Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Decentralizovaná směnárna v solidity

Plzeň 2024 Jakub Burian

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: 2023/2024

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Jakub BURIAN**
Osobní číslo:	**A20B0069P**
Studijní program:	**B0613A140015 Informatika a výpočetní technika**
Specializace:	**Informatika**
Téma práce:	**Decentralizovaná směnárna v Solidity**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Prostudujte principy fungování blockchain distribuovaných databází. Analyzujte nejpoužívanější decentralizované směnárny na blockchainu Etherea nebo EVM (Ethereum Virtual Machine) kompatibilních.
2. Na základě provedených analýz navrhněte vlastní decentralizovanou směnárnu. Návrh realizujte v programovacím jazyce Solidity.
3. Vytvořte frontend pro snadnou komunikaci a vizualizaci dat z "chytrých smluv" použitím javascriptové knihovny Next.js.
4. Kriticky zhodnoťte dosažené výsledky.

Rozsah bakalářské práce: **doporuč. 30 s. původního textu**
Rozsah grafických prací: **dle potřeby**
Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

Dodá vedoucí bakalářské práce.

Vedoucí bakalářské práce: **Ing. Miloslav Konopík, Ph.D.**
Katedra informatiky a výpočetní techniky

Datum zadání bakalářské práce: **29. srpna 2023**
Termín odevzdání bakalářské práce: **2. května 2024**

L.S.

Doc. Ing. Miloš Železný, Ph.D.
děkan

Doc. Ing. Přemysl Brada, MSc., Ph.D.
vedoucí katedry

V Plzni dne 29. srpna 2023

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 2. května 2024 Jakub Burian

Poděkování

Rád bych poděkoval Ing. Miloslavu Konopíkovi, Ph.D., za odborné vedení, cenné rady a věcné připomínky při zpracování této práce.

Jakub Burian

Abstract

The first part of this thesis introduces the reader to the principles of the blockchain, the Ethereum network, and exchanges. The reader will acquire basic knowledge about the functioning of the two most widely used decentralized exchanges. On the basis of these information, a design and implementation of a fully decentralized exchange based on a list of buy and sell orders (orderbook). The work focuses on efficient execution considering the costs of executing transactions on the Ethereum network. In the last part of the thesis, the costs of the resulting implementation are analyzed and evaluated.

Abstrakt

První část práce seznámí čtenáře s principy blockchainu, sítě Ethereum a směnáren či burz. Čtenář dále nabude základních znalostí o fungování dvou nejpoužívanějších decentralizovaných směnáren. Na základě těchto poznatků je proveden návrh a implementace plně decentralizované směnárny na principu seznamu nákupních a prodejních objednávek. Práce se zaměřuje na efektivní provedení vzhledem k cenám nákladů na provedení transakcí na síti Ethereum. V poslední části práce jsou analyzovány a zhodnoceny ceny nákladů ve výsledné implementaci.

Obsah

1 Úvod

Blockchain a kryptoměny se dostávají poslední dobou více do povědomí. Hlavním zástupcem kryptoměn je bitcoin. Jednou z hlavních myšlenek kryptoměn je decentralizace. S tím logicky přichází požadavek, aby bylo možné nakupovat a prodávat kryptoměny bez centrální autority. Tento požadavek lze realizovat na Ethereu, což je momentálně druhá největší kryptoměna hned po bitcoinu. Spolu s bitcoinem se považují za nejvíce decentralizované kryptoměny, jelikož jejich provoz zajišťují tisíce strojů po celém světě. Oproti bitcoinu má podstatnou nadstavbu, která umožňuje vytvářet a spouštět téměř jakékoliv programy na síti Etherea, kde momentálně běží nespočetně decentralizovaných směnáren, na kterých se dá obchodovat téměř jakákoliv kryptoměna či token. Cílem této práce je provést analýzu fungování existujících nejpopulárnějších decentralizovaných směnáren a na základě analýz navrhnout a realizovat vlastní decentralizovanou směnárnu na síti Etherea na principu seznamu nákupních a prodejních objednávek (orderbook).

2 Ethereum

Ethereum je distribuovaná síť propojených počítačů známá jako uzly, na kterých běží instance Ethereum klienta. Tento klient verifikuje bloky a transakce a má uloženou celou kopii blockchainu Etherea. Tento klient se označuje jako *full node*.

Také existují *light node* klienti, kteří nemají uloženou celou kopii blockchainu, ale jen hlavičky bloků. Tento typ klienta umožňuje verifikaci dat proti kořenovým uzlům získaných z hlaviček bloků. K fungování nicméně stále potřebuje nějaký externí *full node*, ze kterého přečte všechna potřebná data. Výhodou tohoto klienta je kompaktnost a nízké paměťové a hardwarové nároky [1].

2.1 Blockchain

Blockchain je posloupnost záznamů spojených pomocí hashe předchozího záznamu. Záznamy se nazývají bloky. Základním požadavkem na blockchain je neměnnost, tedy že nejde změnit data již přidaných bloků. Každý blok obsahuje informace důležité pro neměnnost a bezpečnost. Mezi ně patří hlavně hash předchozího bloku a hash tohoto bloku. Pokud by se změnila nějaká informace v již přidaných blocích, hashe by již na sebe nenavazovaly. Hashe jsou určeny na základě jednotlivých dat uložených v blocích. Každý blok obsahuje (mimo již zmíněná) následující data [2]:

Nonce – Je 64bitové číslo, které se musí najít pomocí speciálních algoritmů (nebo náhodných) a prokázat tak práci vykonanou pro vytvoření tohoto bloku.

Beneficiary – Adresa, která úspěšně vytvořila tento blok a našla tak odpovídající *nonce*. Této adrese se připíše odměna ve formě Etherea (Ether je token, kterým se platí za transakce na síti Ethereum).

Difficulty – Určuje, jak složité je najít odpovídající *nonce* pro vytvoření nového bloku. Je možné ji vypočítat na základě složitosti předchozího bloku a časové značky.

State root – Hash kořenového uzlu stavové struktury (trie) po provedení všech transakcí. Tato stavová struktura obsahuje např. konta všech adres, data uložená na blockchainu a kódy chytrých kontraktů.

Transactions root – Je kořenový hash *Merklova stromu* vytvořený z transakcí (viz 2.6) náležících bloku. *Merklův strom* je binární strom, kde každý uzel je hash vzniklý z hashe potomků. Listy stromu vzniknou z hashe dat, v tomto případě jsou těmito daty transakce [3].

Gas limit – Určuje, kolik nákladů (paliva, viz 2.4) je možné maximálně použít v tomto bloku. Tento limit je přesně určen a mění se zřídka.

Gas Used – Určuje, kolik nákladů (paliva) bylo nutné použít pro zpracování všech transakcí zahrnutých v tomto bloku.

Timestamp – Je unixová časová značka (v sekundách) udávající, kdy byl blok vytvořen.

2.2 Stav

Stav v síti Etherea je mapování mezi adresami (160bitové) a uloženými daty této adresy. Každá adresa má následující data [2]:

Nonce – Nonce (neplést s nonce bloku) je číslo reprezentující počet transakcí poslaných z této adresy. Pro každou adresu začíná od 0 a inkrementuje se každou poslanou transakcí.

Balance – Je číslo reprezentující, kolika Etherea tato adresa disponuje. Číslo se udává ve wei jednotkách, což je nejmenší nedělitelná jednotka v Ethereu. Jedno Ethereum je 1e18 wei.

Storage root – Kořenový hash *Merklova stromu* obsahujícího data účtu.

Code hash – Hash kódu kontraktu uložený na adrese. S tímto hashem lze získat kód, který je proveden při interakci s touto adresou. *CodeHash* je nulový, jedná-li se o EOA (externally owned wallet) adresu, a nejedná se tedy o kontrakt.

2.3 Ethereum Virtual Machine (EVM)

EVM je jednoduchá big-endianová zásobníková architektura operující na 256bitových slovech [2]. Probíhají na ní instrukce, které transformují její stavy z jednoho na druhý. EVM je turingově kompletní, ale je limitován maximálním palivem, jež je možné za blok nebo transakci provést. EVM je kompletně izolovaný systém, který nemá přístup na externí soubory např. na webu nebo externím disku. Výhoda tohoto provedení je zvýšená bezpečnost a determinismus systému.

2.4 Palivo

Za zpracování transakce platí odesílatel etherem [2]. To slouží primárně k tomu, aby síť Etherea nebyla zahlcována zbytečnými transakcemi, které by jinak síť zpomalovaly a docházelo by k DoS (denial of service) útokům. Čím vyšší vytíženost sítě je, tím vyšší je cena za palivo. Transakce s vyšší cenou za palivo jsou zpravidla zpracovány přednostně, nicméně není to pevně dané pravidlo. Pořadí zpracování transakcí si určují těžaři, ale je logické, že upřednostní transakce s vyšší cenou, když všechny poplatky za zpracované transakce jdou právě těžařům.

Částka, kterou odesílatel zaplatí, se odvíjí podle jednotky ceny za jednotku paliva a počet jednotek použitého paliva. Nicméně kolik paliva bude použito, se před zpracováním transakce nedá přesně určit, odesílatelé tedy udávají parametr zvaný limit paliva, jenž určuje, kolik je možné na transakci maximálně použít paliva. Před zpracováním transakce se celá částka odečte od odesílatele a nepoužité palivo se ihned po zpracování přičte zpět. Nepoužitým palivem se myslí rozdíl mezi limitem paliva a spotřebovaným palivem.

Každá programová instrukce má přesně daný počet paliva, který je nutný k jejímu provedení. Mezi nejdražší instrukce patří zápis a čtení ze *storage* paměti (viz 2.5). Příklady některých instrukcí:

SSTORE – Je instrukce pro zápis do paměti typu *storage*. Spotřebuje ~5000 paliva, jedná-li se o změnu nějakého slotu. Spotřebuje ~20000 paliva, jedná-li se o první zápis slotu. Jeden slot je velký 32 bajtů.

SSLOAD – Je instrukce pro přečtení nějakého slotu z paměti. Spotřebuje 2100 paliva, jedná-li se o první čtení slotu. Prvním se myslí jen v kontextu zpracování jedné transakce. Pokud se ve stejné transakci čte ten samý slot po několikáté, tato operace spotřebuje jen 100 paliva.

ADD – Je instrukce, která sečte dvě čísla. Spotřebuje jen 3 jednotky paliva. Většina aritmetických a bitových operací spotřebovává jen pár jednotek paliva.

MSTORE, MLOAD – Jsou instrukce pro čtení a zápis do paměti typu *memory*. Zpravidla spotřebují 3 jednotky paliva.

2.5 Typy paměti

V EVM se paměť dělí na následující typy [4]:

Memory – *Memory* je volatilní paměť, neukládá se tedy navždy na blockchain. Životnost této paměti je jen v rámci jedné transakce, po jejím zpracování (vykonání kódu) je tato paměť vymazána. Dala by se přirovnat k paměti RAM. Zápisové operace mohou být 8- až 256bitové, kdežto čtecí operace mohou být pouze 256bitové. To ale nemusí být vždy vhodné, jelikož pokud je potřeba přečíst například pouze 8 bitů, musí se přečíst 256 bitů a vynaložit tak více potřebného paliva na zpracování. V tomto typu paměti může být tolik prvků, kolik limit paliva dovolí.

Zásobník – EVM je zásobníková architektura, provádí se zde tedy všechny výpočty. Velikost zásobníku je 1024 prvků a obsahuje také reference na hodnoty z paměti *memory*. Operuje na 256 bitech.

Storage – *Storage* paměť by se dala přirovnat k pevnému disku. Data jsou založená na klíč–hodnota párech a jsou permanentně uloženy na blockchainu. Klíče a hodnoty jsou 256bitové. Zapisovací a čtecí operace jsou také 256bitové, což nemusí být vždy výhoda, pokud se pracuje jen například s 8bitovou hodnotou. Kvůli bezpečnosti je *storage* přístupná jen jí samotné. Operace jsou zde dražší vzhledem k palivu oproti paměti *memory* a zásobníku.

2.6 Transakce

Transakce je transformace stavu EVM z jednoho na druhý [2]. Je inicializována z nějaké EOA adresy, ale nemůže být inicializována z adresy, která patří nějakému kontraktu. Ke každé EOA adrese náleží privátní klíč, kterým se transakce kryptograficky podepisují. Transakce se dá vyvolat z jakéhokoliv uzlu. Typicky se používají veřejně snadno přístupné uzly, jako je např. Infura (viz 5.1).

Transakce jsou zpracovány atomicky. To znamená, že buď proběhne celá a najednou, nebo vůbec. V praxi to také znamená, že v jednom okamžiku je zpracovávána jen jedna transakce. Nemůže tak dojít ke konfliktům a výsledek je vždy deterministický. Pokud dojde k nějaké chybě při zpracování transakce (např. dělení nulou) a byly již předtím použity zápisové instrukce do paměti *storage*, *storage* se vrátí do stavu před začátkem zpracování té transakce.

2.6.1 Typy transakcí

Regulární transakce – Je jednoduchá transakce tokenu Ether z jedné adresy na druhou. Při této transakci se dají také poslat volitelná data (například zakódovaná zpráva).

Exekuční transakce – Je transakce, kde příjemcem je nějaký kontrakt a prvních 8 bajtů dat je identifikátor nějaké funkce. Tyto identifikátory jsou určeny z hashe, který vzniká z názvu a signatur funkcí.

Nasazující transakce – Je transakce, při které není specifikována adresa příjemce a poslaná data jsou bajtkódem kontraktu. Vznikne nový kontrakt s daným bajtkódem. Adresa vytvořeného kontraktu je obvykle určena z hashe adresy odesílatele a její *nonce*.

2.6.2 Parametry transakcí

Typ – Typ transakce specifikuje, zda se jedná o starou transakci nebo o novou transakci typu EIP-1559 (Ethereum Improvement Proposal). Tento nový typ transakce byl zaveden s cílem zrychlit provedení transakcí a zmenšit soutěživost o to, která transakce proběhne dříve.

Nonce– Je číslo udávající, kolik transakcí odesílatel doposud vytvořil.

Gas Price – Je číslo udávající, kolik chce odesílatel transakce zaplatit za 1 jednotku paliva. Pokud se jedná o EIP-1559 transakci, tento parametr se rozkládá na 2 další, kterými jsou a *maxFeePerGas* a *maxPriorityFeePerGas*. *MaxFeePerGas* určuje, kolik maximálně je odesílatel ochotný zaplatit za jednu jednotku paliva. *MaxPriorityFeePerGas* určuje, kolik maximálně je odesílatel ochotný zaplatit za jednu prioritní jednotku paliva.

GasLimit – Je číslo udávající, kolik maximálně může být na tuto transakci použito paliva. Pokud zpracování transakce vyžaduje více paliva, transakce selže, nicméně odesílatel zaplatí za každé použité palivo. Kolik paliva transakce spotřebuje, je před odesláním transakcí přesně určitelné a mění se pouze tehdy, pokud by došlo mezi posláním a zpracováním transakce k nějaké změně v datech blockchainu, se kterými

transakce interaguje. Minimální limit paliva je 21000, který se typicky používá jen při přesunu Etherea z adresy na adresu bez interakce s kontrakty.

To – Je adresa příjemce transakce. Příjemcem může být EOA adresa nebo adresa kontraktu. V případě vytváření nového kontraktu je tato hodnota nulová.

Value – Označováno jako hodnota transakce. Je to číslo udávající, kolik Etherea se posílá příjemci transakce.

R, s – Jsou hodnoty náležící k podpisu transakce. Z těchto hodnot lze vypočítat adresu, která tuto transakci podepsala.

Chain ID – Číslo, jehož primární účel je oddělení transakcí od jiných EVM sítí.

Data – Při základní transakci, která jen převádí Ether, je tento parametr obvykle nulový. Pokud se jedná o interakci s kontraktem, prvních 8 bajtů je identifikátor funkce, kterou má transakce vyvolat. Zbylá data za identifikátorem jsou použita jako parametry funkce. Pokud se jedná o transakci, která vytváří nový kontrakt, je tento parametr bajtkódem vytvořeného kontraktu.

2.6.3 Eliptická křivka

Eliptická křivka je matematická křivka definovaná rovnicí ve tvaru $y^2 = x^3 + ax + b$, kde a a b jsou konstanty [5]. Důležitou vlastností eliptických křivek je, že jsou grupami, což znamená, že mají způsob, jak přidávat body na křivce k sobě, aby získaly další bod na křivce. Tato vlastnost činí eliptické křivky užitečnými v kryptografii, zejména při generování veřejných a privátních klíčů, které se používají pro zabezpečenou komunikaci a transakce.

2.6.4 Podepisování transakce

Každá EOA adresa má svůj privátní klíč, z něhož se dá odvodit její veřejný klíč, ale z veřejného klíče nejde odvodit privátní. Z veřejného klíče lze snadno určit EOA adresu, která je posledních 20 bajtů z *keccak-256* hashe veřejného klíče. Privátní klíč je 256bitový a používá se primárně při podepisování transakcí.

Ethereum používá kryptografii nad eliptickou křivkou. Jedná se o asymetrickou kryptografii založenou na modulární aritmetice využívající problém diskrétního logaritmu. Každý veřejný klíč je nějakým bodem (x, y) na eliptické křivce. Veřejný klíč se určí podle vzorce K = k * G, kde k je privátní klíč a G je generátor, který je jasně daný. Operace násobení představuje operaci nad eliptickými křivkami.

K podepisování transakcí a zpráv se používá algoritmus ECDSA (Elliptic Curve Digital Signature Algorithm). ECDSA podpisy se skládají ze tří částí: *r*, *s* a *v* [6].

Zjednodušený algoritmus při podepisování zprávy privátním klíčem *p*:

1. Spočítej hash zprávy. Při podepisování transakcí se místo hashe zprávy počítá hash parametrů transakcí, které jsou speciálně kódované.
2. Vygeneruj náhodnou hodnotu *k*.
3. Najdi bod (x, y) na eliptické křivce násobením *k* a *G*.
4. Urči $r = x \bmod n$. Pokud je *r* nulové, jdi zpět na krok 2.
5. Urči $s = k^{-1}(e + rp) \bmod n$. Pokud je *s* nulový, jdi zpět na krok 2.

Jelikož se používá náhodné *k*, podpis je při podepisování té samé zprávy pokaždé jiný. Nicméně existuje způsob [6], jak bezpečně určit hodnotu *k* z privátního klíče *p* a hashe zprávy. Pokud se použije tento způsob, podpisy jsou deterministické a jsou vždy stejné při podepisování té samé zprávy.

Jelikož jen z hodnot *r* a *s* jdou na eliptické křivce určit dva rozdílné body (a tedy 2 veřejné klíče), musela se zavést ještě jedna hodnota *v*, která rozlišuje, o který z těch 2 veřejných klíčů se jedná. Výsledný podpis je spojením hodnot *r*, *s* a v a má 65 bajtů.

3 Solidity

Solidity je objektově orientovaný programovací jazyk pro vývoj kontraktů, které běží na EVM. Solidity je staticky typovaný jazyk. Kód v solidity se zkompiluje do bajtkódu, který se dá poté použít na nasazení na Ethereum. Nasazený bajtkód se liší, jelikož neobsahuje logiku konstruktoru (který již byl použit a není potřeba). Nasazený bajtkód se nazývá runtime bajtkód [7].

3.1 Chytrý kontrakt

Jedná se v podstatě o program běžící na Ethereu. Pojem „chytrý kontrakt" je označován jako protokol či program, který je samovykonávající, důvěryhodný a vykoná danou smlouvu (kód) mezi více účastníky [8]. V Ethereu je garantováno, že transakce vždy proběhnou přesně podle kódu chytrých kontraktů. Ty se dají na Ethereu (ale nemusí) zveřejnit a každý se může ujistit, že opravdu vykonávají to, co mají. V této práci je tento pojem ve zkrácené verzi jako „kontrakt".

3.2 ABI (Application Binary Interface)

S nasazenými kontrakty je potřeba komunikovat ve strojní (binární) podobě. Proto bylo zavedeno ABI, které definuje rozhraní kontraktů. Rozhraní určuje, jakým způsobem lze s nimi komunikovat. Dá se to přirovnat k API (Application Programming Language), ale na nižší úrovni. ABI je množina funkcí a atributů, které jsou typicky uložené ve formátu *json*. Při interakci s kontrakty se nemusí tak přímo znát hexadecimální identifikátory funkce, ale stačí k tomu názvy a parametry funkce, které se získají z ABI. Díky ABI se také interakce správně zakóduje do strojové podoby. Některé z elementů, které ABI obsahuje pro funkce [9]:

Typ – Typ funkce, může být klasická funkce nebo konstruktor, *receive* a nebo *fallback* funkce.
Name – Název funkce.

Inputs – Pole parametrů funkce. Každý parametr má název a datový typ.

Outputs – Pole výstupů funkce. Každý výstup má název a datový typ.

StateMutability – Může nabývat 4 hodnot. Nabývá hodnotu *pure,* pokud se jedná o funkci, která je pouze pro čtení a nepotřebuje ani číst stav blockchainu. Jako příklad takové funkce by mohla být primitivní funkce pro sečtení dvou čísel. Jako další hodnota je *view*, která na rozdíl od *pure* čte stav blockchainu. Poté jsou 2 hodnoty *payable* a *non-payable*, které mění stav blockchainu. Rozdíl mezi nimi je ten, že *payable* může přijmout Ether (hodnota transakce může být nenulová).

3.3 Kontrakt

Kontrakt se dá přirovnat k třídám jiných programovacích jazyků. Kontrakt umožňuje dědění jiných kontraktů. Dědí se pomocí klíčového slova *is*. Je možné dědit od více kontraktů. Kontrakt dále umožňuje implementaci rozhraní, které se implementuje stejným klíčovým slovem *is*. V kontraktech dále existují klíčová slova *this* a *super*. Pomocí *this* se dá získat například adresa kontraktu instance nebo konto Etherea. Pomocí *super* lze volat funkce rodičovského kontraktu [10].

3.4 Funkce

Principy funkcí v solidity jsou stejné jako v ostatních programovacích jazycích. Funkce může mít vstupní argumenty (limitováno 16 kvůli ukazateli zásobníku) a výstupní data. Funkce mají jeden ze 4 typů viditelnosti [11]:

External – Funkce je přístupná všem kromě kontraktu samému. Používá se pro zpřístupnění funkcionality a rozhraní jiným kontraktům.

Public – Funkce je přístupná všem včetně kontraktu samého. Na rozdíl od external může tuto funkci volat i ten samý kontrakt v ostatních funkcích.

Internal – Funkce je přístupná všem kontraktům, které dědí tento kontrakt.

Private – Funkce je přístupná pouze kontraktu samému.

Funkce se dá dále dělit podle toho, zda mění stav blockchainu, či nikoliv. Klíčové slovo *view* označuje, že funkce nezapisuje do blockchainu, ale může pouze číst jeho stav. Klíčové slovo *pure* označuje, že funkce ani nečte stav blockchainu (může být primitivní funkce sčítající 2 čísla předaná jako parametry).

Funkce mohou nabývat klíčového slova *override*, které označuje, že funkce přepisuje implementaci funkce rodičovského kontraktu. Funkce takového rodičovského kontraktu musí obsahovat klíčové slovo *virtual*.

3.5 Modifikátory

Modifikátory vyjadřují, jaké akce nebo omezení funkce mají v deklarativní a čitelné formě [12]. Jsou podobné návrhovému vzoru *dekorátor* v objektově orientovaném programování. Modifikátory tak mění chování funkce. Kód modifikátoru může být spuštěn před implementací funkce samotné, nebo po ní. Modifikátory také slouží k redundanci opakování kódu. Jeden z nejpoužívanějších modifikátorů je například *onlyOwner*, který se používá u funkcí, které nemůže vyvolat jakákoliv adresa, ale pouze jedna určitá (vlastník). Další z nejpoužívanějších modifikátorů je *non-reentrant* modifikátor, který způsobí selhání transakce, jestliže daná funkce je zavolána opět během jejího vykonávání.

3.6 Paměti z pohledu solidity

Z pohledu solidity lze rozdělit paměť na 4 typy [13]:

3.6.1 Storage

Jsou proměnné či data nadefinovaná na úrovni kontraktu, ne funkce. Takové proměnné jsou přístupné všem funkcím kontraktu a jsou tak sdílené a persistentní (jsou uloženy přímo na blockchainu). Jsou viditelné ve všech funkcích kontraktu. Manipulace s touto pamětí je z pohledu paliva nejdražší. Existuje klíčové slovo *storage*, které se používá, pokud chceme ve funkci vytvořit ukazatel na nějaká data ve *storage*. Dále může být toto klíčové slovo použito v parametrech privátních funkcí.

3.6.2 Memory

V paměti *memory* leží lokální nadefinované pole o fixní délce nebo jiné neprimitivní datové typy (struct…). Po skončení vykonávání funkce je tato paměť vymazána. Manipulace s touto pamětí je mnohem levnější než *storage*. Při definování lokálního pole fixní délky nebo nějakého jiného neprimitivního datového typu se musí použít klíčové slovo *memory*. Pokud se toto klíčové slovo použije s referencí na pole ve *storage*, toto pole se překopíruje do paměti. To může být výhodné z hlediska paliva, pokud je potřeba číst celé pole ve *storage* vícekrát než jednou.

3.6.3 Calldata

Je podobná paměti *memory*, ale lze z ní pouze číst. Typicky se používá pro argumenty funkce dynamického typu (struct, array,...), které není potřeba měnit v průběhu vykonávání funkce. Je levnější než paměť *memory*.

3.6.4 Zásobník

V zásobníku jsou uloženy lokální proměnné ve funkcích. Manipulace se zásobníkem z pohledu paliva je velmi levná. Zásobníkový ukazatel vidí pouze na prvních 16 slotů (proměnných) v zásobníku. Pokud se ve funkci nadefinuje např. 17 lokálních proměnných a poté se použije ta první , kompilace kódu selže, jelikož v zásobníku nelze jít tak hluboko (pod 16 slotů).

3.7 Mapping

Mapping je v solidity něco jako hash mapa nebo hash tabulka. Hodnoty jsou uloženy na nějaké adrese ve *storage* paměti. Tyto adresy se získají keccak-256 hashem ze spojení klíče a slotem mappingu. Mapping se typicky používá jako asociace, například jako klíč je adresa a jako hodnota je konto adresy. Jelikož mapping operuje na *storage* paměti, nelze ho použít pro jiný typ paměti (tedy *memory* a *calldata*). Mapping může být vnořený a může být parametrem privátní nebo vnitřní funkce [14].

3.8 Pole

Pole v solidity mohou být jak statická, tak i dynamická. Mohou být vnořená, nicméně typ vnořených polí musí být vždy ten samý. Maximální počet vnořených polí je 15 (z důvodu již zmíněného omezení ukazatele zásobníku). Dynamická pole oproti statickým mohou být jen ve *storage* paměti, mohou měnit délku a jsou opatřena funkcí *push*, která přidá prvek na konec pole, a funkcí *pop,* která odstraní poslední prvek (a nemají například *shift* nebo *unshift*). Nejčastější způsob odstranění prvku z dynamického pole je zaměnění pozice s posledním prvkem a následném zavolání *pop (*to lze však použít jen pro pole, ve kterém nezáleží na pořadí) [15].

3.9 Struct

Jde o datový typ složený z několika proměnných (jako struct v programovacím jazyce C). Označuje se jako „objekt bez funkcionality". Jednotlivé proměnné mohou být jakéhokoliv typu kromě struct samotné. Není tedy povolen rekurzivní struct, jelikož by velikost takové rekurzivní struct byla nekonečná. Struct může být použit s poli (pole struct) nebo mapping (nějaký klíč odkazující na struct) [16].

3.10 Posílání etheru

Posílání etheru z EOA adresy na EOA adresu je přímočaré, ale není tomu tak v případě účasti kontraktů. Aby kontrakt mohl přijmout ether, musí obsahovat speciální

funkci, která je vyvolána při přijetí Etherea (těmi jsou funkce *fallback* a *receive*). V případě posílání etheru při volání nějaké funkce kontraktu musí mít funkce *payable* klíčové slovo. Obecně se ether posílá přes hodnotu *value* v transakci (hodnota transakce). Z kontraktu lze ether posílat 3 způsoby [17]:

Funkce transfer – Limit paliva je 2300 a v případě neúspěchu selže celá transakce.

Funkce send – Limit paliva je 2300 a na rozdíl od funkce transfer neselže celá transakce v případě neúspěchu, ale vrátí *boolean* (úspěch / neúspěch).

Funkce call – Implicitně žádný limit paliva nemá a touto funkcí lze také vyvolat jakoukoliv funkci kontraktu příjemce. Vrací *boolean* (úspěch / neúspěch) a vrácená data vyvolané funkce.

3.11 Speciální proměnné

V solidity jsou speciální proměnné, které lze volat odkudkoli a jež slouží převážně k získání informací o aktuální transakci nebo bloku. Mezi základní se řadí [18]:

tx.origin – Vrací adresu odesílatele transakce. Tímto může být pouze EOA adresa, jelikož kontrakt transakce vyvolat neumí (ale umí vyvolat funkce jiných kontraktů).

msg.sender – Na rozdíl od *tx.origin* vrací adresu, která vyvolala funkci. Tímto může být i kontrakt (když kontrakt interaguje s jiným kontraktem přes veřejné funkce).

block.timestamp – Vrací aktuální timestamp (čas) bloku.

block.number – Vrací číslo aktuálního bloku.

block.coinbase – Vrací adresu aktuálního těžaře. Může být užitečné například k odměnění těžaře za rychlejší zpracování (vytěžení) transakce.

3.12 ERC20 Standard

Kdokoliv dříve mohl vytvořit svůj vlastní token pomocí kontraktů. Nebyl ale zavedený žádný pevný standard nebo rozhraní, které tokeny musí implementovat. ERC20 standard byl vytvořen jako reakce na tento problém pro zaručení kompatibility různých tokenů při používání různých kontraktů na Ethereu.

Jedná se o zaměnitelné tokeny, tedy 1 token (stejného kontraktu) má stejné vlastnosti a hodnotu jako jiný token (stejného kontraktu). Pokud nějaká peněženka vlastní více tokenů, jejich množství lze ukládat pod 1 proměnnou (protože jsou zaměnitelné).

Rozdíl mezi ERC20 tokeny a nativním tokenem (Ether) je ten, že nativní token není kontrakt (nemá tedy ani adresu), má jiný způsob odesílání (posílá se přes hodnotu *value* v transakcích) a nikdo jiný nemůže manipulovat s nativním tokenem z cizí peněženky.

ERC20 token umožňuje kontraktům nebo jiným peněženkám vyjmout ERC20 tokeny z adresy jiné peněženky. Používá se k tomu *approve* mechanismus. Jakákoliv adresa může specifikovat (posláním transakce *approve*), jaká jiná adresa a s jakým množstvím může tato adresa manipulovat. Tato funkcionalita je kritická pro používání tokenů s interakcemi s nějakými kontrakty (například decentralizované směnárny). Kdyby tato funkcionalita nebyla, pro interakci s kontraktem, který manipuluje s tokeny, by bylo potřeba poslat 2 transakce místo jedné.

ERC20 token je kontrakt, který musí implementovat následující funkce (jinak nemusí být kompatibilní s ostatními kontrakty, které očekávají ERC20 rozhraní) [19].

totalSupply() – Vrátí celkové množství tokenů, které všechny adresy dohromady vlastní.

balanceOf(address) – Vrátí, kolik zadaná adresa vlastní tokenů.

allowance(address,address) – Vrátí, kolik tokenů může druhá zadaná adresa přesunout od první adresy.

decimals() – Jelikož solidity operuje pouze na celých číslech, musí se nějak dát najevo například uživatelskému rozhraní, jak má reprezentovat množství tokenů v desetinné

podobě. Nejpoužívanější a výchozí je 18 desetinných míst (tedy množství 1e17 by UI zobrazilo jako 0.1 tokenu).

transfer(address, uint) – Odesílatel transakce pošle tokeny adrese zadané jako parametr.

approve(address, uint) – Odesílatel transakce umožní manipulovat s tokeny adrese zadané jako parametr.

transferFrom(address, address, uint) – Přepošlou se tokeny z první adresy na druhou. Volající adresa této funkce to musí mít povolené (přes funkci *approve*).

Kromě výše zmíněných funkcí mohou existovat různá rozšíření, se kterými se již implicitně nepředpokládá, že token implementuje. Jedná se například o funkci *burn*, která adrese odebere tokeny a sníží tak celkové množství tokenů, nebo se může jednat o funkci *mint*, která je opakem *burn*.

3.13 Hardhat

Hardhat je prostředí pro kompilování, testování a nasazování kontraktů v solidity [20]. Je snadno rozšiřitelný, skládá se z několika pluginů, které lze snadno upravit nebo napsat svůj vlastní plugin (existuje také například plugin pro zobrazení použitého paliva na jednotlivé funkce při spuštění testů).

Hardhat poskytuje svou vlastní síť Hardhat Network, která vytváří blok s každou novou transakcí. Jedná se o lokální EVM síť, lze na ní tedy simulovat jakékoliv operace, jako kdyby se jednalo o hlavní síť Ethereum (například spuštění unit a funkcionálních testů). Pokud je nutné komunikovat s nějakým kontraktem, který je již na hlavní síti, může se použít tzv. fork, kdy Hardhat převede svou síť do stavu hlavní sítě a pokračuje lokálně v tomto stavu.

Jednou z dalších velmi užitečných funkcí, kterou Hardhat Network poskytuje, je console.log(), kterou lze použít přímo v kódu solidity a snadno tak ladit kód (žádná vestavěná metoda pro logování v solidity není).

3.13.1 Mocha

Mocha je populární framework v Javascriptu pro psaní automatických testů [21]. Je založen na dvou základních funkcích *describe* a *it*. Describe se používá k seskupení testů a dává jim kontext. *It* se používá pro definování jednotlivých testů a jejich očekávaných výsledků. Využívá se spolu s Hardhat k testování kontraktů.

4 Bezpečnost a útoky

V této kapitole budou popsány časté bezpečnostní hrozby v kontraktech [22].

4.1 Reentrancy

Jedná se o opakované spouštění funkce z té samé funkce. To samo o sobě nemusí být škodlivé, ale záleží na pořadí vykonaných akcí. Aby *reentrancy* nastalo, funkce typicky musí obsahovat nějakou cestu, jak zavolat potenciálně škodlivý kód. U některých funkcí je žádoucí, aby mohl kdokoliv zavolat nějaký vlastní callback (a tento callback by mohl volat zpátky tu funkci, ze které byl zavolán). Funkce *call*, která posílá ether, otevírá tuto možnost útoku, jelikož kromě posílání etheru může zavolat také libovolnou funkci (může se ošetřit explicitním nastavením, kolik paliva lze na *call* použít).

V případě, že se funkce týká jeden ze zmíněných případů, musí se funkce buď zabezpečit *non-reentrant* modifikátorem (viz 3.5), nebo změnit pořadí řádek kódu, aby ani nemohla nastat nějaká chyba. Typický příklad kontraktu obětí *reentrant* útoku:

Kontrakt, který distribuuje určité množství etherea určitým adresám. Kontrakt tak na začátku obsahuje všechen ether pro všechny adresy. Funkce pro vybrání Etheru:

1) Pošli ether volajícímu funkce (použitím *call* - volající může být jiný kontrakt, který zavolá znovu tuto funkci na vybrání Etheru a může tak vybrat Ether určený pro jiné adresy).
2) Sniž množství, které si volající může vybrat (chyba – snížit množství se má jako první, nebo funkci ošetřit *non-reentrant* modifikátorem).

4.2 DoS (Denial of Service)

Typickým příkladem je posílání etherea neznámé adrese. Adresa může být kontrakt a může tak přijetí etherea zamítnout. V případě, že se ether posílá několika adresám v jednom *for* cyklu a jedna to nepovolí, transakce selže a žádná adresa ether nepřijme.

Dalším typem je DoS za použití limitu paliva bloku. Jelikož v jednom bloku se může spotřebovat jen určité množství paliva, jakoukoliv transakci, která potřebuje více paliva na zpracování, nebude možné zpracovat. Typicky se jedná o *for* cyklus vykonávající náročný kód, který se kvůli špatné implementaci nedá rozložit na více volání.

4.3 TOD (Transaction-ordering dependence)

Jelikož jsou všechny transakce veřejně viditelné před jejich zpracováním v tzv. *mempoolu*, je možné reagovat na nějakou transakci posláním jiné transakce za účelem zisku nebo škody. Jako příklad lze uvést transakci, která po zpracování nakoupí velké množství určitého tokenu. Jako reakci na tuto transakci nějaký škůdce pošle tu samou transakci, jenom s větší cenou za palivo, aby byla zpracována před tou první. Za předpokladu, že druhá transakce bude zpracována před první a obě budou zpracovány, škůdce může nakoupené tokeny ihned prodat se ziskem. Tento typ útoku se také nazývá *sandwich-attack* a typicky jako ochrana na decentralizovaných směnárnách se používá tzv. *slippage* (určuje maximální cenu, na které je možné transakci ještě provést).

5 Front-end

5.1 Infura

Infura je sada nástrojů pro usnadnění vývoje aplikací (kontraktů) na Ethereu. Odstraňuje bariéru, kdy by každý vývojář nebo uživatel musel mít své vlastní Ethereum *full node*. Správa vlastního *node* by pro běžné uživatele byla příliš nákladná a nespolehlivá). Infura spravuje několik svých *node* a dává veřejnosti možnost s nimi komunikovat přes api klíče [23].

5.2 Ethers

Ethers je javascriptová knihovna pro interakci s EVM-kompatibilními sítěmi. Může být také použita s typescriptem [24].

Připojení k síti zajišťuje abstraktní modul *ethers.provider*. Abstraktní, protože je více možností se připojit k nějakému nodu. Například se lze připojit k vlastnímu nodu nebo nejběžněji k Infura. Přes tento modul se posílají podepsané transakce nebo získávají data z blockchainu (zajišťuje tedy veškerou komunikaci s frontendem a sítí Etherea). Nevyžaduje privátní klíč, sám o sobě tedy poskytuje pouze operace pro čtení.

Pro komunikaci bude také potřeba mít nějakou adresu a její privátní klíč. V *ethers* existuje modul *ethers.wallet*, který vytváří objekty s rozhraním peněženky. Pro vytvoření je třeba privátní klíč a provider. V tomto modulu lze také náhodně vygenerovat nějakou adresu (a tedy i privátní klíč).

Pro komunikování s nějakým kontraktem slouží modul *ethers.contract*. Pro instanci třídy kontraktu se potřebuje adresa kontraktu, jeho ABI a instance *ethers.wallet* (připojení k síti s privátním klíčem). Tato instance poté poskytne funkce, které lze na kontraktu spustit (v této třídě se jmenují stejně jako v předaném ABI). Za vývojáře vytváří transakce připravené pro komunikaci s EVM (tedy v bajtové podobě) a používá podepisovací algoritmus. Přes tento modul lze také nasazovat nové kontrakty.

5.2.1 Web3

Web3 je javascriptová knihovna pro interakci s EVM-kompatibilními sítěmi. Je to plnohodnotná alternativa ke knihovně *ethers*, liší se pouze ve struktuře a pojmenování metod.

5.3 Next.js

Next.js framework je nadstavba knihovny React. Doplňuje React o nějaké další funkcionality. V next.js lze spojit renderování html na klientovi a na serveru. Díky SSR (server side rendering) je tento framework vhodný pro SEO (search engine optimization). Next.js dále obsahuje vestavěný routing (která stránka se má vyrenderovat podle URL). Cesty routingu jsou pevně dány strukturou a názvy adresářů a souborů [25].

React je deklarativní a flexibilní javascriptová knihovna pro tvorbu UI (uživatelských rozhraní). UI jako komplexní celek lze rozložit do jednotlivých komponent, které zapouzdřují logiku jednotlivých částí. Existují dva typy komponent – funkcionální a třídové (komponenta může být buď funkce, nebo třída). Výstup, který má komponenta vykreslit, se určuje v metodě *render*.

6 Burzy

Jedná se o jakékoliv místo, které umožňuje směnit jedno finanční aktivum za druhé. Dochází zde k setkání kupců a prodejců, kteří mohou provést nějaký obchod za stanovených podmínek. Může se jednat o devizové, akciové či kryptoměnové burzy.

6.1 Orderbook

Orderbook je základem centralizovaných burz, ale je používána i v decentralizovaných směnárnách. Každý trh (například trh s etherem proti dolaru ETH/USD) má svůj orderbook, ve kterém jsou uloženy všechny nákupní a prodejní objednávky (ale zpravidla se orderbook zobrazuje pouze s limitními objednávkami). Objednávky jsou zde seřazeny podle ceny, na kterých jsou objednávky nastaveny, a zobrazuje se také množství, které objednávky mají nakoupit/prodat. Aktuální cena trhu se zpravidla stanovuje na polovinu součtu nejvyšší ceny nákupní limitní objednávky a nejnižší ceny prodejní limitní objednávky (v tomto základním stanovování ceny není uvažována velikost objednávky) [26].

6.2 Objednávka

Každá objednávka v sobě nese 4 základní parametry [27]:

Typ objednávky – Zda se jedná o limitní, stop a nebo market objednávku (viz 6.3).

Strana objednávky – Zda se jedná o nákupní, nebo prodejní objednávku.

Cena objednávky – Tržní cena, na které se má objednávka aktivovat.

Velikost objednávky – Jaké množství nákupního/prodejního aktiva se má směnit za druhé.

6.3 Základní typy objednávek

6.3.1 Market

Objednávka je instantně vyplněna (zpracována) na aktuální nejlepší tržní ceně (negarantuje zpracování na nějaké konkrétní ceně).

6.3.2 Limit

Jestliže se jedná o nákupní objednávku, jde o takovou, která má cenu níže, než je aktuální tržní cena, a čeká na vyplnění, nebo na zrušení objednávky. Objednávka slouží jako protistrana market objednávce a vyplní se, jakmile se vyplní všechny ostatní limitní objednávky s vyšší cenou (objednávka je na řadě, bude vyplněna s další market objednávkou). Jestliže se jedná o prodejní objednávku, jde o objednávku, která má cenu výše, než je aktuální tržní cena.

6.3.3 Stop

Jestliže se jedná o nákupní objednávku, jde o objednávku, která má cenu výše, než je aktuální tržní cena, a čeká na vyplnění nebo na zrušení objednávky. Objednávka se změní v market objednávku, jakmile se tržní cena dostane nad cenu této stop objednávky. Není garantováno, že se objednávka vyplní na této ceně (nemusí být dostatečná likvidita). Jestliže se jedná o prodejní objednávku, jde o objednávku, která má cenu níže, než je aktuální tržní cena.

6.3.4 Typy objednávek podle času platnosti

Objednávky lze dále rozdělit do skupin podle času platnosti (tzv. *time-in-force*) [28]. Tento typ určuje, jak dlouho je objednávka aktivní nebo jak má být s objednávkou naloženo při jejím zpracování.

GTC (good-till-canceled) – Objednávka je aktivní, dokud není zrušena tím, kdo ji zadával.

GOT (good-til-time) – Objednávka je aktivní do určitého času, poté je automaticky zrušena.

FOK (fill-or-kill) – Objednávka musí být na zadané ceně vyplněna celá najednou (nesmí být vyplněna jen parciálně). Pokud by k tomu nedošlo, dojde k jejímu okamžitému zrušení .

IOC (immediate-or-cancel) – Objednávka musí být na zadané ceně vyplněna okamžitě. Oproti FOK povoluje parciální vyplnění objednávky.

6.4 Likvidita

Likvidita obecně určuje, jak rychle je možné nakoupit, nebo prodat za peníze nějaké aktivum bez velkého ovlivnění aktuální ceny aktiva [29]. Nejlikvidnějším aktivem jsou peníze samy. Příklad méně likvidního aktiva může být například nemovitost. Likviditu v trhu lze vyčíst z jeho orderbooku. Čím více a čím větší velikosti objednávek tam jsou, tím více je aktivum likvidní (mohu prodat/nakoupit hodně bez značného ovlivnění ceny).

6.5 Volatilita

Za volatility se považuje, jak rychle a jak moc se mění cena v trhu [30]. Například za hodně volatilní trh se může považovat takový, který by během dne spadl o 90 %. Volatilita souvisí s likviditou – čím menší likvidita, tím větší volatilita.

6.6 Margin a Spot obchodování

Směnárny (primárně centralizované) se dělí na dva hlavní typy obchodování [31]. *Spot* obchodování znamená, že pro uskutečnění obchodu musí mít zadavatel objednávky přímo k dispozici celé množství objednávky (mít na obchodním účtu celé množství). *Margin* obchodování je obchodování s pákou, to znamená, že obchodník si může například od dané směnárny vypůjčit určité množství prostředků na uskutečnění obchodu.

Obvykle se jedná maximálně o desetinásobky toho, co má obchodník na účtu. Hlavní výhoda margin obchodování je jasná, lze dosáhnout vyššího zhodnocení s menším kapitálem.

6.7 Decentralizované směnárny

Je místo, kde se dají směňovat aktiva bez centrální autority, jež by zpracovávala objednávky a spravovala samotná aktiva [32]. Aktiva spravují buď obchodníci sami, nebo kontrakty. Centrální autority zde zastupují kontrakty. Kontrakty mají za úkol umožnit obchod za předem stanovených podmínek. Decentralizované směnárny mohou fungovat na orderbooku stejně jako centralizované, nicméně mnohem rozšířenější je tzv. automatizovaný tvůrce trhu (automated market maker, AMM). AMM automaticky spravuje aktiva a likviditu, díky níž je možné uskutečnit obchody (nahrazuje limitní objednávky v orderbooku). Fungování jednoho konkrétního AMM je uvedeno v kapitole 7.1.

7 Analýza existujících směnáren

Konkrétní decentralizované směnárny, které budou předmětem analýzy, byly vybrány podle známé stránky ohledně kryptoměn – CoinMarketCap. Směnárny se zde dělí na 3 hlavní typy – swap, orderbook a aggregator. Drtivá většina směnáren je typu *swap*, ze kterých se vybral hlavní zástupce Uniswap V2 (viz 7.1). Většina ostatních směnáren typu *swap* je pouhou kopií nebo malou či větší modifikací jmenovaného zástupce. Dále se vybral jeden zástupce typu *orderbook* dYdX (viz 7.2). Typ *aggregator* se v této analýze vyskytovat nebude, jelikož jak z názvů typu vypovídá, jde jen o směnárny, kde se jen využívá ostatních směnáren k nalezení nejlepší ceny či nejmenších nákladů na obchodování. Za takovou nejznámější se považuje OneInch.

7.1 Uniswap V2

V Uniswapu od verze 2 je možné obchodovat tokeny ERC20 oproti ostatním ERC20 tokenům (ve verzi 1 bylo možné obchodovat jen ERC20 oproti ETH) a navíc se už zde s nativním tokenem ETH téměř vůbec nepracuje, jelikož je nahrazen tokenem nazývaným WETH, což je jednoduchý ERC2 kontrakt, do kterého se vloží ETH a vrátí ERC20 token WETH – výsledkem je kompatibilnost s ERC20 tokeny, není tedy nutné řešit implementaci pro oba typy tokenů, ale jen pro jeden (ERC20). Základní myšlenkou swapu je, že do nějakého kontraktu pošlu nějaké množství nějakého tokenu a za to dostanu jiný token (ve stejné transakci).

7.1.1 Páry

Směňování tokenů se na nízké úrovni děje v tzv. párech Uniswapu. Pár je složen ze dvou ERC20 tokenů, například s názvy WETH a USDC (USDC je ERC20 token, který udržuje cenu 1 dolaru). Pár je sám o sobě kontrakt, který je zároveň ERC20 kompatibilní (sám o sobě je ERC20 token). Každý pár přímo vlastní nějaké množství tokenů WETH a USDC, které páru poskytují tzv. poskytovatelé likvidity. Tyto tokeny poté umožňují

směňování jednoho tokenu za druhý. Čím více tokenů pár vlastní, tím větší likvidita a tím větší objem, který je možné zobchodovat s malým ovlivněním ceny.

Poskytovatelem likvidity se může stát každý výměnou tokenů za nějaké množství LP (liquidity provider) tokenů. LP token je přímo vydávaný párem (jelikož sám o sobě je ERC20 token) a určuje, jaký podíl mají jednotliví poskytovatelé likvidity ve vložených tokenech (WETH, USDC). Pokud někdo vlastní např. 50 % všech LP tokenů nějakého páru, znamená to, že 50 % tokenů, které pár vlastní (WETH, USDC), patří právě jemu a může si je kdykoliv z páru vybrat (a tím snížit likviditu v páru o 50 %).

Uniswap určuje koupené (vyměněné) množství tokenů pomocí tzv. *Constant product formula*:

$$k = x \cdot y \tag{7.1}$$

Kde k je konstanta, x a y je množství tokenů X (např. WETH) a Y (např. USDC), které pár vlastní. Při swapu tokenů tedy musí k zůstat konstantou. To znamená, že pokud do páru vložím (chci vyměnit) nějaké množství tokenu X, množství tokenu Y v páru se musí snížit (poslat tomu, kdo začal swap) tak, aby k zůstalo konstantou. Tedy pokud chci směnit Δx tokenů za Δy, rovnice bude vypadat takto:

$$\begin{aligned} k &= (x + \Delta x) \cdot (y - \Delta y). \\ x \cdot y &= (x + \Delta x) \cdot (y - \Delta y). \end{aligned} \tag{7.2}$$

Rovnice množství tokenu Y, které pár odešle za přijaté X tokeny (za předpokladu nulových poplatků):

$$\Delta y = y - \left(\frac{x}{(x+\Delta x) \cdot y}\right) \tag{7.3}$$

Cena tokenů se tedy určuje jako poměr množství tokenů, jimiž pár disponuje (množství tokenů, které patří adrese páru).

$$p_x = \frac{y}{x}$$

$$p_y = \frac{x}{y}$$

(7.4)

Ale pozor, *k* není konstantou v případě poskytování (přidávání likvidity). V takovém případě se zvyšuje jak *k*, tak *x* i *y*, kdy poměr *x* a *y* se zachovává. To znamená, že při poskytování likvidity nelze poskytnout libovolné množství tokenu X a libovolné množství tokenu Y, ale musí to být ve stejném poměru množství tokenů, který pár aktuálně vlastní (tedy na aktuální ceně). Výjimkou je případ, kdy do páru přidává likviditu někdo první. Ten si určuje cenu sám, může tedy přidat libovolný poměr tokenů.

7.1.2 Funkce páru

Swap (uint amount0Out, uint amount1Out, address to, bytes calldata data) – Funkce na nízké úrovni, která provádí výměnu tokenů. Je implementována velmi flexibilně a efektivně. Parametry neobsahují množství, která se mají páru pro výměnu poslat. Důvodem je, že funkce očekává, že tyto tokeny obdržela již před zavoláním této funkce a množství lze snadno zjistit (podle předchozí a aktuální *balance*). Jeden z parametrů *amount0Out* a *amount1Out* bude vždy nulový, a to ten, který token do páru posíláme. Na první pohled se může zdát, že si vlastně může každý určit, kolik tokenů chce dostat (pomocí parametrů amountOut). Není tomu ale tak, jelikož ve funkci dochází ke kontrole *Constant product formula* rovnice.

Parametr *to* určuje, na jakou adresu se mají tokeny poslat. To přináší flexibilitu, kdy je možné provést například více výměn mezi více páry v jedné transakci – tokeny se nepošlou hned tomu, kdo výměnu začal, ale pošlou se na jiný kontrakt, který je poté schopen provést další výměny (např. Uniswap Router).

Pokud funkci volá nějaký kontrakt, je možné pro větší flexibilitu zavolat libovolnou funkci (tzv. callback) volajícího kontraktu před dokončením výměny. K tomu slouží poslední parametr *data*, který obsahuje signaturu funkce spolu s parametry, která se má zavolat. Pokud se žádná funkce volat nechce, parametr data zůstane jednoduše prázdný. Callback funkce se zavolá po poslání tokenů z páru. Spolu s možností zadat jakýkoliv *amountOut* vzniká možnost provedení tzv. *Flash Swapu.*

Flash Swap – Kdokoliv si může vypůjčit tolik tokenů, kolik pár vlastní, ale musí je před dokončením volání funkce vrátit. Každý si tedy může zvolit jakýkoliv *amountOut* a vrátit je (poslat je zpět páru) v callback funkci.

Mint (address to) – Funkce na nízké úrovni, která přidává likviditu. Parametr má jen jeden, tím je adresa *to,* na kterou se pošlou nově vzniklé LP tokeny. Jelikož funkce nemá žádné jiné parametry, funkce očekává, že jsou tokeny na přidání likvidity poslány již předem. Posílat je předem může na první pohled znít riskantně, nicméně je třeba si uvědomit, že transakce probíhají atomicky, a pokud se ve stejné transakci pošlou tokeny a zavolá funkce *mint*, o tokeny nelze přijít.

Burn (address to) – Funkce na nízké úrovni, která odstraňuje likviditu. Parametr má jen jeden, a tím je adresa *to,* na kterou se pošlou tokeny A a B po odstranění likvidity. Pro odstranění likvidity funkce již předpokládá, že LP tokeny jsou poslány předem.

7.1.3 Poplatky

V praxi se při provádění výměny odvádí poplatek poskytovatelům likvidity (aby z toho něco měli, jelikož se vystavují riziku (viz 7.1.4). Jedná se o 0,3% poplatek, který mírně ovlivní rovnici pro vypočtení tokenů po výměně. Zvyšuje se tedy i k (o poplatek), zvyšuje se likvidita, kterou poskytovatelé vlastní (jelikož množství LP tokenů se při výměně zachovává).

7.1.4 Riziko poskytování likvidity

Poskytovatelé likvidity se vystavují riziku zvanému *impermanent loss*. To vzniká, když se cena poskytnutých tokenů mění. Čím více se cena změní od doby, kdy někdo poskytl likviditu, tím větší vzniká ztráta. Zatímco cena jednoho tokenu roste, poskytovatel tento token ztrácí a získává druhý, jehož cena klesá. Tyto ztráty mají vynahradit poplatky za výměny (ale zpravidla poplatky nemusí tyto ztráty pokrýt).

7.1.5 Factory

Páry se vytváří v Uniswap továrně. Adresy páru jsou vypočteny deterministicky, tedy ze dvou adres ERC20 tokenů ve stejném pořadí se vždy vypočítá stejná adresa. To je možné pomocí speciálního operačního kódu *CREATE2* (který vytváří kontrakty s deterministickou adresou podle vstupu). To také zajišťuje, že pro stejné tokeny bude existovat vždy jen jeden pár, nebude jich nekonečně mnoho (bylo by nepříjemné, jelikož by byla likvidita stejných párů tokenů rozložena do několika párů atd.).

7.1.6 Router

Jelikož funkce kontraktu páru jsou poměrně na nízké úrovni, byl vytvořen také kontrakt na vyšší úrovni za účelem snadné interakce s páry. Jedná se o kontrakt Uniswap Router, přes který jde většina transakcí. Každý si však může vytvořit svůj vlastní router pro interakci s páry (není nutné ho používat, např. ani nejde používat pro *Flash Swapy*). Mezi hlavní funkce se řadí možnost provést směny ve více párech v jedné transakci, vypočítává přesné množství, za kolik se smění nějaké množství tokenů a umožňuje snadný převod z ETH nativního tokenu na WETH.

7.1.7 Funkce routeru

addLiquidity (address tokenA, address tokenB, uint amountADesired, uint amountBDesired, uint amountAMin, uint amountBMin) – Funkce přidá likviditu do páru s tokeny A a B (pokud ještě neexistuje, automaticky se vytvoří). Nejprve se vypočítá přesné (optimální) množství (podle aktuální ceny), které je páru potřeba poslat. Pokud je

vypočítaná hodnota menší než předaná minimální hodnota v parametrech (5. a 6. parametr), transakce selže. Poté funkce přepošle optimální množství tokenů od odesílatele do kontraktu páru a zavolá funkci *mint* s parametrem *to* adresu odesílatele (odesílatel transakce dostane LP tokeny).

swapExactTokensForTokens – (uint amountIn, uint amountOutMin, address[] calldata path, address to, uint deadline) – Funkce nejdříve vypočítá, jak velké množství druhého tokenu se dostane za *amountIn* množství. Pokud je vypočítaná hodnota menší než *amountOutMin*, transakce selže. Parametr *amountOutMin* se může na první pohled jevit jako zbytečný, ale je velmi důležitý a používá se jako ochrana proti velkým změnám cen v intervalu od poslání transakce po zpracování (mezitím se může cena drasticky změnit, ať už přirozeně, nebo za účelem provedení útoku). Parametr *path* v sobě nese adresy tokenů, které se mají směnit. Musí být minimálně dva, ale může jich být více – podpora více swapů v různých párech v jedné transakci.

swapTokensForExactTokens – (uint amountOut, uint amountInMax, address[] calldata path, address to, uint deadline) – Funkce funguje podobně jako předchozí s tím rozdílem, že podporuje myšlenku obdržení přesného množství tokenů po swapu (v předchozí není možné přesně určit).

swapExactETHForTokens (uint amountOutMin, address[] calldata path, address to, uint deadline) – Funguje stejně jako *swapExactTokensForTokens* s tím rozdílem, že odesílatel posílá ETH (hodnota transakce je nenulová) a ETH se před swapem zabalí do WETH (jelikož páry neumí pracovat s ETH).

7.1.8 Kopie, alternativy Uniswapu

Na základě Uniswapu vzniklo nespočetně mnoho dalších směnáren (např. Pancakeswap, Sushiswap, ...), které jsou pouhou kopií nebo se snaží vyřešit nějaký problém Uniswapu nebo ukrást uživatele Uniswapu (dát uživatelům lepší podmínky např. při poskytování likvidity). Často se také snaží modifikovat *Constant product formula* za účelem zlepšení

efektivity páru (kdy pár má sice hodně tokenů, ale s většinou z nich se nemanipuluje a leží tam jakoby zbytečně). Modifikovaný vzorec může vypadat např. takto:

$$k = x^2 \cdot y^2$$
$$k = \sqrt{x} \cdot \sqrt{y}$$
$$k = x \cdot y \cdot z \tag{7.5}$$

7.2 DyDx

DyDx má v době psaní největší tržní objemy v okruhu decentralizovaných směnáren s orderbookem. Typ obchodování je zde pouze *margin*. To znamená, že se musí půjčit prostředky k uskutečnění obchodu. Orderbook zde však není zapsaný na blockchainu, používají se kryptograficky podepsané zprávy s informacemi o objednávkách, které tvoří orderbook. Tyto zprávy jsou uložené v centrální databázi DyDx.

7.2.1 Protokol margin obchodování

Obchodník si půjčuje nějaký token a použije ho k nákupu či prodeji jiného tokenu. Půjčené tokeny je nutné zpět vrátit půjčovateli s úrokem. Toto má dvě hlavní použití. Jedním je, že lze token prodat, ačkoliv ho obchodník nevlastní (tedy půjčí si od půjčovatelů). Druhým je, že může koupit token s větší kupní sílou a mít tak potenciální větší zisky, jichž by pouze vlastním kapitálem nedosáhl. Vystavuje se však také větším potenciálním ztrátám (všechny tokeny musí být půjčovateli vráceny).

7.2.2 Kontrakty

K obchodování s marginem jsou použity tři kontrakty:

Proxy kontrakt slouží pro přenos tokenů od obchodníků.

Margin kontrakt obsahuje veškerou obchodní logiku. Jsou v něm uloženy všechny otevřené pozice (obchody).

Vault kontrakt drží všechny tokeny zamčené v pozicích. Má funkce určené přímo pro *margin* kontrakt.

7.2.3 Zpráva nabídky půjčky

Pro uskutečnění *margin* obchodu je potřeba někdo, kdo je ochoten půjčit svoje tokeny. Půjčovatel kryptograficky podepíše určitou zprávu, která může být následně přenášena bez blockchainu. Zpráva má následující povinné informace:

- owedToken – adresa tokenu, který chce půjčit.
- heldToken – adresa tokenu, ve kterém jsou drženy obchodní pozice po půjčení.
- payer – adresa toho, kdo poskytuje tokeny na půjčku.
- signer – adresa, která tuto zprávu podepisuje.
- owner – adresa, která vlastní půjčku a všechny platby půjdou na tuto adresu.
- maxAmount – maximální množství tokenu, který je ochoten půjčit.
- minAmount – minimální množství tokenu, které musí být půjčeno, aby půjčka (a tedy i *margin* obchod) začala.
- minHeldToken – minimální množství *heldToken*, které musí být zamčeno v pozici po půčjení.
- interestRate – úrok za rok, který půjčovatel dostane.
- expirationTimestamp – časové razítko, do kdy lze půjčku realizovat.
- callTimeLimit – minimální množství času, za které musí být pozice uzavřena, pokud je aplikován *margin call* (viz 7.2.6) půjčovatelem.
- maxDuration – maximální doba, po kterou je půjčka platná.

7.2.4 Otevírání pozic

Pro otevření pozice musí obchodník poslat transakci (vyvolat funkci) *margin* kontraktu.
Parametry funkce jsou:

- Zpráva nabídky půjčky.
- Zpráva nákupu *owed* tokenu za *held* token.
- Adresa kontraktu *ExchangeWrapper,* který se použije se zprávou nákupu. *ExchangeWrapper* je kontrakt, který obaluje volání funkce jakékoliv již existující decentralizované směnárny, kde je možné realizovat nákupní a prodejní obchody.
- Množství *owed* tokenu, které si chce obchodník půjčit.
- Množství tokenu, které obchodník použije jako svůj vklad.
- Adresa, která bude vlastnit pozici po její otevření (typicky ten samý obchodník).

Při zpracování této transakce *margin* kontrakt postupuje následovně:

- Verifikace podpisu a vstupů zprávy nabídky půjčky.
- *Proxy* kontrakt přesune vkladové tokeny obchodníka do kontraktu *Vault*.
- Zapíše se použité množství v půjčce v *margin* kontraktu. To je nutné, aby se při příštím otevřením pozice nepřekročilo maximální množství, které chce půjčovatel půjčit.
- *Margin* kontrakt zavolá funkci v *ExchangeWrapper* kontraktu pro výměnu *owed* tokenu za *held* token. *ExchangeWrapper* verifikuje podpis a vstupy zprávy nákupu a provede výměnu tokenů.
- *Margin* kontrakt použije *Proxy* kontrakt pro přenos *held* tokenů (nakoupených za *owed*) a přesune je do *Vault* kontraktu, kde zůstanou zamčené po celou dobu trvání pozice.
- Detaily o pozici jsou uloženy v *margin* kontraktu v mappingu, kde každá pozice má svůj veřejný identifikátor, který může být využit pro interakci s pozicí ze strany obchodníka či půjčovatele.

Všechny tyto kroky se provedou atomicky a buď všechny budou úspěšné, nebo se neprovede ani jeden. Všechny tokeny jsou po úspěšném zpracování transakce zamčeny ve *Vault* kontraktu.

7.2.5 Uzavírání pozic

Obchodník může kdykoliv uzavřít jakoukoliv část pozice. Obchodník pošle transakci do *margin* kontraktu se zprávou prodejní objednávky (tj. způsob, jakým získat *owed* token za *held* token). Tato objednávka musí získat minimální množství *owed* tokenu, které je potřeba splatit půjčovateli (i s úroky).

Při zpracování této transakce *margin* kontrakt postupuje následovně:

- Spočítá celkové množství *owed* tokenu i s úroky, které je potřeba splatit půjčovateli.
- *Margin* kontrakt zavolá funkci v *ExchangeWrapper* kontraktu pro výměnu *held* tokenu za *owed* token. Po výměně je ve *Vault* kontraktu množství *owed* tokenů, které je potřeba splatit, a vkladové tokeny a profit pro obchodníka.
- Pošlou se dlužné tokeny půjčovateli.
- Pošlou se vkladové tokeny a profit obchodníkovi.
- Detaily o pozici jsou vymazány.

Po úspěšném zpracování má půjčovatel i obchodník své tokeny a ve *Vault* kontraktu není žádný nadbytek ani nedostatek tokenů.

7.2.6 Margin call

Margin call znamená, že je do určité doby nutné do pozice přidat *held* tokeny, jinak bude celá pozice zrušena. To je nutné v případě, kdy pozice je ve ztrátě. V zájmu půjčovatele je použít *margin call* v situaci, kdy cena *owed* tokenu roste oproti *held* tokenu a v pozici skoro není dostatek množství *held* tokenu, které lze směnit za *owed* tokeny pro splacení půjčky.

Půjčovatel může kontraktu *margin* poslat transakci indikující *margin call*. Tato transakce obsahuje identifikátor pozice a kolik množství *held* tokenu je nutné do pozice doplnit, aby se *margin call* přerušil. Na to má obchodník určité množství času (definováno ve zprávě nabídky). Obchodník má dvě možnosti, buď *held* tokeny doplnit,

nebo hned splatit půjčku (tedy zrušit pozici). Pokud neudělá ani jedno, celé množství *held* tokenu v pozici bude patřit půjčovateli.

7.2.7 Bezpečností rizika

Pro obchodníka je zde riziko, že může být zavolán margin call před tím, než obchodník chce obchod zrušit, a zároveň je v pozici dostatek *held* tokenů.

Pro půjčovatele je zde riziko, že cena *owed* tokenu oproti *held* tokenu vzroste tak rychle, že nebude možné provést *margin call* včas tak, aby se za *held* token koupilo celé dlužné množství *owed* tokenu. Toto riziko se může snížit vysokým minimálním vkladem a nízkým *callTimeLimit* (definované ve zprávě nabídky půjčky).

8 Návrh směnárny

Nebyla nalezena žádná decentralizovaná směnárna na bázi orderbooku, která by zároveň ukládala a zpracovávala objednávky pouze na blockchainu. Uniswap sice je pouze na blockchainu (nepotřebuje žádný centrální bod, aby fungoval), ale nemá orderbook. DyDx sice má orderbook, ale k vyplňování objednávek navíc používá jiné decentralizované směnárny. Navíc zde musí být nějaký centralizovaný bod, který umožňuje předávání DyDx zpráv (což zajišťuje api server DyDx).

Na základě těchto poznatků bylo rozhodnuto vytvoření decentralizované směnárny s orderbookem, která je decentralizovaná a leží pouze na blockchainu. To znamená, že objednávky budou uloženy na blockchainu, ne pouze v podepsaných zprávách, jako používá DyDx. Tím, že směnárna bude ležet pouze na blockchainu, se také zaručí deterministické zpracování limitních i stop objednávek. Budou mít jasně dané pořadí a pravidla pro zpracování. V porovnání se směnárnou DyDx objednávky existují formou podepsané zprávy, ale na blockchainu není dané jejich pořadí ve zpracování. Například tam může existovat nějaká stop objednávka, ale už není jasně zaručeno, že bude zpracována. V případě vytvářené směnárny bude zaručené, že žádná objednávka nebude vynechána, ale bude zpracována podle jasně daného pořadí (seřazené podle ceny a data vytvoření).

Je také potřeba vyřešit, jak efektivně orderbook ukládat, jelikož zapisování dat a jejich měnění přímo na blockchain spotřebovává nejvíce paliva. Celá směnárna bude napsána v solidity jako kontrakt. Budou k dispozici 3 základní typy objednávek – limit, stop a market objednávky. Každý typ objednávky může být buď prodejní, nebo nákupní. Limitní objednávky tvoří orderbook a poskytují tak likviditu pro market a stop objednávky. Stop objednávky čekají a stanou se market objednávkou, pokud cena trhu bude nad/pod cenou stop objednávky.

8.1 Výhody orderbooku oproti AMM

Jedna z nejpodstatnějších výhod, která je také motivací pro vytvoření této směnárny, je variabilita a flexibilita trhu. Tím je myšleno, že tvůrci trhu mohou limitní

agregovat objednávky (likviditu) na určitých (pro ně výhodných) cenových hladinách. Naopak na nějakých cenových hladinách je výhodné udržet likviditu nízkou pro snadné ovlivnění ceny.

Na rozdíl od orderbooku je AMM deterministický model a využití likvidity v něm není efektivní, je konstantně rozložená. Je možné vypočítat, kolik přesně tokenů je potřeba prodat/koupit, aby se cena trhu dostala na požadovanou úroveň.

8.2 Efektivní ukládání objednávek

K uložení objednávek (celého ordebooku) bude potřeba datová struktura, která dokáže efektivně seřadit prvky podle nějakého atributu. V tomto případě je tím atributem cena objednávky, a pokud je cena objednávek stejná, prioritní bude ta, která byla dříve vytvořena. Všechny typy objednávek budou mít tuto datovou strukturu odděleně (tedy limit a stop objednávky nebudou ve stejné datové struktuře). Market objednávky není třeba ukládat do těchto struktur, jelikož jsou aktivní ihned při jejich poslání.

Datová struktura, která by toto dokázala, je rozhodně prioritní fronta. Prioritní fronta využívá heap sort algoritmus, který má časovou komplexitu pouze O(n log(n)). Avšak nevýhodou tohoto přístupu v tomto případě je to, že manipuluje se *storage* polem (tj. zapisuje a mění v poli), což je v solidity nejdražší operace, a nebylo by realizovatelné pro velké množství objednávek.

Jedním z řešení tohoto problému je vlastní datová struktura, která byla nazvána jako prioritní list. Tato struktura je v základu spojovaný list (jednosměrný), ve kterém jsou prvky ukládány seřazeně. Ve *storage* se tedy mění vždy konstantní množství prvků (změní se konstantní počet odkazů na další prvky). Aby se vložený prvek správně zařadil, je nutné v základu projít celou datovou strukturu, což vede na lineární složitost. Ale dochází zde pouze ke čtení prvků, ne k zapisování. Jak je známo, čtení *storage* prvků je levnější než jejich změna.

Avšak to pořád nestačí, při vkládání limitních nebo stop objednávek daleko od aktuální ceny by bylo stále náročné na palivo (např. čtení 100 prvků, než se narazí na místo, kam by se nová objednávka měla zařadit). Řešením tohoto problému je zavedení operace nad prioritním listem, která umožní vložit prvek na aktuální index, a nemusí se

tak číst všechny prvky listu. Tato operace vložení má konstantní časovou složitost, jelikož nemusí číst všechny prvku seznamu a iterovat od prvního prvku, ale ví přesně, na jaký index se má nový prvek vložit. Na základě daného indexu se jednoduše najde předchozí a následující prvek, které budou použity k porovnání s vkládaným prvkem. Vkládaný prvek musí být menší/větší než předchozí a větší/menší než následující. Pokud tato kontrola projde, je bezpečné vložit prvek na zadaný index. S touto operací je možné vkládat limitní a stop objednávky daleko od aktuální ceny (např. za 1000 objednávek) s konstantní spotřebou paliva, jelikož není třeba číst všechny prvky prioritního (ale pouze 2).

8.3 Ukládání jednotlivých trhů

Trhy je třeba nejdříve vytvořit a inicializovat. Trh se vytváří na základě dvou parametrů. Prvním je adresa ERC20 tokenu, který bude *base* tokenem trhu. Druhým je adresa ERC20 tokenu, který bude *quote* tokenem trhu. Rozdíl mezi *base* a *quote* tokenem je následující. Cena trhu se udává ve formátu, kolik *quote* tokenů je třeba ke koupi jednoho *base* tokenu. Nákupní objednávky kupují *base* token za *quote* token, prodejní objednávky prodávají *base* token za *quote* token. Vytvořené trhy se uloží do jednoduché struktury pole nebo mappingu. Každý trh má 4 prioritní listy pro objednávky (nákupní limit objednávky, prodejní limit objednávky, nákupní stop objednávky a prodejní stop objednávky).

8.4 Exekuce market objednávky

Market objednávka je taková objednávka, která se vyplní ihned při jejím poslání. Slouží jako protiobjednávka limitním objednávkám. Ne vždy je ale možné ji vyplnit, jelikož v trhu nemusí být vždy dostatek limitních objednávek, které by dokázaly market objednávku celou vyplnit (ale to by se stalo pouze v málo likvidním trhu).

Exekuce market objednávky povede na lineární časovou složitost, jelikož je třeba přečíst a vyplnit tolik limitních objednávek, dokud není market objednávka vyplněna. Může se stát, že na market objednávku bude potřeba pouze jedna limitní objednávka na

vyplnění, ale také jich nemusí stačit ani 10, záleží na velikosti objednávek. Limitní objednávky jsou již seřazené v prioritním listu podle ceny a data vytvoření a vždy se bere ta, která je na vrcholu. Pokud jsou limitní objednávky vyplněny, dojde k jejich odstranění z prioritního listu v konstantním čase (i když jich bylo vyplněno 10, změní se pouze konstantní počet odkazů na prvky).

V tomto případě zápisy v prioritním listu jsou tedy v konstatním čase, ale čtení v lineárním. Je třeba dodat, že se při exekuci market objednávky nemanipuluje pouze s prioritními listy, ale musí se přeposlat tokeny na adresy, což je zápis do *storage* v lineárním čase. Konta adres ERC20 tokenů jsou uložena ve *storage mappingu* a pro každou vyplněnou objednávku se musí přesunout od tvůrce market objednávky k tvůrcům limitních objednávek. Tedy vyplnění hodně limitních objednávek povede k velké spotřebě paliva (což nemusí být tvůrce market objednávky ochoten zaplatit).

Řešením tohoto problému je zavedení poplatků za vytvoření limitní nebo stop objednávky. Poplatek, který bude poslán tvůrci market objednávky, by měl pokrýt jeho náklady na vyplnění limitních objednávek. Tedy při vytváření spolu s tokeny objednávky by se odeslalo i nějaké množství nativního tokenu etheru (kterým se také platí poplatky). Zde je třeba dbát na variabilní cenu paliva, jejíž rozptyl je poměrně velký. Cena paliva se může zdvojnásobit během sekundy (to závisí na aktuální vytíženosti sítě).

Avšak je celá řada situací, kdy může být výhodné vytvořit market objednávku, i když to nepokryje celé náklady na vyplnění limitních objednávek. Například obchod, kdy na jiné směnárně by se prodával nějaký token za vyšší cenu, než na této směnárně. V tom případě lze nakoupit na této směnárně i za cenu vysokého nákladu na vyplnění limitních objednávek, ale pak prodat za cenu vyšší na té jiné směnárně (takový obchod se nazývá *arbitráž*).

8.5 Exekuce stop objednávky

Stop objednávky jsou stejně jako limitní objednávky seřazeny podle ceny v prioritním listu. Po každé market objednávce se zkontroluje, zda cena trhu nedošla k ceně nejbližší stop objednávky. Pokud ano, tato stop objednávka se sama stane další

market objednávkou ve stejné transakci a dále je s ní nakládáno stejně jako s původní market objednávkou.

Se stop objednávkami lze mít dokonce více market objednávek v jedné transakci, což po zavedení poplatků není problém z hlediska pro tvůrce market objednávky, ale je třeba si dát pozor na gas limit bloku, aby nedošlo k trvalém zablokování trhu (bylo by tam tolik stop objednávek, že by transakci nebylo možné zpracovat v jednom bloku). To se dá ošetřit tím, že se nastaví maximální počet objednávek, které mohou být v jedné transakci vyplněny. Není tedy možné trh zablokovat tím, že by někdo cíleně vytvořil tisíce stop objednávek a způsobil tak DoS (viz 4.2).

9 Implementace v solidity

Program obsahuje dva hlavní kontrakty, OrderBlock a PriorityList. Název vytvořené decentralizované směnárny je OrderBlock, který vznikl z kombinace slov orderbook a blockchain. OrderBlock kontrakt se stará o vytváření trhů, objednávek a jejich exekuci. Kontrakt PriorityList obsahuje implementaci prioritního listu, stará se o ukládání a řazení objednávek.

9.1 Kontrakt PriorityList

9.1.1 Struct objekty

Kontrakt obsahuje jeden hlavní objekt Data. Obsahuje 2 hodnoty – *value* a *next*. Z pole těchto objektů se skládá list. *Next* hodnota odkazuje, na kterém indexu leží následující prvek. *Value* jednoduše uchovává hodnotu prvku. V OrderBlock kontextu *value* představuje id objednávek. První dva prvky listu jsou rezervované pro speciální účely. *Value* prvního prvku je index, na kterém leží první prvek (seřazený). *Value* druhého prvku je index, na kterém leží volné místo pro nový prvek.

9.1.2 Funkce

compare – Funkce porovná 2 předané hodnoty a určí tak, jak mají být tyto hodnoty seřazené. Funkce volá implementaci compare funkce kontraktu (OrderBlock), která prioritní list používá. Tam dochází k porovnání cen a dat vytvoření objednávek.

init – Tato funkce musí být zavolána ještě před použitím prioritního listu. Vytvoří pole listu o předané délce. Toto pole se dále rozšiřuje, pokud není místo na další prvky.

insert – Funkce vloží novou hodnotu do listu. Funkce nejprve vloží hodnotu na první volný prvek a nastaví nový první volný prvek (následník nově obsazeného prvku). Poté funkce řeší, kam zařadit nově přidanou hodnotu. Iteruje se od prvního prvku a volá se funkce compare, dokud nevrátí *true*. To znamená, že se v cyklu narazilo na místo, kam se

má prvek zařadit, a nastaví se prvkům odkazy na prvky. Čím menší je priorita prvku, tím více iterací, dokud se nenarazí na správné místo.

insertAt – Stejně jako funkce insert vloží novou hodnotu, ale v konstatním čase. Tato funkce již neiteruje prvky, ale pouze zkontroluje, zda hodnota opravdu patří na předený index v poli listu. Volá se funkce compare pro 2 prvky obklopující nový prvek. Pro předchůdce funkce compare musí vrátit *false* a pro následníka *true*. Kdyby docházelo ke kontrole jen u jednoho z prvků, docházelo by ke špatnému zařazení.

getFirst – Vrátí hodnotu prvního prvku listu.

getByIndex – Vrátí hodnotu n-tého prvku listu.

removeFirst – Odstraní první prvek z listu. Jako první prvek se nastaví následník odstraněného prvku. Odstraněný prvek se zařadí do volných prvků jako první.

removeByIndex – Podobný princip jako *removeFirst* funkce, ale odstraní několik prvních prvků v konstatním čase. Tato funkce je užitečná v OrderBlock při odstraňování několika vyplněných limitních nebo stop objednávek.

9.2 Kontrakt OrderBlock

9.2.1 Struct objekty

Kontrakt má nadefinované tyto 3 hlavní objekty:

Market – Obsahuje adresy tokenů trhu (*base* a *quote* token) a 4 prioritní listy pro objednávky daného trhu.

Order – obsahuje všechny potřebné údaje o objednávce.

- orderId – id objednávky.

- marketId – k jakému trhu patří.
- price – na jaké ceně se má objednávka aktivovat. Pro market objednávku je cena nulová, jelikož se aktivuje instantně na aktuální ceně trhu.
- createdAt – timestamp, kdy byla objednávka vytvořena. Slouží primárně pro porovnání objednávek, pokud by jejich cena byla stejná.
- side – zda-li je objednávka nákupní, nebo prodejní.
- typee – určuje typy objednávky. Jsou možné následující typy: limit, stop, market, cancelled, filled, failed.
- slippage – maximální cena, na které se objednávka může vyplnit. Užitečný pouze pro market a stop objednávky. Slouží jako ochrana před prudkým pohybem ceny po odeslání transakce a také před TOD útokem (viz 4.3). Například kdyby byla spatřena čekající transakce s nákupní market objednávkou a jako reakce by se poslala další nákupní market objednávka s vyšší cenou za palivo – nakoupila by dříve a byla instantně v zisku.

OrderQueue – obsahuje všechny informace o vyplněných limitních objednávkách. Neukládá se do *storage*, pouze do *memory* při zpracování market objednávek.

- orderId – id vyplněné limitní objednávky.
- stopOrder – značí, zda-li je limitní objednávka vyplněna market nebo stop objednávkou. Tato informace poslouží při výměně tokenů, zda-li se mají poslat odesílateli transakce (tvůrce market objednávky), nebo tvůrci stop objednávky.
- filled – značí, zda-li limitní objednávka byla vyplněna přes celé množství objednávky nebo jen parciálně.
- sender – je adresa tvůrce market nebo stop objednávky.
- creator – je adresa tvůrce vyplněné limitní objednávky.
- amountIn – množství tokenů, které má být posláno na adresu *sender*.
- amountOut – množství tokenů, které má být posláno na adresu *creator*.

9.2.2 Storage proměnné

Kontrakt obsahuje následující *storage* proměnné:

- markets – mapping, který pro id trhu ukládá objekt trhu.
- orders – mapping, který pro id objednávky ukládá objekt objednávky.
- freeMarketId – volné id trhu, které se inkrementuje s každým vytvořeným trhem.
- freeOrderId – volné id objednávky, které se inkrementuje s každou vytvořenou objednávkou.
- pairs – ukládá id trhu podle tokenů. To je užitečné, pokud se potřebuje najít id trhu pro 2 konkrétní tokeny (pokud vůbec takový trh existuje).

9.2.3 Funkce

Kontrakt obsahuje následující veřejné (external) funkce, které mění stav blockchainu (nejsou read-only):

createMarket – Funkce očekává 2 argumenty, kterými jsou *base* token a *quote* token. Vytvoří se objekt trhu a inicializují se prioritní listy trhu.

createOrder – Funkce nejdříve validuje vstupy od uživatele. Mezi to patří například ověření, že pro zadanou limitní nákupní objednávku s nějakou cenou neexistuje limitní prodejní objednávka s nižší cenou. Po validaci vytvoří objekt objednávku podle předaných parametrů do určitého trhu. Pokud se jedná o limitní nebo stop objednávku, tak se vloží do prioritního listu a přesunou se tokeny od tvůrce do tohoto kontraktu. Kontrakt tokeny musí vlastnit přímo, aby bylo garantováno, že pro všechny objednávky je vždy dostatečné (přesné) množství tokenů. Jako nepovinný parametr funkce je index v poli prioritního listu, kam se má objednávka přímo vložit. Pokud se jedná o market objednávku, zavolá se funkce _marketOrder.

cancelOrder – Funkce zruší limitní nebo stop objednávku podle předané id objednávky jako parametr. Nejdříve se validuje, že objednávka patří tomu, kdo chce objednávku zrušit, a že se jedná o aktivní (nevyplněné ani nezrušené) limitní, nebo stop objednávku. Tvůrci objednávky jsou vráceny tokeny, které byly poslány kontraktu při jejím vytvoření.

Objednávka je označena jako zrušená a dojde k jejímu odstranění z prioritního listu (nebo okamžitě, nachází-li se zrušená objednávka v listu jako první).

Kontrakt dále obsahuje privátní funkce:

_marketOrder – Funkce nejprve zavolá funkci _buildOrderQueue, která najde limitní objednávky pro vyplnění. Dále se zjistí, zda se cena trhu po vyplnění market objednávky zvýší, a pokud ano, tak jestli se mají aktivovat nějaké stop objednávky. V cyklu se volá funkce _getExecutableStopOrder, dokud nevrátí, že již žádná další stop objednávka na aktivování není. Pro všechny aktivovatelné stop objednávky se zavolá funkce _buildOrderQueue, která najde limitní objednávky k jejich vyplnění. Nakonec se zavolá funkce _executeOrderQueue s předanými daty získané z funkce _buildOrderQueue.

_buildOrderQueue – Funkce pro zadanou market objednávku (nebo stop objednávku, která se stala market objednávkou) najde limitní objednávky k jejímu vyplnění. Pokud market objednávku nejde vyplnit, celá transakce selže. Je zde cyklus, který iteruje přes limitní objednávky, dokud zadaná market objednávka není vyplněna. Přitom se vytváří pole objektů OrderQueue, které obsahují informace o vyplněných limitních objednávkách. Nakonec z prioritního listu odstraní všechny plně vyplněné limitní objednávky.

_executeOrderQueue – Funkce zodpovědná primárně za výměnu tokenů mezi tvůrci objednávek. Funkci je předané pole OrderQueue objektů, podle kterých se zjistí, kolik a na jaké adresy tokeny poslat. Tvůrce vyplněných limitních objednávek obdrží tokeny přímo od tvůrce market objednávky. Tvůrci market objednávky obdrží tokeny přímo od kontraktu, jelikož kontrakt vlastní tokeny tvůrců limitních objednávek.

_getExecutableStopOrder – Funkce zkontroluje, zda cena první stop objednávky v prioritním listu vyhovuje aktuální tržní ceně, a pokud ano, objednávku vrátí.

_stopOrderFailed – Funkce je zavolána, pokud selže vyplnění nějaké stop objednávky. Selže, pokud není dostatek limitních objednávek za přívětivou cenu (určuje parametr objednávky *slippage*). Tvůrci selhané stop objednávky jsou zpět poslány tokeny a objednávka je označena jako selhaná.

10 Implementace front-endu

Ke komunikaci se sítí Ethereum a kontrakty byla použita knihovna web3 (viz 5.2.1). Je třeba zmínit, že k fungování směnárny není žádný front-end zapotřebí. Jedná se pouze o usnadnění interakcí mezi směnárnou a běžnými uživateli. Webová aplikace je rozdělena do následujících komponent.

10.1 App

Je kořenovou komponentou aplikace. Probíhá zde připojení k síti Etherea přes krypto peněženku Metamask. Poskytuje dalším komponentám možnost komunikovat se sítí Ethereum a kontraktem OrderBook přes react kontext. Dále zde probíhá načtení vytvořených trhů. Probíhá zde také routing, tedy renderování komponent podle url adresy. Aplikace má 3 hlavní stránky – nalezení trhu podle názvu tokenů (kořenová stránka), stránka na vytvoření trhu a stránka trhu, kde lze vytvářet a zobrazovat jednotlivé objednávky.

10.2 Market creator

Komponenta na vytváření nových trhů. UI obsahuje 2 vstupy od uživatele a těmi jsou adresy tokenů. Při stisknutí tlačítka *create* se vygenerují data pro transakci vytvoření trhu a pošlou se do peněženky Metamask, kde transakce bude čekat na potvrzení.

10.3 Market selector

Komponenta na výběr existujících trhů. Je zde jeden vstup, kam se zadávají symboly tokenů. Pokud trh se zadanými tokeny existuje, je možné jít na stránku tohoto trhu stisknutím tlačítka *select*.

10.4 Market

Kořenová komponenta pro trh. Jsou zde 3 funkce, které naslouchají událostem v kontraktu OrderBook. Těmi událostmi jsou vytvoření nových objednávek, jakákoliv

změna limitních nebo stop objednávek a zrušení objednávek. Při výskytu těchto událostí komponenta komunikuje s komponentou *Orderbook*, kde se změny promítnou. Komponenta dále načítá objednávky z kontraktu, které předá dalším komponentám. Komponenta renderuje 3 další komponenty: *Orderbook*, *ControlPanel* a *UserOrders*.

10.5 Orderbook

Komponenta na základě získaných dat o objednávkách od rodičovské komponenty Market zobrazuje orderbook daného trhu a aktuální cenu trhu. Na levé straně jsou zobrazené prodejní objednávky, na pravé nákupní. Oproti ostatním směnárnám se zde zobrazují také stop objednávky (běžně pouze limitní) kvůli plné transparentnosti. Pro zlepšení orientace uživatelů jsou zde také malé animace. Např. při zvýšení ceny trhu cena zezelená, při změně velikosti nějaké objednávky daná objednávka zabliká.

10.6 ControlPanel

Komponenta pro vytváření objednávek. Je nutné vybrat typ objednávky a stranu objednávky (nákupní, nebo prodejní). Komponenta dále obsahuje vstupy pro cenu a množství objednávky. Při stisknutí tlačítka vytvoření objednávky se cena a množství objednávek převedou do celočíselných jednotek tokenu (Ethereum neoperuje na desetinných číslech) a na základě všech zadaných parametrů se vygeneruje transakce pro vytvoření objednávky přes knihovnu *web3*.

10.7 UserOrders

Komponenta pro zobrazování objednávek, které patří připojené adresy peněženky. Tyto objednávky jsou získány od rodičovské komponenty. Objednávky jsou zobrazeny v tabulce a každou aktivní limitní a stop objednávku lze zrušit. Dojde tak k vygenerování transakce zrušení objednávky s daným ID a pošle se do peněženky k potvrzení.

11 Ověření funkcionality

V jazyce javascript byly napsány testy pro kontrakty OrderBlock a PriorityList. Byla využita testovací knihovna Mocha (viz 3.13.1) pro Hardhat vývojové prostředí (viz 3.13). Je zde jeden soubor s unit testy pro kontrakt PriorityList, jeden unit test pro kontrakt OrderBlock a jeden funkcionální test pro kontrakt OrderBlock. Funkcionální test zahrnuje komplikovanější situace (např. exekuce 50 objednávek). Testy se zaměřují na správnost vytvoření všech typů objednávek a jejich správnou exekuci.

Jednou ze základních funkcí z Mocha knihovny je *describe*, které se jako první parametr předává název testované komponenty (v jednotkovém testu je to typicky název funkce) a jako druhý parametr funkce, která se spustí při testování (*describe* mohou být vnořené).

V předané funkci v *describe* se očekává výskyt druhé základní funkce z Mocha knihovny *it*. Tato funkce opět očekává 2 parametry – název testu a funkce, ve které proběhnou jednotlivá ověření testu pomocí funkce *except.*

Funkce *except* očekává jeden parametr a tím je testovaná hodnota. Testovanou hodnotu lze testovat proti očekávané hodnotě různými způsoby. Například se musí rovnat přesně nebo aproximačně nebo že při volání funkce transakce selže s určitým důvodem selhání. V testech byla nejčastěji použita přímá rovnost a test selhání transakce. Aproximační rovnost byla použita pro hodnoty, kde je očekáváno hodně desetinných míst.

Pro testování kontraktu PriorityList byl vytvořen jednoduchý mock kontrakt, který umožňuje práci s prioritním listem (implementuje metodu compare, která řadí číselné hodnoty vzestupně). Testuje se zde, zda se prvky korektně zařazují a odstraňují z prioritního listu.

Do vývojového prostředí Hardhat lze nainstalovat plugin, který spustí testy, zjistí a zpracuje pokrytí testů v html formátu. Z výsledku vyšlo, že testy mají 98% pokrytí.

11.1 Spotřeba paliva

Do vývojového prostředí Hardhat lze nainstalovat plugin, který pro spuštěné testy vypíše spotřebu paliva sdruženou podle funkcí. Jelikož pod jednou funkcí je napsáno vytváření všech typů objednávek, pro potřebu získání spotřeby paliva byly testy pouštěny po částech, aby se získala přesnější množství na každou část testu.

11.1.1 Vytvoření trhu

Vytvoření trhu má spotřebu ~6,000,000 paliva. Důvodem tak vysoké spotřeby je inicializování prioritních listů. Inicializují se 4 (pro každý typ limitních a stop objednávek) o počáteční velikosti 64 prvků. Musí se tedy vytvořit pole o délce 64 a do každého napsat odkaz (*next*) na další prvek.

11.1.2 Vytvoření limitní a stop objednávky

Pro vytvoření těchto objednávek je potřeba téměř shodné množství paliva, které je zároveň konstantní, využije-li se metoda prioritního listu *insertAt* s předaným indexem, kam se má přesně objednávka zařadit. Mezi nejdražší operace se zde řadí zápis objektu objednávky do *storage* paměti mapping (~70,000 paliva), poslání tokenů od adresy odesílatele do kontraktu OrderBlock (~25,000 paliva), vložení ID objednávky do prioritního listu (~28,000 paliva) a inkrementace ID volné objednávky (~5,000 paliva). Celková průměrná spotřeba paliva je ~200,000.

11.1.3 Exekuce market objednávky

Je-li market objednávka vyplnitelná pouze jednou limitní objednávkou (a tedy implicitně nemůže být aktivována žádná stop objednávka), celková průměrná spotřeba paliva je ~150,000. Mezi nejdražší operace se tu řadí výměna tokenů mezi účastníky obchodu (~40,000 paliva) a modifikace zbývajícího množství limitní objednávky (~5,000).

Je-li market objednávka vyplnitelná dvěma limitními objednávkami, dojde ke zvýšení celkové spotřeby přibližně na ~200,000 (tedy o ~50,000). Toto zvýšení je

způsobeno odstraněním první vyplněné limitní objednávky z prioritního listu (~20,000 paliva) a výměnou tokenů, která zde proběhla dvakrát (mezi třemi účastníky). Dá se tedy aproximovat, že za každou další potřebnou limitní objednávku se spotřeba zvýší o ~30,000 paliva (odstranění více prvků z prioritního listu má konstantní potřebu). Pro porovnání se směnárnou Uniswap má realizace obchodu celkovou spotřebu přibližně ~140,000 paliva.

11.1.4 Exekuce stop objednávky

Exekuce stop objednávky je úzce spojena s exekucí market objednávky. Dá se nad tím přemýšlet tak, že stop objednávky zvýší množství původní market objednávky a povede to k vyplnění většího množství limitních objednávek. Za předpokladu, že došlo k vyplnění dvou limitních objednávek, je celková průměrná spotřeba paliva ~225,000. Došlo tedy o zvýšení o ~25,000 oproti verzi bez stop objednávky. Toto zvýšení je primárně způsobeno odstraněním stop objednávky z prioritního listu. A dá se říci, že počet zpracovaných stop objednávek nemá na celkovou spotřebu paliva velký vliv, jelikož odstranění několik prvků z prioritního listu je konstantní.

Akce	**Spotřeba paliva**
Vytvoření trhu	~6,000,000
Vytvoření trhu na Uniswapu	~4,000,000
Vytvoření limitní/stop objednávky	~200,000
Vytvoření market objednávky – základ	~150,000
Vytvoření market objednávky – vyplněno více limitních objednávek	~150,000 + x · 30,000, kde x je počet vyplněných limitních objednávek
Vytvoření market objednávky – vyplněno	~150,000 + x · 30,000 + y · 25000, kde x

více limitních objednávek, Spuštění stop objednávek	je počet vyplněných limitních objednávek a y je počet spuštěných stop objednávek
Obchod na Uniswapu	~140,000

Tabulka 11.1: Tabulka znázorňuje průměrné spotřeby paliva pro dané akce.

12 Zhodnocení

Bakalářská práce představila základní principy fungování blockchainu, sítě Ethereum, kontraktů a burz či směnáren. Hlavním cílem této práce bylo využít nabytých znalostí k návrhu plně decentralizované směnárny na bázi orderbooku pro obchodování ERC20 kompatibilních tokenů na sítí Ethereum.

Jako největší výzva této práce bylo efektivní navržení směnárny z hlediska spotřeby paliva, jelikož zapisování dat do blockchainu je nejdražší operace. Každá vytvořená limitní nebo stop objednávka je zapsána na blockchain, market objednávky se nezapisují, jelikož jejich účel je splněn ihned při zpracování transakce a dále nejsou potřeba. Navíc objednávky musí být seřazené podle ceny a data vytvoření, aby jejich exekuce proběhla vždy ve správném pořadí.

Pro ukládání objednávek a celého orderbooku byla navržena zčásti vlastní datová struktura prioritní list, která kombinuje abstraktní datovou strukturu, prioritní frontu a jednostranný spojený list. Všechny operace kromě vypsání všech prvků je možné provést v konstantním čase (jen při zpracovávání transakce). Avšak exekuce market objednávky vede na lineární složitost, jelikož je potřeba vyměnit tokeny mezi tvůrcem market objednávky a všemi tvůrci vyplněných limitních objednávek.

Směnárna byla naprogramována jako kontrakt v solidity. Byly napsány automatické testy, které byly spuštěné na lokální síti vývojového prostředí Hardhat. Bylo také vytvořeno uživatelské rozhraní v javascriptové knihovně React. Toto rozhraní umožňuje náhled orderbooku a vytváření všech typů objednávek na vybraném trhu. Pro testovací účely byla směnárna nahrána pouze na testovací síť Ethereum (Goerli a Sepolia), kde není potřeba reálné Ethereum na spotřebu paliva.

Literatura

[1] Nodes and clients | ethereum.org. [online]. Dostupné z: https://ethereum.org/en/developers/docs/nodes-and-clients/

[2] Wood, G. Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper. 2020, 3e2c089. Dostupné z: https://github.com/ethereum/yellowpaper.

[3] What Is a Merkle Root (Cryptocurrency)? How It Works in Blockchain. Investopedia [online]. Copyright © Investopedia [cit. 01.05.2023]. Dostupné z: https://www.investopedia.com/terms/m/merkle-root-cryptocurrency.asp

[4] [online]. Dostupné z: https://www.datasciencecentral.com/the-ethereum-virtual-machine-evm/

[5] Eliptická křivka – Wikipedie. [online]. Dostupné z: https://cs.wikipedia.org/wiki/Eliptick%C3%A1_k%C5%99ivka

[6] Digital Signatures on Ethereum | MyCrypto. Medium [online]. Dostupné z: https://medium.com/mycrypto/the-magic-of-digital-signatures-on-ethereum-98fe184dc9c7

[7] [online]. Dostupné z: https://blog.chain.link/what-are-abi-and-bytecode-in-solidity/

[8] Chytrý kontrakt – Wikipedie. [online]. Dostupné z: https://cs.wikipedia.org/wiki/Chytr%C3%BD_kontrakt

[9] [online]. Dostupné z: https://www.quicknode.com/guides/smart-contract-development/what-is-an-abi

[10] [online]. Dostupné z:
https://jeancvllr.medium.com/solidity-tutorial-all-about-contracts-e8e30bd1b289

[11] [online]. Dostupné z:
https://jeancvllr.medium.com/solidity-tutorial-all-about-functions-dba2ccb1e931

[12] [online]. Dostupné z:
https://medium.com/coinmonks/solidity-tutorial-all-about-modifiers-a86cf81c14cb

[13] [online]. Dostupné z:
https://betterprogramming.pub/solidity-tutorial-all-about-data-locations-dabd33212471

[14] [online]. Dostupné z:
https://medium.com/coinmonks/solidity-tutorial-all-about-mappings-29a12269ee14

[15] [online]. Dostupné z:
https://jeancvllr.medium.com/solidity-tutorial-all-about-array-efdff4613694

[16] [online]. Dostupné z: 14
https://jeancvllr.medium.com/solidity-tutorial-all-about-structs-b3e7ca398b1e

[17] Secure Ether Transfer | solidity-patterns [online]. Dostupné z:
https://fravoll.github.io/solidity-patterns/secure_ether_transfer.html

[18] Solidity – Special Variables - GeeksforGeeks. GeeksforGeeks | A computer science portal for geeks [online]. Dostupné z:
https://www.geeksforgeeks.org/solidity-special-variables/

[19] What Are ERC-20 Tokens on the Ethereum Network?. Investopedia [online]. Dostupné z: https://www.investopedia.com/news/what-erc20-and-what-does-it-mean-ethereum/

[20] Hardhat Explained – Moralis Web3 | Enterprise-Grade Web3 APIs. Moralis Web3 - Enterprise-Grade Web3 APIs [online]. Copyright © Moralis Copyright 2023 [cit. 08.03.2023]. Dostupné z: https://moralis.io/hardhat-explained-what-is-hardhat/

[21] Mocha – the fun, simple, flexible JavaScript test framework. Mocha - the fun, simple, flexible JavaScript test framework [online]. Dostupné z: https://mochajs.org/

[22] Known Attacks – Ethereum Smart Contract Best Practices. [online]. Dostupné z: https://ethereum-contract-security-techniques-and-tips.readthedocs.io/en/latest/known_attacks/

[23] Ethereum API | IPFS API & Gateway | ETH Nodes as a Service [online]. Dostupné z: https://www.infura.io/faq/general

[24] Infura Blog | Tutorials, Case Studies, News, Feature Announcements. Infura Blog | Tutorials, Case Studies, News, Feature Announcements [online]. Copyright © [cit. 08.03.2023]. Dostupné z: https://blog.infura.io/post/ethereum-javascript-libraries-web3-js-vs-ethers-js-part-i

[25] Next.js by Vercel – The React Framework [online]. Dostupné z: https://nextjs.org/learn/foundations/about-nextjs/

[26] What Is an Order Book? Definition, How It Works, and Key Parts. Investopedia [online]. Dostupné z: https://www.investopedia.com/terms/o/order-book.asp

[27] 3 Order Types: Market, Limit and Stop Orders | Charles Schwab. Charles Schwab | A modern approach to investing & retirement [online]. Copyright © 2023 [cit. 08.03.2023]. Dostupné z: https://www.schwab.com/learn/story/3-order-types-market-limit-and-stop-orders

[28] Time in Force: Definition, Types, and Examples. Investopedia [online]. Dostupné z: https://www.investopedia.com/terms/t/timeinforce.asp

[29] Understanding Liquidity and How to Measure It. Investopedia [online]. Dostupné z: https://www.investopedia.com/terms/l/liquidity.asp

[30] Volatility: Meaning In Finance and How it Works with Stocks. Investopedia [online]. Dostupné z: https://www.investopedia.com/terms/v/volatility.asp

[31] Crypto Spot Trading vs Margin Trading: What Is the Difference?. Crypto.com: The best place to buy Bitcoin, Ethereum, and 250+ altcoins [online]. Copyright © 2018 [cit. 08.03.2023]. Dostupné z: https://crypto.com/university/crypto-spot-vs-margin-trading

[32] Square – Groupe de conseil en stratégie et organisation [online]. Copyright © [cit. 08.03.2023]. Dostupné z: https://www.square-management.com/wp-content/uploads/2022/01/square_white-paper-decentrated-exchange-1.pdf

Programátorský manuál

Manuál k instalaci a spuštění směnárny.

Prvně je třeba si vytvořit etherovou peněženku Metamask.

- Do prohlížeče nainstalovat rozšíření Metamask.
- Otevřít rozšíření Metamask a vytvořit peněženku.
- Poté je třeba exportovat privátní klíč, který bude potřeba při nasazování kontraktů (pro exportování je třeba kliknout na 3 tečky vlevo nahoře a detaily účtu.
- Pro testovací účely byla zvolena testovací síť Ethereum nazvaná Sepolia. V Metamasku je třeba se přepnout na tuto síť (nahoře uprostřed, show test networks).
- Aby bylo možné provádět transakce, je třeba mít nějaké Ethereum. Zadejte vaši veřejnou adresu zde https://sepolia-faucet.pk910.de/ a po chvilce můžete kliknout na tlačítko „claim rewards" a obdržíte testovací Ether na Sepolia (doporučeno alespoň množství 0.1).

Následuje nasazení kontraktů na Sepolia. V adresáři *orderblock-contracts*:

- Spustit příkaz *npm install*, který nainstaluje všechny potřebné javascriptové balíčky včetně vývojového prostředí Hardhat.
- Je třeba zkopírovat soubor *.env.sample* a pojmenovat ho jako *.env* a zadat zde privátní klíč od Vaší vytvořené peněženky.
- Pro spuštění testů slouží příkaz *npx hardhat test.* Testy běží na lokální síti Hardhat.
- Pro nasazení kontraktů na testovací síť Sepolia slouží příkaz *npx hardhat run –network sepolia .\scripts\deploy.js*. Nasadí se 2 testovací ERC20 tokeny a kontrakt směnárny.
- Po úspěšném nasazení se vypíší adresy nasazených kontraktů, které jsou dále použity ve frontendu směnárny.

Následuje spuštění frontendu. V adresáři *orderblock-frontend:*

- Spustit příkaz *npm install*, který nainstaluje všechny potřebné javascriptové balíčky včetně vývojového prostředí React.
- Nahradit adresu Vámi nasazeného kontraktu OrderBlock v souboru *./src/contracts/OrderBlock.json*. Je třeba zde změnit klíč *address*.
- Spustit příkaz *npm start*, který spustí React stránku na *localhost:3000*.
- Přihlásit se na stránce přes Metamask.
- Ve vyhledávacím poli je možné vyhledávat trhy podle názvů tokenů, jeden již vytvořený trh je s tokeny *TEST1* a *TEST2* (příklad url trhu*localhost:3000/TEST1-TEST2*)
- Na stránce trhu se zobrazí orderbook a UI rozhraní pro vytváření objednávek.
- Při vytváření objednávek je třeba potvrdit 2 transakce, které se ukážou postupně v Metamasku.