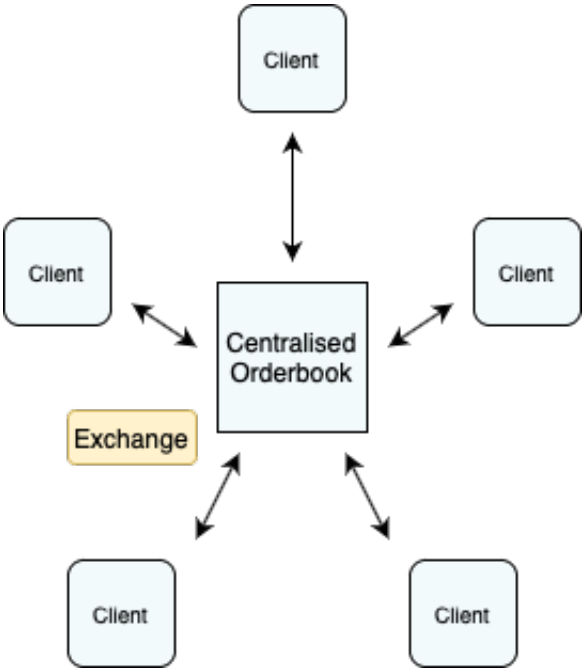

Figure 1.4: A simplified architectural overview of centralised exchanges

Zero Knowledge Rollups

Zero Knowledge Rollups, or ZK-Rollups, are an alternative scaling proposal for public blockchains and aim to ”bundle” a large set of transactions into one transaction which is then submitted to the chain[26]. Users broadcast their transactions off chain to ”relayers”, who bundle large amounts of transactions together and derive a hash representing the state delta between the current chain state and the aggregated bundle state changes. Relayers submit the transaction bundle along with the hash to the blockchain at periodic intervals at which point the individual transactions are definitively committed.

ZK-Rollups rely on a sophisticated cryptographic method known as zero-knowledge proofs to proof veracity of the bundled transactions[9].

1.3.18 Centralised Exchange Platforms

Centralised exchange platforms allow users to exchange tokens with one another in a manner that replicates brokerages present in traditional finance. (Trad-Fi) They maintain a centralised ledger of transactions between participants and use in-house algorithms to match orders between traders.

Centralised exchange platforms *by definition* maintain custody over the user’s assets on their behalf, as the exchanges need to be able to exchange token holdings between various participants in real-time.

[9]See [45] for a well-known informal introduction to zero-knowledge proofs

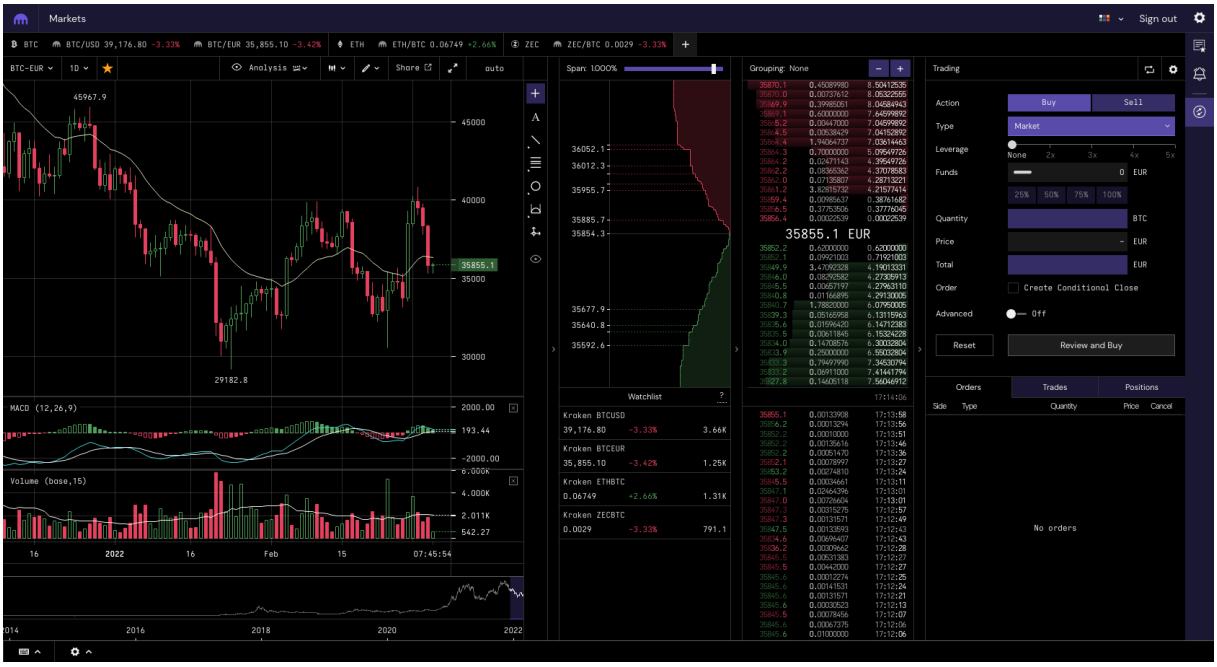

Figure 1.5: An example of a typical centralised exchange interface (www.kraken.com). Centralised exchanges offer large UX advantages over current decentralised protocols

1.3.19 Decentralised Exchange Protocols

In contrast to centralised exchange platforms, decentralised exchange protocols offer a manner of trading that does not rely on a central ledger or taking custody of the user's assets. There are two main models in use today, Automated Market Makers (AMM) and Non-Custodial Orderbooks (NCO).

Automated Market Makers (AMM)

Automated Market Makers, such as Uniswap[10], work by pooling two tradeable assets together into pools via a smart contract. First, participants in the pools commit an asset pair to the pool and lock their assets for a specified amount of time, or according to some conditions.

Traders can then interact with the staked tokens in the pool, exchanging one token for another, or vice versa. The ratios of exchange (it would not be 100% accurate to say "price", as the tokens are exchanged in proportion directly to each other) are set by the rules of the smart contract and usually vary according to the relative amounts of each asset in the pool.

It is the staker's tokens that are exchanged when traders interact with the pool. To incentivise this, stakers are usually compensated for pooling and locking their assets with some sort of reward system that grants them new tokens, or a fee for every transaction

[10] `https://uniswap.org/`

that interacts with the pool. These new tokens can often be called "LP" tokens, or liquidity pool tokens. When traders withdraw their token pair, they receive their tokens in the new ratio set by the smart contract according to actual trading activity.

These types of Automated Market Makers were a brilliant innovation that helped facilitate the advent of decentralised trading as well as spark the greater "DeFi" ecosystem. (See below)

Nonetheless, they do maintain a few drawbacks:

- Liquidity Pool smart contracts involve complex logic and a greater amount of transitions in the overall Ethereum state than a simple transfer of tokens. This leads to a high amount of gas required for traders to execute against the pool.
- Stakers can be victim to "impermanent loss", which is an economic loss in overall portfolio value caused by committing tokens into a liquidity pool. Assume the market price of one of the tokens in the pool changes. The associated ratio between the two tokens will also change, and the appreciating token will then represent a smaller portion of the pool. (As it increases in value in relation to the other token, and market dynamics ensure that this occurs across all exchanges) When stakers go to withdraw their staked tokens, they receive a different proportion than originally committed, and this new proportion can represent a lower total value than the funds they originally committed.
- Traders are dependent on stakers committing the funds they want to trade. In lower liquidity markets, this could mean that pools are very small, or even non-existent. Smaller pools mean less liquidity and more volatile ratios between assets.

Non-Custodial Orderbooks (NCO)

Non-Custodial Orderbooks are another method of decentralised exchange.

Users submit signed messages representing a willingness to trade certain assets under certain conditions. For instance, a user may submit a message representing "I am willing buy 10 of token X at a price of 1 ether". (The actual message would use a standard encoding) By producing an ECDSA signature of the message, the message can be verified to have originated from the signer.

Smart contracts can read these order messages and then perform exchanges between parties accordingly. For instance, a smart contract might receive two matched orders, verify their respective signatures, and then swap tokens between their originators. Users usually need to approve the contract to swap tokens on their behalf.

Orders are submitted to the chain via relayers. How relayers are designed, implemented, and compensated are dependent on the respective protocol.

NCO's can be very diverse, as the only categorical requirement is that they do not take custody of the user's assets, relying instead on the signed proof. For instance, dYdX is a

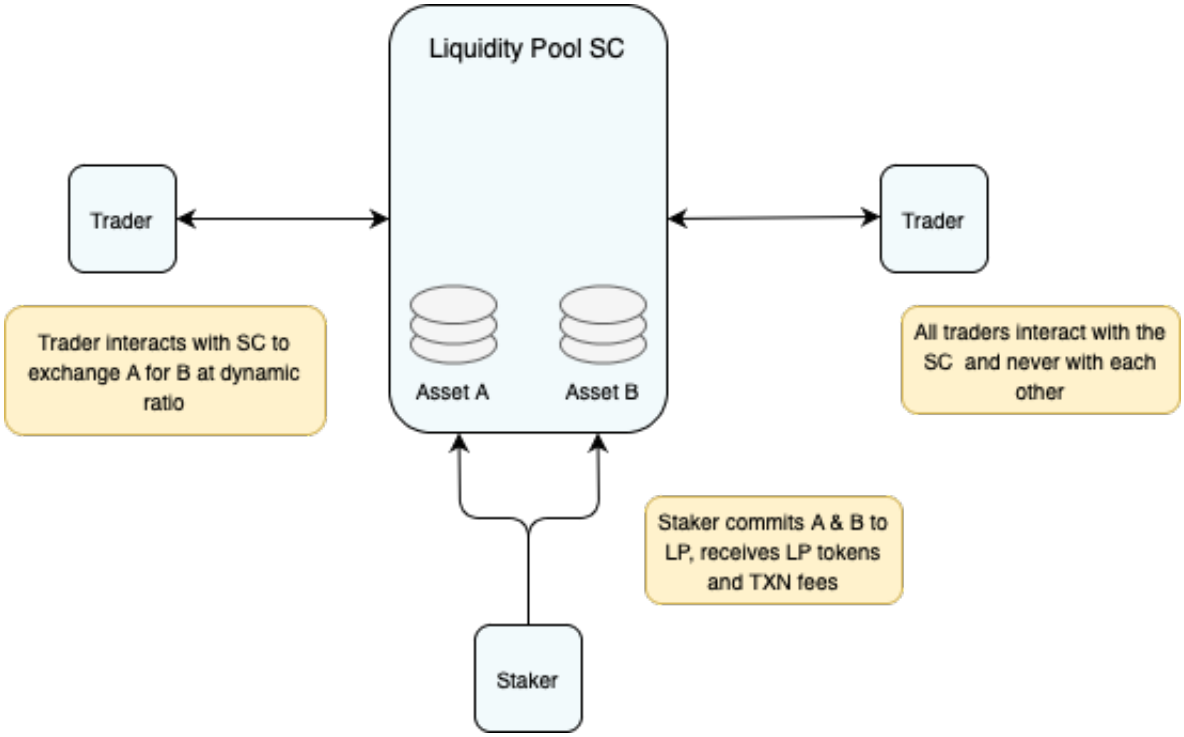

Figure 1.6: The basic interactions of a Liquidity Pool SC

NCO that maintains a centralised orderbook and submits orders to the chain via its own relayers. See Section 2.6.

1.3.20 DeFi

"DeFi" stands for Decentralised Finance, and is the loose labeling of a set of protocols, blockchains, cryptocurrencies, tokens, tooling, and communities that have emerged as an ecosystem supporting tokenised finance without dependence on any centralised entity.

At the time of this writing DeFi represented approximately $80 B in market size as measured by "Total Value Locked" (TVL)[11].

Our project can be considered a DeFi trading protocol.

1.3.21 Initial Coin Offerings

Initial Coin Offerings or "Token Sales" as they are sometimes referred to, are the digital asset equivalent of traditional share issuance, or an IPO.

Initial Coin Offerings are a smart contract implementation that allows the contract holder to raise funds (via a base protocol token, such as Ether) in exchange for tokens, usually at a set fixed price.

[11] `https://www.defipulse.com/`

1.3.22 IPFS

IPFS, the Inter-Planetary File System, is a distributed file system running over a peer-to-peer network of interconnected nodes. Content in the network is addressed not with locations (such as url based routing used by HTTP) but via a *content hash* of itself and all of its children, recursively (See Merkle Trees above).

Users can request content from the network via its content hash. The content is not maintained by one entity but by an arbitrary number of peers that lend their disk space to the network and maintain a copy.

Our system is deployed in a decentralised manner via IPFS.

1.4 Our Approach

Our project attempts to build a novel implementation of the NCO model. We explicitly prioritise decentralisation in our design & implementation.

We also attempt to replicate and match both the capabilities and UX offered by centralised trading platforms. We believe that proper UX is a key component towards driving DEX adoption among retail traders.

In the following section we give a brief overview of the capabilities of our system and the various architectural aspects that make up its design. These are further expanded upon in Chapter 3.

Browser-based Peers

Our entire client-side application runs as a Javascript application in the user's web browser. In order to join the network and participate in the exchange, users only need to "download" the application via their web browser. We place download in quotes here as the user only needs to visit a url. (The application is solely Javascript)

Decentralised Deployment

Our application is deployed via IPFS as a hashed directory of files that represents a single-page browser application (SPA) built using React (cf. 3.6).

Our full application is thus available over the IPFS network at the specific hash:

`QmYszj7GyMvKLJPzZWaop9szx4caZTeuHgip45sa2uV3yn`.

We have also deployed an IPFS gateway to accommodate users with browsers which do not natively route IPFS content. [12][13]

[12] `https://ipfs.thresholdholdings.com/ipfs/QmYszj7GyMvKLJPzZWaop9szx4caZTeuHgip45sa2uV3yn/`

[13] The domain name belongs to one of the authors and is coincidental

Non-Custodial Orderbook

Our project is a non-custodial protocol, meaning at no point in time until order execution does anyone have custody of the user's assets besides the user themselves.

Users submit signed messages containing their order details which are then executed against corresponding orders on-chain. Order execution entails actually swapping token balances between two users.

Order Propagation

In order for peers to communicate orders with one other, they form an open peer-to-peer network utilising a gossip protocol over WebRTC[28]. A PubSub abstraction over the peer-to-peer network is utilised, allowing peers to communicate with each other on specified channels.

Peers pass orders to each other via messages over these channels and each peer maintains a "best effort copy" of the entire orderbook. We detail some possible more advanced techniques for implementing the shared order book state in the Future Work section.

Peer Discovery

Peers uses a gossip protocol in order to communicate with one another. Peers also communicate metadata with these messages, including the location of other peers in the network. Peers maintain co-peer locations in a distributed hash table (DHT).

In order to establish connection details over WebRTC, we utilise a WebRTC signaling server. The WebRTC signaling server also handily doubles as rendezvous point to find initial peers in the network on startup.

Our WebRTC signaling server is the only centralised portion of our architecture. It is important to note that the WebRTC signaling server only handles negotiation between peers when they initially connect over WebRTC. Thereafter peers communicate solely between themselves via the WebRTC protocol.

Order Persistence

We utilise the web platform as much as possible. Peers maintain their copy of the orderbook in their local browser-based IndexedDB[14].

By using IndexedDB and storing the orderbook in the user's browser, we avoid the need for a centralised storage solution for each peer.

[14] `https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API`

A downside of this approach is that if a user unintendedly deletes their browser's storage, their is no way to restore the orderbook to its previous state, and it must be constructed again from scratch. As mentioned earlier, restoring the orderbook to its previous state via synchronisation will be covered as part of implementation of the shared order book state in Chapter 7.

Order Matching & Execution

We allow users to become special "validator nodes" which validate, match, and send orders to the chain for execution. Validator nodes pay the gas for transactions on the chain and in return are compensated via a commission on the order amount.

Any node can become a validator node by submitting a deposit (which is used to pay the gas fees) and generating a signer to submit transactions on their behalf.

We use a novel technique which we term "delegated signing" in order to avoid the need for the validator node to manually approve every transaction and facilitate a superior user experience. See section 3.4 for more details.

Token Creation

We implement a smart contract solution that allows users to generate entity identifiers and create an arbitrary number of tokens linked to an entity. Tokens can be created as fungible tokens or non-fungible tokens, and in the fungible case they can be created with an arbitrary supply number.

Combined with the trading capabilities, our system facilitates easy creation of tokens for fundraising in the style of an ICO.

Order Event History

We utilise the Graph Protocol's Subgraph implementation in order to index order execution events in our exchange contract (cf. Chapter 3). By doing so, we allow the client application to easily query the exchange state history in order to observe token price trends and order history from other peers.

This is a necessary feature in order to facilitate the price analysis features offered by Centralised Exchanges.

1.5 Description of Work

The project is envisioned to serve as prototype for a real-world application. Accordingly, it should boast a feature set that real users would find attractive and strive for the non-functional characteristics that are expected of a production grade system.

Importantly, the system shall:

- allow users to quickly oversee their account balances available to trade on the platform,
- give users the tools to analyze market activity in a near real time manner,
- seamlessly interface with common methods of managing token wallets, most notably the Metamask browser extension,[15]
- allow users to trade with the market in a P2P manner, notably without any noticeable delays. This functionality should remain close in spirit and "feel" to leading modern exchanges,
- provide a manner for users to issue tokens on the protocol that can be traded on the platform,
- not persist centralised data in any manner and maintain the "spirit" of decentralisation in its approach to all implementation details,
- maintain security as a fundamental design philosophy so that user funds remain safe from attack or loss,
- boast a smooth & modern design that, combined with modern Javascript libraries, promotes a pleasant user experience,
- be deployed and distributed on IPFS, a decentralised file store, to maintain a true "end-to-end" decentralised experience.

Our application bundle consists of:

- The front-end JavaScript application which contains the entire client-side application code necessary to run a peer, and is deployed to IPFS,
- Three Smart Contracts, representing our on-chain functionality deployed to the Ethereum blockchain,
- A WebRTC signaling server,
- and a Subgraph implementation

This project also led to the development of a novel technique for blockchain developers:

- Delegated Signing - A technique for signing transactions on behalf of a user that
  - obviates the need for manual transaction approvals, facilitating high throughput and good UX
  - can be performed without knowledge of the user's private key
  - can be implemented without any storage of the user's signing details by the application

[15] `https://metamask.io/`

1.6 Work Outline & Milestones

A brief outline of our work follows: We first turn our attention in Chapter 2 to other exchange implementations currently under active development in order to present an overview of current exchange methodologies. Details of each project are analysed as well as their feature set for end users.

Chapter 3 justifies our main architectural decisions behind the design, while Chapter 4 presents the specific implementation details as well as the novel techniques we developed. Chapter 5 presents a framework for evaluating our solution. Chapter 6 concludes, and Chapter 7 presents a few promising avenues of future work.

Authorship Statement

All participants contributed to the success of the project. The following rough specialisations emerged:

Corey Bothwell was responsible for the development of the client-side application, including the UI, state management, and chain communication. He assisted with the Peer implementation including the client-side database. He also implemented the signing mechanisms for orders and led the development of the delegated signing approach. He implemented details of the order matching algorithm and also deployed the client application to IPFS. He was instrumental in the completion of this report especially Chapters 1, 3, 4, and 5.

Ankan Ghosh led the development of the client application's peer discovery aspects via WebRTC. He led the modelling of the IndexedDB, implemented backend and client interaction with DB, order validation logic and pub-sub gossiping via protocol buffer for information serialisation. He helped in integrating different modules of the project. He also helped in dockerising the code base and creating a continuous deployment channel in google cloud platform cluster used for development/testing of the project.

Saiteja Pottanigari was responsible for developing the smart contracts and leading the initial design decisions and setup. He also contributed to the Graph Protocol's Subgraph implementation. He was responsible of providing ideas and implementing the EIP-712 off-chain order signing mechanism, as well as IndexedDB and Web Workers. He was responsible for the deployment of the smart contracts on-chain, deployment of MATIC's testnet full node and deployment of the WebRTC signaling server. He implemented the initial version of order matching algorithm. He also guided the others with his expertise and experience in blockchain development.

All participants contributed to the associated report and presentations.

1.6.1 Milestones

Milestone 1 - Midterm Presentation

The midterm presentation will be given on 30 September, 2021. The midterm presentation will detail our design decisions and implementation prototype at the midway point in the project.

Milestone 2 - Hand-in & Final Presentation

The final hand-in is set for 15 March, 2022. The final presentation will take place 7 April, 2022, where the final implementation will be presented along with our design decisions and conclusions.

Chapter 2

Related Work

This section of the report focuses on the other decentralised exchange platforms which attempt to solve similar problems. For each we provide a description of how important functionalities are implemented.

A comparative analysis is also provided at the end.

2.1 Loopring

Loopring is an open source protocol founded in 2017 based on SCs for decentralised exchanges on the Ethereum BC. Loopring allows for multiple exchanges to mix and match orders, off-chain order-matching and on-chain TX clearing and payment. Loopring is BC agnostic and can be deployed on any BC with SC functionality (at present in Ethereum and NEO).

Protocol

Loopring is a modular protocol to build a DEX on multiple BC's. It follows a Unidirectional Order Model where each order has an exchange rate between two tokens, this exchange rate is calculated based on the portion of Token A trader wants to exchange for Token B and on the assumption that trade is possible whenever the buyer is willing to pay an equal or higher price than the seller's price. A prominent feature of the protocol is the mixing and matching of multiple orders in circular trade forming in Order Ring[49].

Suppose 2 trader wants to trade on Token A and Token B, Alice has 15 token A and she wants 4 token B for them, Bob has 10 token B and he wants 30 token A for them. So if we fix the Token A as the reference, then Alice want to buy Token B at 3.75A, while Bob is selling 10 token B for the price of 3.00A. On contrary, referencing Token B, Alice is selling 15 token A for the price of 0.266B and Bob is buying 10 token A for the price of 0.333B. Hence, making the buyer or seller role irrelevant as long as buyer pays a equal or higher price. Similarly, it is possible to club more orders forming a order ring from the

same pair of tokens. This order ring is only valid if all component TXs can be executed at an exchange rate equal to or better than the original rate specified implicitly by the user. To verify order-ring validity, Loopring protocol SC's must receive order-rings from ring-miners where the product of the original exchange rates of all orders is equal to or greater than 1.

$$\frac{x_1 \cdot x_2 \cdot x_3}{y_1 \cdot y_2 \cdot y_3} \geq 1$$

Ecosystem

Loopring is not only limited to providing a network for DEX. It has rich set of features and services for the DEX to flourish DApps to move in and use the features with easy integration to existing platform. Loopring offers a common wallet service which will be incentivised to produce orders by sharing fees with ring-miners. Incentivisation is trustless due to the implementation of LPSM.

Loopring protocol SCs (LPSC)[49] are public contracts present in the network to which checks order-rings received from ring miner. This also emit events which interact relay browsers with events to keep their order books and trade history up to date. Relays are nodes that receive orders from wallets or the relay-mesh, maintain public order books and trade history, and optionally broadcast orders to other relays.

Asset Tokenisation service is another feature offered to token assets (real, fiat or tokens from other chains) and get tokens issued, which can be redeemed for the deposit in the future.

Order Mechanism

Loopring uses an order book model but in a distributed manner and heavily relies on relays to explore and order matching. Initially the order gets authorised by Loopring Protocol Smart Contracts(LPSC) to trade certain amount of tokens from the account. Then the order creation takes place on current rate and order book for tokens are provided by relays or other agents hooked up to the network such as Order Book browsers. A certain amount (LRx token) for order processing is added to incentivise the miners. Wallet sends the order and its signature to one or more relays. Relays update the public order book and broadcast the order to other relays through any arbitrary communication medium. To facilitate a certain level of network connectivity, there is a built-in liquidity sharing relay-mesh using a consortium BC. Miner tries to fill the order fully or partially at the given exchange rate. The algorithm also makes the trade profitable by matching with multiple orders forming a loop. Post this LPSC checks to verify the miner's data for order filling and post checks finalises the verification and settlement.

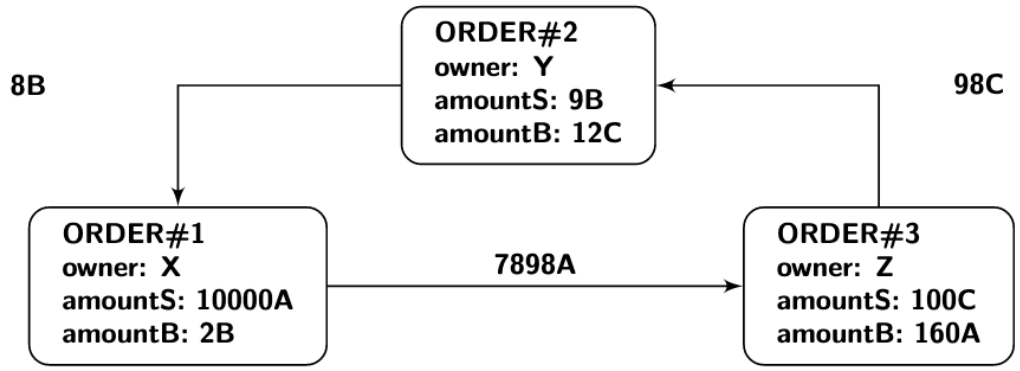

Figure 2.1: Loopring order matching

Fraud Protection

Loopring also prevents front-running TXs where someone tries to copy another node's trade solution, and have it mined before the original TX that is in the pending TX pool. This can be achieved by specifying a higher TX fee (gas price). Loopring also claims to protect the network from DOS Attack and Insufficient Balance attack.

2.2 Deversifi

Deversifi was initially called ETHFINEX which is founded by BitFinex a popular DEX on Bitcoin. BitFinex expansion plan to introduce a similar decentralised exchange in Ethereum was the base idea for ETHFINEX which later optimised their technology and renamed to Deversifi. DeversiFi is a self-custodial orderbook based DEX built using StarkWare's StartEx layer 2 scaling engine. StarkWare StarkEx is a ZK scaling solution that batches TXs on the Ethereum BC, increasing throughput and privacy without compromising on security. StarkEx is an Ethereum L2 scalability solution that is based on validity proofs which can be operated using both ZkRollup or Validium mode[2] and is a mature platform that has been deployed on Ethereum Mainnet. DeversiFi recently released v2.0 which can handle 9000+ TXs per second without compromising privacy, also provides access to aggregated liquidity and claim to be taking nearly 0% TX fees.

Blockchain Platform

Deversifi uses an unique layer 2 scaling solution Validium by STARKWare StarkEx[47] which uses a Zero Knowledge Transaction Rollup just like ZKRollup but the data is not submitted to the chain in Validium, data availability is kept off-chain. The StarkEx system has an off-chain component and an on-chain component as in the fig. The off-chain component stores the state, orders transaction execution in the system, and sends state updates to the on-chain component. The on-chain component stores the state commitments, system assets, and it is also responsible for enforcing the validity of state transition.

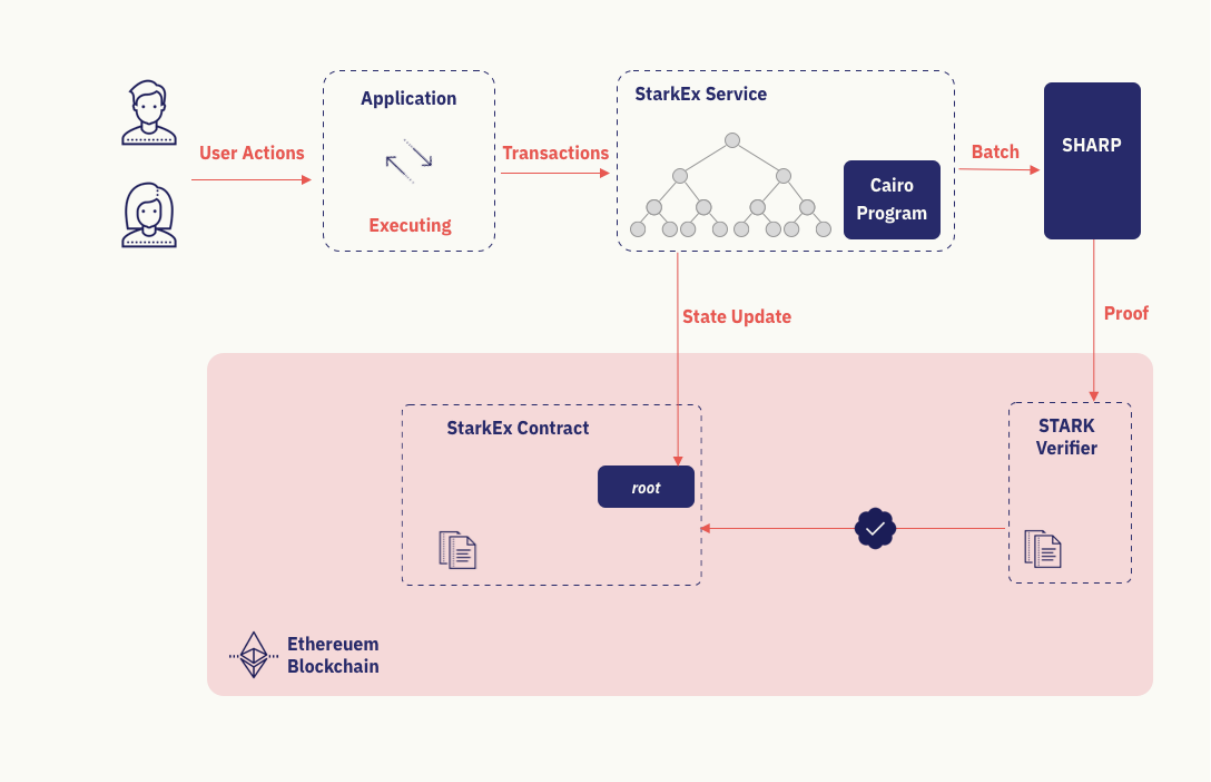

Figure 2.2: StarkEx

All the transactions in the system are executed by the application and then sent to the StarkEx Service. The StarkEx Service sends the batched transactions to SHARP, a shared proving service, to generate a proof attesting to the validity of the batch. SHARP sends the STARK proof to the STARK Verifier to verify it. The service then sends an on-chain state update transaction to the StarkEx Contract, which will be accepted only if the verifier finds the proof valid.

Technical Implementation

A distributed, multi-threaded & multi-process off-chain matching engine built around speed, stability, and scalability named Hive developed by BITFINEX[11]. Through an increasingly efficient order allocation process, matching engine have worked to drive down TX fees to pennies whilst improving market quality (lower spreads, improved order book depth). The Data Availability Committee is a collection of trusted entities who hold customer balance data off-chain and take part in signing the commitments which update the BC state. This ensures that even if both DeversiFi and StarkWare go offline (highly unlikely), there will be an easy way to make sure that customers can withdraw their funds. The committee is a necessary part of the StarkWare V1 to ensure traders can trade privately, without their trading history being published on-chain. Other ways of ensuring private trading without compromise (speed, trust, *etc.*) are currently being researched and will likely be available in later iterations.

Order Matching Algorithm and Counterparty Discovery Mechanism

Once a trader has locked his tokens into a SC, whilst still maintaining control at all times, the trader can submit, modify, and change orders just by signing each change with their Ethereum key (as simple as clicking a confirm button with user friendly wallets like MetaMask). This is instead of having to submit a full TX to the Ethereum BC every time would like to update an order or trade. This saves gas fees and enables traders to employ more advanced trading strategies that are dependent on quickly being able to update orders. DeversiFi OTC is a valuable tool for those looking to swap larger amounts of tokens in a single trade. Executing such a trade on a regular exchange could cause a big swing in the token price, potentially costing the user a significant amount of money, not to mention the possibility for a longer settlement time. DeversiFi OTC effectively solves this by keeping the order off the exchange orderbook. Users deposit their assets in the DeversiFi zkSTARK SCs where the exchange engine executes trades off-chain (connecting orders with deeply liquid external order-books at scale and speed) and settling final balances on-chain afterwards. This results in rapid execution speed with new, updated balances instantly available for trading as well as lower trading fees owing to increased gas-cost efficiency. Evolution of Ethfinex Trustless to DeversiFi and the fast-approaching Margin, Lending and new Fee features - giving DeversiFi a competitive edge against other exchanges.

2.3 Binance

After Binance being in the market of Centralised Orderbook Cryptocurrency exchange and being one of the largest Cryptocurrency markets, Binance had plans to launch its own Orderbook Decentralised Exchange tackling all the issues like front running, slow throughput, and higher transaction fee that were in the market during their initial version launch. The initial version of Binance Dex was based on the single Blockchain. To create Binance DEX, the team behind the exchange created a new blockchain and peer-to-peer distributed system which has a decentralised order-matching engine based upon Binance Chain Technology also ensures that all transactions are registered in a chain within the blockchain nodes. This aims to establish a full, auditable ledger of activity and permits users to exchange the digital properties issued and specified on the DEX and Binance Coin (BNB) is no longer a token of the ERC20 standards and is now migrating to the Binance Chain in Binance Chain Evolution Proposal 2 (BEP2) standard and becoming the native asset of the current blockchain. Binance uses a periodic auction matching concept which resolves issues like front running and ensures that most of the orders in the orderbook are filled.

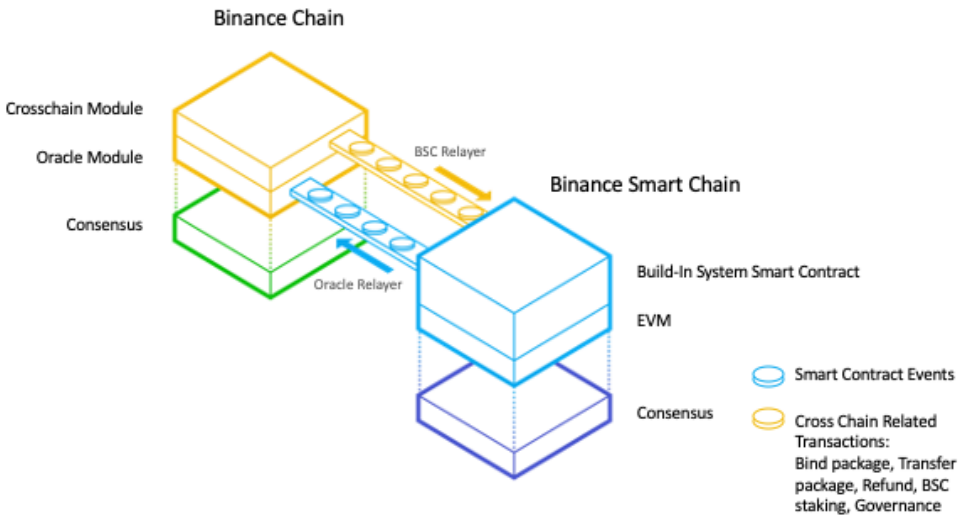

Figure 2.3: Binance Blockchain Platform

Blockchain Platform

Binance DEX is a Decentralised Exchange built on top of two side by side BCs developed by Binance. First one is the Binance Chain, a Tendermint[1]-based, instant finality BC, which has a one second block time with a high speed and large throughput design. The second one is Binance SmartChain, which exhibits similar functionalities as Ethereum, *i.e.,* it supports SC and tokenisation with different standards like BEP2 (Binance Chain Evolution Proposal) and other tokens which they are planning to introduce in future.Binance Chain adopts a consensus mechanism called Delegated Proof of Stake and Binance Smart Chain[8] relies on a system of 21 validators with Proof of Staked Authority (PoSA) consensus that can support short block time and lower fees. The most bonded validator candidates of staking will become validators and produce blocks. The double-sign detection and other slashing logic guarantee security, stability, and chain finality. There is a Native Cross-Chain Communication protocol employed by Binance which is using relayers between both Binance Chain and Binance SmartChain, which is bi-directional, decentralised, and trustless. The DEX now supports both ERC20 tokens as BEP20E tokens.

Technical Implementation

The Cross Chain Communication is enabled by the usage of relayers and an oracle module because they are responsible to submit Cross-Chain Communication Packages between

[1]Tendermint is software for securely and consistently replicating an application on many machines. Tendermint is a general purpose blockchain consensus engine that can host arbitrary application states

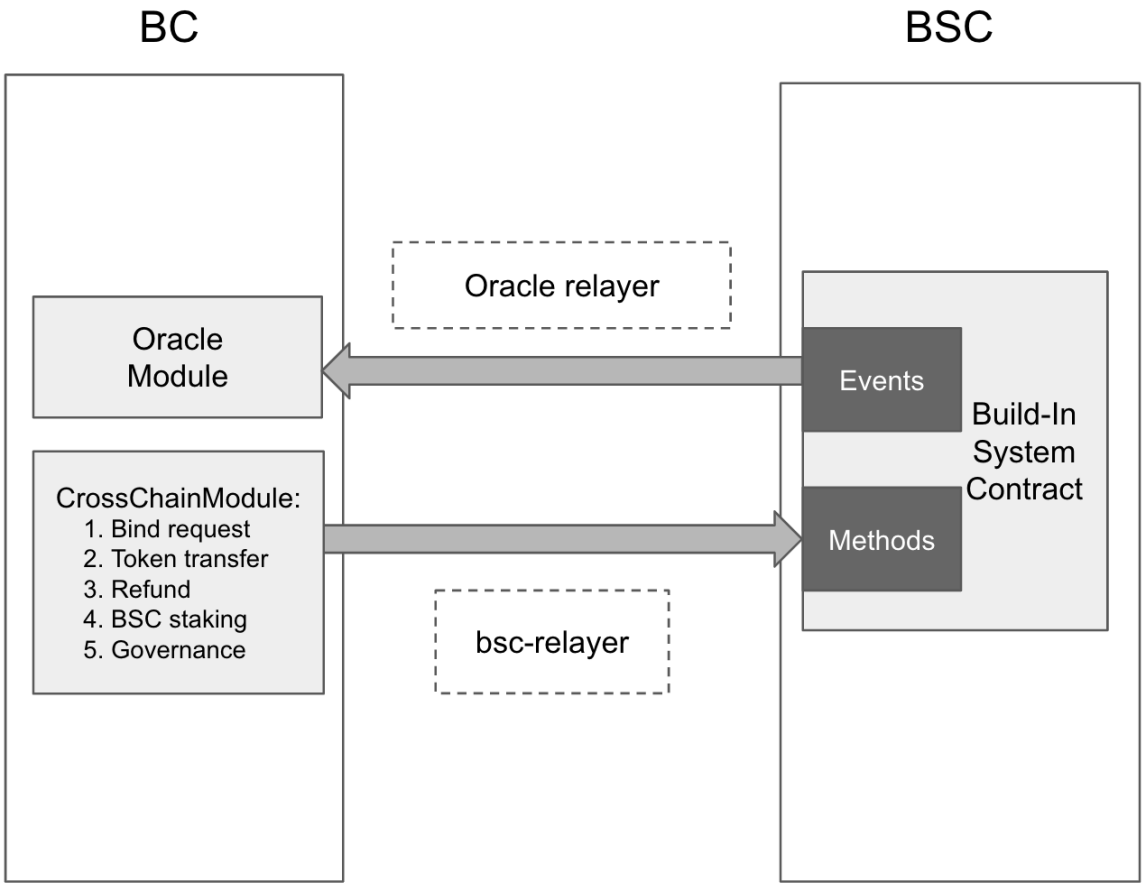

Figure 2.4: Binance Cross Chain Communication

Binance Chain and Binance Smart Chain. Binance uses cross-chain interoperability property to enable one BC to function as a light client of another BC. In Binance DEX, Cross-chain transfers[7] only support bound BEP2 or BEP8 tokens on Binance Chain and BEP20 tokens on Binance SmartChain. We have two relayers that handle different functionalities. One, BSC Relayer[6] is a service that relays cross-chain packages from the Binance Chain to the Binance Smart Chain. These relayers must sign up with BSC and deposit a certain amount of BNB. BSC can only consider relaying requests from licensed Relayers. Relayers may also track BSC for double sign activity and apply proof to Binance Chain. All BSC relayers build their stable infrastructure, watch any event happened on the Binance Chain, and act timely to get paid accordingly. Secondly, Oracle Relayer[9] is a service which monitors events on BSC, builds and broadcasts transactions to BC. Finally, oracle module is a common module like governance which is used to handle the validators who want to reach a consensus on cross chain transfers.

Counterparty Discovery Mechanism

Binance DEX uses periodic auction to match[10] all open orders. Maker/Taker concepts are introduced to enhance the current periodic auction match algorithm. A match is still

executed only once in each block, while the execution prices may vary for maker and taker orders. Orders meet any of the below conditions would be considered as the candidates of next match round. New orders that come in just now and get confirmed by being accepted into the latest block. Existing orders that came in the past blocks before the latest and have not been filled or expired. Candidates would be matched right after one block is committed. Each block has one round of matching. The match only takes place when the best bid and ask prices are 'crossed', i.e. best bid > best ask. There would be only 1 price selected in one match round as the best prices among all the fillable orders, to show the fairness. All the orders would be matched first by the price aggressiveness and then block height that they get accepted. The execution price would be selected as the below logic, in order to maximise the execution quantity and execute all orders or at least all orders on one side that are fillable against the selected price.

Order Matching Algorithm

Binance employs an on-chain periodic auction matching[5] concept, which resolves problems such as front running even though it is on-chain and guarantees that the majority of orders in the order book are filled. As mentioned in periodic auction matching per block, fast orders are less important, i.e. fast orders do not benefit from high-frequency trading activities despite being hundreds of milliseconds faster. Fast orders have little benefit. Orders are client orders to purchase or sell tokens into other tokens on the Binance DEX. The order would initially lock the properties to be traded with, and the Binance DEX would then begin the process of order matching against any orders from the block. If the order matches the opposite hand, the trade is created and the properties are moved. Completely completed orders would be withdrawn from the order book, while unfilled or partly filled GTE's would remain on the order book until they were filled by others. The matching occurs within the BC nodes, and all transactions are registered on-chain, resulting in an accurate, auditable ledger of operations.

2.4 IDEX

IDEX[1] has been one of the longest running DEXs on Ethereum, processing over 3 billion trades since launch in late 2017.Until the existence of Automated Market Marker DEXs, IDEX 1.0 has set many records in Decentralised Exchange space as a DApp including like Most DEX Transactions, Largest Contract State which means holding many accounts in a single contract state, Most Unique Users. In attempts to fix the underlying infrastructure of centralised exchanges by facilitating trades with smart contracts, removing third-party control of trader funds, few companies have started developing DEX platforms and the earlier ones to the market implementing the on-chain order book mechanism had to compromise on speed and performance. IDEX came up with a hybrid solution IDEX 1.0. The first version of the IDEX exchange combined off-chain matching and validation with on-chain settlement. There were still few issues with IDEX 1.0 like trade execution speed, limited scalability, and a few UX challenges. Then IDEX launched its IDEX 2.0 built on

Optimised Optimistic Rollup (O2 Rollup) is a novel, open-source layer-2 design for bringing scalable applications to public blockchains. This design enables the escrow of funds and coordination of trade settlement necessary to support instant, off-chain execution.

Blockchain Platform

IDEX operates on Ethereum, supporting ETH and ERC-20 assets, and Binance Smart Chain (BSC), supporting BNB and BEP-20 assets.As we see the growing number of number of new smart contract platforms emerge, each with a unique set of capabilities and assets. As these platforms grow,IDEX see increased demand for trading these assets and a need for non-custodial trading solutions that support these networks. Initially IDEX was supporting only Ethereum. Over the past few months, IDEX has expanded to support new layer-1 blockchains, starting with Polkadot and Binance Smart Chain. IDEX customers can now be able to trade assets on all of these different chains from the safety and security of a single IDEX account. IDEX is aiming to develop a multichain will continue to expand to other layer-1 networks, such as Algorand, as they launch and grow in popularity. With O2 Rollup, IDEX 2.0 can bring this scalable architecture into any blockchain that supports smart contract capabilities.

Technical Implementation

In IDEX 2.0 the Optimised Optimistic Rollup (O2 Rollup)is implemented which is a novel, open-source layer-2 design for bringing scalable applications to public BCs. IDEX 2.0 [2] combines these two concepts to support the batch settlement of trades and drastically. reduce gas costs. The first critical change is the removal of balances from the on-chain contract. The core contract becomes responsible solely for escrowing funds, while the actual account and balance information are stored off-chain in a public, layer-2 ledger maintained by IDEX and cryptographically guaranteed to be available to the public. IDEX 2.0 combines the hybrid design of IDEX 1.0 with Merkle roots to support the batch settlement of trades, drastically reduce gas costs and congestion issues. Instead of settling each of these changes individually to the Ethereum network, they are settled in batches. Every few minutes, all of the previous transactions, known as an O2 block, are hashed together into a fixed-length Merkle root. It is this Merkle root that is submitted to the network and stored for public validation. The O2 Rollup design achieves this through the combination of cryptographic fraud proofs and a validator network with well-designed economic incentives.

Counterparty Discovery Mechanism and Order Matching Algorithm

Earlier in IDEX 1.0, the implementation was in the maker and taker concept where a maker creates and taker used to fulfill an order existing order placed by the maker in the order book. Later in IDEX 2.0[31], the implementation is quite similar to traditional, non-crypto exchanges, limit orders are filled at the specified price or better with no risk of order collisions or trade failures IDEX uses an off-chain matching engine that leads to both the

maker and taker into limit order transactions.This design approach, combined with an off-chain matching engine, is necessary to enable non-custodial trading with the speed and UX that modern trading environments demand. IDEX employs a high-performance central limit order book design that continuously matches user orders on a price-time priority basis. IDEX 2.0 also upport for partial fills enables seamless matching against multiple orders. As a result, the user experience is similar to a high-performance centralised exchange. Instead of users who designate their status as a maker or taker at the start of a process, users are able to sign an order with a price window (i.e. sell at or above this price or buy at or below this price) as IDEX 2.0 made changes in smart contract architecture. IDEX would also have to balance consumers with competing pricing windows.

2.5 Wave Exchange

Waves DEX (now Waves.Exchange) [23] is a decentralised cryptocurrency exchange that launched in June 2017. It is a hybrid assets exchange that allows users to transfer, trade, issue and stake crypto. Wave exchange blends the security capabilities of a decentralised framework with the benefits of centralised exchange's optimised order matching. Wave exchange also provide staking opportunities, low TX cost and API easy to integrate with applications.

Ecosystem

Waves DEX have matured over time and now offers lightweight multicurrency wallet services as well as have integration with major third party wallets like Ledger and with major exchanges. WaveDEX also provides the feature of algo based trading, staking, leverage trading and it can be used on mobile application as well making it more accessible.

Architecture

Wave DEX [15] have a hybrid architecture, it follows a heavy and light node architecture. The heavy node keeps the full TX record of the BC and light node just keeps a partial copy of the TXs and it can verify a TX with a heavy node. However, new nodes are open to join heavy node group if node have enough resources. WAVES is built on the Scorex platform, which develops an approach based on using current network state as an alternative to full TX history. A simplified payment verification procedure will be realised for the lightweight node, adding another security layer. Wave DEX uses a Proof of Stake protocol as a consensus algorithm. The lightweight node is a browser plugin written in JavaScript which interacts with Scorex based full nodes.

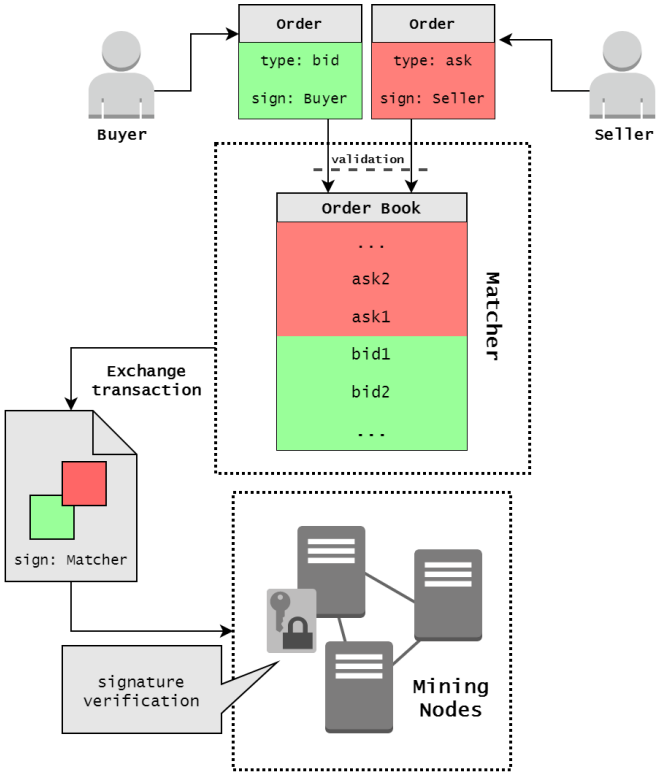

Figure 2.5: Wave order matching

Order Matching

Wave exchange is account based [22], all transactions are signed, the user's wallets are not controlled by a single entity as orders are digitally signed by the owners, as an authorisation process. User place a buy or sell order in the orderbook and other user can add a digitally signed counter-order and send the complete TX with a pair of orders to the BC. Then the finality of the order is recorded in the ledger and transferred between the buyer and seller. Order processing takes place in centralised fashion, hence boosting the performance. Wave allows limit, partial filled & cancel order TX . Users are allowed to set validity of the order, once expired the order gets cancelled. Wave charges order TX cost of 0.003 wave irrespective of the amount of the order. Order collection takes place in decentralised manner and matching TX takes places in a centralised order book model.

Fraud Protection

Wave DEX prevents front-running TXs due to centralised order matching algorithm which also arbitrageur with cancelled order.

2.6 dY/dX

dXdY is a decentralised trustless perpetual contract markets offering margin trading, spot trading, borrowing and lending with no locking period launched in April 2019(around). dYdX runs on SCs on the Ethereum BC. Currently, the lending is supported only for ETH, DAI and USDC.

Ecosystem

dYdX platform [33] focused towards trader experience on lending, borrowing and taking custom margin positions. dYdX claims to short and long trade with leverage up to 4x. Funds can be automatically borrowed from lenders on the platform for Margin trading. Collateral used to secure margin trades continuously earns interest, making sure that the user gets the maximum yield on his capital. During lending dYdX can help the user easily earn passive income on your crypto holdings.

dYdX also provide analytical dashboards for tracking portfolio performance. dYdX platform also provides APIś for using the spot, margin, lending and borrowing protocol. Also provide client in TypeScript and Python for programmatic trading.

Architecture

dYdX exchange uses a centralised order book but remains non-custodial and settles trades and liquidations in a trustless manner. Exchange runs on an L2 (layer-2) BC system, and operates independently of previous dYdX protocols and systems, including the v1 and v2. Trades are settled in an L2 (layer-2) system, which publishes ZK (zero-knowledge) proofs periodically to an Ethereum SC in order to prove that state transitions within L2 are valid. Funds must be deposited to the Ethereum SC before they can be used to trade on dYdX [20].

L2 solutions was developed and operated jointly with Starkware. ZK proofs increases the TX throughput and lower minimum order sizes compared with systems settling trades directly on Ethereum (i.e. L1). This is achieved while maintaining decentralisation, and fully non-custodial nature of the exchange.

Order Matching

The protocol [19] initially setup a business model based on TX volume not on TX amount and later on improvement in protocol made sure the consistent trade volume. Transaction fees incurred is paid by dYdX, when orders are matched, dYdX submits a TX to execute the matched trades on-chain. This allows dYdX to offer a smoother and faster trade experience, but it also means dYdX incurs gas costs proportional to the number of trades on dYdX.

On dYdX, there are two types of orders: Maker and Taker. Orders that do not fill automatically and are left on the order book are known as maker orders. These orders give the order book more depth and liquidity. Taker orders, on the other hand, automatically fill Maker orders that have already been placed. They deplete the order book's liquidity. To save money on processing fees, dYdX would charge higher fees for smaller orders than for larger orders. In the Maker hand, there will be no costs to allow consumers to place limit orders on the order book.

2.7 Conclusion

As we discuss more about the DEX in operation, centralised DEX do hold the king share in amount and transaction counts. Centralised DEX are still plagued with shortcoming of a centralised system. Binance being the most used among all the DEX has multiple allegations for market manipulations in the past. Server downtime and hacked custodial DEX have resulted in the loss of customer funds has adverse effect on the crypto investment in the past globally. In recent times, the change and awareness of these problems are being noticed and many investors are now very careful while using the DEX for investment.

With institutional investors pouring in millions into their crypto portfolios, it is very critical to supply the flow with liquidity for the DEX. In DEX market, the liquidity provider are holding the bargaining chip as the liquidity pools has now been seen as more than a successful partnership. DEX are now partnering with liquidity providers to address the liquidity crunch in their respective platform. A further study can be useful to investigate to see how much of negotiating power is dictated by liquidity providers.

With the current analysis of the respective DEX, most of the DEX using centralised order broadcast mechanism are successful in providing higher throughput in number of transactions. Also, it is evidenced that there is increasing trend towards providing staking options by the DEX and rewards the liquidity providers. The entire market worth of all staking platform tokens is $633 billion, but locked-in staking accounts for just around 24% of that, indicating that there is still a sizable addressable market[42].

Comparison Criteria	**Loopring**	**Wave.exchange**	**DyDx**	**Binance**	**Deversifi**	**IDEX**
Protocol	Loopring	Waves-NG	dydx decentralised protocol	Tendermint	Starkware StarkEX	IDEX
Native Token	LRC	Wave	-	BNB	ETH	IDEX
Order Execution Nature	Custodial	Non-Custodial	Non-Custodial	Non-Custodial	Non-Custodial	Non-Custodial
Order Matching Cost	>0.2	0.003 WAVES	ETH gas price	BNB	ETH gas	-
Order Matching Algorithm	Relays	Matcher Engine	Centralised	Periodic Matching	Hive	Server-side
MultiSig Transaction	-	Yes	-	-	-	-
Scalability	zkRollup(L2)	Lightweight Node	zkRollup(L2)	-	Validum	O2 Rollup
Listed Tokens	120+	-	9	154+	-	-
Enlisting ICO's	No	Yes	No	yes	No	No
Instant tx notification	No	Yes	-	3-sec	-	-
Supported token standard	ERC-20	-	ERC-20	BEP2	ERC-20	ERC-20 & BEP2
Order Broadcast	Centralised	Decentralised(0x)	Centralised	Decentralised	Centralised	Centralised
Fraud Prevention Mechanism	-	-	-	-	DAC community	-
Analysis Dashboard	No	Yes	Yes	Yes	Yes	Yes
Staking Options	Yes, 85+	Yes	No	Yes	No	Yes
Tx throughput	2,025 tx/sec	100 tx/sec	45 tx/sec	500+ tx/s	9000 tx/s	-
Consensus Mechanism	-	LPoS	-	DPoS	-	-

Table 2.1: Comparison Chart

Chapter 3

Architecture & Design

In the following chapter we give a high level overview of the system's architecture and its various components. We also justify some of our design decisions and enumerate our technology stack along with key dependencies.

3.1 Overview

Figure 3.1 gives a bird's-eye view of the system.

This project uses a novel implementation of an orderbook distributed over a peer-to-peer network. Peers communicate by gossiping with one another over WebRTC in PubSub "channels".

Client nodes in our network run as a Single Page Typescript Application (SPA) in the user's browser. Users request this application bundle from the IPFS network. Users can retrieve the files via any IPFS node with the content hash available, nonetheless we run an IPFS gateway[1] to provide a familiar User Experience as well as facilitate clients with browsers which are not compatible with IPFS.

Clients communicate with each other in a peer-to-peer manner via a gossip protocol over WebRTC. We use a library to assist with the peer communication called libp2p[36]. Clients exchange metadata with one another and maintain a directory of peer locations via a Kademlia Distributed Hash Table.

We utilise a PubSub protocol from libp2p which allows peers to broadcast orders to specific topic channels. When a peer broadcasts a message on a channel, libp2p attempts to deliver that message to all peers listening on that channel. We use these channels to broadcast orders between peers.

Validators watch the distributed orderbook and attempt to match orders and submit them to the chain for execution. Validators pay the necessary gas fees and are in return

[1] `https://ipfs.thresholdholdings.com/ipfs/QmYszj7GyMvKLJPzZWaop9szx4caZTeuHgip45sa2uV3yn/`

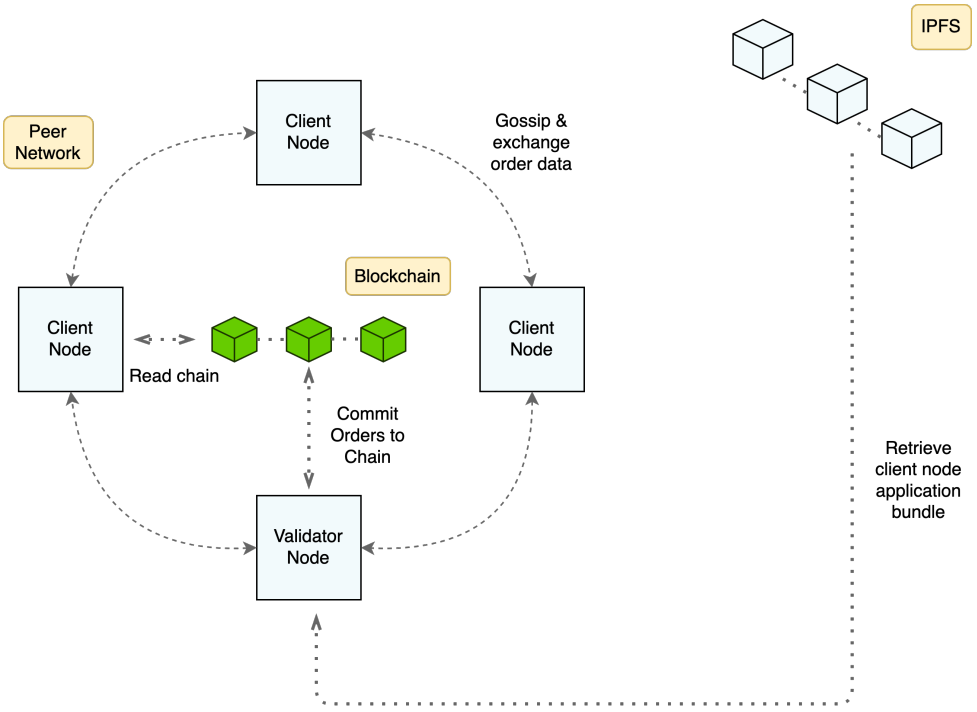

Figure 3.1: Overall System Architecture

compensated via a commission on the order. Validators then communicate back to the network the execution status of any matched orders.

True to our project goals, at no point is there a true point of centralisation which leads to single point of failure. Client nodes do rely on a WebRTC signalling server for brokering the communication information between them. The WebRTC signalling server also doubles as a bootstrap server for finding initial peers. After a session is established between peers, nodes are completely capable of communicating amongst themselves over WebRTC.

3.1.1 User Roles

In our architecture, users can assume three roles in the trading ecosystem: Token Set Owners, Traders, and Validators.

Traders

All users can easily assume the role of a trader by submitting an order to the network.

Token Set Owners

Users can easily create their own Token Set using the client-side interface. By doing so they become a Token Set Owner and can create independent tokens and offer them for

sale on the exchange. Token Set Owners can also commit dividends to the individual tokens so that token holders may claim dividends.

Validators

Users who have enrolled themselves as Validators execute client side algorithms to determine the optimal matching of orders. Once orders are matched, they become paired into a transaction.

Validators send the transactions to the chain which eventually executes the orders. We use a novel and sophisticated technique in order to sign transactions on the validator's behalf without knowledge of their private key. Validators pay the gas fees required to commit orders to the chain and are compensated via a commission payment from the buyer matched in the transaction. Transactions are committed to the Polygon layer 2 plasma chain, which eventually checkpoints to the Ethereum main chain.

3.2 Architecture Stack

This section serves as a high level overview of each of the "big ideas" in our system as well as the technology choices lying underneath. For a formal list of dependencies, please see Section 3.6.

3.2.1 Client Browser-based Application

Save for the blockchain itself and the WebRTC signalling server, our system runs completely in the client's browser. There is no traditional server back-end.

The client begins by downloading our application bundle from IPFS. Once the application has been fully downloaded, the application renders the GUI to the client and begins the bootstrap process. In the background, the application establishes connections to other peers via WebRTC. We use libp2p to manage the connections between peers. Once the peer has established connections to others in the network, the user is ready to submit orders to the protocol.

Application state is controlled via a central Redux store(cf. 3.6). The Redux store maintains a cohesive global state tree representing the overall state of the application. The application UI is "subscribed" to the Redux store and thus updates in real time as the application state changes. Additionally, the Redux store provides an interface for modifying the state tree, which we use to handle user interactions.

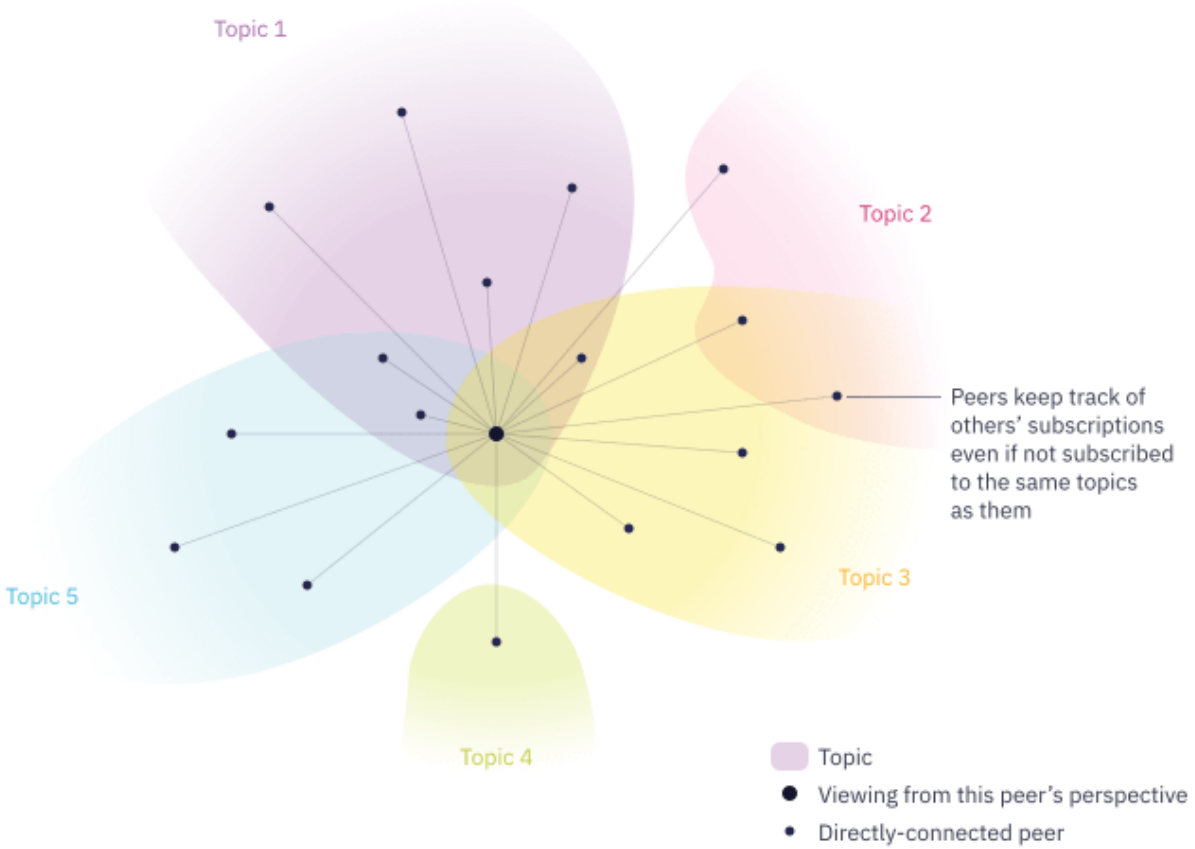

Figure 3.2: Libp2p's PubSub implementation is a clever abstraction over a sparsely-connected network[35]

3.2.2 Peer Order Communication

Peers communicate via a PubSub implementation of libp2p library through WebRTC. All the peers discovered using the WebRTC protocol are listening/subscribed to a topic channel. Messages are broadcast to all peers using the pub-sub channels. See Figure 3.3 for details.

It is important to note that the PubSub implementation of libp2p is a clever abstraction that allows developers to utilise the familiar PubSub design pattern, whereby nodes publish a message on a channel, and subscribers to the channel receive the message. Under the hood, peers are connected in a sparse network which is not fully connected. Peers gossip with their neighbours in order to ensure that messages posted to a topic eventually reach all subscribers of that topic.

Our implementation uses two different channels for propagating messages:

- The Order Topic: '/bb/order/1.0.0'
- The Match Topic: '/bb/match/1.0.0'

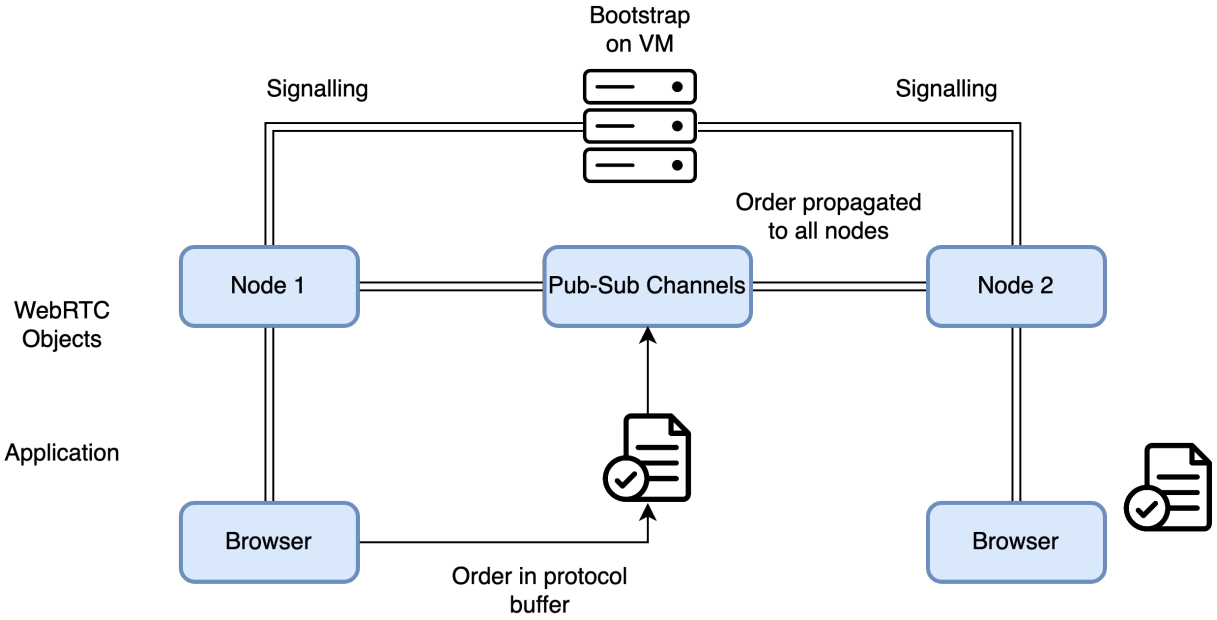

Figure 3.3: Peers in our system communicate over WebRTC & Pub-Sub Channels

All peers are subscribed to both topics. All peers share orders with each other on the Order Topic. Validator nodes post notifications of order matches on the Match Topic.

We note here that these topics are open by nature, any party implementing the libp2p protocol, connecting to our signaling server, and listening on these topics could listen for and publish messages on these channels. We view the open nature of these protocols positively and see this as an avenue for further works and implementations to add liquidity and build on top of our protocol.

Once order messages are received by a peer, they are stored in the IndexedDB. Validators especially rely on these saved order representations to match orders and submit them to the chain.

Efficient serialisation is achieved by using protocol buffers for message encoding. Protocol buffers are language-neutral, platform-neutral, extensible mechanism for serialising structured data (cf. Section 1.3). The orders are streamed across the nodes subscribed to the pub-sub channel in structured format.

Order Matching

Before transactions can be executed on-chain, they need to matched and verified. The order matching algorithm is run by validator nodes in the user's browser using a multithreaded worker process. Our implementation currently only supports a simple implementation of a Limit Order. Future work might implement a more sophisticated matching algorithm for other order types.

The order matching algorithm takes the state of the orderbook and attempts to match as many orders as possible at suitable prices and quantities. As order prices and quantities

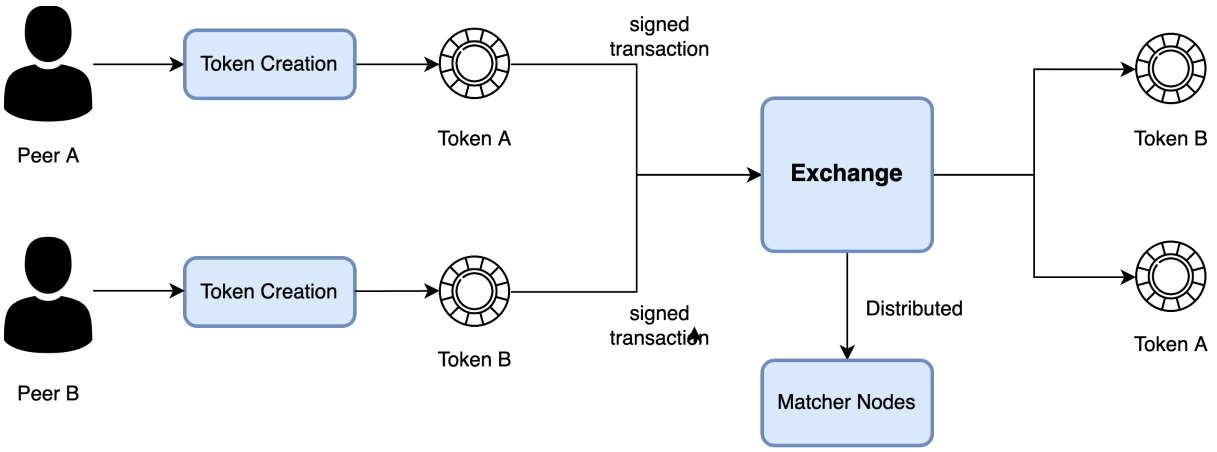

Figure 3.4: Token Trading

should theoretically only rarely match up exactly, the order matching algorithm must make compromises in selecting which orders can be matched together.

The order matching algorithm matches the orders and then performs some verification checks. Confirmed orders are then executed against the Polygon layer 2 chain. Finally, after transaction confirmation, the validator nodes notify the network that the orders contained in the execution have been matched accordingly.

3.2.3 Smart Contracts

Our system uses 3 smart contracts which maintain token state, balances, and execute orders against counterparties.

- TokenFactory.sol
- BBToken.sol
- Exchange.sol

TokenFactory.sol

TokenFactory.sol is a special factory contract used to instantiate instances of **BBToken.sol**. TokenFactory allows users to instantiate their own BBToken Contract which is linked to their address. Upon creation of a BBToken Contract, the Token Factory takes a unique identifier called an Entity URI as an argument. Entity URI's are stored in the TokenFactory Contract and linked to a respective BBToken Contract address. Thus, the entity URI is all that is needed in order to locate a specific BBToken Contract.

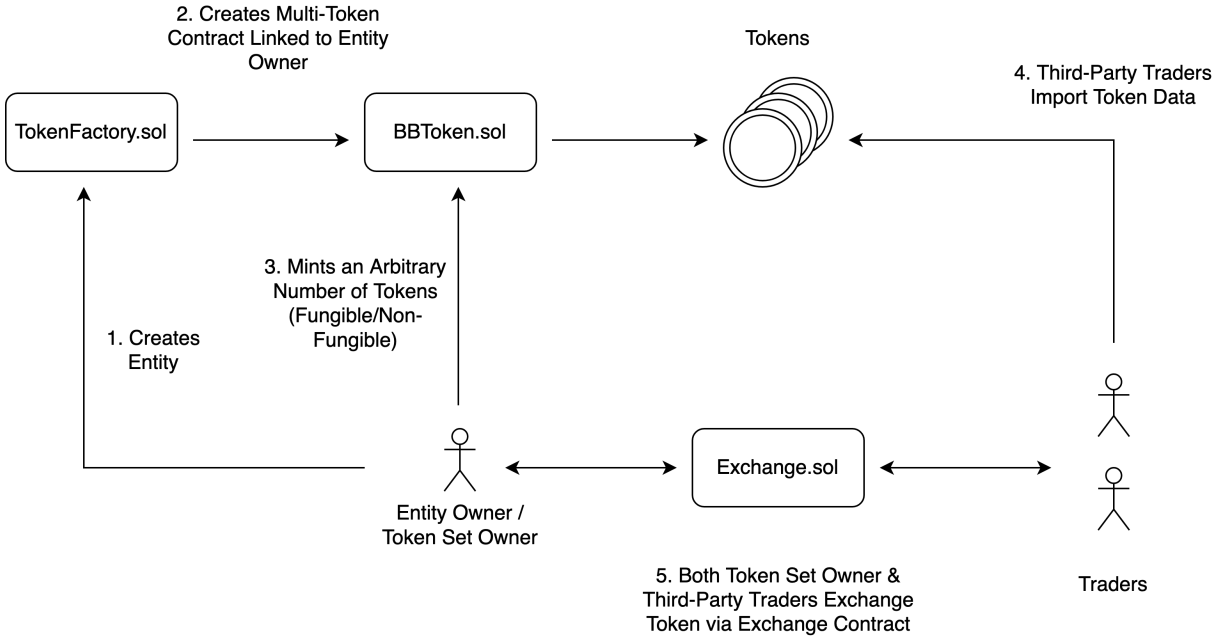

Figure 3.5: Our smart contract architecture. Entities create a Multi-Token contract via TokenFactory.sol. Entities can mint an arbitrary number of tokens via the Multi-Token contract and then exchange them via Exchange.sol

BBToken.sol

BBToken.sol is our specific implementation of the ERC-1155 Multi-Token interface. Each individual contract owner is the owner of the entire token set, and interacts with this contract to mint individual tokens with their own individual metadata. This is also where token dividends are added and claimed, token holders are tracked, and exchange approvals for token holders can be set.

Exchange.sol

Finally, **Exchange.sol** is where our exchange logic takes place. The Exchange contract takes care of validating and executing orders submitted from validators. The Exchange contract also keeps track of trader balances, and allows deposits & withdrawals. The Exchange contract also contains functionality for ensuring that the sellers have granted the exchange authority to transfer a specific Multi-Token on their behalf.

Figure 3.5 gives a visual overview of how the three smart contracts interact with one another.

3.2.4 Contract Event Indexing

The Graph[29] is a protocol which provides services for indexing and querying data from the blockchain. The Graph Protocol provides implementations of Subgraphs, which define

how data from the blockchain will be indexed by a graph node, which listens for events, calls and on-chain data from blocks added to the blockchain. This allows both clients and other protocols to efficiently query indexed data from the blockchain without the need to query the chain directly. We have written our own Subgraph implementation for indexing both the Exchange and TokenFactory contracts, which listen for the events from the Mumbai Polygon Testnet. The details of the implementation are described in the Chapter (cf. 4)

3.3 Economics

Our system is a protocol for exchanging tokens, so economics play a central role in our system's design. Due to the prototypical nature of our project, there is room for further research & development around the economic aspects of our system. Nonetheless we have attempted to cover the most salient aspects involved in our system design. We touch on three such aspects here.

3.3.1 Order Matching

When considering our order matching algorithms, a natural question arises: How do you efficiently match orders when the probability that two counterparties agreeing exactly on a price and quantity to trade is very low?

Centralised exchange platforms solve this by maintaining a market price in real-time based upon the relative quantity of buy and sell orders. Traders can place "limit" orders which will only execute once a certain price threshold is crossed, or they can place "market" orders, which will execute immediately at the currently calculated market price. This is similar to how traditional financial markets function as well.

If trades start to congregate too much on one side of the market (i.e. buy orders are outstripping sell orders, or vice versa) the exchange reacts by adjusting these prices until they are back in balance.

We take a slightly simpler approach. In our system, *all orders are limit orders*. Our order matching algorithms place orders into two respective queues, one for buy orders and one for sell orders. Orders are sorted based upon price, and our algorithms prioritise orders with the most aggressive pricing, so in the case of buy orders, the highest offered prices are pushed to the front of the queue. Similarly for sell orders, the lowest asked prices are prioritised at the front of the queue.

We allow buy orders to specify a *limit range*, with a lower bound offer and an upper bound limit. In contrast, sell orders specify a *simple limit* without a range.

The order matching algorithms take the most aggressively matched orders, so the orders at the head of each respective queue (buy/sell) and attempt to match them at the sell order's simple limit price. If the sell order's simple limit price is within the buy order's

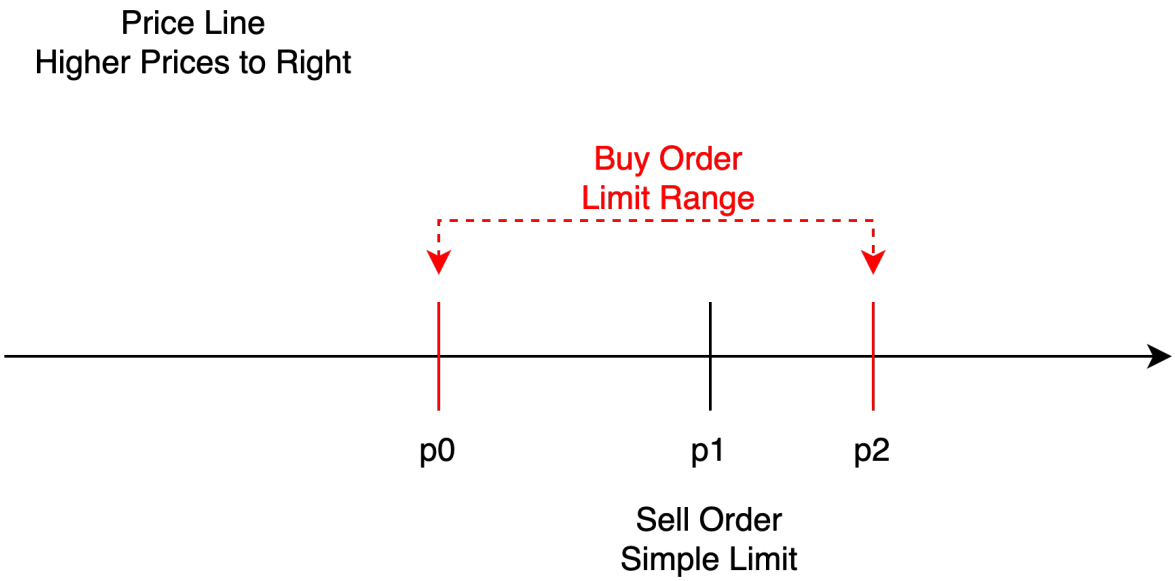

Figure 3.6: Potential Order Match. Here the buy order's limit range overlaps with the sell order's simple limit. These orders may match if other order parameters match up

limit range, the orders are a potential match. Further checks are carried out before determining if the orders suitably match with one another (cf. 4). Figures 3.6 and 3.7 demonstrate the two scenarios where orders might match or fail to match.

What this means is that is up to the discretion of the user to ensure that their placed order maximises their preferences regarding the tradeoff between maximising the chances of order execution and a more preferential price.

We pause to note here that these limits are also enforced on-chain by the Exchange contract, so validator nodes cannot simply modify the order matching algorithms to submit orders contrary to these constraints. In the traditional parlance of "Make/Take" commonly seen in Non-Custodial Orderbook protocols, here sellers **make** orders, while buyers **take** them.

It is difficult to say whether or not this arrangement specifically favors buyers or sellers, although we intended to slightly favor sellers in this scenario. Sellers are guaranteed to have their orders executed at their specified price, while buyers have greater flexibility regarding the tradeoff between execution price and order priority. (As they can more easily provide a range that is higher and therefore prioritised by the validator nodes)

We note that this implementation serves as the groundwork for more advanced and sophisticated implementations and we welcome further research into this topic.

3.3.2 Front-Running

Another economic consideration is front-running. There is not any mechanism that could prevent validator nodes from prioritising their own orders against others.

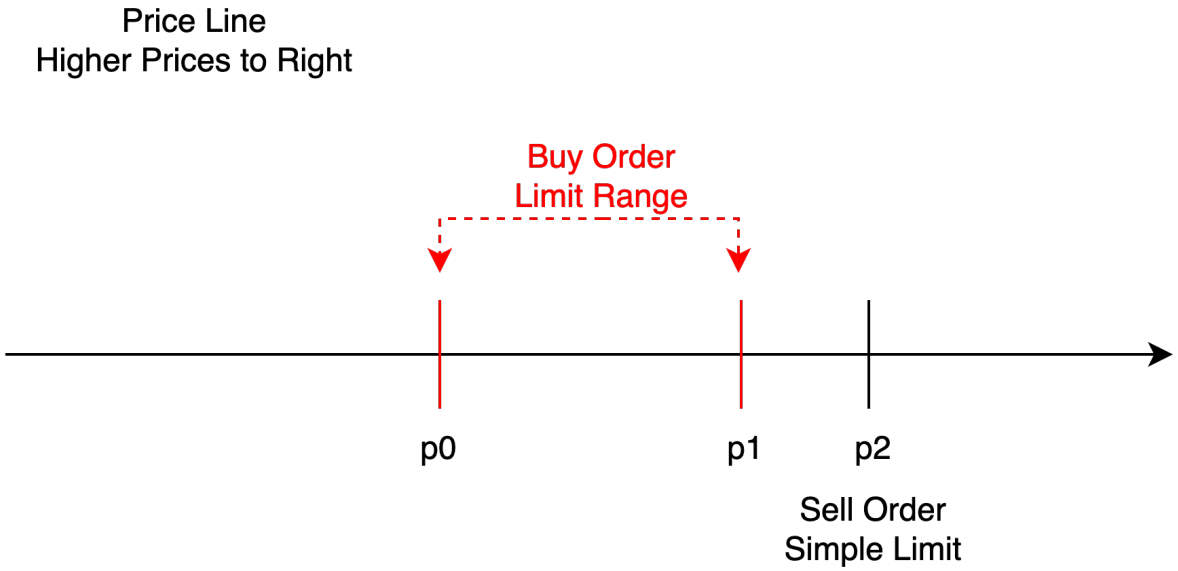

Figure 3.7: Unmatched orders. Here the buy order's limit range does not overlap with the sell order's simple limit. These orders will not match and are sent to an overflow queue

Luckily, in the case of sell orders, sellers are protected from any adverse effects, as the sell orders can only be filled at their specified limit price. (Again, this constraint is enforced at the smart contract level)

In the case of buy orders, there exists a scenario where validator nodes could relatively easily prioritise their own sell orders at the highest price possible, submitting orders exactly at the corresponding buy orders limit price. The risk to the buyer is then proportional to the size of their limit range. Higher limit ranges means that a validator could secure a larger advantage by prioritising their own order at the maximum possible price.

Luckily, in an efficient market with large amounts of liquidity, the necessary limit range for buyers to see their orders executed should be relatively small. As liquidity grows, we expect the prices of submitted orders to efficiently converge around a market price. In this scenario, only small limit ranges would be needed for buyers to have their order filled, as small deviations are enough to move their order to the front of the queue.

3.3.3 Commission Percentage

A final economic consideration we discuss is the commission percentage paid to validators.

Validators submit orders to the chain and incur a gas cost for doing so. In order to incentivise them to do so, commissions are paid out for every successful order execution. In this way validators are incentivised to only submit orders to the chain which they believe will be executed.

In a completely efficient market, we would expect that enough order validators would join the network until the gas cost of submitting an order to the chain were approximately equal

to the commission. Further developments to our system would do well to incorporate a market-variable commission structure that dynamically adjusts commission rates to match market activity.

For simplicity and true to the exploratory nature of our work, in our system 1% of the purchase price from the buy order is paid to the validator as a commission. This means that buyers bear the burden of the commission payment. However once again, economic theory tells us a more nuanced story: In an efficient market, the commission burden becomes equally shared by buyers and sellers as buyers incorporate the necessary commission payment into their purchase price.

3.4 Novel Techniques

In this section we present a few of the novel techniques we developed during the course of this work.

3.4.1 Delegated Signing

We were presented with an exciting problem during the development of the validator nodes. Like all users of the system, validator nodes run in the user's browser. Validators verify and send orders to the chain. It is neither possible nor desirable to retrieve the user's private key for security reasons.

Thus, in a naive implementation, validators would need to manually approve every transaction they send to the chain, an unacceptable user experience and a bottleneck for transaction throughput. We therefore developed an innovative solution to sign transactions on the user's behalf, a solution which we term *Delegated Signing*. We generate a random private key on the user's behalf using ethers.js, which we call the signer key. The signer key has an associated signer address. By allowing the user to send ether to this signer address, the signer key can now send transactions to the chain. We can now send transactions on behalf of the user without prompting them for transaction approval.

Theoretically, the process could stop here. However for obvious reasons, the signer key is not persisted anywhere. The signer key is maintained only in memory and immediately passed to a wallet object from ethers (cf. 4). Thus when the user would refresh the browser or navigate away from the application page, the web browser would clear the application memory and the signer key would be lost and render any ether at the signer address unrecoverable.

We developed this technique further in order to overcome this problem. In our system, the only truly persistent storage is the chain itself. Of course, as a public blockchain, any persisted information is exposed to the outside world.

Therefore we encrypt the signer key upon generation and store it on-chain in a mapping linked to the user's address. We ask the user to encrypt the signer key by requesting their

public encryption key. Fortunately, Metamask exposes a little-known API for doing this, see[2].

Once we have obtained the public encryption key, we perform a Diffie-Hellman exchange with an additional ephemeral private key using the tweetnacl.js library resulting in a "shared secret". (Shared is in quotes here as the ephemeral address is discarded) The result is a secret that can be used to encrypt messages in which only the user can decrypt via their "true" private key. We use this secret to encrypt the signer key in memory and then send it to the chain.

The user has now created a delegated signing authority which they have safely stored encrypted, on chain. Importantly, as only the user can decrypt the encrypted signer key, the user has the ability to selectively release access to the signer key to the application. (or more generally, other apps the user trusts)

When the user initialises the validator process in our application, we retrieve the encrypted signer key from the chain and ask the user to decrypt it. Upon decryption, we initialise a signer object in-memory and use this signer object to send orders to the chain. We view this innovation as one of the most promising outcomes of our work, and more generally, we envision this technique being useful for both delegated signatures (no transaction/gas) and delegated transactions (gas required). The only difference is that the signer requires a deposit of ether to initiate transactions.

We note that the usefulness of this technique is further compounded by the fact users can limit the amount of ether they send to the signer address. Additionally, users can continuously send ether as the signer sends transactions to the chain, or perhaps third parties could also deposit Ether into the signer address.

In summary, we see this technique as a powerful tool for application developers to enhance user experience and request a secure but limited signing authority where individual signing approvals may not be desirable.

3.5 System Interactions & Client Views

In this section we enumerate the main functions of our system using sequence diagrams. We also demonstrate the accompanying client side view from the application where appropriate. We include some miscellaneous views at the end for completeness.

3.5.1 Initialisation

Upon startup, the application already begins to perform necessary startup functions in the background. This includes initialising import global state variables & objects, including *ethersStore*, a component for managing interactions with the users wallet and the

[2] `https://docs.metamask.io/guide/rpc-api.html#eth-getencryptionpublickey`

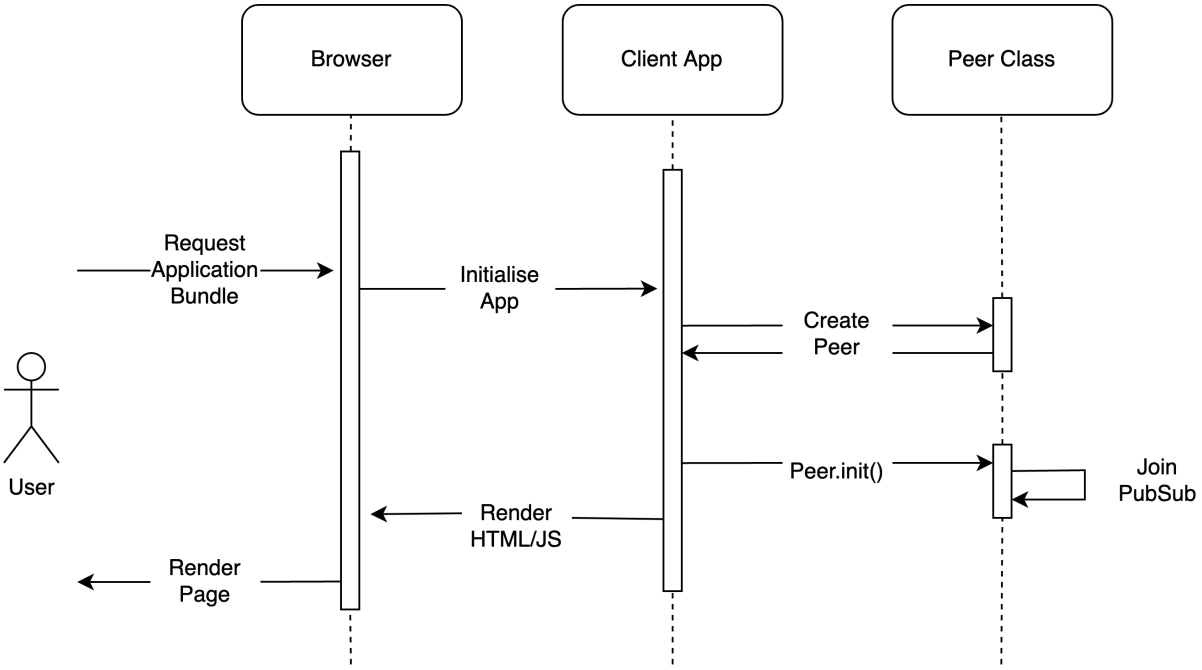

Figure 3.8: Peer Initialisation

blockchain. Additionally, the browser-based database (IndexedDB) and the peer networking functionalities are initialised.

The peer itself is initialised via configuration variables (cf. 4.3.2) and then connects with other peers in the network and joins our PubSub channels.

The user is presented with the dashboard view upon startup. (Figure 3.9) The dashboard presents a list of common functions that the system can perform for the user, as well as any recent orders present in the orderbook.

3.5.2 Token Administration

Users can easily create their own tokens and offer them for sale. In order to do so, users first create an entity linked to their address. The creation of an entity involves the creation of an ERC-1155 Multi-Token Contract linked to the user's address. Afterwards, the user can create an arbitrary number of multi-tokens from the linked contract. Users can create fungible or non-fungible tokens. Users can add dividends to their non-fungible tokens.

On the client-side, users navigate to the Token Administration section by using the header tab. The administration functions are contained therein.

Creating an Entity & Token Set

When creating an entity, the user supplies a unique identifier which is linked to the Multi-Token Contract and used to identify tokens associated with the entity. Once an address

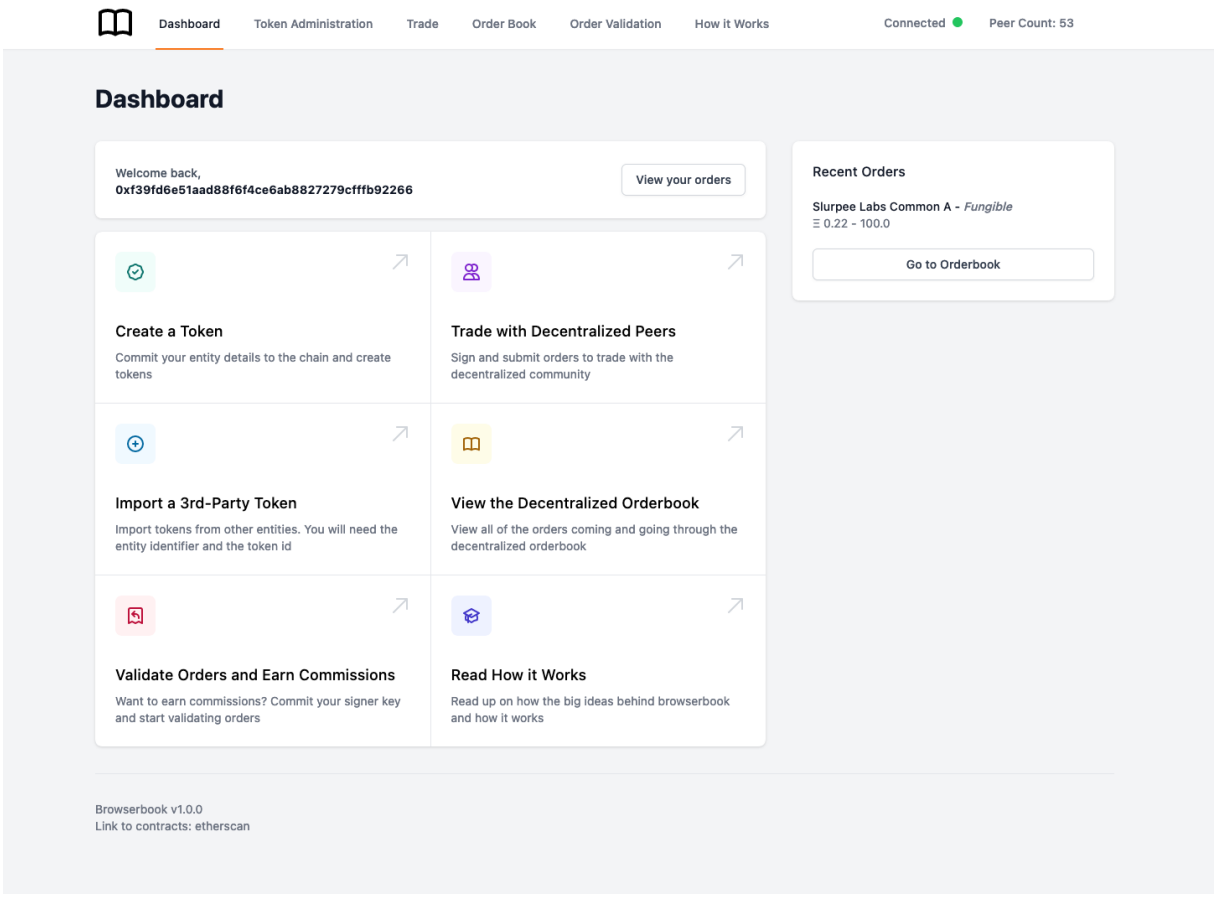

Figure 3.9: The client facing dashboard

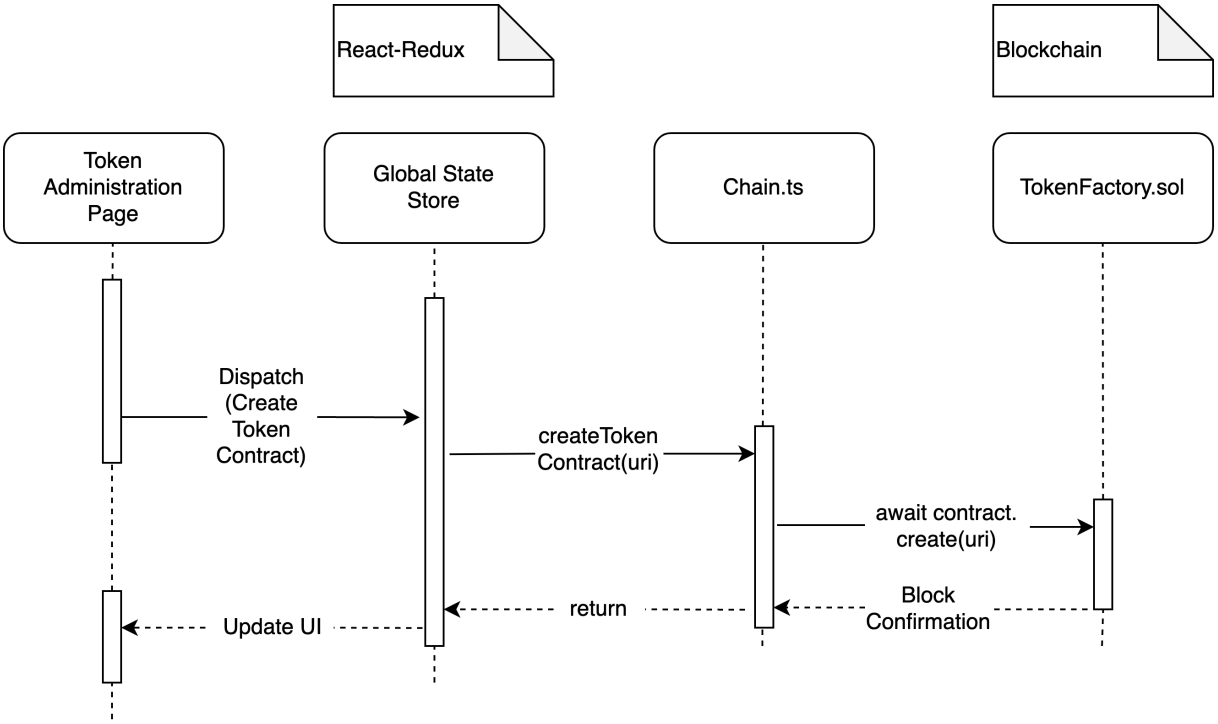

Figure 3.10: Creating an Entity

creates an entity, it cannot modify or create a new entity. If a user wishes to create a new entity, they would need to use a new address.

Creating a Token

Once the user has created an Entity, the user can now proceed to create individual tokens for sale. The user creates a token by specifying various parameters including:

- Token Name
- Token Metadata URI
- Token Type (Fungible or Non-Fungible)
- Token Supply

The user is free to give their token a memorable name as well as a metadata URI. We envision the metadata URI as an enabler of token metadata. For instance, fungible tokens might include information on the dividend schedule and initial purchase price. Non-fungible tokens might include information on specific rights gained by the token holder.

The metadata URI is left intentionally open-ended. Users are free to determine the most suitable manner for storing token metadata, such as on-chain, via IPFS, or traditional

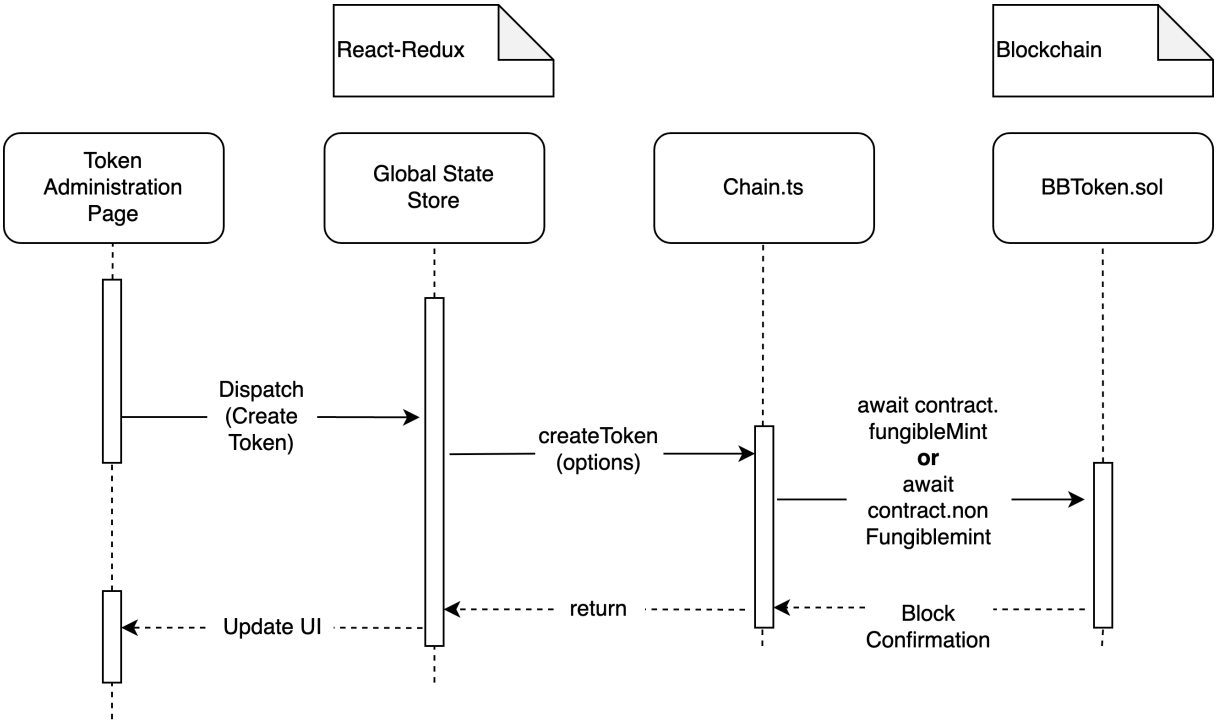

Figure 3.11: Creating a Token

server methods. The user also specifies the type of token (Fungible or Non-Fungible), as well as the token supply. The application automatically defaults to a supply of 1 if the Non-Fungible token type is selected.

Once a user has successfully created an entity, the token creation form will automatically display. Figure 3.12 shows the client's token creation view.

Adding Claimable Dividends

Users can provide dividends to their fungible token holders by providing a dividend amount to the Token Contract. The contract requires that clients specify which token is due to receive a dividend by giving the token id. Our UI lets users simply select the token by name; we map token ids to their respective names to provide a more seamless user experience.

Users must specify the amount of dividend to be provided in aggregate. Our system converts this amount to a per-holder amount, as this is what the contract expects.

Token holders can easily add dividends to their tokens by clicking on the "View" link on the token list. A modal popup will display presenting token information, as well as the option to add dividends to the token. Figure 3.14 shows the user interface presented via the token information modal.

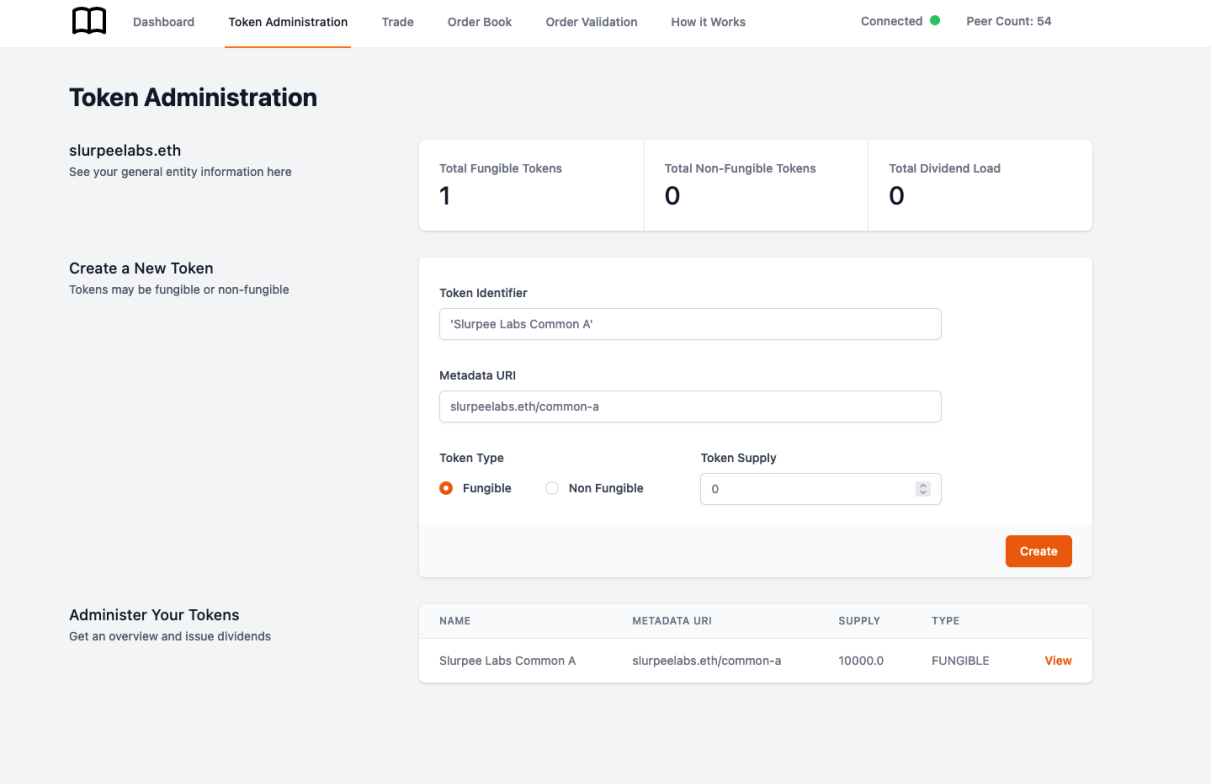

Figure 3.12: The token creation view. Users can specify a token name, metadata identifier, type, and supply. Token supplies default to 1 for non-fungible tokens

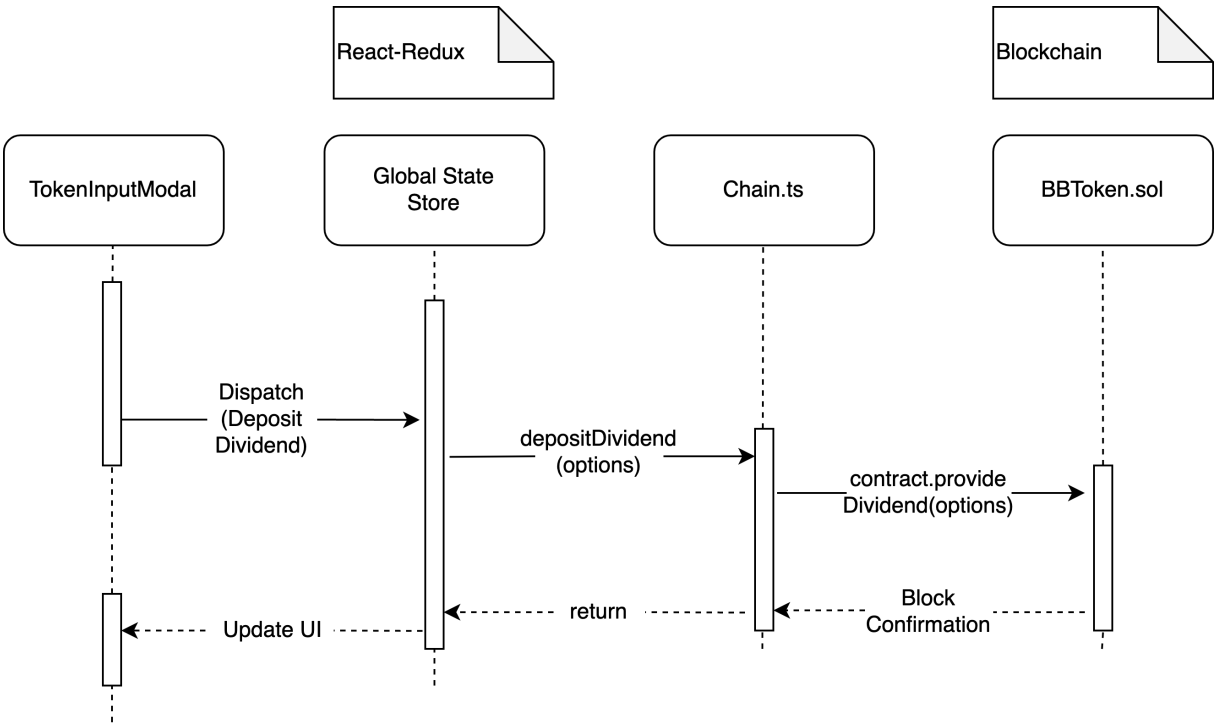

Figure 3.13: Providing a Dividend to Token Holders

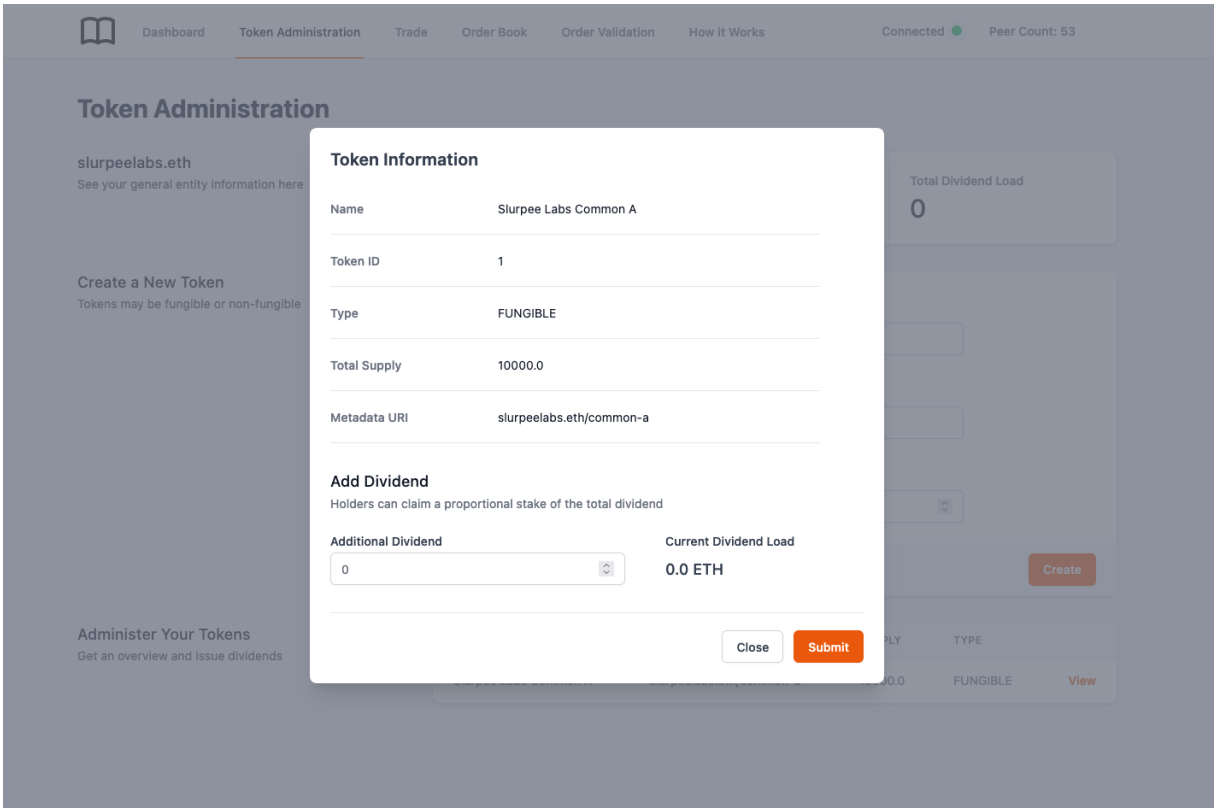

Figure 3.14: Adding dividends to a token for claiming by holders

3.5.3 Token Trading

Users can trade both their own tokens, and third-party tokens. In order to trade third-party tokens, they must first ”import” the token. Users import tokens by providing the entity URI and a token id.

There are some additional steps to be taken before a user is fully ready to trade. Users who will submit buy orders must submit a deposit to the exchange contract which will be used to make purchases and transferred to sellers. Users who will submit sell orders must grant the exchange contract approval to transfer their tokens on their behalf. Most users would perform both actions as most users will want to submit buy and sell orders.

Users can also withdraw their exchange balance at any time.

Users also have the ability to simply transfer a token to a third-party directly by supplying an address. This is useful as a UX enhancement and for situations where tokens need to be transferred without a sale taking place. (i.e. inter-company transfers).

In order to navigate to the trading functionalities, the users select ”Trade” from the header menu, and the application will route them to the trading portal. The trading portal contains a sub-menu used for sub-navigation within the trading portal. The user is presented with a list of links on the left hand side of the page:

- *Trade*: Submit buy and sell orders to the marketplace

- *Transfer*: Transfer tokens to other users of the system
- *Tokens*: View both created tokens and third-party imported tokens. Add a new third-party token
- *My Orders*: View all placed orders, their details and status
- *Balance*: View, deposit, and withdraw against the user's exchange balance

Submitting an Order

To submit an order, users navigate to the first sub-navigation tab *Trade* and enter order details into a form. Upon submission, order details are collated into an object and the order is given a randomly generated id. The following details make up an order:

- A randomly generated order id
- The sender
- The traded token contract address (ERC-1155 Multi-Token Contract address)
- The traded token id
- The type of order (Buy or Sell)
- The offered price
- The limit price
- The offered quantity
- The expiry in milliseconds

Once the order has been generated, the user must sign the order with their Metamask wallet. We utilise an implementation of EIP-712[12], which is an Ethereum standard for signing structured data objects(cf. 1.3).

Signed orders are then persisted to the user's database (IndexedDB) and broadcasted to peers in the network. If an order is filled, a validator node notifies all peers of execution and the peers update their respective orderbooks locally to reflect the status that an order has been filled. Note that in the sequence diagram the balance check for buy orders is not shown, as it performed prior by the TradeForm component directly.

Figure 3.17 shows the client view for entering and submitting a trade.

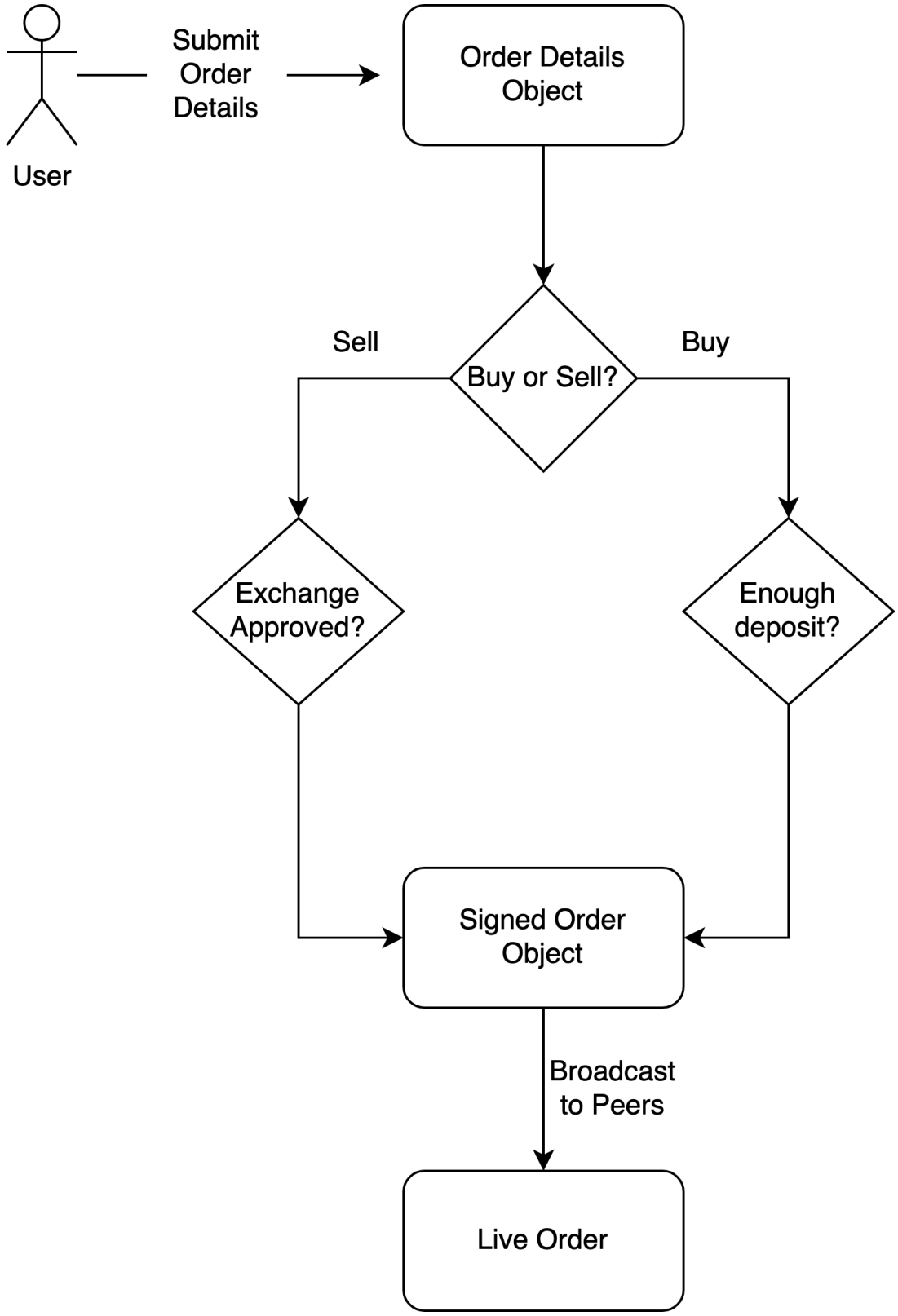

Figure 3.15: The Order Lifecycle

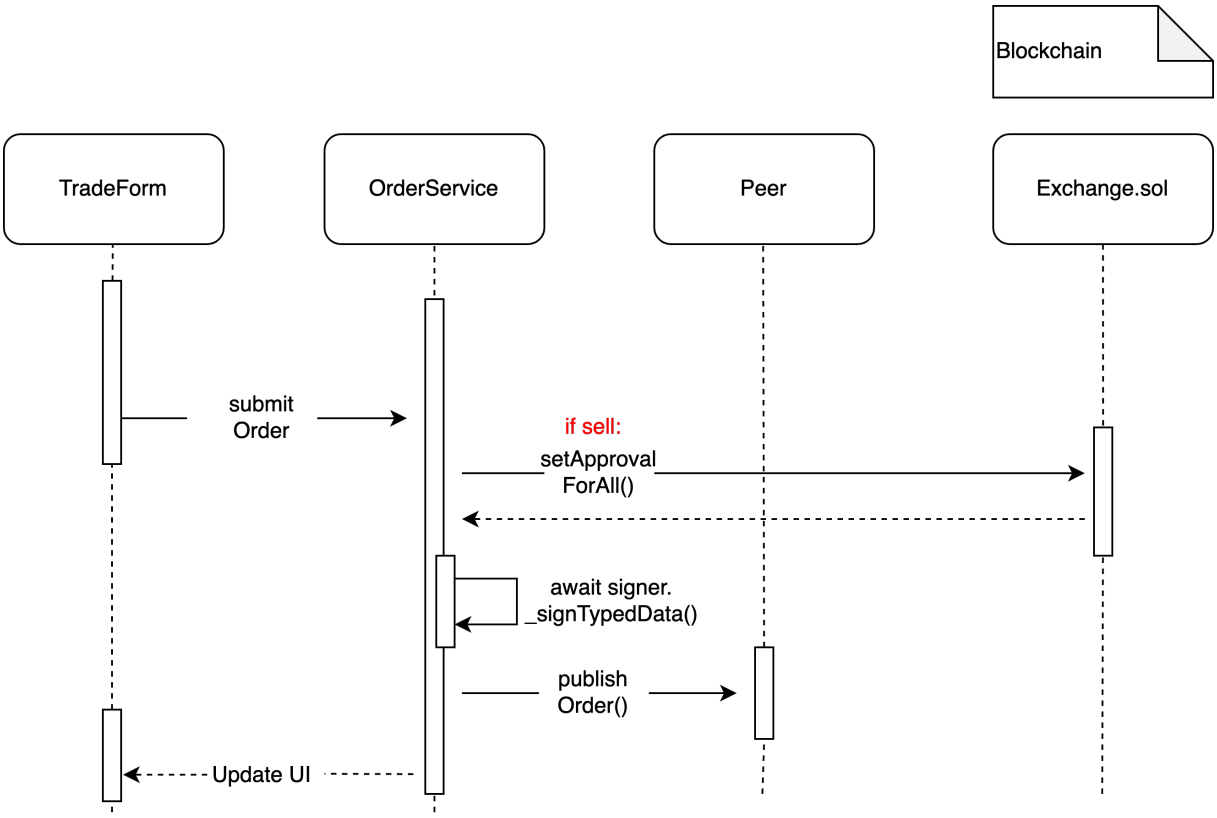

Figure 3.16: Submitting an Order

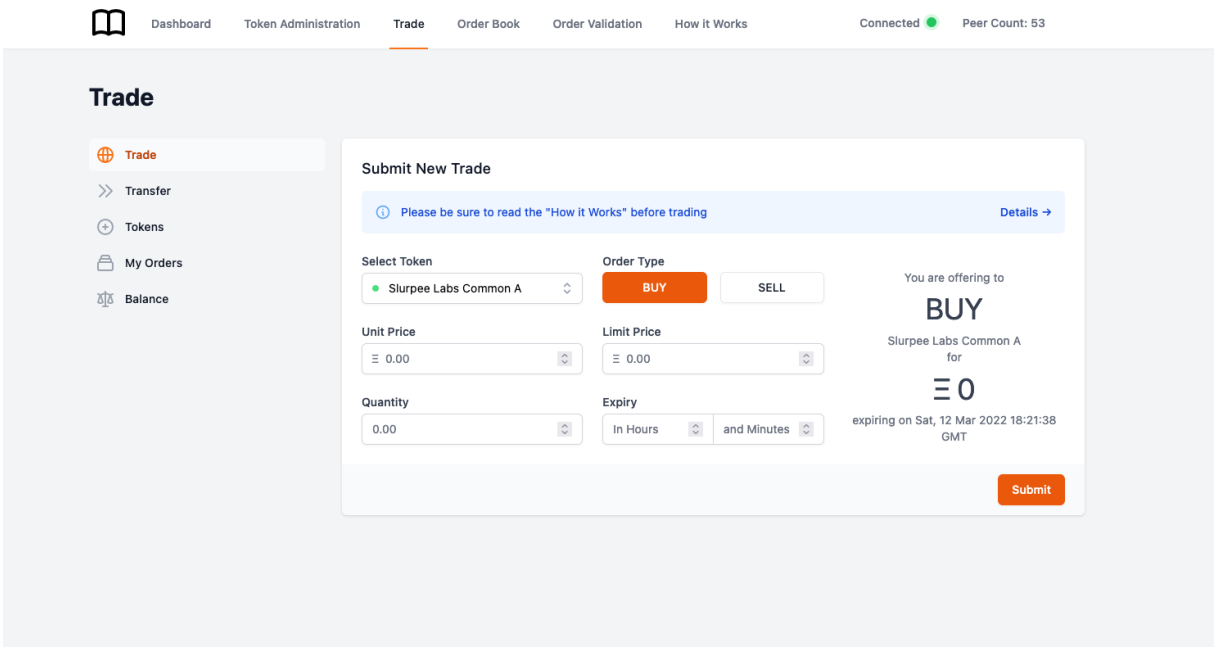

Figure 3.17: The client order submission form

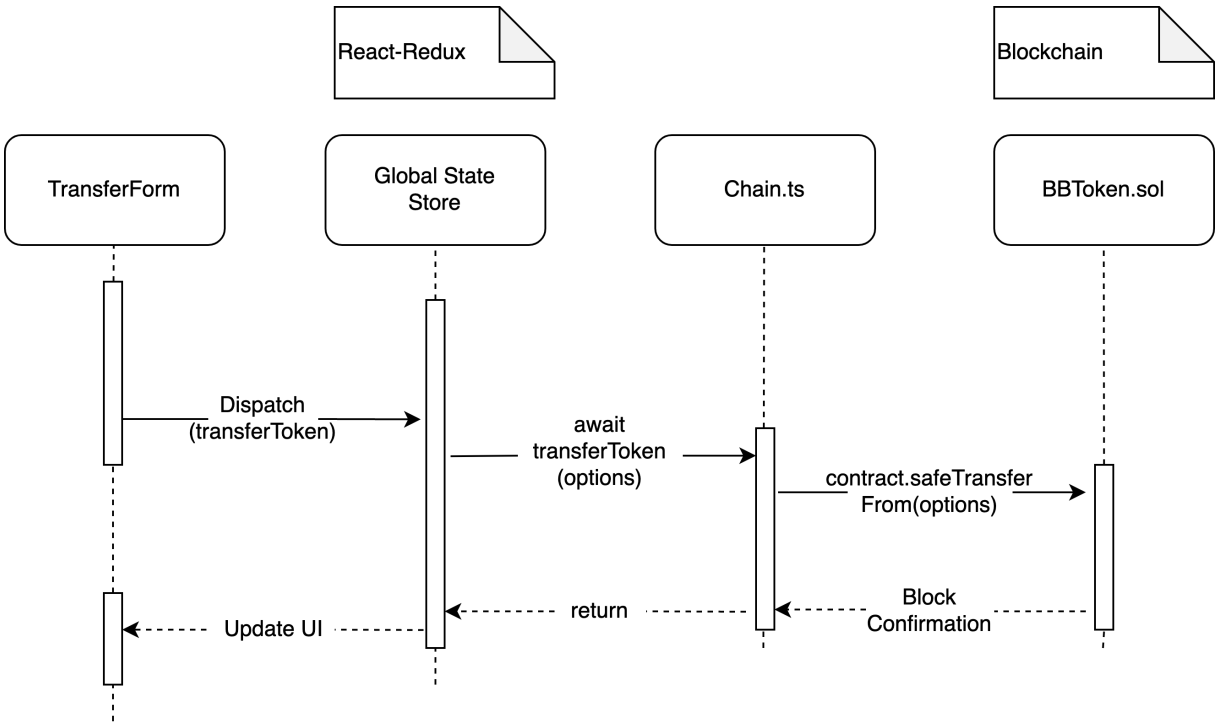

Figure 3.18: A direct token transfer

Transferring a Token

To facilitate transfers of tokens without the necessity of entering into a trade, we allow users to transfer tokens directly to another address without compensation.

Figure 3.18 shows the procedural flow of a direct token transfer. Users can transfer tokens by visiting the *transfer* tab.

Importing a Third-Party Token

Users need to be able to trade tokens created from third-party entities. All token information is stored on-chain. We rely on users to specify which tokens they intend to hold and trade so that we can import this information from the chain.

In order to import a third-party token, users first navigate to the *Tokens* tab in the sub-menu navigation, and then click a button in the top-right hand portion of the screen. This presents users with a new token modal. (See Figure 3.20)

To facilitate a pleasant and seamless user experience, users simply need to input the entity URI and the token id. The system retrieves the appropriate token by first querying the appropriate contract address of the ERC-1155 Multi-Token associated with the entity id. Thereafter it retrieves the token information.

In order to persist the token information between browser sessions, the token information is saved in the client's IndexedDB.

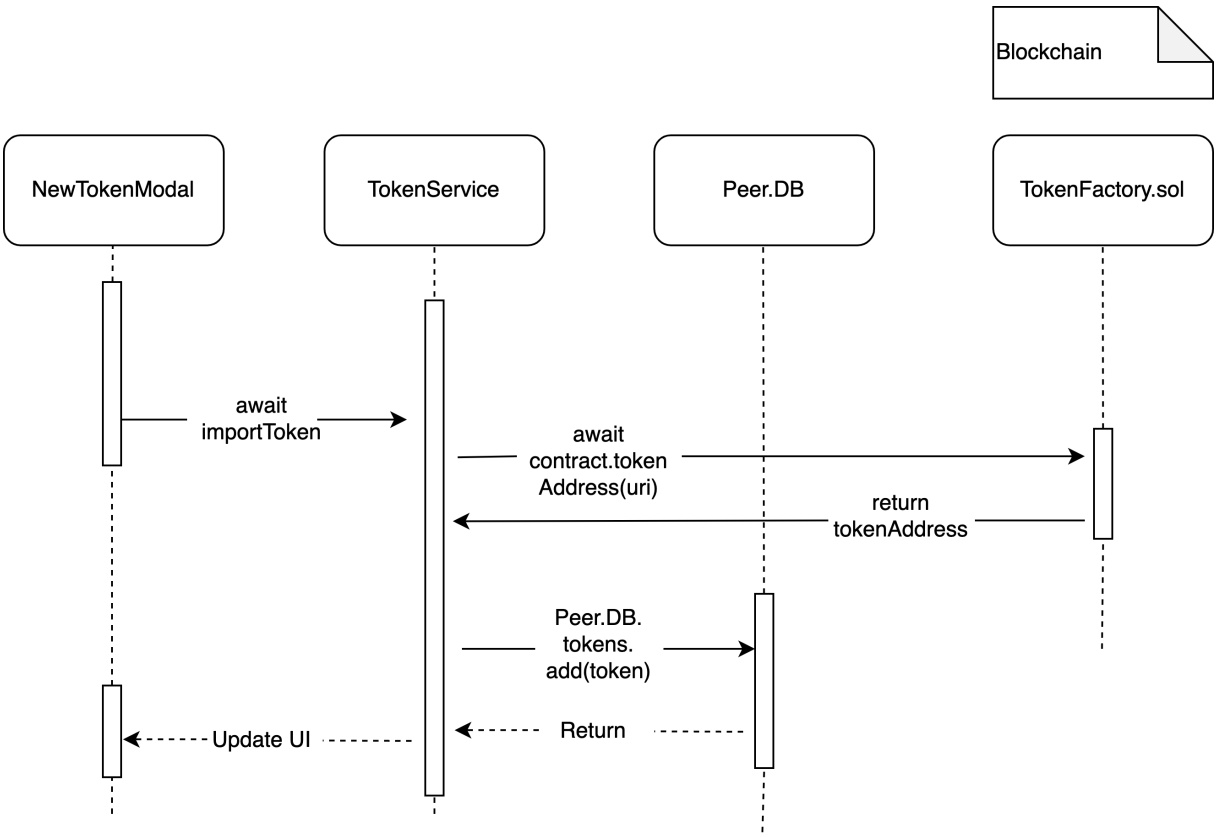

Figure 3.19: Importing a Third-Party Token

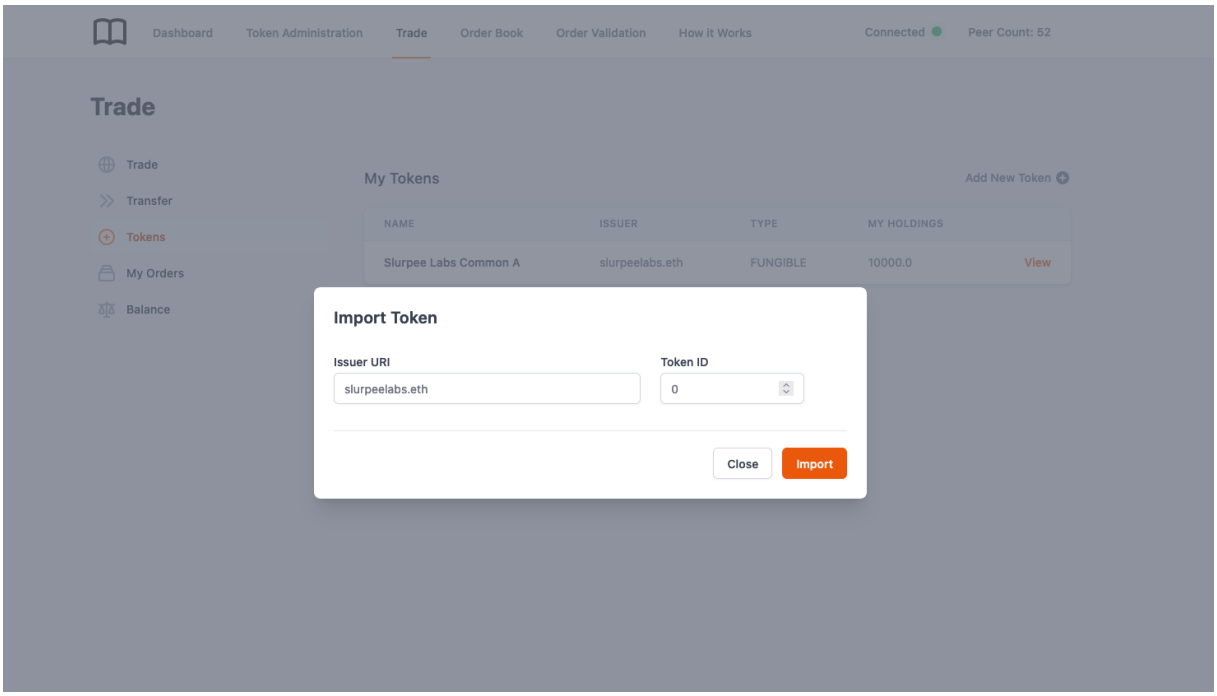

Figure 3.20: Client-side view for importing a new token. A client only needs to import an entity URI and a token id, the system handles the appropriate queries to the blockchain that locates the actual ERC-1155 Multi-Token contract address

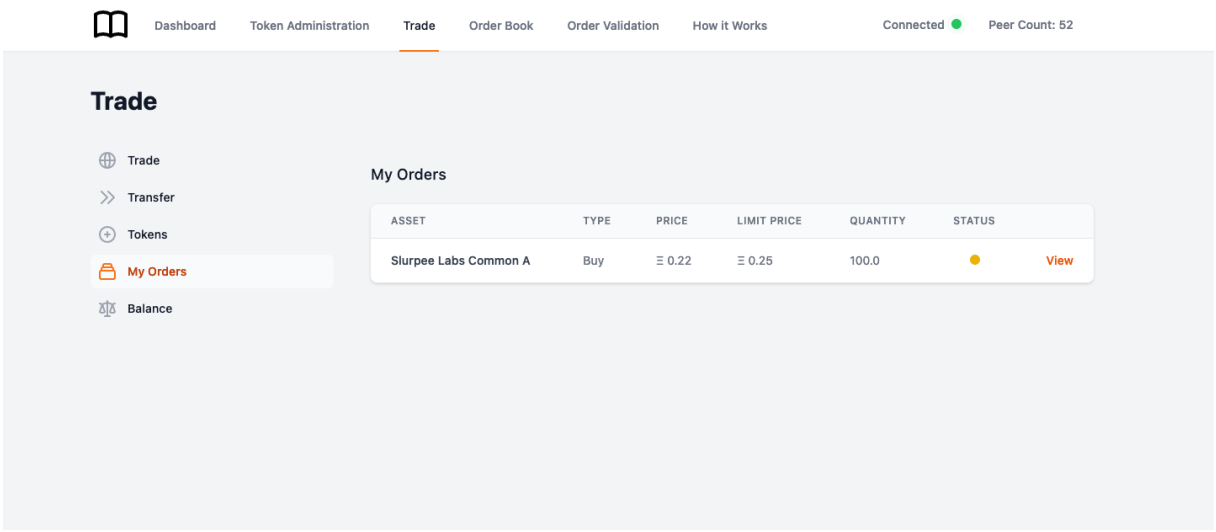

Figure 3.21: Client view of own orders. Here one order is visible which has a status of ”Pending” as indicated by the yellow bulb. This color would change in real-time to green or red depending on messages from the validator nodes once the order is executed against a counterparty

Viewing Orders

Users can view their own orders by visiting the *Orders* tab. There users are presented with an overview of their orders as well as their status. An order can be ”Pending”, ”Matched”, or ”Expired/Rejected” based upon messages received from validator nodes regarding the execution status of an order. An order is pending when no match message has been yet received. Upon a match message being received, the status will change to ”Matched”. If an order is rejected by a validator or expires, it’s status changes to ”Expired/Rejected”.

We use a user-friendly dot representation with a variable color to indicate order status in the orderbook.

- Green: Order Matched
- Yellow: Order Pending
- Red: Order Rejected/Expired

Adding a Trading Deposit

Users submit an exchange deposit by navigating to the *Balance* tab and specifying an amount. Users submit this value to the Exchange Contract in a transaction.

Figure 3.23 shows the client-side interface for adding a deposit to the exchange contract.

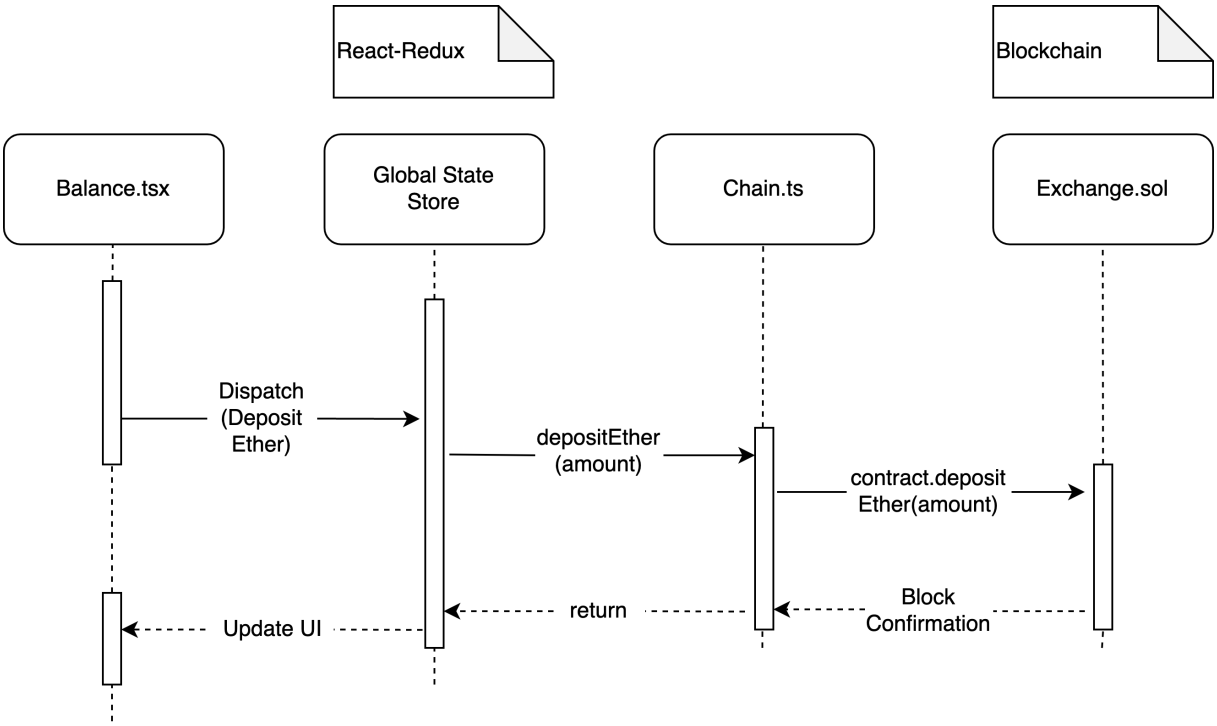

Figure 3.22: Depositing Ether into the Exchange Contract for Trading

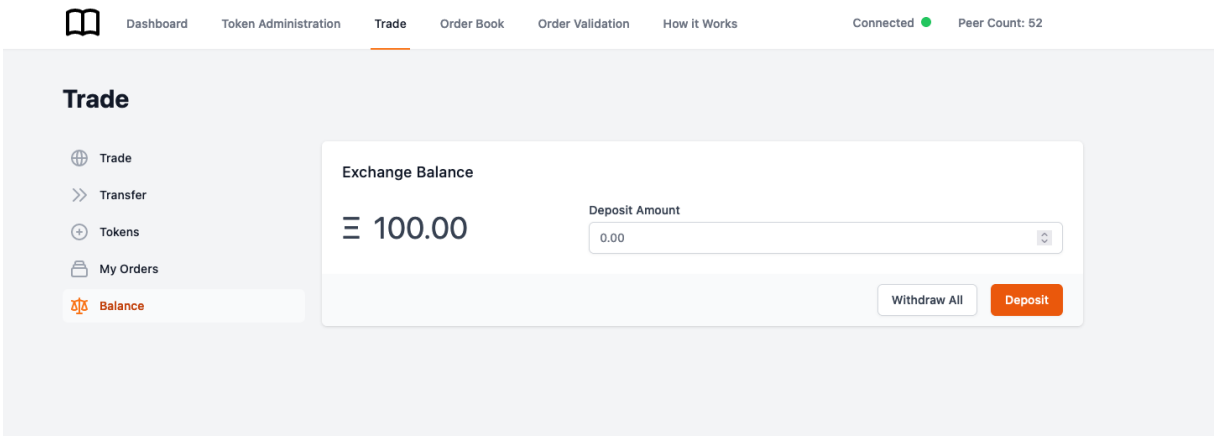

Figure 3.23: The client-side view to deposit ether for trading

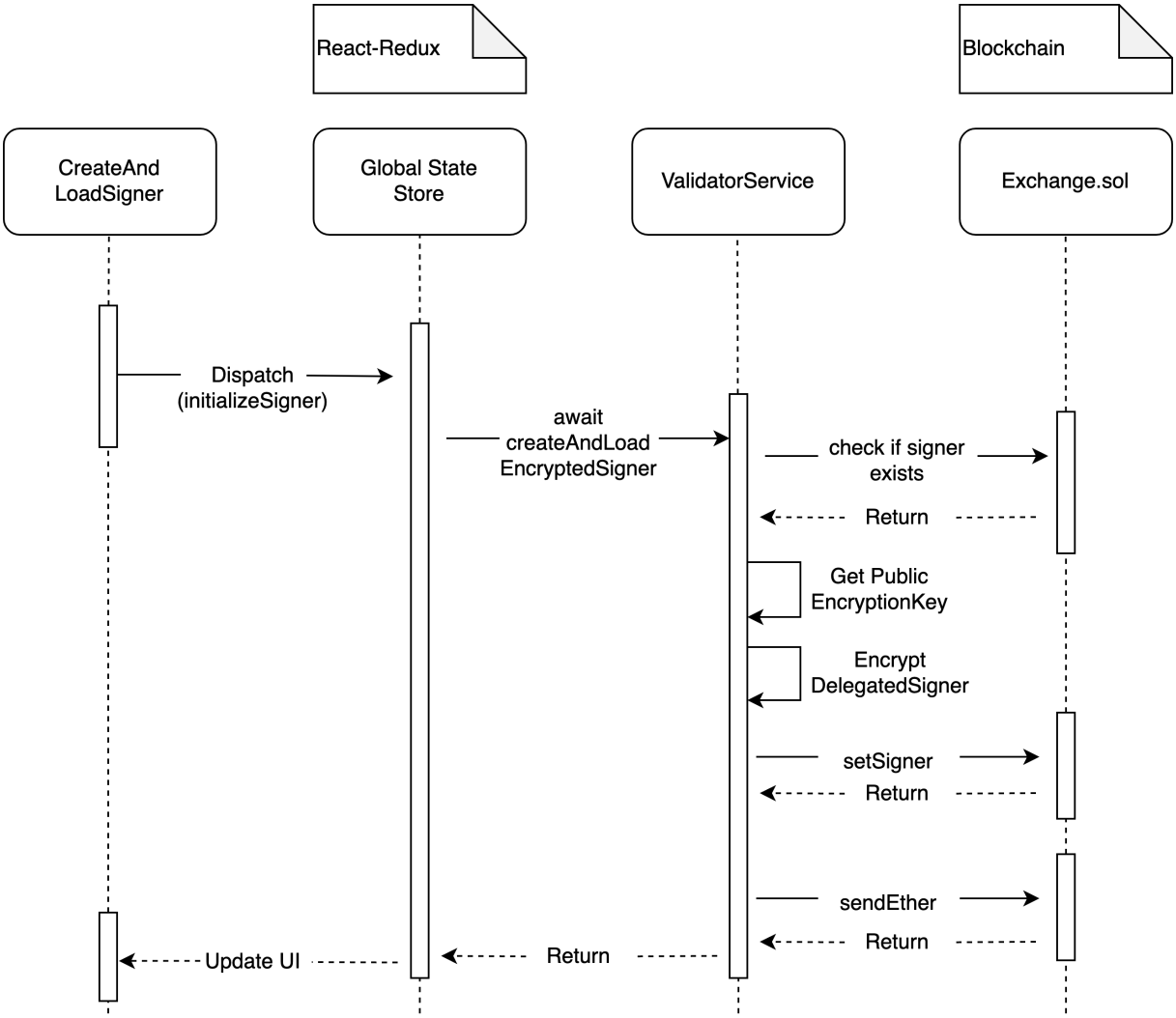

Figure 3.24: Initialising a Delegated Signing Authority

3.5.4 Order Validation

Validators watch the orderbook and attempt to send orders to the chain. Validators pay the necessary gas fees and in return receive a commission on trades.

We use the aforementioned delegated signing technique in order to obviate the need for the validator to manually sign individual transactions. Validators initialise and encrypt a delegated signing key and store it on-chain. Validators must send ether to the signer address in order that the signer can pay the gas fees.

Initialising a Signer

During the initialisation process, the system first checks to ensure that the signer has not yet been initialised. If it were, the initialisation process would overwrite the existing signer key, and any funds held by the signer would be lost.

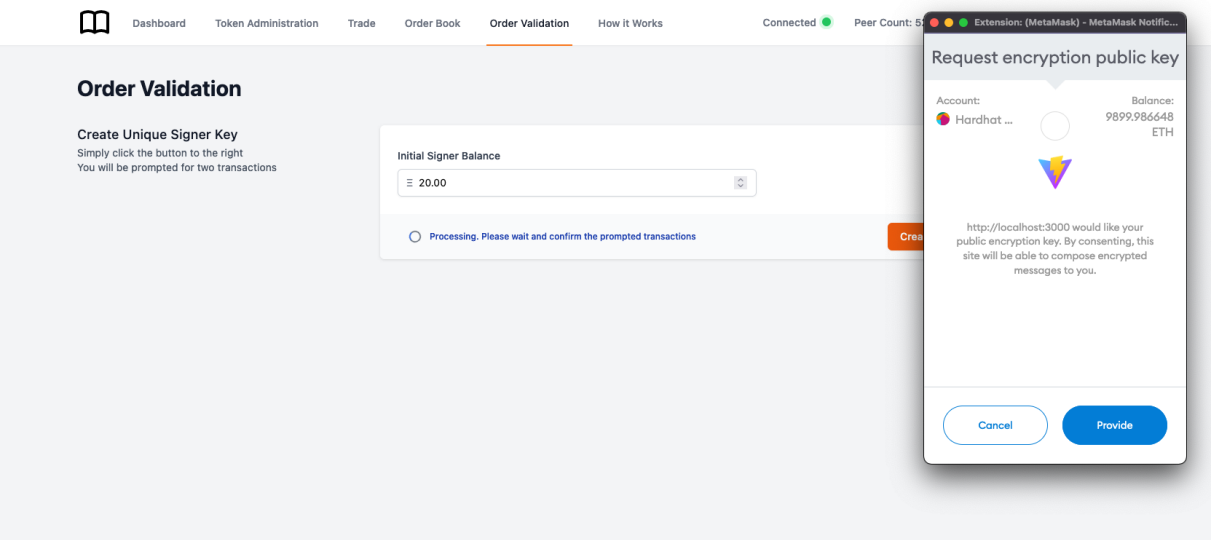

Figure 3.25: The client approving a transaction to encrypt their delegated signer

Despite the relatively complex interactions happening behind the scenes - the interaction for the user is very straightforward. The user simply needs to submit an initial deposit amount to the contract.

The process then proceeds in three steps and requires three transaction approvals from the user:

- Request the public encryption key from the user's Metamask wallet
- Encrypt a delegated signing key and store it on chain (requires gas)
- Send an initial deposit of ether to the signer address

Initialisation is a one-time process. Once the initialisation has completed, upon revisiting the page the user only needs to decrypt the on-chain signing key (no gas required) to re-initialise the signer and begin matching orders again. This means that to re-initialise order matching only requires one UI interaction, which is the ideal in terms of user experience.

Matching & Submitting Orders

In order to begin matching orders, the validator needs to send a signal to a web worker process. We use web workers to run our Order Management System so as to not block the main Javascript thread, which could cause an unsuitable delay in the rendering of the UI and an associated decrease in UX quality.

The main process communicates with the web worker process via messages. The main process sends the "start" message to instruct the web worker to begin processing orders. The main process also sends new orders to the web worker as they arrive from other peers.

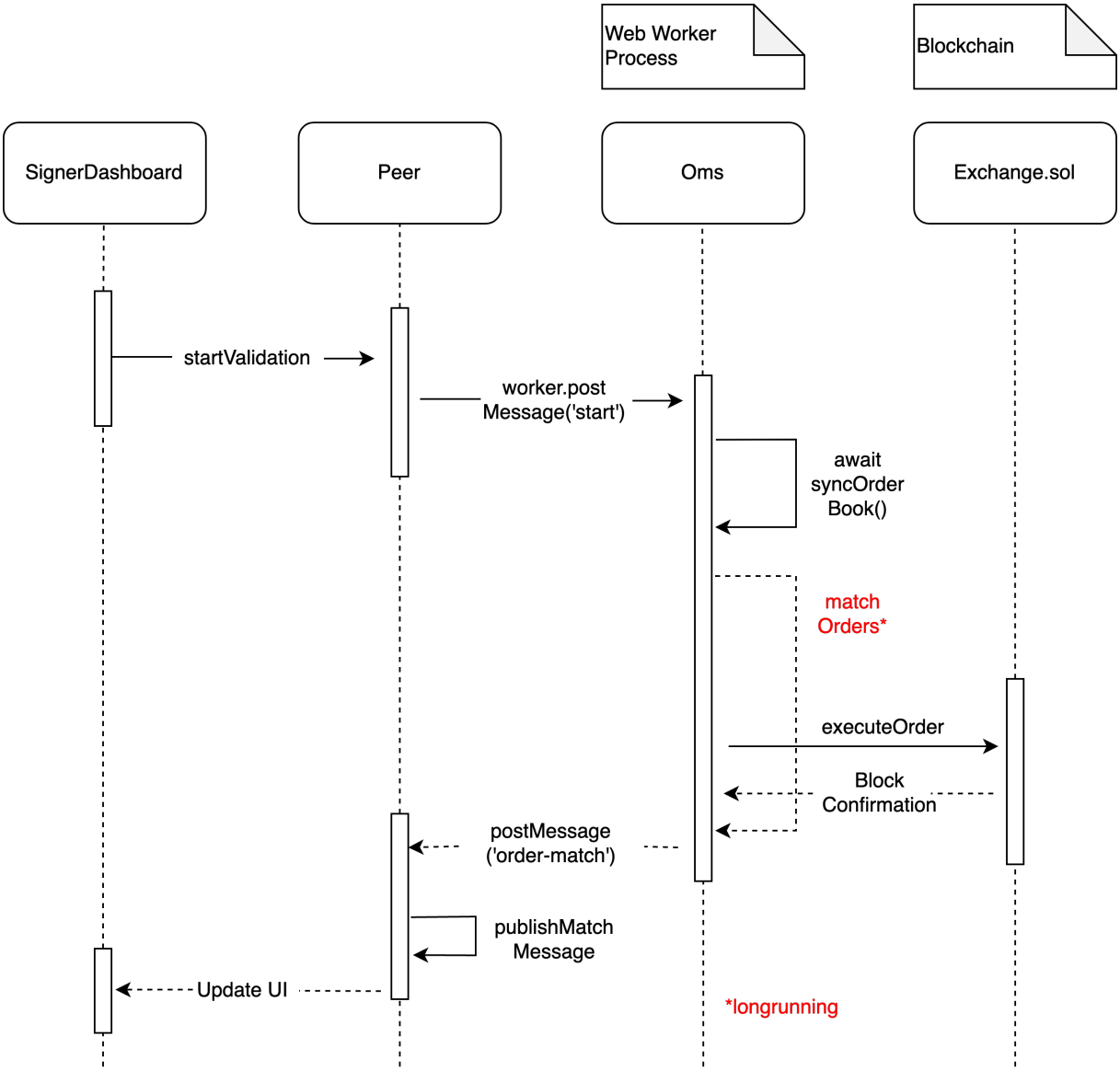

Figure 3.26: Matching Orders via the OMS in a Web Worker process

As orders are executed, the web worker process sends messages back to the main process notifying it of the match. The main process then broadcasts a message to other peers notifying them of the match. The peers can then update their own orderbook accordingly.

Figure 3.27 shows the delegated signer in action matching orders and sending them to the chain.

3.6 Dependencies

Our project depends on a lot of other great software. Here we pause to illuminate some of the key dependencies in our stack.

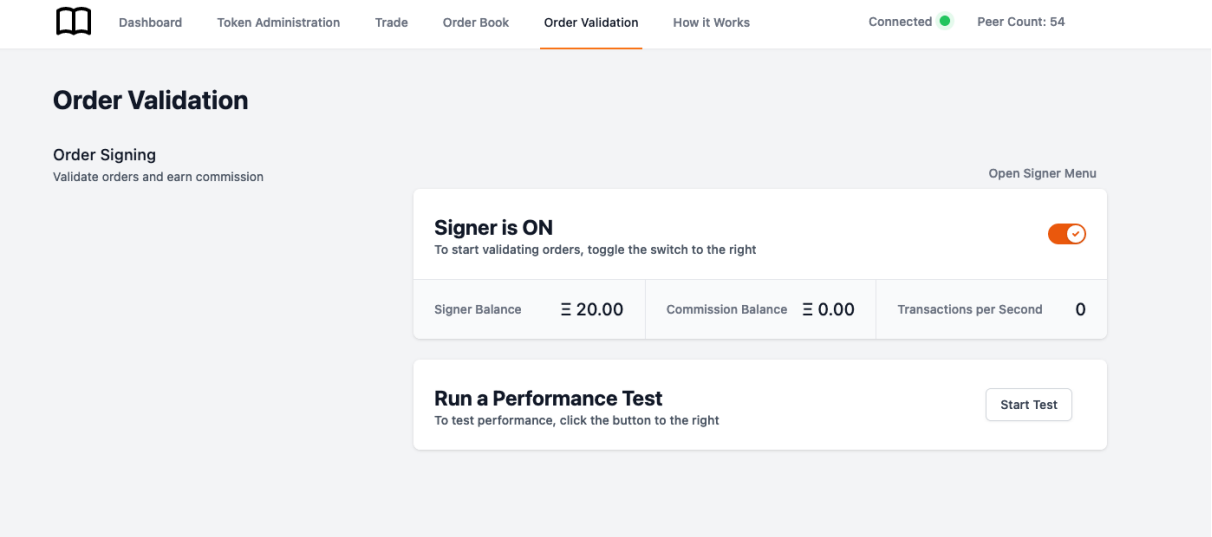

Figure 3.27: The client view of the order matching process

Libp2p

Libp2p is a set of networking libraries that facilitate the creation of peer-to-peer applications and protocols[36]. Libp2p includes utilities for bootstrapping nodes, initialising the PubSub and gossip protocols, storing peer locations in a distributed hash table, and more. Our system utilises libp2p to facilitate peer discovery and communication.

Dexie.js

Dexie.js[3] is a wrapper around browser-native IndexedDB implementations. Dexie provides a more usable abstraction for application developers to create, modify, update, and delete IndexedDB stores.

We use Dexie to handle all of our interactions with the client-side database in the browser's IndexedDB.

Ethers

Ethers[4] is a robust and user-friendly library for interacting with the Ethereum blockchain. Ethers provides utilities and abstractions for creating contract objects, reading the blockchain state, making RPC calls, interacting with browser-based signing authorities like Metamask, integrating with smart contract ABI's, and other common utilities. We use ethers for all application interactions with the blockchain.

[3] `https://dexie.org/`
[4] `https://docs.ethers.io/v5/`

Tweetnacl.js

Tweetnacl.js[5] is a high level cryptography library which provides utilities for private key encryption, public key encryption, hashing, and signatures.

We use tweetnacl.js to encrypt our delegated signing keys before sending them to the blockchain for storage. For details please see the Implementation chapter.

React

React.js[6] is a very well-known and popular javascript framework for creating user interfaces. Our client-side application uses react to manage application layout and state.

React uses a special syntax to compile javascript to HTML called "JSX".

Redux

Redux.js[7] is a global state container for javascript applications. Redux centralises application state into a global state tree and provides an API for both subscribing to state changes and modifying the state. Redux also facilitates asynchronous UI updates and easier debugging. We use Redux to manage global application state throughout the application.

Redux additionally comes with first-party bindings with React, making it a pleasant choice for development.

Ts-proto

Ts-proto[8] is a smaller library we use for encoding peer messages before publishing on our PubSub channels. Ts-proto reads a protobuf specification and generates Typescript classes along with utilities for encoding/decoding objects into/from a binary representation.

Tailwindcss

Tailwindcss[9] is a popular css library that provides "atomic css styles". Tailwindcss eschews the traditional accepted best practice of separating markdown and styling information and instead provides developers with a wide suite of utility classes for styling components directly in their markdown specification. We use tailwind to style all of our project's components.

[5] `https://tweetnacl.js.org`
[6] `https://reactjs.org/`
[7] `https://Redux.js.org/`
[8] `https://github.com/stephenh/ts-proto`
[9] `https://tailwindcss.com/`

Typescript-collections

Typescript-collections[10] is a small library offering a variety of well-known datastructures for developers. We use typescript-collections for its queue implementation, which we utilise in our order matching algorithms.

Infura

Infura[11] is an Infrastructure-as-a-Service provider that facilitates easy access to the Ethereum network via JSON-RPC calls[32]. The service provider runs Ethereum nodes on behalf of the application developer in order to free the development team to focus on their smart contract and web3.js functionality.

We use Infura to process our Ethereum RPC's.

Hardhat

Hardhat[12] is a suite of tools and libraries that provide an Ethereum development environment, assisting developers in committing smart contract code, running migrations, deploying to local test networks[30].

We use Hardhat to develop, compile, test, and debug their smart contracts.

Metamask

Metamask[13] is a Ethereum-based wallet and signing authority that runs as a browser extension in the user's browser[39]. While our project does not use Metamask directly, our interactions with the chain depend on it, so it's worth mentioning here.

We should also note that Metamask is not the only option for performing these functions, but it is the most popular and the one we developed around.

[10] `https://github.com/basarat/typescript-collections`
[11] `https://infura.io/`
[12] `https://hardhat.org/`
[13] `https://metamask.io/`

Chapter 4

Implementation

4.1 Blockchain

We determined that Ethereum represents the best choice for our system due to its high number of users and already existing decentralised applications. Ethereum is well-known for its diverse utility in the blockchain ecosystem but does suffer from slower block finality and higher gas fees. Our implementation specifically remains compatible with any EVM compatible chain, including compatible L2 solutions, rather than being rigid and supporting only one particular Layer 2 solution. To support any network other than the current implementation on Polygon Network, users simply need to switch the RPC url in their browser based signing authority.

4.2 Smart Contracts

A small set of smart contracts is used to facilitate the creation and exchange of tokens and stores the only persistent state in our protocol. Figure 4.1 demonstrates the architecture and their dependencies. We adopt the common notation that bold contracts represent singleton contracts whereas those shown in normal font can be deployed an arbitrary number of times.

4.2.1 BBToken

The BBToken contract is the ERC-1155[46] compatible Multi-Token contract which implements the OpenZeppelin ERC1155 contract specification[50], and can be used to create an arbitrary number of fungible and non-fungible tokens. We find it helpful to refer to BBToken as a ”Token Set”, as itself is a contract which allows the creation of an arbitrary number of individual tokens. The BBToken contract extends the ERC1155 Multi-Token standard, which facilitates up to 2**256-1 individual tokens. Tokens can be either fungible or non-fungible.

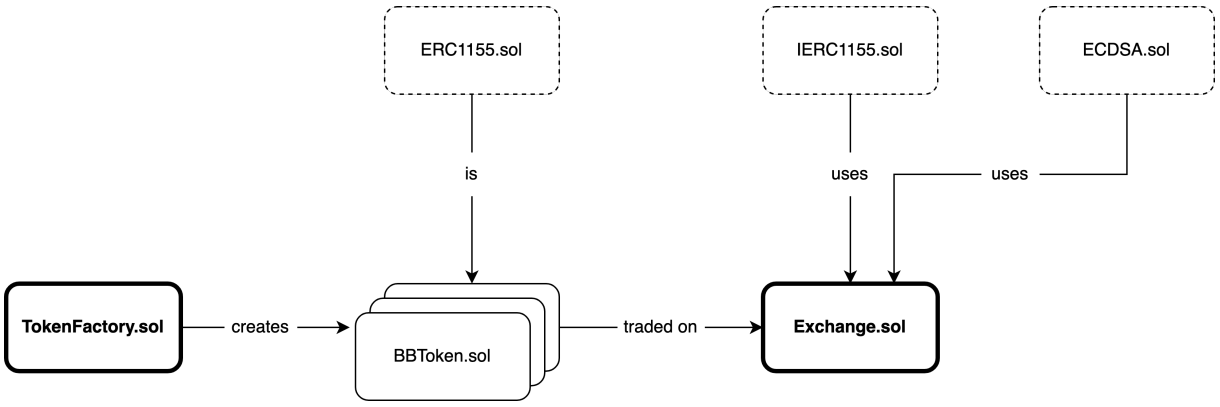

Figure 4.1: Smart Contract Implementation Overview

Our contract allows the inclusion of additional properties to a particular token, specifically, we allow Token Set Owners to create fungible tokens with dividend capabilities. Token Set Owners can provide dividends to Token Holders, and Token Holders can query and claim any available dividends. Additionally, Token Set Owners can provide a unique name and metadata URI to both fungible and non-fungible tokens. Figure 4.2 details some of the state variables present in an individual BBToken contract.

For the avoidance of doubt we clarify our usage of the following:

- **Token Set Owners**: Owners of a BBToken Multi-Token Contract. There can only one Token Set Owner per BBToken contract, and they can create an arbitrary number (up to the maximum) of individual multi-tokens. The Token Set Owner mints and ”owns” the entire BBToken contract, hence the name.
- **Token Holders**: Any wallet address that holds an individual multi-token, created within a BBToken contract. A Token Holder need not be a Token Set Owner.

```
contract BBToken is ERC1155 {
    ...
    // Token contract owner address (Token Set Owner)
    address private _contractOwner;
    string public contractURI;
    ...
    // Keep track of unique token identifiers
    uint256 public tokenNonce;
    // Token variables
    mapping(uint256 => string) public tokenNames;
    mapping(uint256 => uint256) public tokenSupply;
    mapping(uint256 => string) public tokenMetadata;
    mapping(uint256 => bool) public isNonFungible;
    ...
    // State variables for tracking dividend claims
    // tokenId => address => amount
    mapping(uint256 => mapping(address => uint256)) public
        fungibleHolderAmount;
    mapping(uint256 => mapping(address => uint256)) public
        fungibleHolderDividendClaim;
    ...
}
```

Figure 4.2: State variables of BBToken smart contract

4.2.2 TokenFactory

The TokenFactory contract is a factory contract which lets any user create or mint their own Multi-Token contract, BBToken, or a "Token Set". Wallets can only mint exactly one Token Set, no more. Upon minting a Token Set contract, users become the Token Set Owner by default. It is not possible to transfer ownership of a Token Set after creation.

The created Token Set contract has a specific contract address at which it is deployed on the blockchain. We use a two-step process in order to allow clients to easily query the address location of deployed contract. When a client calls the *create* function, we first collect an string-based URI which we refer to as the Token Set identifier, or sometimes the "entity" identifier. (It is useful to think of owners of a Token Set as "entities", for instance as an individual or firm) This entity identifier can be viewed as a unique identifier for the entire Token Set. The *create* function deploys a new instance of the BBToken contract and stores the deployment address in a mapping indexed by the unique identifier provided by the minter. We also store the identifier in its own mapping indexed by the minter's wallet address. This facilitates the following:

- Clients can query the address location of Token Set contract solely based on the entity identifier
- Token Set owners have a unique & immutable pointer from their wallet address to their Token Set contract address via the entity URI

```
contract TokenFactory {
    // Only allows creation of a single token
    mapping(address => bool) private _createdToken;
    // Address to token set URI
    mapping(address => string) public addressURI;
    // Token set URI to ERC-1155 token address
    mapping(string => address) public tokenAddress;

    // Event emission aids client-side discovery
    // (owner address, URI, token set address)
    event TokenContractCreated(address indexed, string, address indexed);

    function create(string calldata URI) public returns (address) {
        require(_createdToken[msg.sender] == false, "CANNOT_CREATE_MULTIPLE_TOKENS_PER_ADDRESS");

        addressURI[msg.sender] = URI;

        BBToken token = new BBToken(msg.sender, URI);
        tokenAddress[URI] = address(token);

        _createdToken[msg.sender] = true;
        emit TokenContractCreated(token.owner(), URI, address(token));
        return address(token);
    }
    ...
}
```

Figure 4.3: TokenFactory state variables and creation function

Seen in this light, TokenFactory can be envisioned as the root contract which provides location information to all deployed Token Sets in our protocol. Figure 4.3 demonstrates our *create* function as well as the important mappings that index the entity URI's and Token Set addresses.

4.2.3 Exchange

The Exchange contract is a stateful contract that handles the validation and execution of orders between traders. The Exchange contract utilises functionality from the OpenZeppelin IERC1155 interface[50] in order to facilitate the swaps of multi-tokens between individual traders. All orders between traders are priced in Ether. We considered two different approaches when considering how to price orders in our exchange contract:

- All orders are priced in Ether. This means that sellers offer a token for sale at a specific ratio of that token to Ether. For instance, selling 100 of Token A at 1 Ether each. Similarly, buyers submitting a buy order price their offers in Ether terms, so a buyer might offer to buy 50 of Token B at 1.5 Ether each.

- All orders are priced in terms of ratios between two tokens. Buyers and sellers are free to make trade offers for tokens quoted *in relation* to other tokens. So a Buyer might offer to purchase 100 of Token C in exchange for 50 Token D. In this case the price is then quoted at 2 Token D per Token C.

While the approach using ratios offers a lot of flexibility for markets, we felt it added too much complexity both in terms of the implementation and the user experience. Additionally, the ratio approach could present problems with liquidity in smaller token markets.

The Exchange contract also handles further functionality needed by the protocol, namely:

- Validating signed orders before execution
- Storing exchange deposits of buyers in the protocol
- Storing encrypted delegated signer keys
- Storing state variables representing validator's commission claims

It is important to note that it is possible for an arbitrary number of traders to possess and trade individual tokens from a Token Set; it is not necessary that traders be Token Set owners themselves.

Storing Exchange Deposits

When two orders are executed in the exchange contract, the buyer transfers Ether to the seller in exchange for a specific amount of some token at an agreed upon rate. In order to execute an order on the buyer's behalf, the exchange contract needs to possess the indicated Ether amount in the buy order, which is eventually transferred to the seller. Recall that orders are simply signed messages submitted to the Exchange contract by validator nodes; there is no Ether value contained therein. If there is no Ether contained in the buyer's signed messages, how does the Exchange contract send the indicated amount to the seller? The solution is to simply require that buyers submit an Ether deposit which is held in the exchange contract on their behalf. Validation logic in the order execution function ensures that buyers have enough deposit to cover the entire order price before execution is carried out.

```
contract Exchange {
    // Used for tracking Ether balances of traders, specifically buyers
    mapping(address => uint256) public balances;
    ...

    function depositEther() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdrawEther() public {
        (bool sent, bytes memory data) = msg.sender.call{
            value: balances[msg.sender]
        }("");
        require(sent, "Failed to send Ether");
        balances[msg.sender] = 0;
    }
    ...
}
```

Figure 4.4: Exchange Contract: Buyer Deposits

Validating Signed Orders & Order Execution

Order execution takes place when validator nodes submit a transaction the Exchange contract which calls the *executeOrder* function. The *executeOrder* function takes five arguments:

- **bidOrder** - an Order struct representing a signed buy order
- **askOrder** - an Order struct representing a signed sell order
- **price** - an unsigned 256 bit integer representing the price per token to fill the order at
- **quantity** - an unsigned 256 bit integer representing the amount of tokens to be transferred
- **data** - arbitrary byte data passed to further functions and included for compatibility reasons

Order structs represent a signed order from a peer(cf. 4.3.3); they are demonstrated in Figure 4.5.

```
contract Exchange {
    ...
    enum OrderType {
        BID,
        ASK
    }

    struct Order {
        string id;
        address from;
        address tokenAddress;
        uint256 tokenId;
        OrderType orderType;
        uint256 price;
        uint256 limitPrice;
        uint256 quantity;
        uint256 expiry;
        bytes signature;
    }
    ...
}
```

Figure 4.5: Order Struct: Represents a signed order from a buyer or seller

The *executeOrder* function performs a wide variety of validations to ensure the submitted orders are a suitable match before actually swapping any tokens. These checks include:

- Verifying that the submitted orders for execution represent an order for the same token (by Token Set address and token id)
- Verifying that the orders are not expired (cf. 4.3.3)
- Verifying that the execution parameters submitted by the validator (price & quantity) are acceptable to the buyer/seller
- Verifying that the buyer has sufficient liquidity, i.e. that they possess a great enough deposit to cover the total order price
- Verifying that the seller has sufficient liquidity, i.e. that they possess a sufficient number of tokens in question to fill the order at the specified quantity
- Verifying the respective signatures of each order
- Verifying the signatures have not been used for a previous order (i.e. order can only be executed once)

For brevity we detail a subset of the validation checks.

Verify the orders are not expired: Order expiry times are given in the Order Struct in Unix time in milliseconds, or milliseconds since January 1st, 1970. The Exchange contract validates that the order expiry time is greater than the current time, i.e. the order is not expired. In a centralised system this relatively trivial, however in a decentralised state machine such as the EVM, there is no trivial way to calculate an exact system time. Therefore we use block time with a sufficient buffer of 900 seconds in order to account for any drift.

```
1 contract Exchange {
2     ...
3     // Orders can be valid for up to 15 minutes past their expiry time
          in order to account for "block time shift" (CB words)
4     function verifyExpiry(Order memory order) private view returns (bool
          ) {
5         return ((order.expiry / 1000) + 900) > block.timestamp;
6     }
7     ...
8 }
```

Figure 4.6: Verifying that orders are not expired before execution

Verify that the execution parameters submitted by the validator (price & quantity) are acceptable to the buyer/seller: Validators are responsible for submitting orders to the chain, and thus also specify the execution parameters that determine how those orders will be executed against one another, namely the price and quantity of tokens in question. Recall from Section 4.3.5 that we adopt the convention that sellers "make orders while buyers "take" them. The Exchange contract is responsible for ensuring that the the submitted execution parameters fall into the acceptable ranges for the buyer and seller. The Exchange contract verifies that the submitted execution price is equal to the seller's simple limit price and also falls within the buyer's range limit price (cf. Section 4.3.5). The Exchange contract also verifies that execution quantity is not above the quantity specified by the buyer and seller in their respective orders. See Figure 4.7

Verify the respective signatures of each order: The Exchange contract also validates the provided order signature using ECDSA signature recovery. By recovering the address of the provided signature, the Exchange contract can ensure it is equal to the "from" field in the Order Struct. In doing so the Exchange contract validates that the submitted order was indeed signed by the address in the "from" field. See Figure 4.8.

If all of the checks pass, the Exchange contract then calls *collectCommission* which first collects the commission for the validator from the buyer, and then finally *exchangeTokens* which performs the actual swap between the buyer and the seller. See Figure 4.9.

```
contract Exchange {
    ...
    // Buyers take
    function verifyBuyerPrice(uint256 price, Order memory bidOrder)
        private
        pure
        returns (bool)
    {
        return bidOrder.limitPrice >= price;
    }

    // Sellers make
    function verifySellerPrice(uint256 price, Order memory askOrder)
        private
        pure
        returns (bool)
    {
        return askOrder.price == price;
    }

    function verifyQuantity(uint256 quantity, Order memory order)
        private
        pure
        returns (bool)
    {
        return quantity <= order.quantity;
    }
    ...
}
```

Figure 4.7: Verifying order price and quantity details fit the execution details submitted by a validator

```
contract Exchange {
    bytes32 public DOMAIN_SEPARATOR;

    bytes32 public constant EIP712DOMAIN_TYPEHASH = keccak256(
        "EIP712Domain(string name,string version,uint256 chainId,address
            verifyingContract)"
    );

    bytes32 public constant ORDERS_TYPEHASH = keccak256(
        "Order(string id,address from,address tokenAddress,uint256
           tokenId,uint orderType,uint256 price,uint256 limitPrice,
           uint256 quantity,uint256 expiry)"
    );
    ...
    constructor() {
        uint256 chainId = 31337;
        DOMAIN_SEPARATOR = keccak256(
            abi.encode(
                EIP712DOMAIN_TYPEHASH,
                keccak256(bytes("BrowserBook")),
                keccak256(bytes("1")),
                chainId,
                address(this)
            )
        );
    }
    ...
    function verifySignature(Order memory order) public view returns (
       bool) {
        bytes32 orderHash = keccak256(
            abi.encode(
                ORDERS_TYPEHASH,
                keccak256(bytes(order.id)),
                order.from,
                order.tokenAddress,
                order.tokenId,
                order.orderType,
                order.price,
                order.limitPrice,
                order.quantity,
                order.expiry
            )
        );

        bytes32 digest = keccak256(
            abi.encodePacked("\x19\x01", DOMAIN_SEPARATOR, orderHash)
        );

        return digest.recover(order.signature) == order.from;
    }
    ...
}
```

Figure 4.8: Verifying an order's signature

```
contract Exchange {
    ...
    function executeOrder(
        Order calldata bidOrder,
        Order calldata askOrder,
        uint256 price,
        uint256 quantity,
        bytes calldata data
    ) public {
        // Ensure we are trading the same tokens
        ...
        // Ensure orders are not expired (to some reasonable degree)
        ...
        // Ensure the specified exchange price falls within limits
        ...
        // Ensure the specified exchange quantity falls within limits
        ...
        // Verify buyer has sufficient funds
        ...
        // Verify seller has sufficient tokens
        ...
        // Verify signatures
        ...
        // Verify only used once
        ...
        collectCommission(bidOrder, askOrder, price, quantity, msg.
            sender);
        exchangeTokens(bidOrder, askOrder, price, quantity, data);
    }
    ...
}
```

Figure 4.9: Exchange Contract: Abbreviated *executeOrder* function

Storing Encrypted Delegated Signer Keys

The Exchange contract is also responsible for storing the encrypted signer keys of the validator nodes. Validator nodes submit their encrypted signer keys to the Exchange contract for long-term storage after their generation (cf. 4.3.4).

```
1  contract Exchange {
2      ...
3      mapping(address => address) public signerAddresses;
4      mapping(address => string) public encryptedSignerKeys;
5      ...
6      function setSigner(address signerAddress, string calldata
           encryptedSignerKey)
7          public
8      {
9          signerAddresses[msg.sender] = signerAddress;
10         encryptedSignerKeys[signerAddress] = encryptedSignerKey;
11     }
12     ...
13 }
```

Figure 4.10: Exchange Contract: Encrypted signer key storage

Storing Validator's Commission Claims

Finally, the Exchange contract is also responsible for tracking validator's commission claims. Commission claims are stored in a simple mapping from the validator's signer address.

```
1  contract Exchange {
2      ...
3      mapping(address => uint256) public signerCommissionBalances;
4      ...
5  }
```

Figure 4.11: Exchange Contract: Validator commission balances via signer address

4.3 Client Application

4.3.1 State Management and Client-Chain Communication

Chain communication in our application is facilitated by one of our key dependencies ethers.js (or "Ethers"). Ethers allows the creation of objects, called providers or signers, derived from the user's browser based signing authority which provide an easy-to-use API for calling JSON-RPC's[1]. By passing in contract information, specifically a contract address and ABI, Ethers even provides us with abstracted contract instances with methods that map to specific contract functions in the referenced smart contract. We maintain a constant map of contract metadata in `client/src/app/chain/ContractMetadata.json` and dynamically pass the appropriate contract metadata to Ethers whenever we need to communicate to the blockchain.

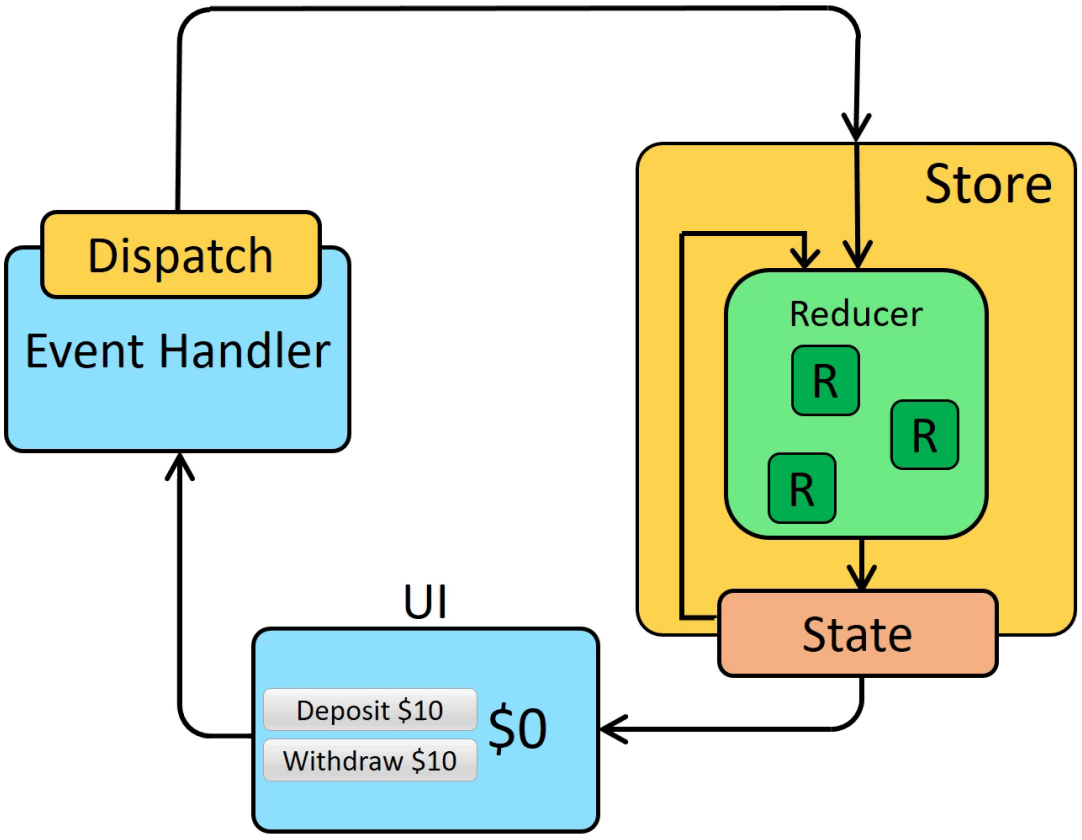

Figure 4.12: Example state flow in a Redux application. The client UI dispatch events to the global state store, where reducers act on the state to modify it in accordance with the action. Finally, the store notifies the UI of the updated state

[1]The main difference between providers and signers is that providers are read-only, i.e. they have no signing authority

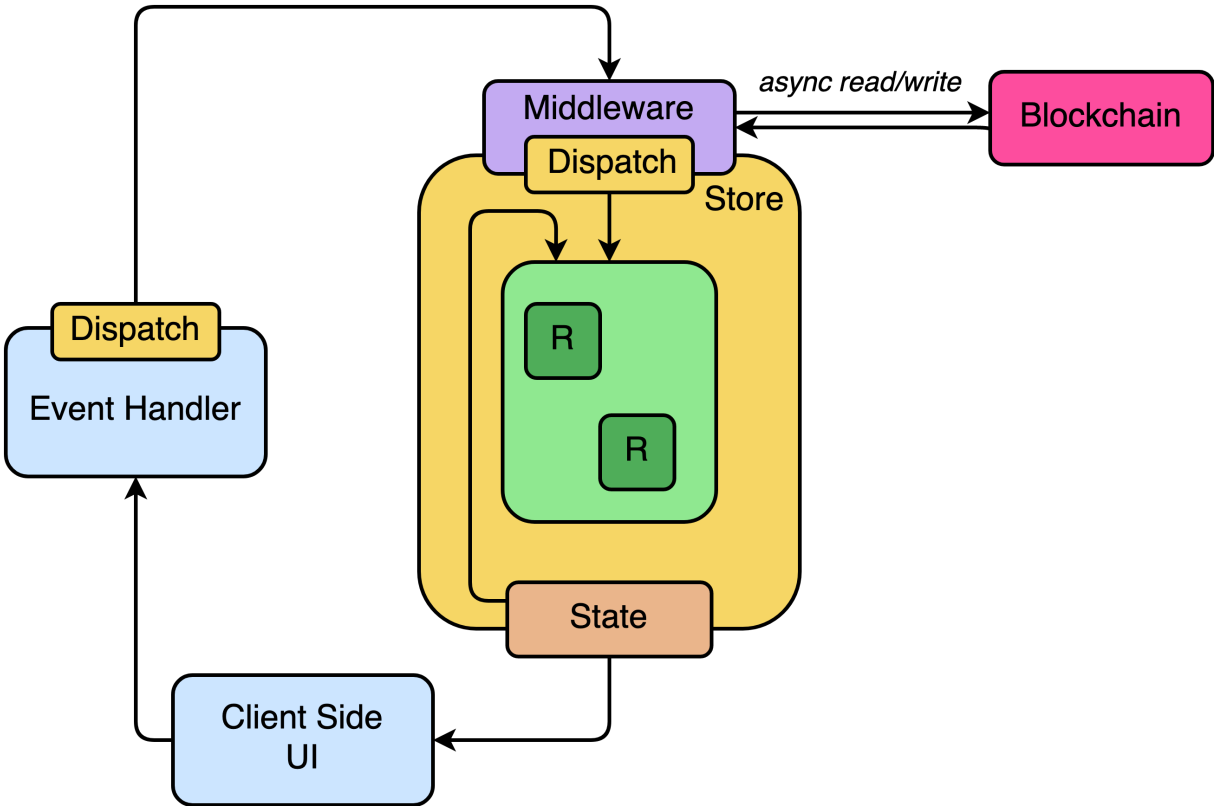

Figure 4.13: Our modifications to the general Redux pattern for querying and writing to the blockchain. Asynchronous queries & operations are passed to a special asynchronous middleware called a *Thunk* which awaits asynchronous chain operations and then synchronously dispatches them to the store

We use Redux to serve as a single source of truth for all of our application state. As noted in Section 3.6, Redux is a javascript library providing a global state container for applications, with utilities for both subscribing to state updates and dispatching updates to the state container. With Redux, the client watches for events in the UI from the user, and upon receiving an event dispatches an *action* to the global state store which describes *how* the state should be modified. Functions in the global store called *reducers* read the action sent from the client and modify the state accordingly. Finally, the global store notifies the client of the updated state and the client re-renders the front-end with the new state data accordingly. Figure 4.12 demonstrates this pattern in practice[2].

The benefits of such a pattern are that, by separating reading & observing state from it's modification, we can better reason about the progression and mutation of state in our application over time, and we have introduced the fundamentals of Redux in order to demonstrate how we have extended this state management technique in our application to incorporate data from the blockchain. We start by noting querying the blockchain for state data and event logs is an asynchronous operation. Fortunately, it is possible to extend the general Redux pattern to handle asynchronous operations and data fetching by splitting

[2]This image was taken directly from the excellent Redux documentation, available at `https://redux.js.org/tutorials/fundamentals/part-2-concepts-data-flow`

an asynchronous query into two steps: 1) Dispatching the asynchronous operation/query itself and, 2) Dispatching the result of the operation to the store once the operation completes[3].

This is the pattern we extend to handle blockchain data. When the user requests an operation that reads or writes to the blockchain, we dispatch a special asynchronous function, called a *Thunk* which calls our *Chain.ts* module to read or write to the chain. The *Chain.ts* module initialises an Ethers.js provider or contract (depending on the operation) object to communicate with the JSON-RPC node provided by the browser based signing authority. The *Chain.ts* module carries out the operation specified and awaits the result of the transaction. Once the transaction completes, the *Chain.ts* module returns and the asynchronous Thunk can finally dispatch the result of the operation to the store. We found this to be a very maintainable way of incorporating asynchronous logic and querying of the blockchain into our application. Figure 4.13 shows an overview[4].

By using this pattern, we can easily incorporate blockchain data into our normal state management flow. UI interactions that read or write to the blockchain have the same interface as any other state query or modification, and they all begin by dispatching an action. By using this uniform interface, we greatly simplify the reasoning we need to do about our application state and produce more maintainable and flexible code.

We use a small example from our application to demonstrate. Token Set Owners create a new token by navigating to the Token Administration Page, where they are provided with a TokenInput Form. This form takes in the user input and upon submission dispatches a *createTokenThunk* to the store. The *createTokenThunk* calls the *Chain* module which submits a transaction to the blockchain via the user's browser based signing authority, calling the appropriate mint function respectively. Figure 4.14 shows an excerpt of how this pattern looks in the application code for creating a new token in a Token Set.

[3]The Redux documentation has a terrific general example of this, available at: `https://redux.js.org/tutorials/fundamentals/part-6-async-logic`

[4]We note that this image is our own creation inspired by the style of the Redux documentation

```
1
2  /* UI Form Handler */
3  const handleSubmit = () => {
4      ...
5      if (...) {
6          // Error handling for invalid input
7      } else {
8          dispatch(
9              createTokenThunk({
10                 contractAddress: props.tokenContract.address,
11                 tokenType,
12                 tokenSupply,
13                 tokenIdentifier,
14                 tokenMetadataURI,
15             }),
16         )
17     }
18 }
19
20 /* Special Thunk Middleware */
21 /* Awaits async call to chain */
22 export const createTokenThunk = createAsyncThunk(
23     'tokens/createToken',
24     async (options: CreateTokenOptions, thunkAPI: any): Promise<void> =>
25     {
26         await createToken(options)
27     },
28 )
29
30 /* Chain Module */
31 export const createToken = async (options: CreateTokenOptions) => {
32     const { contractAddress, tokenIdentifier, tokenType, tokenSupply,
          tokenMetadataURI } = options
33     const wrapper = new EtherContractWrapper()
34
35     const contractName = ContractName.Token
36     const contract = await wrapper.getContract(contractName,
          contractAddress)
37
38     if (tokenType === TokenType.Fungible) {
39         const tx = await contract.fungibleMint(
40             tokenIdentifier,
41             ethersLib.utils.parseEther(tokenSupply),
42             tokenMetadataURI,
43             ethersLib.utils.toUtf8Bytes(''),
44         )
45         await tx.wait() // Returns upon block confirmation
46     } else {
47     ... // Same but for NonFungible
48     }
49 }
```

Figure 4.14: State Management in Our Application: Example of our extended Redux architecture when creating a new token

4.3.2 Peer Initialisation

Peer initialisation takes place in the background after the user starts the application. Creation of a peer involves setting important libp2p configuration variables which determine how the peer connects to other peers in the network. These include specifying the transport layer, the connection encryption mechanism, the peer discovery mechanism, and the protocol for communication.

We provide a class wrapper over the libp2p peer implementation (`client/src/app/p2p/Peer.ts`) in order to provide additionally functionality, namely:

- Interfacing with the IndexedDB via dexie (cf. 3.6)
- Encoding & decoding message information from their protocol buffer representation to a native JSON object
- Methods for initialising and controlling a validator node

Initialising this class with a valid configuration and calling the setup methods creates a peer and performs the following actions:

- Creates a new unique peerId and libp2p multi-address[5]
- Connects with the WebRTC Signaling Server/Bootstrap Server fetches available peer details
- Attempts to establish a web socket connection with all available peers
- Initialises the IndexedDB and any schemas needed
- Subscribes to both the ORDER_TOPIC & MATCH_TOPIC (cf. 3.2.2)
- Sets up handlers for receiving serialised messages from other peers

Our application initialises a peer and calls the relevant setup methods in the background on startup. A commented representation of our libp2p configuration is shown in Figure 4.15 while an excerpt from our peer class is shown in Figure 4.16.

[5] A multi-address is a specific implementation format of libp2p used for defining peer connection details in a network

```
{
    addresses: {
        listen: [
            // Configures the WebRTC Signaling server address
            '/dns4/simpleweb3.ch/tcp/443/wss/p2p-webrtc-star',
        ],
    },
    modules: {
        transport: [Websockets, WebRTCStar], // Sets transport protocol
        streamMuxer: [Mplex], // Stream multiplexing
        connEncryption: [NOISE], // Encryption protocol
        peerDiscovery: [Bootstrap], // Instructs the peer to bootstrap
            via the WebRTC Signaling server
        dht: KadDHT, // Peer addresses are stored in a local kademlia
            DHT
        pubsub: Gossipsub, // The PubSub protocol is initialised with a
            GossipSub implementation
    },
    config: {
        peerDiscovery: {
            bootstrap: {
                list: [
                    // The WebRTC Signaling server doubles as a
                        bootstrap rendezvous
                    '/dns4/simpleweb3.ch/tcp/443/wss/p2p-webrtc-star',
                ],
            },
            dht: {
                enabled: true,
                randomWalk: {
                    enabled: true,
                },
            },
            pubsub: {
                // Additional PubSub configuration
                enabled: true,
                emitSelf: false,
            },
            transport: {
                [transportKey]: {
                    filter: Filters.all,
                },
            },
        },
    },
}
```

Figure 4.15: Configuration of the libp2p node

```
export class Peer {
    // PubSub topics
    static ORDER_TOPIC: string = '/bb/order/1.0.0'
    static MATCH_TOPIC: string = '/bb/match/1.0.0'
    // Client Database
    static DB: P2PDB = P2PDB.initialize()

    // Config & instance variables
    config: Libp2pOptions
    node: Libp2p | null = null
    isValidator: boolean = false
    signerAddress: string | null = null
    chalk: number = 0

    constructor(config: Libp2pOptions) {
        this.config = config
        ...
    }

    async init() {
        this.node = await Libp2p.create(this.config)
        ...
        await this.node.start()
    }
    ...
    join() {
        ...
        this.node.pubsub.on(Peer.ORDER_TOPIC, this.processOrder)
        this.node.pubsub.subscribe(Peer.ORDER_TOPIC)
        this.node.pubsub.on(Peer.MATCH_TOPIC, this.processMatchMessage)
        this.node.pubsub.subscribe(Peer.MATCH_TOPIC)
    }
    // Methods for publishing/processing messages on the PubSub channels
    async processOrder(encodedOrder: any) {
        ...
    }

    async publishOrder(order: Order) {
        ...
    }

    async processMatchMessage(encodedMatch: any) {
        ...
    }

    async publishMatchMessage(match: Match) {
       ...
    }
    ... // Further methods
}
```

Figure 4.16: An excerpt from our Peer class which provides a layer of functionality on top of libp2p's peer implementation. On initialisation and startup the application calls the *init* and *join* methods which initialises the peer and joins the relevant PubSub topics. Notice as well the initialisation of the P2PDB which is a simple implementation of IndexedDB via dexie

4.3.3 Order Creation, Signing & Propagation

The data required is to submit an order is analogous to the Order Struct defined in 4.5. Recall from Figure 3.17 that clients submit orders by navigating to the *Trade* page and submitting a form with the data of their order. Upon submission this form sends the data to the *OrderService* where the data types are converted to their respective Solidity counterparts[6]. A random alphanumeric identifier is generated to refer to the order. The order representation is now complete and awaiting signature.

The order is then signed by invoking the Ethers *_signTypedData* function, which is an implementation of ERC-712 signing (cf. 1.3) and prompts the user for a signature via their browser based signing authority. Once the signature has been obtained, the order is passed to the Peer's *publishOrder* method, which handles broadcasting the order to all peers on the order topic as well as persisting the order to Peer's own IndexedDB. (As peers do not receive messages from their own broadcasts) Figure 4.17 demonstrates the end-to-end process of submitting an order.

[6]In the form itself we use strings to represent every piece of user input related to their order. This is done to maximise the usability of the browser based form controls

```
/* TradeForm.tsx */
const handleSubmit = () => {
    setError('')

    if (...) {
        // Error handling
    } else {
        submitOrder(
            primaryAccount,
            selected,
            orderType,
            price,
            limitPrice,
            quantity,
            expiryHours,
            expiryMinutes,
        )
    }
}

/* OrderService.ts */
export const submitOrder = async (
    fromAddress: string,
    token: Token,
    orderType: OrderType,
    price: string,
    limitPrice: string,
    quantity: string,
    expiryHours: string,
    expiryMinutes: string,
) => {
    const signer = ethers.getSigner()
    ...
    const unsignedOrder: Omit<ChainOrder, 'signature'> = {
        id: (~~(Math.random() * 1e9)).toString(36) + Date.now(), //
            Random id
        from: fromAddress,
        tokenAddress: token.contract.address,
        tokenId: ethersLib.BigNumber.from(token.id),
        orderType: orderType === OrderType.BUY ? 0 : 1,
        price: ethersLib.utils.parseEther(price),
        limitPrice: ethersLib.utils.parseEther(limitPrice),
        quantity: ethersLib.utils.parseEther(quantity),
        expiry: ethersLib.BigNumber.from(getDateExpiryInMs(expiryHours,
            expiryMinutes)),
    }

    const signature = await signer?._signTypedData(OrderDomain,
        OrderTypes, unsignedOrder)
    const signedOrder: Order = { ...chainOrderToPeerOrder(unsignedOrder)
        , signature }
    await peer.publishOrder(signedOrder)
}
```

Figure 4.17: Signing and broadcasting an order to the network

4.3.4 Validator Nodes

Validator nodes watch the orderbook and attempt to submit orders to the chain. Before a peer can become a validator node they need to initialise a signer and encrypt the signer key and store it on chain. See Sections 3.4 and 3.5.4 for more details. Users initialise a signer by interacting with a standard UI control (a toggle) that dispatches an operation to the ValidatorService module using our extended Redux state as detailed in 4.3.1.

Signer initialisation includes generating a random signer key and encrypting this signer key via the user's public encryption key provided by their browser based signing authority. The functions *encryptDelegatedSigner* and *getEncryptionKey* handle this process and are invoked by the Redux Thunk middleware. They are appropriate detailed in Figure 4.18.

The encryption itself takes place via the nacl.js library (cf. 3.6), via the *nacl.box* method[7].

Validator nodes also initialise a web worker to work with the Order Management System (OMS). Web workers, as described in 1.3, are a manner for browser-based Javascript to run compute intensive applications without blocking the main thread. Web workers communicate with the main process via messages. The Order Management System is initialised in web worker context in order to prevent the long running order matching algorithms from blocking the rest of the application. In this way, the user can freely continue to use the application without interrupting the order execution & submission algorithms running in the background. We detail the exact specifics of the OMS in the next subsection.

The following messages are passed to the web worker from the validator:

- **Start**: Starts the web worker & the OMS
- **Stop**: Stops the web worker & the OMS
- **New Order**: Notifies the web worker of a new order received after startup

The web worker sends the following messages back to the Peer at periodic intervals:

- **Order Match**: Notifies the Peer of a matched order pair along with the respective order details
- **Order Rejection**: Notifies the Peer of a rejected order pair along with the respective order details

The application receives these messages and can update the IndexedDB and then application UI accordingly.

[7]See `client/app/src/chain/Encryption.ts`

```
1
2  getEncryptionKey = async (account: string): Promise<string> => {
3      ...
4      // Retrieve the user's public encryption key from Metamask
5      return await this.provider.send('eth_getEncryptionPublicKey', [
           account])
6  }
7
8  encryptDelegatedSigner = async (account: string) => {
9      // Calls above
10     const encryptionKey = await this.getEncryptionKey(account)
11     const { address: signerAddress, privateKey } = ethers.Wallet.
           createRandom()
12
13     const encryptedSignerKey = ethers.utils.hexlify(
14         ethers.utils.toUtf8Bytes(
15             JSON.stringify(
16                 encrypt({
17                     publicKey: encryptionKey,
18                     data: privateKey,
19                     version: 'x25519-xsalsa20-poly1305',
20                 }),
21             ),
22         ),
23     )
24
25     return [signerAddress, encryptedSignerKey]
26 }
```

Figure 4.18: Encrypting a delegated signer key before sending it to the chain for persistance

4.3.5 Order Matching

The Order Matching System, or OMS, runs in a web worker process environment so as to not block the main application thread as detailed above. The OMS is responsible for sorting, matching, verifying, and submitting the orders to the chain. The OMS maintains its own internal state of the orderbook and communicates with the main application thread via messages. To start the OMS, the application sends the start message along with the signer key as a parameter so that a wallet object can be initialised which will send transactions to the JSON-RPC endpoint in order to execute orders.

Upon startup, the OMS first synchronises the orderbook stored in IndexedDB and sorts the existing orders by token. The OMS then further sorts the orders into two priority Queues, one for buy orders, and one for sell orders[8]. The priority queue sorts the orders in place so that the highest-priced buy orders and lowest-priced sell orders are at the front of each respective queue.

The OMS then proceeds through each of queue-pairs for each token in the orderbook and

[8]Queues are a standard abstract datatype in computer science and operate on the Last-In, First-Out principle, or LIFO, i.e. elements are added sequentially and then de-queued in the order they were entered

attempts to match 100 buy and sell orders for that token. It then moves to the next token type and attempts to process those orders. The OMS keeps track of its own state and indicates that it is in a "waiting" state once 100 orders have been processed for every token type. If there are additional orders still remaining in the orderbook, the OMS is restarted via an interval function which runs every ten milliseconds[9]. It is during this brief pause that the OMS incorporates new messages from the application process. If the OMS has received new orders in the meantime, it adds them to their respective sorted queue at the position in sorted-price order.

When the OMS attempts to match an order, it first runs a series of validation checks to ensure that the orders *should* match on-chain. The checks are not exhaustive but we felt they covered the most prominent cases. If the orders are deemed to be an invalid match, the OMS sends the invalid party into an *overflow* queue where it can be re-added to the orderbook queues and re-attempted at a later time with a different counterparty.

If the validation checks succeed, the OMS submits the matched orders to the chain and awaits the block confirmation. If the transaction was successful, the OMS notifies the application process with the "Order Match" message along with the specific details of the matched orders. If the transaction fails for any reason, the OMS notifies the client with the "Order Rejected" message and adds both orders to a "stale" queue where they are then discarded.

An abridged overview of the OMS algorithm is shown below in Figure 4.19.

[9]An interval function is a first-class concept in Javascript that allows developers to continuously call a specified function at some set interval. The reason for doing so is because we need to incorporate a very brief "pause" in the execution in order to check for new messages from the application process. It is tempting to implement the OMS as a simple while loop, but doing so means that no new messages can reach the process when it is looping through the orders

```
// Internal OMS state
type OMS = {
    running: boolean
    waiting: boolean
    orderbook: Map<string, { bid: PriorityQueue<Order>; ask:
        PriorityQueue<Order> }>
    overflow: Array<Order>
    stale: Set<string>
    signerNonce: number
    stats: { startTime: number; endTime: number; success: number; fail:
        number }
}
// Start the OMS algorithm on 'start' message
const start = async (signerAddress: string, decryptedSignerKey: string)
    => {
    ...
    // Sync the orderbook
    await syncOrderBook()
    ...
    setInterval(() => {
        if (oms.running && oms.waiting) {
            matchOrders(delegatedExchangeContract)
        }
    }, 10)
    ...
}
// Match orders
const matchOrders = async (delegatedExchangeContract: ethers.Contract)
    => {
    for (const [_, { bid, ask }] of oms.orderbook) {
        for (let i = 0; i < 100; i++) {
        // Match 100 orders
        if (bid.peek() !== undefined && ask.peek() !== undefined) {
            const highestBid = bid.peek() as Order // Safe as we check
                the undefined case above
            const lowestAsk = ask.peek() as Order

            if (validMatch(highestBid, lowestAsk) === MatchValidity.
                Valid) {
                await matchOrder(delegatedExchangeContract, bid.dequeue
                    () as Order, ask.dequeue() as Order)
            } else {
                // Send to overflow
            }
        }
        }
    }
...
}
```

Figure 4.19: Excerpts from our OMS algorithm

4.4 Hosting and Additional Infrastructure

4.4.1 Signalling Server a.k.a Bootstrap Server

Connections between our peers via WebRTC depends on sharing SDP data and establishing an ICE connection (cf. Section 1.3). In order to exchange SDP information, peers need to initially communicate via a WebRTC signalling server, which assists in the SDP handshake. After a connection has been established, peers communicate directly over WebRTC. Our WebRTC signaling server also doubles as our bootstrap server, where peers query the locations of initial peers in the network.

Occasionally, peers cannot communicate with one another directly due to Symmetric Network Address Translation, or NAT. In this case our bootstrap server also serves as a TURN server, relaying the traffic on behalf of the two peers. In this case, some aspects of decentralisation are lost.

We implement our own bootstrap server based upon the provided implementation provided by Protocol Labs in their implementation of libp2p. The bootstrap server is publicly provided as a docker image. We use Nginx as a reverse proxy to provide a SSL certificate and assist tunneling the web socket request over HTTP. Figure 4.20 shows the Dockerfile we used to setup our bootstrap server.

```
version: "3.3"
services:

    js-libp2p-webrtc-star:
        image: libp2p/js-libp2p-webrtc-star
        environment:
            - VIRTUAL_HOST=${DOMAIN}
            - LETSENCRYPT_HOST=${DOMAIN}
            - VIRTUAL_PORT=9090
        networks:
            service_network:

    nginx-proxy:
        image: jwilder/nginx-proxy
        ports:
            - 443:443
            - 80:80
        container_name: nginx-proxy
        networks:
            service_network:
        volumes:
            - /var/run/docker.sock:/tmp/docker.sock:ro
            - nginx-certs:/etc/nginx/certs
            - nginx-vhost:/etc/nginx/vhost.d
            - nginx-html:/usr/share/nginx/html
        depends_on:
            - js-libp2p-webrtc-star

    nginx-proxy-letsencrypt:
        image: jrcs/letsencrypt-nginx-proxy-companion
        environment:
            NGINX_PROXY_CONTAINER: "nginx-proxy"
        networks:
            service_network:
        volumes:
            - /var/run/docker.sock:/var/run/docker.sock:ro
            - nginx-certs:/etc/nginx/certs
            - nginx-vhost:/etc/nginx/vhost.d
            - nginx-html:/usr/share/nginx/html

networks:
    service_network:

volumes:
    nginx-certs:
    nginx-vhost:
    nginx-html:
```

Figure 4.20: Dockerfile for our WebRTC Signaling/Bootstrap server

4.4.2 Subgraph Implementation

Indexing the blockchain data via the Graph Protocol helps in efficiently reading the on-chain data. A Subgraph is a way of defining rules used to describe the index of data retrieval in a block by block manner.

As we are only focusing on using the Subgraph for the event handler of two use cases:

1. Track the token details based on the token creation.
2. Track the token price based on the exchange of tokens.
3. Track of number of user and their addresses.

The definition consists of three important files that define the Subgraph; Firstly, `subgraph.yaml` manifests the smart contracts your Subgraph indexes, which events from these contracts to listen for, and how to convert the event data to entities that the Graph Node stores and allows you to query. In layman's terms it defines the which, when, and how the graph node is to index the data. It includes details such as from which blockchain the data should be fetched, which network, which contract to be indexed, which events to be handled, to which graphql schema they should refer, and whether we are handling events, calls, or entire block data. We index the Polygon Testnet, also known as Mumbai. We are using a Token entity and a User entity for handling the TokenCreation event and using a TokenPrice entity for handling the TokensExchangedAt event, which are both handled by the handler functions in the mapping file specified in 4.21.

`schema.graphql` contains the GraphQL schema specifying what data is stored for the Subgraph and how to query this using GraphQL. All queries will be performed against the Subgraph schema's data model and the entities indexed by the Subgraph. It is preferable to define the Subgraph structure in a way that fits the requirements of the application. In this file we have created the three entities as show in Figure 4.22 :

1. **User** represents the wallet address of an account
2. **Token** represents the owner of the token, its creation date and the URI of the particular token
3. **TokenPrice** represents a relation to a Token and information regarding a specific exchange event which includes the prices at which the tokens were executed during the exchange event

`mapping.ts` is a mapping file that determines how incoming event data is converted into a structured format as specified in the GraphQL schema file. The mappings convert the Ethereum data sourced by the mappings into entities described in the schema. Mappings are authored in a TypeScript file that may be compiled to WASM (WebAssembly). In the mapping defined in Figure 4.23, when a TokenCreation event is emitted, the function is called using the event details with the block & transaction information, We then create a new Token entity, load the details of the user, and take the emitted event parameters and save the entity for querying.

```yaml
specVersion: 0.0.2
schema:
  file: ./schema.graphql
dataSources:
  - kind: ethereum
    name: TokenFactory
    network: mumbai
    source:
      address: "0xB82B54316ac597baA4a494b124c081af45E48233"
      abi: TokenFactory
    mapping:
      kind: ethereum/events
      apiVersion: 0.0.5
      language: wasm/assemblyscript
      entities:
        - Token
        - User
      abis:
        - name: TokenFactory
          file: ./abis/TokenFactory.json
      eventHandlers:
        - event: TokenCreated(indexed address,indexed address,
          string)
          handler: handleTokenCreated
      file: ./src/mapping.ts
  - kind: ethereum
    name: Exchange
    network: mumbai
    source:
      address: "0x9B59bFc98Cdc48cB4a0E73037eEd9CFf827D58ae"
      abi: Exchange
    mapping:
      kind: ethereum/events
      apiVersion: 0.0.5
      language: wasm/assemblyscript
      entities:
        - TokenPrice
      abis:
        - name: Exchange
          file: ./abis/Exchange.json
      eventHandlers:
        - event: TokensExchangedAt(indexed address,indexed
          address,uint256,uint256)
          handler: handleTokenExchanged
      file: ./src/mapping.ts
```

Figure 4.21: Subgraph Setup

```
type User @entity {
    id: ID! #Address
}

type Token @entity {
    id: ID! #Token Address
    owner: User! # address
    createdAt: BigInt!
    URI: String!
}

type TokenPrice @entity {
    id: ID! # random generation
    token: Token!
    inToken: Token!
    priceInToken: BigDecimal!
    priceAt: BigInt!
}
```

Figure 4.22: GraphQL schema

```
export function handleTokenCreated(event: TokenCreated): void {
    let token = Token.load(event.params.param1.toString());

    if (!token) {
        token = new Token(event.params.param1.toString());
    }

    let user = User.load(event.params.param0.toString());

    if (!user) {
        user = new User(event.params.param0.toString());
        user.save();
    }
    token.owner = user.id;
    token.createdAt = event.block.timestamp;
    token.URI = event.params.param2;
    token.save();
}
```

Figure 4.23: Example mapping for Token Creation event

Chapter 5

Evaluation

5.1 Order Throughput

In order to evaluate our system's order throughput, we perform a basic series of performance load tests in order to test how fast our implementation can reliably execute orders. Our system processes orders by exchanging order data between peers, sorting and matching those orders by validator nodes, and finally executing and sending those orders to the chain. We identified three main performance criteria that would serve as bottlenecks on order throughput in our system:

- How fast peers can reliably exchange order messages with one another
- How fast validator nodes can run the order matching algorithms to reliably sort and match orders
- How fast validator nodes can actually execute and send orders to the chain

We envisioned an end-to-end testing pipeline to test all three potential bottlenecks in concert with one another. We faced issues however in reliably creating a suitable test infrastructure for simulating orders being exchanged between peers. Ideally, we would have constructed a network of "psuedo-nodes", that would simply submit signed orders on our peer-to-peer network channels. These nodes could have been run without any browser application and been run on virtual machines spread out over different geographic locations to simulate real world network latencies.

Sadly, due to limitations with libp2p and difficulties in fostering a connection between browser and non-browser nodes, this plan proved to be beyond the scope of this initial work. Essentially, incompatibilities between browser and non-browser nodes stem from the fact that the browser cannot open a direct TCP connection to non-browser based peers. Browsers themselves communicate via WebRTC, but compatibility layers for non-browser based nodes are still in active development.

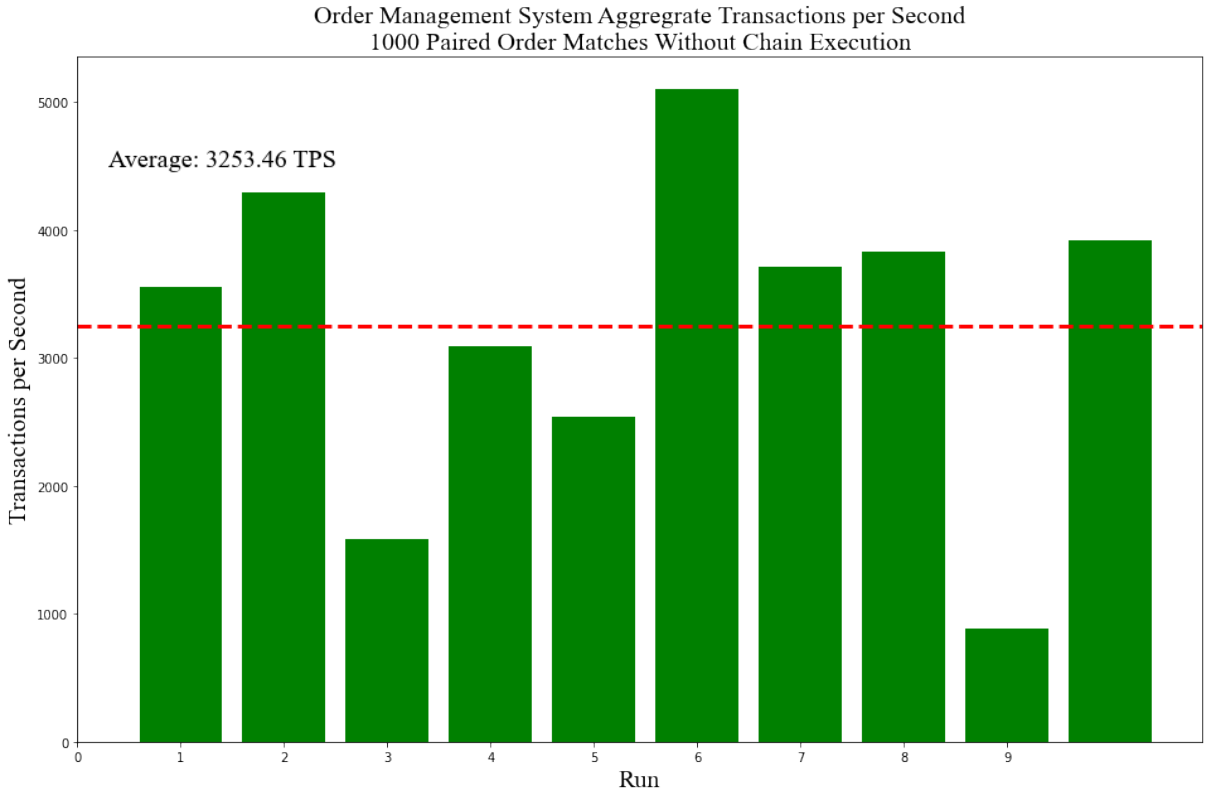

Figure 5.1: Testing OMS throughput without chain execution

Given this limitation, we focused our efforts on analysing the order throughput in a more isolated fashion. We first attempted to determine how fast our order matching algorithms can reliably sort and match orders. We then consider overall performance again with the additional constraint that the orders must be submitted to the chain.

Order Matching Algorithm Analysis

We created an in-application testing framework in order to send mock orders through our system. The framework fills the orderbook with 1000 buy and 1000 sell orders that are compatible and fit the criteria to be successfully matched by the OMS[1]. We run the OMS via a validator node as normal and record the time required to fill and execute all 2000 orders (1000 each of buy/sell, so 1000 matches) against each other, as well as if any of the executions fail.

In order to test our order matching algorithm in isolation, we first run the test without sending the matched orders to the chain for execution. That is, we simply match the orders against each other, log a success or failure, and record the overall throughput. We run the test over ten runs and record the average result. Figure 5.1 shows the result our of test runs.

[1]In fact the testing framework is available to evaluators. It requires a bit of setup, see the application *"How it Works"* page for more details

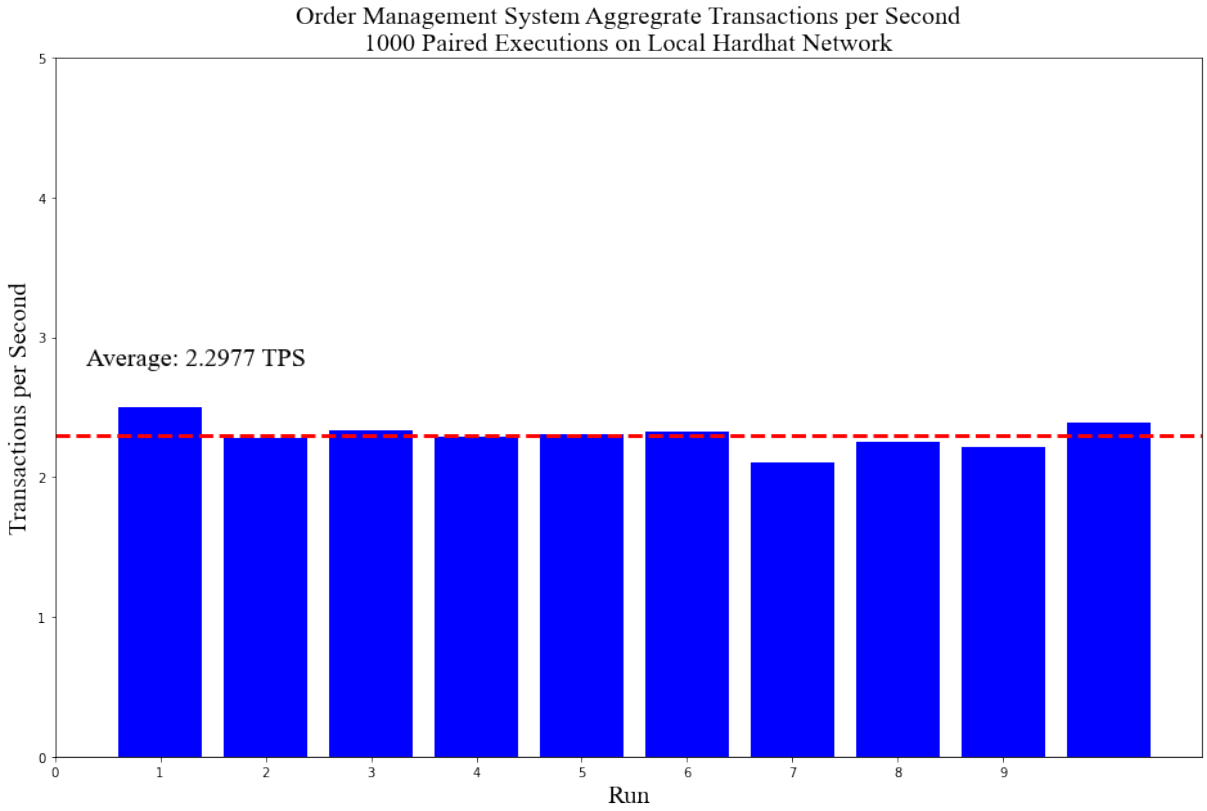

Figure 5.2: Testing OMS throughput including chain execution

Order Execution & Chain Submission

We then turned our attention to testing throughput of the OMS while considering chain execution. We re-ran the same testing framework again over 10 runs but submitted the orders to a local Hardhat blockchain with the default network settings. Figure 5.2 shows the results of these test runs.

Clearly throughput suffers by orders of magnitude when orders are actually submitted to the chain. Throughput averages only 2.3 Transactions per Second when run via the local Hardhat network, compared to over 3000 Transactions per Second when not submitting the orders to the chain. This result suggests our algorithms are well designed and sufficient for the task at hand; in order to increase scalability, the focus should remain on improving throughput on the blockchain side of the equation.

It's important to note that these isolated performance numbers may not be wholly indicative of transaction throughput achievable by the system as a whole. As the validator nodes are distributed throughout the network, overall order throughput of the system as a whole could be much larger when considered in aggregate across all distributed validators. We elaborate on this slightly in Section 5.2.

Testing Throughput on Mumbai Testnet

For economic reasons, it was not possible to replicate the testing procedure on the Mumbai Testnet, as filling the the orderbook with thousands of orders over ten runs presented a significant cost in MATIC between the buyer and the seller. Nonetheless, we engaged in isolated smaller runs of our testing procedure to compare performance to the local Hardhat network and observed throughput rates in the range of 0.1 to 0.4 Transactions per Second. Our conclusions regarding this figure remain the same as those in the prior subsection.

Tables 5.1 & 5.2 provides a summary of our results. In the next section we elaborate with some further discussion.

Run	**Time (ms)**	**Success**	**Failure**	**TPS**
1	281	1000	0	3558.719
2	233	1000	0	4291.845
3	631	1000	0	1584.786
4	323	1000	0	3095.975
5	394	1000	0	2538.071
6	196	1000	0	5102.041
7	269	1000	0	3717.472
8	261	1000	0	3831.418
9	1124	1000	0	889.680
10	292	1000	0	3424.558

Table 5.1: Summary of Performance Testing: No Chain Execution

Run	**Time (ms)**	**Success**	**Failure**	**TPS**
1	399649	1000	0	2.502
2	439404	1000	0	2.276
3	428472	1000	0	2.334
4	437919	1000	0	2.284
5	434280	1000	0	2.303
6	430527	1000	0	2.323
7	475176	1000	0	2.104
8	443981	1000	0	2.252
9	452000	1000	0	2.212
10	418875	1000	0	2.387

Table 5.2: Summary of Performance Testing: With Chain Execution

5.2 Discussion

Our system represents an exciting development in the DEX space that we hope helps pioneer more advanced techniques and serves as inspiration for further development of

our architecture and Decentralised Exchanges in general. We present many novel aspects of a prototypical exchange architecture that build on existing techniques and DEX implementations.

Particularly noteworthy we feel are:

- Our architectural concept of building peers into a peer-to-peer network, who then exchange orders with one another.
- The seamless "drop-in, drop-out" nature of validator nodes, along with economic incentives to encourage them to validate and submit orders, and the associated distributed nature of order validation and execution.
- Our novel delegated signing technique for signing messages on a user's behalf without knowledge of their private key and without any centralised storage.

We briefly discuss some other noteworthy points of the system below.

5.2.1 Throughput

It is clear from our load testing results that the current bottleneck on throughput lies in submitting orders to the chain. With the current architecture, there exist a variety of possible techniques available that may help increase order throughput. The first and most salient would be to batch transactions. In our implementation, the OMS waits for block approval of a submitted order before continuing to process more orders in the queue. This simple implementation was preferable during the development of the OMS, as it facilitated a more consistent messaging model between peers regarding the status of their order.

If our system were to be further developed, a more sophisticated OMS could eventually evolve that batches orders together into transactions which are submitted to the chain in unison. By combining multiple orders into single transactions, a higher throughput can be achieved. Care needs to be taken when implementing this technique however as the logic for handling failed transactions becomes more complex. For instance, if an order fails to match on-chain (because of a failed validation check) the contract would either need to roll back all orders in the batched transaction or specifically notify the client of the singular failed transaction. Emitting an event could be a possible solution to this problem. Additionally, the OMS needs to take greater care in estimating the necessary gas required for the transaction when submitting orders in batches.

5.2.2 Consensus

Our system could benefit from a more sophisticated consensus model. Currently, when validator nodes broadcast a match message, they simply flood the network with their notification that an order has been matched. Other validator nodes take this message and notify their respective OMS to drop the associated order from the queue.

Economic incentives mean that a validator is indeed inclined to heed these messages from other validator peers, as to submit an order that was already matched would lead to a rejection of the transaction and wasted gas by the validator.

Nevertheless, a more sophisticated architecture could be envisioned where validator nodes participate in some sort of consensus scheme with more formal guarantees about sharing the state of their respective orderbooks. One can imagine a scenario where validators act as the source of truth for the distributed orderbook while participating in a strong consensus scheme and serving as relayers of orders submitted from traders[2].

5.2.3 Security

Our system takes advantage of interesting economic properties in order to protect the state of the distributed orderbook. In peer-to-peer systems with no central authority, any node is free to change the contents of their local datastore or attempt to submit malicious information to other nodes. In most cases however, we find that it is in a peer's best interest to act honestly and remain guided by market forces. Peers submitting exaggerated orders to the network that deviate too greatly from market price will either be aggresively filled by the validator nodes or simply ignored (depending on their relation to other orders).

So while peers are free to manipulate their own datastore and submit exaggerated data, the economic incentives suggest that they should mostly not do so. We believe our project only scratches the surface of this exciting topic and hope to see more research along these lines.

5.3 Deviations

This section briefly describes any deviations from the initial description of work as stated in Chapter 1.

5.3.1 Various Order Types

Our initial implementation envisioned a variety of different order types that would effectively mimic the order types available from centralised exchanges, including Day Orders, Open Orders, Good until Date Orders, Fill or Kill Orders, and Immediate or Kill Orders. Over time we found that the implementation of such varied and niche order types would benefit our protocol more after the characteristics and general pros and cons of our architecture have become more crystallised.

[2]We note that this conceptually resembles the idea of the blockchain itself, and care would need to be taken with the consensus scheme that scalability is not too heavily sacrificed

5.3.2 WASM

While not contained in our task description, we initially planned to deploy our application to the browser via WebAssembly, or WASM[3]. While excited by the innovation this demonstrated, we found it difficult to find suitable libraries while developing the peer networking aspects of our system and we evenutally dropped WASM in favor of a pure Javascript approach approximately midway through the project.

[3]WebAssembly is a binary- based compilation target which aims to supplant the Javascript scripting capabilities offered by web browsers. See `https://webassembly.org/`

Chapter 6

Summary and Conclusions

The DEX implementation we propose presents a novel architecture for exchanging orders directly via the browser in a peer-to-peer manner. As an open and decentralised exchange that allows users to create their own Token Sets, our platform offers a unique bundle of features in the non-custodial DEX market. Our system hopes to build on the fundamental techniques used for building decentralised exchange platforms and solve an essential problem for traders and investors alike, while empowering organisations to efficiently raise funds. Throughout the entire lifecycle of the platform, we have managed to keep the protocol decentralised in nature.

Our system is a web application platform hosted on IPFS. There is no centralised entity hosting the system, no centralised entity storing the orders, and no centralised entity matching the orders. A simple platform abstracts away all the complexities of web3, DeFi and DHT networking for users. All the packages & modules used to build our system are open source projects, and thus open to review. Peers in the network can submit signed messages that broadcasts their order to the network, and clever limits and market forces minimise problems with front running.

Our system uses the well known libp2p networking library developed by the creators of IPFS. Peers communicate via WebRTC, a protocol for enabling direct connections between devices in peer-to-peer networks. We have configured our own WebRTC signaling server to assist with the SDP handshake necessary to establish a WebRTC connection. Additionally, our WebRTC signaling server doubles as a bootstrap node to aid initial peer discovery. A Kademlia Distributed Hash Table is used to maintain an address book of available peers, and a PubSub abstraction is used as over a gossip protocol to exchange order information between peers in the network. For efficient serialisation, we utilise protocol buffers to encode order message information, and we persist orders from peers in the user's browser-based IndexedDB.

All orders are validated and submitted to the chain in a decentralised manner by special validator nodes in the network. We incentivise validator nodes by providing a commission payment on the total order price. Validator nodes run special order matching algorithms in a Web Worker in order to validate and match orders without blocking the main application thread. Validation logic on-chain verifies that orders submitted for execution by validator

nodes are indeed valid and safe to execute against one another. We developed a novel and sophisticated technique for signing transactions on behalf of validators without the need to prompt for signature authorisation and without knowledge of their private key, and we present methods that can be used to limit the economic risk associated with this technique which we term *Delegated Signing*.

We also give due attention to demonstrating how high scalability and transaction throughput might be achieved in our system. We use a plasma chain layer 2 solution, Polygon[1], in order to maximise transaction throughput when committing orders to the chain. Seamless future research and developments surrounding the scalability of our architecture should be possible thanks to its modular nature, and we present some initial performance measures and as well as potential bottlenecks in transaction throughput along with suggestions for improvement going forward.

Apart from enabling trading in a decentralised manner, our system goes one step further in allowing the user to create their own Token Sets that follow ERC-1155 multi token standards, allowing an entry point for the entire retail market to the Ethereum ecosystem with only a Metamask wallet as a prerequisite. Owners of tokens can also elect to reward the token holders with dividends, providing a powerful platform for finance without any barriers to entry.

Almost all the requirements mentioned in the initial project descriptions are implemented, and detailed explanations are provided with each module along with a justification on design choices made over the course of the project. This project has demonstrated a novel DEX architecture and sophisticated techniques that we hope can push current DEX implementations to evolve towards the next frontiers of functionality, usability, and scalability. In the final section which follows, we detail some important future research topics.

[1] also known as MATIC

Chapter 7

Future Work

7.1 Upgrades to the Existing Protocol

This section describes potential protocol improvements that could be further integrated into the existing architecture.

7.1.1 Order Analysis Chart

Currently users lack a bird's eye view of the moving prices of tokens in the marketplace. Analytic dashboard upgrades is an important UX upgrade as this gives the users a one-stop dashboard to make decisions regarding their portfolios. A breakdown of all investments along with price candles with moving averages could be a good starting point for a basic analytic overview. An implementation could easily build on the already existing indexing capabilities provided by the Graph Protocol indexer.

7.1.2 Order Matching Improvements

The results of our current order matching algorithm's performance are promising, but there exists room for improvement in order to match available throughput offered by leading DEX's in the market. As a starting point, we envision that effectively bundling the transactions could greatly increase possible order throughput via the validator nodes. Event logs could potentially be used to notify clients of rejected orders in a batch of transactions, obviating the need to reject the entire bundle.

Additionally, our system could benefit from additional types of orders, such as pure market orders[1], or more sophisticated order types like "Fill or Kill". These changes would need to be friendly with regards to resource requirements, as the matching algorithm is distributed on validator nodes and these nodes may have compute power limitations.

[1] Order that fills immediately at the prevailing market price

7.2 New Feature Implementation

This section lists out potential upgrades which would constitute a major protocol upgrade and thus require substantial changes to our system design.

7.2.1 A Strengthened Consensus Model

As detailed in Chapter 5, our system currently does not implement any strong consensus model and instead relies on a simple "flooding" mechanism to propagate match messages between peers. Our system could benefit from a stronger consensus model to prevent two or more validators from attempting to execute the same order. Additionally, peers might implement a stronger consensus model for the orderbook in general. These improvements would need to be considered in the light that they may have an adverse affect on scalability. Luckily, at this time the throughput bottleneck in our architecture is still the submission of orders to the chain, so it is feasible this area might be explored without the risk of greatly harming order throughput. We note that PouchDB might be a suitable option for an implementation as it supports similar guarantees to traditional ACID in a distributed setting[14].

7.2.2 Staking Option for the Token Holders

Our system is a non-custodial DEX focused on trading, so users don't have an opportunity to earn a return from the tokens held in their wallets. Staking is a very lucrative investment opportunity and can assist with solving liquidity problems on exchanges. Adding a staking functionality could make the platform more attractive for users and allow the platform to serve as a gateway for a friendly integration with liquidity pools offered by protocols such as Aave[2] or Uniswap.

7.2.3 Alternative Layer 2 Methods: ZK Rollups

One of our application's shortcomings is the necessary delay in awaiting block confirmations when validators submit orders to the chain. The polygon plasma layer is efficient and suitable for retail trading, but not high performance enough to serve as a high frequency trading platform. We have already enumerated one technique which we believe could make a great difference in bundling the transactions. Additionally, roll-up technology could make a significant difference in the performance and experience of the users using the platform, and the technical implementation of rollups merges well with our decentralised orderbook architecture. Our system faces the classic blockchain trilemma of scalability, security and decentralisation, and as in the more general blockchain space, ZK-Rollups offer a very promising avenue to push the frontier of these three limitations.

[2]A decentralised lending protocol, see: `https://aave.com/`

Bibliography

[1] Jason Ahmad, Shamsundar Dileep, and Branden Napier. IDEX 2.0: The Next Generation of Non-Custodial Trading. White paper, 2019. Available at: https://idex.io/document/IDEX-2-0-Whitepaper-2019-10-31.pdf.

[2] Jason Ahmad, Shamsundar Dileep, and Branden Napier. IDEX 1.0 vs 2.0 - So Much More than Scaling. https://blog.idex.io/all-posts/idex-10-vs-20-so-much-more-than-scaling, 2020.

[3] Andreas Antonopoulos and Gavin Wood. *Mastering Ethereum: building smart contracts and DApps*. O'Reilly Media, Inc., 1st edition, 2018.

[4] Martin Becze and Hudson Jameson et al. EIP-1: EIP Purpose and Guidelines, Ethereum Improvement Proposals, no. 1. https://eips.ethereum.org/EIPS/eip-1, 2015.

[5] Binance. Binance DEX matching logic. https://docs.binance.org/match.html, 2022.

[6] Binance. BSC Relayer. https://docs.binance.org/smart-chain/guides/concepts/bsc-relayer.html, 2022.

[7] Binance. Cross Chain Communication. https://docs.binance.org/smart-chain/guides/concepts/cross-chain.html, 2022.

[8] Binance. Introduction of Binance Smart Chain. https://docs.binance.org/smart-chain/guides/bsc-intro.html, 2022.

[9] Binance. Oracle Relayer. https://docs.binance.org/smart-chain/guides/concepts/oracle-relayer.html, 2022.

[10] Binance. Periodic auction matching. https://docs.binance.org/anti-frontrun.html, 2022.

[11] Bitfinex. Introducing "Hive" - A distributed matching engine for digital asset trading. https://blog.bitfinex.com/announcements/introducing-hive/, 2018.

[12] Remco Bloemen, Leonid Logvinov, and Jacob Evans. EIP-712: Ethereum typed structured data hashing and signing, Ethereum Improvement Proposals, no. 712. https://eips.ethereum.org/EIPS/eip-712, 2017.

[13] Corey Bothwell, Jerome Faessler, and Matteo Gamba. Group 07: ERC-721 NFT with ERC-20 & Off-Chain Signing / Report. Technical report, University of Zurich, Faculty of Business, Economics and Informatics, Blockchain Programming Seminar, 2021.

[14] Gareth Bowen. PouchDB 7.2.1: New IndexedDB Adapter. https://pouchdb.com/2020/02/12/pouchdb-7.2.0.html, 2020.

[15] Steven Buchko. What Is the Waves Platform? | The Ultimate Guide. https://coincentral.com/waves-platform-beginner-guide/, 2018.

[16] Vitalik Buterin. Ethereum Whitepaper. White paper, Ethereum Foundation, 2013.

[17] Vitalik Buterin and Fabian Vogelsteller. EIP-20: Token Standard, Ethereum Improvement Proposals, no. 20. https://eips.ethereum.org/EIPS/eip-20, 2015.

[18] Whitefield Diffie and Martin Hellman. New Directions in Cryptography. *Transactions on Information Theory*, IT-22 - No. 6:644, 1976.

[19] dYdX. dYdX Documentation. https://legacy-docs.dydx.exchange/#trading-api, 2021.

[20] dYdX. Layer 2 Alpha. https://dydx.exchange/blog/alpha, 2021.

[21] William Entriken, Dieter Shirley, Jacob Evans, and Nastassia Sachs. EIP-721: Non-Fungible Token Standard, Ethereum Improvement Proposals, no. 721. https://eips.ethereum.org/EIPS/eip-721, 2018.

[22] Waves Exchange. Matcher Fee. https://docs.waves.exchange/en/waves-matcher/matcher-fee, 2022.

[23] Waves Exchange. Waves.Exchange Protocol. https://docs.waves.exchange/en/waves-exchange/waves-exchange-protocol#motivation, 2022.

[24] Ethereum Foundation. Ethereum.org - Proof-of-stake (PoS). https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/, 2022.

[25] Ethereum Foundation. Layer 2 Scaling. https://docs.ethhub.io/ethereum-roadmap/layer-2-scaling/zk-rollups/, 2022.

[26] Ethereum Foundation. ZK-Rollups. https://docs.ethhub.io/ethereum-roadmap/layer-2-scaling/zk-rollups/, 2022.

[27] Google. Protocol Buffers. https://developers.google.com/protocol-buffers, 2022.

[28] Google. WebRTC - Real-time communication for the web. https://webrtc.org/, 2022.

[29] Graph. Graph Protocol Introduction. https://thegraph.com/docs/en/about/introduction/, 2021.

[30] Hardhat. Ethereum development environment for professionals. `https://hardhat.org/`, 2022.

[31] IDEX. Matching Engine. `https://docs.idex.io/#matching-engine`, 2021.

[32] Infura. The Worlds Most Powerful Blockchain Development Suite. `https://infura.io/`, 2022.

[33] Antonio Juliano. Introducing the new dYdX. `https://medium.com/dydxderivatives/introducing-the-new-dydx-9675719bacb6`, 2019.

[34] A. Keranen, C. Holmberg, and J. Rosenberg. Interactive connectivity establishment (ice): A protocol for network address translator (nat) traversal. RFC 8445, RFC Editor, July 2018.

[35] Protocol Labs. Publish/Subscribe Documentation. `https://docs.libp2p.io/concepts/publish-subscribe/`, 2021.

[36] Protocol Labs. Libp2p - A modular network stack. `https://libp2p.io/`, 2022. See GitHub repository: `https://github.com/libp2p`.

[37] Peter Maymounkov and David Mazires. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. *Independently Published*, 2002.

[38] Ralph Merkle. A Digital Signature Based on a Conventional Encryption Function. *LNCS*, 293:69–37, 1987.

[39] Metamask. A crypto wallet & gateway to blockchain apps. `https://metamask.io/`, 2022.

[40] Mozilla. Introduction to WebRTC protocols. `https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Protocols#ice`, 2022.

[41] Mozilla. Using Web Workers. `https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers`, 2022.

[42] Nasdaq. Fintech: Staking is the Quiet Giant of Crypto Yield. `https://www.nasdaq.com/articles/staking-is-the-quiet-giant-of-crypto-yield-2021-06-23`, 2021.

[43] Matic Network. Whitepaper. White paper, 2020. Available at: `https://github.com/maticnetwork/whitepaper`.

[44] Joseph Poon and Vitalik Buterin. Plasma: Scalable Autonomous Smart Contracts. Technical report, `plasma.io`, 2022.

[45] Jean-Jacques Quisquater, Louis Guillou, and Thomas Berson. How to Explain Zero-Knowledge Protocols to Your Children. *Advances in Cryptology - CRYPTO '89: Proceedings*, 435:628–631, 1990.

[46] Witek Radomski, Andrew Cooke, Philippe Castonguay, James Therien, Eric Binet, and Ronan Sandford. EIP-1155: Multi Token Standard," Ethereum Improvement Proposals, no. 1155. `https://eips.ethereum.org/EIPS/eip-1155`, 2018.

[47] Starkware. StarkEx Docs V2. https://docs.starkware.co/starkex-docs-v2/overview, 2022.

[48] The Tokenist. Centralized Crypto Exchanges Had 14x More Trading Volume Than DEXes in 2021. https://tokenist.com/centralized-crypto-exchanges-had-14x-more-trading-volume-than-dexes-in-2021/.

[49] Daniel Wang, Jay Zhou, Alex Wang, and Matthew Finestone. Loopring: A Decentralized Token Exchange Protocol. White paper, 2018. Available at: https://loopring.org/resources/en_whitepaper.pdf.

[50] Open Zeppelin. Open Zeppelin Contract Library. https://github.com/OpenZeppelin/openzeppelin-contracts, 2022.

List of Figures

List of Tables

Appendix A

Installation Guidelines

We include instructions for a full installation of our system. To avoid redundancy, these instructions mirror those found in our repository README[1].

A.1 Using the Application

Our application is deployed to IPFS. To use the application, simply navigate to the following url: PLACEURLHERE. Alternatively, you can navigate to the application by requesting the following content hash from IPFS: HASH GOES HERE Once at the application, navigate to the *"How it Works"* page for an overview on how to use the system.

Note: You will need to have Metamask installed in order to access the application.

Alternatively, you can build the application locally.

A.2 Building and Running a Local Copy of the Application

In order to build the system locally, the following requirements must be met:

- Node.js 16.3.2 or equivalent version
- Node Package Manager (NPM) 8.1.2 or Yarn 1.22.17 (or equivalent versions)
- Any browser supporting MetaMask Eg: Firefox, Brave, Chrome, etc.
- Install the Metamask browser extensions in the browser of choice

After cloning the repository, please follow the steps below.

[1] `https://github.com/map-bgp/browserbook`

A.2.1 Compiling and Deploying the Smart Contracts

The first thing to do is to navigate to the chain directory and install all dependencies:

```
1 cd chain && yarn
```

Option A: Building for Hardhat (locally)

We use Hardhat as a development environment for compiling, building, testing, and deploying the Smart Contracts. Using Hardhat simplifies the deployment of the contracts to a local Ethereum node or any EVM-based Network. For local deployments, Hardhat mimics a public blockchain on your private machine for development. In the `./chain` directory, run the following command:

```
2 yarn hardhat node
```

to start a local node. You are now running a local blockchain and can deploy Browserbook against it for testing purposes. To continue, open another terminal and navigate again to `./chain` and run

```
3 yarn hardhat run ./scripts/deploy.ts --network localhost
```

Hardhat will compile the contracts and deploy them to your local node. Be sure to note the console output which should display something like the following:

```
4 TokenFactory deployed to: 0x5FbDB2315678afecb367f032d93F642f64180aa3
5 Exchange deployed to: 0xe7f1725E7734CE288F8367e1Bb143E90bb3F0512
6 Done in 4.00s.
```

Note that in your output the provided addresses may differ.

Option B: Building for Mumbai (Testnet)

Before deploying to Mumbai, a few environment variables must be set. Copy the `env.example` file to `.env` via

```
7 cp env.example .env
```

You should see the following variables.

- `ETHERSCAN_API_KEY`
- `PRIVATE_KEY`

The Etherscan API Key is optional; the private key variable must be set and is the private key of the account with which you would like to deploy the contracts to the Mumbai testnet. Note that this account must have a sufficient balance of MATIC to cover the deployment gas fees.

We need to make one manual adjustment to our contract in order to ensure it is compatible with the Mumbai testnet. Navigate to `./chain/contracts/Exchange.sol` and change the `chainId` on line 48 from `31337` to `80001` (We optimised for users deploying to Hardhat. 31337 is the chainId for the local Hardhat node, while 80001 is the chainId for Mumbai). Be sure to save the file.

Once the above is complete, run the following command from `./chain`:

```
8 yarn hardhat run ./scripts/deploy.ts --network mumbai
```

You should see something like the following:

```
9  TokenFactory deployed to: 0x5FbDB2315678afecb367f032d93F642f64180aa3
10 Exchange deployed to: 0xe7f1725E7734CE288F8367e1Bb143E90bb3F0512
11 Done in 4.00s.
```

Be sure to note the deployment addresses as they are used to successfully build the client.

A.3 Building and Running the Client Application

In order to interact with the smart contracts, we need to build and run the client application. Once again the first thing to do is install required dependencies by navigating to `./client` and running yarn.

```
12 cd client && yarn
```

Before we can build the application, we need to set some important environment variables. Copy the example environment file from `env.example` via:

```
13 cp env.example .env
```

You will need to set the following environment variables in order to successfully build the application:

- `TOKEN_FACTORY_ADDRESS` - Use the deployment address output from the console above
- `EXCHANGE_ADDRESS` - Use the deployment address output from the console above
- `SIGNER_RPC_URL` - Use `http://34.134.45.184:8545`

- `PERF_TEST_KEY_BUY` - *See Application README*
- `PERF_TEST_KEY_SELL` - *See Application README*

We use the above private keys to perform dummy transactions in the performance test. The private keys above are known private keys provided by the local Hardhat node, so they are safe to distribute here. When running the Hardhat node, they have an initial balance of 10000 Ether on the local network, so they are well suited for running the performance test. (Never send live Ether to the addresses associated with these private keys) If you want to use different private keys for a performance test (say you are running on a different network), you can change them here. There are some additional steps to get up and running with performance testing as detailed on the application's *"How it Works"* page.

Note: If you are running the application to talk to the Mumbai testnet, we need to similarly make one manual change to the codebase. Navigate to `./client/src/app/oms/OrderService.ts` and change the `chainId` on line 22 from `31337` to `80001`.

After optionally performing the above and setting the appropriate environment variables, we are ready to run the application.

To run the application locally simply run:

```
14 yarn dev
```

Please be sure that port 3000 is not occupied. To visit the application, navigate to `localhost:3000` in your browser. Please ensure that Metamask is installed. Also note that you will need to ensure Metamask is configured to either talk to your local Hardhat node or an appropriate Mumbai RPC. For Hardhat, the appropriate RPC url is `http://localhost:8545`, while an example Mumbai RPC is `https://rpc-mumbai.matic.today`. The chainId's are `31337` and `80001` respectively.

If you would like to create a production build, run the following:

```
15 yarn build
```

This will create a `./dist` directory with a production build which is deployable to any sort of static file hosting, including IPFS. **Note:** Because of a particularity in the way that static file servers (including IPFS) handle our build tools output, it is necessary to make a small change to the `./dist/index.html` file before hosting in order to ensure that nested routes/content are properly referenced. Navigate to the `./dist/index.html` file and replace all root paths such as `<scriptsrc="/index.96e2502d.js"defer=""></script>` to be relative paths i.e. `<scriptsrc="./index.96e2502d.js"defer=""></script>` We recommend performing this manually as there should only be three such instances to change in your output. Once complete the files are ready for upload to a hosting provider.

For any questions related to these instructions the project maintainers are always available.

A.4 Running Your own Bootstrap Server

In order to run a bootstrap server, ensure that Docker is installed and that port 9090 is free. Execute the following two lines of code. The application is started on the network details which are consoled out.

```
16 DOMAIN=${DOMAIN_NAME} docker -compose up
```