TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Optimalizace využití výpočetního výkonu při těžení kryptoměn

Diplomová práce

M13000174

Studijní program: N2612 – Elektrotechnika a informatika
Studijní obor: 1802T007 – Informační technologie

Autor práce: **Bc. Pavel Bucháček**
Vedoucí práce: Ing. Petr Kretschmer

Liberec 2016

TECHNICAL UNIVERSITY OF LIBEREC
Faculty of Mechatronics, Informatics
and Interdisciplinary Studies

Usage optimalization of the computation performance in the crypto currency mining process

Master thesis

M13000174

Study programme: N2612 – Electrical Engineering and Informatics
Study branch: 1802T007 – Information technology

Author: **Bc. Pavel Bucháček**
Supervisor: Ing. Petr Kretschmer

Liberec 2016

TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky a mezioborových studií
Akademický rok: 2015/2016

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Pavel Bucháček**
Osobní číslo:	**M13000174**
Studijní program:	**N2612 Elektrotechnika a informatika**
Studijní obor:	**Informační technologie**
Název tématu:	**Optimalizace využití výpočetního výkonu při těžení kryptoměn**
Zadávající katedra:	**Ústav nových technologií a aplikované informatiky**

Zásady pro vypracování:

1. Seznamte se s problematikou kryptoměn
2. Navrhněte aplikaci pro optimalizaci těžby kryptoměn (možné postupy, kritéria optimalizace)
3. Realizujte vlastní aplikaci vhodnými vývojovými nástroji. Postupujte následovně:
a) Vytvořte proxy server pro sledování statistických ukazatelů při těžbě kryptoměn
b) Získané ukazatele z průběhu těžby ukládejte do vhodné databáze
c) Na proxy serveru implementujte algoritmus pro přepojování těžebního stroje mezi více pool servery podle uživatelem definované konfigurace (procentuální rozdělení výkonu).
d) Na základě analýzy statistických ukazatelů zajistěte přepojení těžebního stroje na jiný pool server v případě neefektivní těžby nebo výpadku spojení.
4. Ověřte funkčnost systému. Zhodnoťte změnu efektivity těžby bez optimalizace a s optimalizací

Rozsah grafických prací: **dle potřeby**

Rozsah pracovní zprávy: **30 - 40 stran**

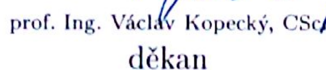
Forma zpracování diplomové práce: **tištěná/elektronická**

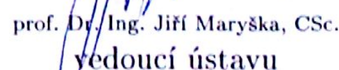
Seznam odborné literatury:

[1] PLÍVA, Zdeněk; DRÁBKOVÁ, Jindra. Metodika zpracování diplomových, bakalářských a vědeckých prací na FM TUL. Vyd. 1. Liberec : Technická univerzita, 2007. 40 s. Dostupné z WWW: <http://www.fm.tul.cz/files/jak_psat_DP.pdf>. ISBN 978-80-7372-189-3.
[2] NAKAMOTO, Satoshi. Bitcoin: A Peer-to-Peer Electronic Cash System
[3] DECKER, C.; WATTENHOFER, R. Information propagation in the bitcoin network, IEEE International Conference on Peer-to-Peer Computing (P2P), Trento, Italy, September 2013. <http://www.tik.ee.ethz.ch/file/49318d3f56c1d525aabf7fda78b23fc0/P2P2013_041.pdf>

Vedoucí diplomové práce: **Ing. Petr Kretschmer**
Ústav nových technologií a aplikované informatiky

Datum zadání diplomové práce: **20. října 2015**
Termín odevzdání diplomové práce: **16. května 2016**


prof. Ing. Václav Kopecký, CSc.
děkan


prof. Dr. Ing. Jiří Maryška, CSc.
vedoucí ústavu

V Liberci dne 20. října 2015

Prohlášení

Byl jsem seznámen s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu TUL.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Diplomovou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím mé diplomové práce a konzultantem.

Současně čestně prohlašuji, že tištěná verze práce se shoduje s elektronickou verzí, vloženou do IS STAG.

Datum: 13. 5. 2016

Podpis:

Poděkování

Rád bych poděkoval vedoucímu práce Ing. Petru Kretschmerovi za cenné rady při vytváření diplomové práce. Dále bych rád poděkoval doc. RNDr. Pavlu Satrapovi za zpracování LaTeX šablony, kterou jsem využil při vytváření této zprávy.

Abstrakt

V práci se zabývám optimalizací těžebního procesu kryptoměn. Rozebírám zde možná rizika, kterými jsou výpadky spojení a nízká efektivita probíhající těžby. Jejich negativní dopad se snažím minimalizovat zařazením proxy serveru umístěným mezi těžebního klienta a cílový pool server. Cílem proxy serveru je monitorovat a analyzovat probíhající těžbu.

V případě výpadku spojení nebo zjištění neefektivní těžby proxy server přepojí těžebního klienta na jiný pool server. Kromě těchto vlastností systém uživateli umožňuje definovat několik měn, které bude jeho těžební klient těžit. U každé měny může uživatel nastavit určitý příděl výkonu.

Systém tak přináší vyšší efektivitu těžby a umožňuje jednomu klientovi těžit více měn. Tím dojde k maximalizaci šance na získání odměny za ověřené bloky.

Klíčová slova

kryptoměny, těžení kryptoměn, Bitcoin, Getwork, Getblocktemplate, Stratum, pooled mining

Abstract

This thesis focuses on the optimization of the cryptocurrency mining process. I discuss all possible threats in this process, which can be a network connection errors or a low effectivity of the mining process. I try to minimalize the impact of these events by using a proxy server placed between a mining worker and a target pool server. The aim of the proxy server is to analyze all events in the mining process.

If the proxy server detects any network fail, or an ineffective mining process is detected, the proxy server will reconnect the mining worker to a different pool server immediately. Except of these features user can also define a several currencies which wants to mine. User can define an amount of performance for each of these currencies.

The system boosts the effectivity of the cryptocurrency mining process and allows the possibility to mine more currencies by one worker. These features will maximalize the chance to earn the block reward.

Keywords

crypto currency, cryptocurrency mining process, Bitcoin, Getwork, Getblocktemplate, Stratum, pooled mining

Obsah

Seznam zkratek 11

1 Úvod **12**

2 Kryptoměny **13**

2.1 Co je to kryptoměna 13

2.2 Uživatelé kryptoměn 14

2.3 Těžaři kryptoměn 14

2.4 Těžařská seskupení 14

2.5 Těžební stroje 15

2.6 Blockchain 15

2.7 Bitcoin 15

2.8 Alternativní kryptoměny 16

2.9 RPC protokoly 16

2.9.1 Getwork 16

2.9.2 Getblocktemplate 17

2.9.3 Stratum 17

3 Návrh proxy serveru pro optimalizaci těžby **18**

3.1 Motivace 18

3.2 Struktura systému 19

3.2.1 Abstrakce RPC protokolů 19

3.2.2 Úložiště dat 20

3.2.3 Správa spojení 21

3.2.4 Analýza nasbíraných dat 22

3.2.5 Konfigurace systému 23

3.3 Možnosti systému 23

3.3.1 Algoritmy a měny 24
3.3.2 Pool servery 24
3.3.3 Těžební skupiny 24

4 Realizace systému pro optimalizaci těžby **26**
4.1 Knihovna pro práci s RPC protokoly 27
4.1.1 Princip činnosti knihovny 27
4.1.2 Identifikace RPC protokolu 28
4.1.3 Autentizace uživatele 30
4.1.4 Handlery RPC protokolů 32
4.1.5 Přepojování klienta mezi pool servery 32
4.2 Úložiště nasbíraných dat 34
4.2.1 Verzování databáze 35
4.2.2 Model aplikace 35
4.2.3 Mapování dat 36
4.2.4 Testování servisní vrstvy 39
4.3 Proxy server 40
4.3.1 Výpadky spojení 41
4.3.2 Plánovač 42
4.3.3 Výpočet efektivity spojení 42
4.3.4 Rozdělování výkonu 44
4.4 Webové rozhraní pro konfiguraci systému 45
4.4.1 REST API 45
4.4.2 Webová aplikace 47
4.5 Testování systému 48
4.5.1 Testovací prostředí 48
4.5.2 Ověření funkčnosti systému 49

5 Závěr **51**

Literatura **52**

A Přiložené DVD **54**

Seznam obrázků

3.1 Schéma systému 25

4.1 Rozdělení systému na jednotlivé projekty 26
4.2 Handlery umístěné ve frontě frontendového kanálu 29
4.3 Hierarchie rozhraní adaptérů pro abstrakci RPC protokolů 33
4.4 Struktura modelových tříd pro ukládání konfiguračních dat 37
4.5 Struktura modelových tříd pro ukládání statistických ukazatelů . . . 38
4.6 Struktura modelových tříd pro ukládání stavových dat aplikace 38

Seznam zkratek

AJAX	Asynchronous JavaScript and XML, metoda pro asynchronní zpracování dat na pozadí webové stránky.
API	Application Programming Interface, rozhraní pro programování aplikací.
BTC	Zkratka pro elektronickou měnu Bitcoin.
CQL	Cassandra Query Language, dotazovací jazyk používaný pro manipulaci s daty v databázi Cassandra.
DOM	Document Object Model, objektový model dokumentu reprezentující HTML stránku.
GH/s	Giga hashes per second, počet spočtených hashů za vteřinu. Jednotka která se používá pro vyjádření výkonu těžebního stroje.
HTTP	Hypertext Transfer Protocol, inernetový protokol pro přenos hypertextových dokumentů.
JSON	JavaScript Object Notation, způsob zápisu dat.
JSON-RPC	RPC protokol, který používá data zakódovaná do formátu JSON.
JSX	Způsob zápisu kódu, který umožňuje v javascriptovém kódu používat HTML značky.
REST API	Representational State Transfer, metoda pro přístup k datům za pomoci HTTP volání.
RPC	Remote procedure call, vzdálené volání procedur.
TTL	Time to live, doba omezující platnost záznamu v databázi Cassandra. Po její uplynutí je záznam automaticky smazán.

1 Úvod

V této práci čtenáře nejdříve seznamuji se základními pojmy a teorií týkající se kryptoměn. Kromě principů fungování kryptoměn se zmiňuji i o RPC protokolech využívaných při jejich těžbě.

Ve třetí kapitole se věnuji teoretickému návrhu systému. Po definici všech požadavků postupně řeším návrh jednotlivých komponent, které budou nezbytné k jeho realizaci. Na závěr této kapitoly stanovuji možnosti systému, které bude svým uživatelům nabízet.

V kapitole s číslem čtyři popisuji realizaci navrženého systému. Jako první se věnuji knihovně pro abstrakci RPC protokolů, která tvoří základ celého systému. Poté, co systém naučím rozumět všem RPC protokolům, řeším ukládání monitorovaných statistických ukazatelů do databáze. Kromě těchto údajů je potřeba perzistentně ukládat i konfigurační a stavová data. Po vytvoření zmíněných částí mám k dispozici již vše potřebné pro analýzu efektivity těžby a přepojování klientů.

V rámci proxy serveru implementuji logiku pro přepojování klientů při výpadcích spojení. Dále zde realizuji periodicky spouštěný skript starající se o analýzu efektivity těžby a procentuální rozdělování výkonu mezi různé měny. Na základě jeho výstupu poté dochází k vytváření požadavku pro přepojení klienta na jiný pool server.

Aby byl systém uživatelsky přívětivý a umožňoval snadné nastavení všech parametrů těžby, vytvořil jsem webové rozhraní, kterému se věnuji v další podkapitole. Závěrem popisuji testování systému v průběhu vývoje a ověření jeho funkčnosti po dokončení realizace.

2 Kryptoměny

Bitcoin a kryptoměny obecně se za posledních několik let dočkaly obrovské popularity. Kromě růstu uživatelské základny a míst, kde lze pomocí kryptoměn platit, roste také výpočetní výkon sítě. Těžaři investují nemalé peníze do svého těžebního vybavení. Pryč jsou doby, kdy se těžilo pomocí procesorů. Ani výpočty pomocí grafických karet již nebyly dostačující, a tak se začali používat programovatelná hradlová pole – FPGA. Největšího výkonu spolu s nejpříznivější spotřebou energie se dnes dosahuje pomocí ASIC integrovaných obvodů. Jedná se o obvod, který je navržen pro výpočet jednoho konkrétního algoritmu. Jeho pořizovací cena je však hodně vysoká.

Jak postupně rostl výkon sítě, měnily se i nároky na používané RPC protokoly. Původně navržený Getwork byl časem nahrazen Getblocktemplate, u kterého nebylo nutné zasílat tolik požadavků v průběhu výpočtu. Stroj tak nebyl omezen kapacitou sítě. Dnes nejpoužívanější protokol Stratum vytvořil český programátor Marek Palatinus (vystupující pod přezdívkou Slush). Protokol je založen na myšlence, že celý proces těžby řídí pool server, který má nejlepší přehled o tom, co je potřeba počítat.

2.1 Co je to kryptoměna

Kryptoměny jsou digitální měny, které nemají žádnou fyzickou podobu (bankovky, mince). Jsou generovány pomocí počítačové sítě o velikém výpočetním výkonu. Hlavními vlastnostmi elektronických měn je plná decentralizace. Měna tedy není závislá na vlivu státu, autora, nebo skupiny lidí. U těchto měn nehrozí inflace, padělání a zabavování účtů. Na rozdíl od ostatních měn, nejsou kryptoměny založeny na důvěře jejímu vydavateli (bance, státu). Veškeré finanční toky jsou sdílené pomocí veřejné peer to peer sítě. Neexistuje žádné centrální místo, kde by byly transakce uloženy.

2.2 Uživatelé kryptoměn

Každý koncový uživatel vlastní jednu, nebo více peněženek. Každé peněžence náleží veřejný a privátní klíč. Uživatel se k síti připojí pomocí programu, který si udržuje distribuovanou databázi se všemi transakcemi v síti (tzv. blockchain). Z toho lze zjistit, které peněžence náleží jaké mince (nebo jejich části). Pokud chce uživatel poslat peníze někomu jinému, vezme patřičný obnos peněz a vytvoří z nich transakci. Transakce [1] může, ale nemusí obsahovat poplatek za provedení. Transakci klient následně podepíše svým soukromým klíčem. Takto vytvořenou transakci odešle do sítě a transakce čeká na potvrzení.

2.3 Těžaři kryptoměn

Úkolem těžařů je potvrzování transakcí. Sbírají všechny aktuálně nepotvrzené transakce a skládají je do bloků. K takto sestavenému bloku hledají kryptografickou nonci [2] (anglicky cryptographic nonce). Musí najít takovou nonci, aby se hash výsledného bloku vešel pod stanovený limit. Obtížnost se stanovuje dynamicky tak, aby se blok v síti potvrdil průměrně jednou za 10 minut. Obtížnost tedy roste spolu s výpočetním výkonem celé sítě.

Motivací těžařů je odměna za potvrzený blok. Ta je aktuálně stanovena na 25 BTC [3] a každé čtyři roky (každých 210 000 bloků) se snižuje na polovinu. Těžaři také náleží všechny poplatky ze zahrnutých transakcí. Z tohoto principu vyplývá, že v současné době je většina transakcí bez poplatku. Jak se postupem času bude snižovat odměna za vytěžený blok, bude také růst cena za transakci. Vzhledem k tomu, že si těžař může vybrat, které transakce do bloku zahrne, jsou obvykle transakce s poplatkem potvrzeny dříve než ty bez poplatku.

2.4 Těžařská seskupení

Pokud je těžař připojen přímo k distribuované síti (tzv. solo mining), je jeho pravděpodobnost na vytěžení bloku velmi malá. Z tohoto důvodu se těžaři sdružují do uskupení (tzv. pooled mining [4]). Těžaři se připojují k pool serveru, který je připojen na Bitcoin síť. Pool server sestavuje transakce do bloků, které následně pomocí RPC protokolů zasílá těžařům k nalezení nonce. Každý těžař má přidělený

rozsah, ve kterém má nonci hledat. Těžař postupně odesílá dílčí výsledky, které mají nižší složitost, než má aktuální blok. Poté, co pool server od některého z těžařů obdrží výsledek odpovídající složitosti bloku, odešle informaci do sítě a získá odměnu za vytěžený blok. Tato odměna je poté rozdělena mezi těžaře podle toho, kolik zaslali mezivýsledků.

2.5 Těžební stroje

Uživatel, který si zřídí účet na pool serveru, si definuje těžební stroje (tzv. workery). Jedná se o jednotlivé těžební stroje, kterými se bude k poolu připojovat.

2.6 Blockchain

Blockchain obsahuje kompletní historii všech transakcí v síti [5], která začíná takzvaným genesis blokem [6]. Každý blok obsahuje kromě transakcí také hash předchozího bloku. Je tedy garantováno, že jsou bloky řazeny chronologicky za sebou. Tento princip zabraňuje dvojímu utrácení měny (double-spending [7]). Pokud by byl některý z bloků změněn, musela by se přepočítat jeho nonce a tím pádem by se změnil i jeho hash. Poté by se musely přegenerovat všechny následující bloky. Pokud je tedy blok umístěn v blockchain dostatečně hluboko (zhruba 6 potvrzení), je tento proces z výkonnostního hlediska nemožný.

2.7 Bitcoin

Bitcoin je první a zároveň nejznámější, nejsilnější a nejpoužívanější kryptoměnou. Síť vytvořil v roce 2009 člověk pod pseudonymem Satoshi Nakamoto [8]. Hlavní jednotkou měny je Bitcoin, který se značí zkratkou BTC. Nejmenší přípustnou hodnotou měny je tzv. satoshi, který je definován jako 0.000 000 01 BTC. Počet bitcoinů je předem stanoven na 21 milionů BTC [9]. Tohoto čísla by se podle předpokladů mělo dosáhnout v roce 2140.

2.8 Alternativní kryptoměny

Na základě úspěchu Bitcoinu vznikly postupem času i další měny, které jsou založeny na stejných principech. Tomu nahrává i to, že všechny zdrojové kódy bitcoinu jsou volně k dispozici. V podstatě každý si tak může vytvořit vlastní elektronickou měnu. Alternativní měny se od sebe liší obtížností, algoritmem pro těžbu, celkovým počtem jednotek, které lze vytěžit a dalšími parametry. Patrně nejvýznamnější alternativní kryptoměnou je Litecoin.

2.9 RPC protokoly

Aby mohl pool server přidělovat práci svým uživatelům (těžařům), musí být schopný s nimi komunikovat. Tato komunikace probíhá pomocí RPC protokolů (remote procedure call, vzdálené volání procedur). Pool server takto informuje uživatele, kteří zahájí výpočet a následně pošlou zpět vypočítaný výsledek. Postupem času vznikly celkem tři protokoly, které se u kryptoměn používají.

2.9.1 Getwork

Jedná se o historicky nejstarší používaný protokol [10]. Přenos probíhá pomocí HTTP komunikace. Protokol využívá JSON-RPC specifikaci – veškerá data se tedy přenášejí ve formátu JSON.

Poté, co se těžební klient připojí k pool serveru, zašle žádost o přidělení práce. Od serveru následně obdrží odpověď, která obsahuje sestavený blok a obtížnost. Když klient nalezne požadovanou nonci, odešle pomocí další zprávy výsledek serveru. Server mu odpoví, jestli tento výsledek přijímá (accepted), nebo nepřijímá (rejected). Příčinou odmítnutého výsledku může být například to, že tento blok již vyřešil jiný stroj (tzv. stale share).

Aby se předcházelo zbytečně velkému počtu odmítnutých výsledků, zavedla se možnost longpoll. Poté, co klient obdrží první zprávu se zadanou prací, vytvoří další žádost na server (longpoll request). U tohoto spojení nastaví klient velmi dlouhou dobu, po kterou bude čekat na odpověď. Ve chvíli, kdy pool server obdrží od jiného stroje validní výsledek, zašle na všechny otevřené longpoll spojení zprávu, která obsahuje novou práci. Po obdržení klient přeruší aktuální výpočet a začne počítat

nový blok. Klient si také otevře nové longpoll spojení.

Hlavní nevýhodou tohoto protokolu je nutnost zasílat novou žádost pro každou dílčí práci. Tento fakt omezuje maximální výkon těžebního stroje zhruba na 4 GH/s.

2.9.2 Getblocktemplate

Motivací pro vznik tohoto protokolu [11] byla kromě výkonnostního hlediska také možnost decentralizace. U Getwork protokolu řídí sestavování transakcí do bloků pool server a těžební stroj nemá možnost to ovlivnit. Protokol je navržen tak, aby bylo snadné jej do budoucna rozšiřovat. Veškerá rozšíření jsou zdokumentována pomocí BIP dokumentů (Bitcoin Improvement Proposal) [12].

Protokol je stejně jako Getwork, založen na HTTP komunikaci a JSON-RPC specifikaci. Po připojení pošle klient žádost o šablonu (template). Odpověď od pool serveru obsahuje obtížnost, hash předchozího bloku, aktuální nepotvrzené transakce a další informace nutné ke složení bloku. Klient tedy může provádět mnohem více výpočtů bez toho, aby si musel serveru říkat o další práci. Spočítané výsledky průběžně posílá serveru. Protokol taktéž obsahuje možnost longpoll.

2.9.3 Stratum

Protokol Stratum je nejnovější a v současnosti také nejpoužívanější protokol [13]. U předchozího protokolu veškerou komunikaci řídí těžební klient. Nejlepší přehled o tom jaká práce je již hotová a co je aktuálně potřeba počítat má však pool server. Klient otevře TCP soket, po kterém komunikuje s pool serverem. Tento soket zůstává otevřen po celou dobu těžby. Pool server tak může klientovi kdykoliv poslat zprávu. Protokol je nazýván `line based`. Jednotlivé zprávy se od sebe totiž oddělují znakem konce řádky (`\n`). Protokol využívá JSON-RPC 2.0 specifikace. Pomocí tohoto protokolu lze přes jedno TCP spojení připojit stroj o výkonnosti až 18 EHash/s (Exa-hashes/s).

3 Návrh proxy serveru pro optimalizaci těžby

3.1 Motivace

Pokud se člověk rozhodne zabývat se těžbou kryptoměn, musí si opatřit výpočetní stroje s patřičným výkonem. To je spojeno s náklady na provoz, které nejsou malé. Odměnou za vynaložené úsilí by měla být provize za ověřené bloky. V průběhu těžby se ale může objevit několik jevů, které mají negativních dopad na výslednou efektivitu těžby. Mohou to být například výpadky internetového spojení mezi těžebním strojem a cílovým pool serverem. Vzhledem k velké konkurenci při těžbě kryptoměn bývají také poměrně časté útoky na pool servery, které mají za cíl vyřadit je z provozu. V takovém případě není pool server schopný s těžebním klientem komunikovat a ten tím pádem nemá co počítat. Může se také stát, že se těžební klient snaží těžit měnu, která má na něj příliš vysokou obtížnost. V takovém případě nikdy nedosáhne požadované odměny za ověřený blok.

Na základě těchto poznatků vznikl nápad na vytvoření systému, který by se tyto negativní jevy snažil co nejvíce eliminovat. Klient by se nepřipojoval k cílovému pool serveru přímo, ale pomocí mezičlánku ve formě proxy serveru. Proxy server by poté na základě uživatelské konfigurace a aktuální situace zvolil cílový pool server. Veškerou komunikaci mezi těžebním klientem a cílovým pool serverem by monitoroval a řešil všechny výše popsané situace. Kromě toho by uživateli umožnil definovat více měn, které chce těžit. Uživatel by poté měl možnost výkon těžebního stroje procentuálně rozdělovat mezi jednotlivé měny.

3.2 Struktura systému

Základem pro celý systém bude abstrakce RPC protokolů (viz kapitola 2.9), pomocí kterých komunikuje těžební klient s pool serverem. Bude potřeba rozlišovat jednotlivé akce, které jsou specifické pro konkrétní protokoly. Jednotlivé události bude potřeba ukládat do perzistentního úložiště a v periodických intervalech je vyhodnocovat. Na základě analýzy nasbíraných ukazatelů se vyhodnotí efektivita aktuálního spojení. V případě zjištění neefektivního spojení systém přepojí klienta na jiný pool server. Z tohoto důvodu musí systém evidovat všechny aktivní spojení. Systém také musí hlídat výpadky spojení a okamžitě na ně reagovat. Kromě ukládání statistických ukazatelů těžby je nutné ukládat také konfigurační údaje, na základě kterých se zvolí cílový pool server.

3.2.1 Abstrakce RPC protokolů

Základním předpokladem pro realizaci systému na optimalizaci těžby kryptoměn je porozumění protokolům, pomocí kterých těžební klient komunikuje s pool serverem. Tato komunikace se bude muset po celou dobu těžby monitorovat. Z přenášených zpráv se budou určovat jednotlivé události protokolů a ty se budou ukládat do databáze.

Aby byl celý systém přehlednější a lépe testovatelný, bude tato část navržena jako knihovna. Ta se bude starat pouze o abstrakci RPC protokolů a nebude tedy obsahovat žádnou aplikační logiku. Přínosem tohoto řešení bude to, že knihovna bude snadno použitelná i pro účely jiných projektů pracujících s těmito RPC protokoly.

Knihovna bude muset umět analyzovat příchozí zprávu od klienta a určit u ní jeden ze tří používaných RPC protokolů (Getwork, Getblocktemplate, Stratum). Z přijaté zprávy musí přečíst jméno a heslo klienta. Na základě těchto informací provede jeho autentizaci. Podle obsahu zprávy určí událost protokolu a zprávu dále předá na pool server. Podobným způsobem bude zpracovávat i odpovědi přijímané od pool serveru.

Z definovaných požadavků je patrné, že veškerá logika týkající se síťového provozu bude zapouzdřena přímo v knihovně. Naopak aplikační logika týkající se autentizace klientů a zpracování událostí protokolů bude implementována mimo tuto knihovnu. Pro tyto účely bude knihovna definovat několik rozhraní. Konkrétní implementace rozhraní s aplikační logikou se knihovně předá při její inicializaci.

Knihovna se bude starat o navazování spojení mezi klientem a pool serverem. Všechna takto vzniklá spojení bude dále spravovat. Pomocí rozhraní bude informovat o událostech RPC protokolů i o výpadcích spojení. Dále bude umět připojeného klienta přepojit na jiný pool server. Požadavky na přepojení klienta budou opět vznikat externě, knihovna je bude pouze zpracovávat. Definováno bude také rozhraní pro úložiště konfiguračních dat. Na jeho základě bude knihovna schopna určit cílový server pro klienta.

Aby bylo možné knihovnu samostatně testovat, bude v ní pro každé z definovaných rozhraních existovat jedna výchozí implementace.

3.2.2 Úložiště dat

Abych mohl vyhodnocovat efektivitu těžby, musím v jejím průběhu ukládat poměrně velké množství statistických ukazatelů pro každé spojení. Kromě nasbíraných ukazatelů bude nutné ukládat také stavová data aplikace. Ukládání i přístup k datům musí být rychlý. Aby byl systém do budoucna snadno škálovatelný, musím od začátku počítat s možností distribuce systému na více serverů.

Na základě těchto požadavků jsem se rozhodl pro použití NoSQL databáze, konkrétně databáze Cassandra. Jejími přednostmi jsou hlavně vysoká rychlost [14] a snadná škálovatelnost. K manipulaci s daty je možné používat dotazovací jazyk CQL. Cassandra také umožňuje asynchronní zápis dat a jednotlivým záznamům je možné nastavovat hodnotu TTL. Po jejím uplynutí dojde k automatickému odstranění takto označených záznamů.

Jako úložiště pro konfigurační a autentizační údaje se nabízí použití klasické relační databáze. Znamenalo by to ale, že bych v projektu musel používat dvě různé databáze. Tím by se zkomplikovalo mapování databázových dat na model aplikace. Odděleně by se také muselo řešit verzování databází. V neposlední řadě by také vzrostla provozní režie a nutnost řešit zálohování pro každou z databází odděleně. Relační databáze navíc nejde tak snadno škálovat. Dále bych se potýkal s problémy provázání záznamů mezi databázemi a konzistencí dat.

Na základě této úvahy jsem se rozhodl pro použití NoSQL databáze i pro ukládání konfiguračních dat. Návrh databázové struktury bude oproti relační databázi probíhat odlišně. V některých případech bude nutná denormalizace dat a s tím spojená duplicita záznamů. Benefitem tohoto řešení ale bude vyšší rychlost, jednodušší implementace a výborná škálovatelnost.

Abych data logicky odlišil podle jejich charakteru, definuji tři databázové keyspace. Keyspace představuje v terminologii databáze Cassandra obdobu schéma v relačních databázích [15]. Konfigurační a autentizační údaje budou ukládány v keyspace `poolec_core`, statistické ukazatele v `poolec_real_time_statistics` a stavová data v `poolec_state_data`.

3.2.3 Správa spojení

Důležitou funkcionalitou systému je schopnost přepojení těžebního klienta na jiný pool server. Z toho důvodu je nutné evidovat všechna otevřená spojení. Systém bude definovat rozhraní pro vytváření požadavků na přepojení klienta. Pokud vznikne takový požadavek, dojde k uzavření všech spojení daného klienta. Před odpojením klienta se do úložiště stavových dat zaznamená adresa nového pool serveru. Klient následně vytvoří požadavek na proxy server a ten ho připojí k novému pool serveru.

U protokolů Getwork a Getblocktemplate může při tomto procesu dojít k jedné mezní situaci, jejíž vznik je popsán níže.

- Klient je připojen k pool serveru X.
- Vznikne požadavek pro přepojení na pool server Y.
- Dojde ke zpracování požadavku na přepojení klienta. Spojení mezi proxy serverem a klientem se uzavře a do stavových dat se uloží adresa pool serveru Y.
- Klient však stále počítá blok ze serveru X. Po dokončení výpočtu pošle výsledek na server.

V tomto případě by došlo k tomu, že by byl výsledek pro pool server X zaslán na pool server Y. Výsledek výpočtu by s největší pravděpodobností nebyl pool serverem Y akceptován a čas na jeho výpočet by byl k ničemu. Abych této situaci mohl zabránit, bude nutné si do stavových dat ukládat adresu pool serveru pro těžené bloky. Výsledek tak bude zaslán na správný server i po zpracování požadavku na přepojení. Naštěstí nemůže dojít k situaci, kdy by jeden klient těžil více bloků současně, takže bude stačit uchovávat pro každého klienta vždy pouze poslední blok.

Aby přepojování klientů probíhalo efektivně a částečně se předcházelo výše popisovanému problému, nedojde k ukončení spojení ihned po vytvoření požadavku na přepojení. Klient totiž může mít již téměř vypočtený mezivýsledek, který by během

několika málo okamžiků poslal na pool server. Kdybych klienta odpojil okamžitě, byl by tento mezivýsledek zahozen. Přepojení klienta tedy proběhne při splnění první z následujících podmínek:

- Klient odeslal spočítaný výsledek a pool server vrátil odpověď.
- Klient odeslal žádost o přidělení nového výpočtu (u protokolů Getwork a Getblocktemplate).
- Server poslal informaci o novém výpočtu (odpověď na `longpoll request`, metody `difficulty set` a `mining notify`).
- Vypršel časový limit žádosti o přepojení.

3.2.4 Analýza nasbíraných dat

Nasbíraná data se budou v systému periodicky vyhodnocovat. V definovaném intervalu se bude periodicky spouštět skript, který si z databáze načte všechna data vzniklá od jeho posledního spuštění. Nad nimi se provede analýza a vyhodnotí se efektivita těžby za tento časový úsek. Efektivita se bude určovat na základě počítaného skóre. Výpočet skóre bude založen na vybraných statistických ukazatelích těžby. V systému bude definováno mezní skóre, po jehož překročení bude těžba považována za neefektivní.

Periodicky spouštěný skript bude také zajišťovat procentuálního rozdělení výkonu těžebních strojů mezi jednotlivé měny. Pro tuto potřebu budu muset ve stavových datech aplikace evidovat čas začátku těžby pro jednotlivé klienty. Pokud bude těžba vyhodnocena jako efektivní (nebude tedy vytvořen požadavek na přepojení), dojde ke spočítání doby, po kterou klient z aktuálního serveru těží. Rozdělení výkonu bude založeno na definovaném časovém kvantu. Takto definovaná časová jednotka bude považována za 100% výkonu. Podle uživatelského nastavení se dopočte příslušné časové kvantum, po jehož dobu se má těžit. Překročí-li doba těžby toto kvantum, dojde k přepojení klienta.

Při zpracování statistik potřebuji načíst záznamy v časovém rozsahu od jejich posledního zpracování. U každého záznamu bych si tedy měl ukládat čas jeho pořízení a zpracování. Dotaz pro načtení záznamů by poté obsahoval podmínku na časový rozsah. S tím by však byla spojená zbytečná režie. Využiji proto jedné z předností databáze Cassandra a u statistických dat budu nastavovat hodnotu TTL. Každý

statistický záznam bude mít nastavenou časovou platnost, po jejímž uplynutí dojde k jeho automatickému smazání. Doba TTL bude nastavena synchronně s periodou spouštění skriptu pro vyhodnocení dat. Při načítání záznamů tak budu v databázovém dotazu klást podmínku pouze na těžebního klienta. Odpadne tím nutnost ukládat si u záznamů časové údaje. Vzhledem k tomu, že dotaz bude obsahovat pouze podmínku na ID klienta, bude jeho zpracování mnohem rychlejší.

Synchronizace automatického mazání záznamů a spouštění skriptu pro jejich vyhodnocení však nikdy nebude dokonalá. Může se tak stát, že se některé záznamy zpracují dvakrát, popřípadě budou některé přeskočeny. Vzhledem k vysokému počtu nasbíraných dat však můžu tuto nedokonalost zanedbat. Kromě jednoznačné výkonnostní výhody se také zabrání možnému přehlcení databáze záznamy, které budou v čase přibývat velmi rychle.

V dalších verzích systému bych chtěl uživateli nabídnout historickou statistiku těžby rozdělenou podle různých ukazatelů. Uchovávání všech nasbíraných statistických ukazatelů těžby by ale bylo velmi náročné na diskový prostor. Bude tedy nutné data vhodně agregovat a ukládat je do zvláštního keyspace pro dlouhodobé statistiky. Pro tuto potřebu vznikne další periodicky spouštěný skript. Při jeho spuštění proběhne agregace dat nasbíraných od posledního běhu skriptu a jejich uložení do dlouhodobých statistik. Z tohoto datového zdroje se statistiky budou vypisovat v uživatelském rozhraní.

3.2.5 Konfigurace systému

Jak jsem již zmínil v kapitole 3.2.2, budu v databázi ukládat i konfigurační data. Aby aplikace byla jednoduše nastavitelná, vytvořím uživatelské rozhraní ve formě responzivní webové aplikace. Abych mohl v budoucnu jednoduše vytvořit mobilní aplikaci nebo poskytovat data aplikacím třetích stran, rozhodl jsem se pro vytvoření REST API rozhraní. Webová aplikace bude ve formě javascriptového klienta, který bude se serverem komunikovat pomocí AJAXových požadavků.

3.3 Možnosti systému

Pomocí webového rozhraní si uživatel založí účet, pod kterým bude spravovat všechny své těžební klienty a jejich konfiguraci. V budoucnu zde také nalezne statistiky agregované podle různých parametrů. K uživatelskému účtu bude uživateli

vygenerován unikátní kód, pomocí kterého bude schopný autentizovat své těžební klienty.

3.3.1 Algoritmy a měny

V aplikaci si uživatel nastaví, jaké měny bude chtít těžit. Každá měna je založena na konkrétním algoritmu. Jeden těžební klient může těžit pouze měny, které mají společný algoritmus. V databázi budou předdefinovány nejpoužívanější algoritmy a měny. Pokud bude uživatel chtít těžit jinou měnu, může si ji v systému vytvořit sám.

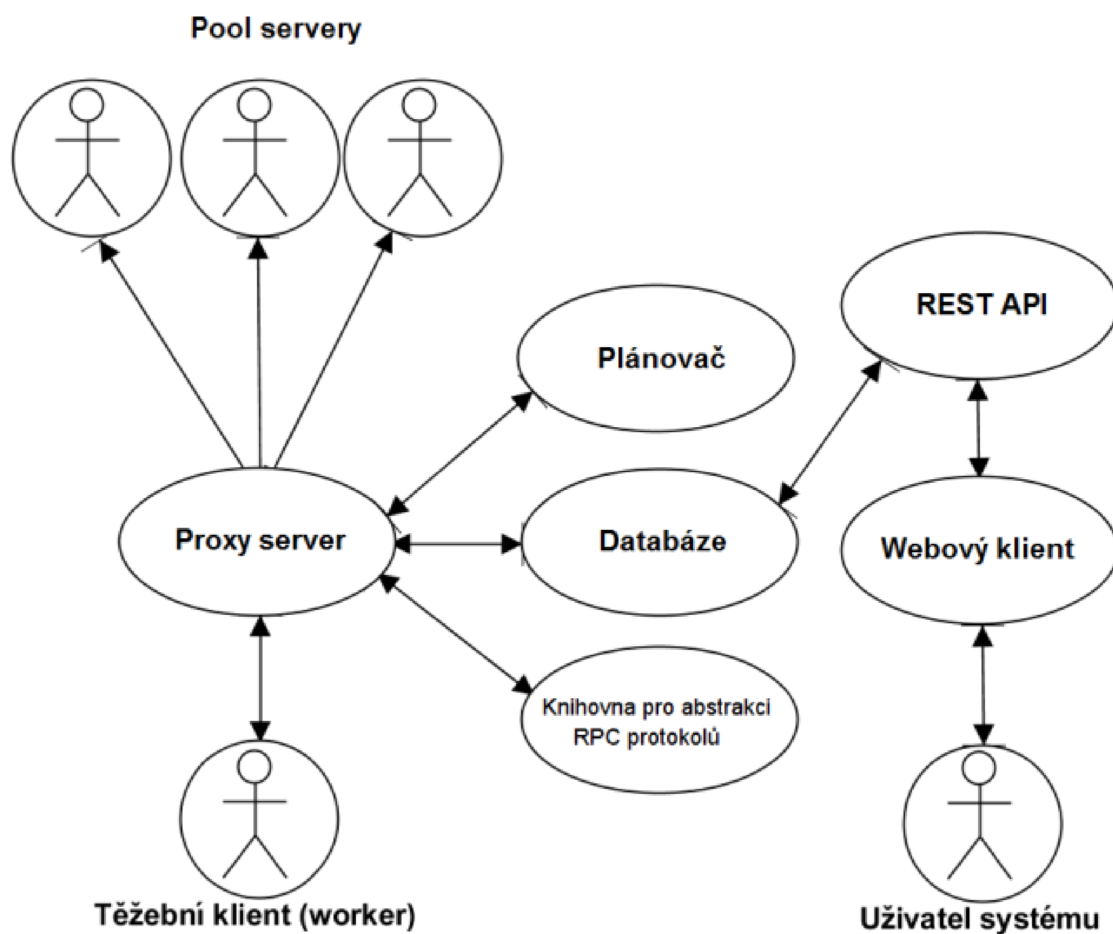
3.3.2 Pool servery

Následně si uživatel definuje všechny pool servery, ze kterých chce těžit. U každého nastaví měnu a autentizační údaje. Pro jednu měnu si uživatel může nadefinovat více serverů. V případě výpadku spojení se použije záložní server.

3.3.3 Těžební skupiny

Účelem těžebních skupin je vytvořit seznam měn, které budou klienti využívající takovou skupinu těžit. Každá skupina má definovaný algoritmus a libovolný počet měn. Uživatel si pro každou měnu zvolí, jaký procentuální podíl těžby jí bude věnován. Každá měna má v rámci skupiny přiřazenou prioritu. K měně je možné přiřadit více pool serverů. Do takto vytvořené těžební skupiny je možné zařadit i více těžebních klientů. Po připojení klienta dojde k nalezení těžební skupiny, do které je zařazen.

Klient bude připojen na první definovaný pool server měny s nejvyšší prioritou. Pokud by došlo k výpadku spojení u aktuálního pool serveru, bude klient přepojen na definovaný alternativní pool server. V závislosti na nastaveném rozdělení výkonu bude klient periodicky měnit těženou měnu. Přepínání mezi měnami bude probíhat v závislosti na nastavených prioritách. Pokud se při periodickém vyhodnocování statistických ukazatelů shledá těžba aktuální měny jako neefektivní, dojde k přepnutí na další měnu definovanou ve skupině.

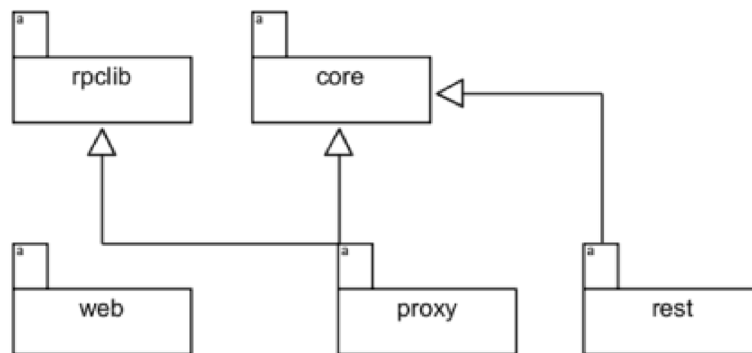


Obrázek 3.1: Schéma systému

4 Realizace systému pro optimalizaci těžby

Systém jsem realizoval v jazyce Java verze 8. V síťové části aplikace využívám frameworku Netty. Na správu závislostí jednotlivých modulů systému používám Maven. Pro dependency injection, REST API a periodické spouštění skriptů využívám frameworku Spring.

Abych logicky oddělil jednotlivé části aplikace, je projekt rozdělen na několik modulů. Jejich vzájemné propojení je znázorněné na obrázku 4.1. Základ tvoří modul `core`, kde je implementován model aplikace a servisní třídy pro mapování na databázová data. Knihovna pro práci s RPC protokoly je implementovaná v modulu `rpclib`. Další částí je modul `proxy`, který využívá oba předchozí moduly systému a je zde implementována veškerá logika proxy serveru. Kromě těchto modulů existuje v systému ještě `rest` modul starající se o REST API a `web` modul s javascriptovou klientskou aplikací.



Obrázek 4.1: Rozdělení systému na jednotlivé projekty

4.1 Knihovna pro práci s RPC protokoly

Jak jsem stanovil v kapitole 3.2.1, bude tato část systému realizovaná jako knihovna. Knihovna bude obsahovat rozhraní definující jednotlivé akce RPC protokolů. Konkrétní implementace těchto rozhraní budou knihovně předány externě. Aby se knihovna dala samostatně testovat, bude obsahovat výchozí implementace těchto rozhraní.

4.1.1 Princip činnosti knihovny

Knihovna funguje na principu proxy serveru. Pomocí `proxy` modulu se k ní připojí těžební klient. Knihovna příchozí zprávu analyzuje a přepošle ji na pool server. V knihovně se pracuje se dvěma kanály. Kanál mezi těžebním klientem a proxy serverem nazývám **`frontendovým kanálem`** a kanál mezi proxy serverem a pool serverem **`backendovým kanálem`**. Zpracování zprávy probíhá pomocí takzvaných **`handlerů`**. Každý z kanálů definuje svojí sadu **`handlerů`**, kterými zpráva postupně prochází [16]. Podle typu zprávy, kterou má **`handler`** zpracovávat, může být definován jako příchozí nebo jako odchozí. Životní cyklus zprávy přicházející od klienta je následující:

- Zpráva postupně projde všemi příchozími handlery umístěnými ve frontendovém kanálu.
- Následně je předána na backendový kanál, kde projde handlery určenými pro zpracování odchozí zprávy.
- Po průchodu posledním handlerem je poslána na proxy server.
- Odpověď následně projde všemi příchozími handlery zařazenými v backendovém kanálu.
- Poté je předána na frontendový kanál a projde všemi jeho odchozími handlery.
- Poslední odchozí handler frontendového kanálu se postará o poslání zprávy klientovi.

Všechny zprávy přijdou jako objekt datového typu `ByteBuf`. Ve frontě jsou proto umístěny handlery pro zakódování, respektive dekódování zprávy

z/do požadovaného formátu (v závislosti na směru zprávy). Některé zprávy navíc mohou být fragmentované na několik částí. Poté je nutné použít agregační handler, který zachycuje jednotlivé fragmenty zprávy a teprve když je zpráva kompletní, předá ji dalšímu handleru.

Každý z handlerů může zprávu modifikovat a na základě své logiky rozhodnout, jestli zprávu předá ke zpracování dalšímu handleru, zahodí ji, nebo ji rovnou zašle zpět klientovi. Zařazení konkrétních handlerů je znázorněno na obrázku 4.2.

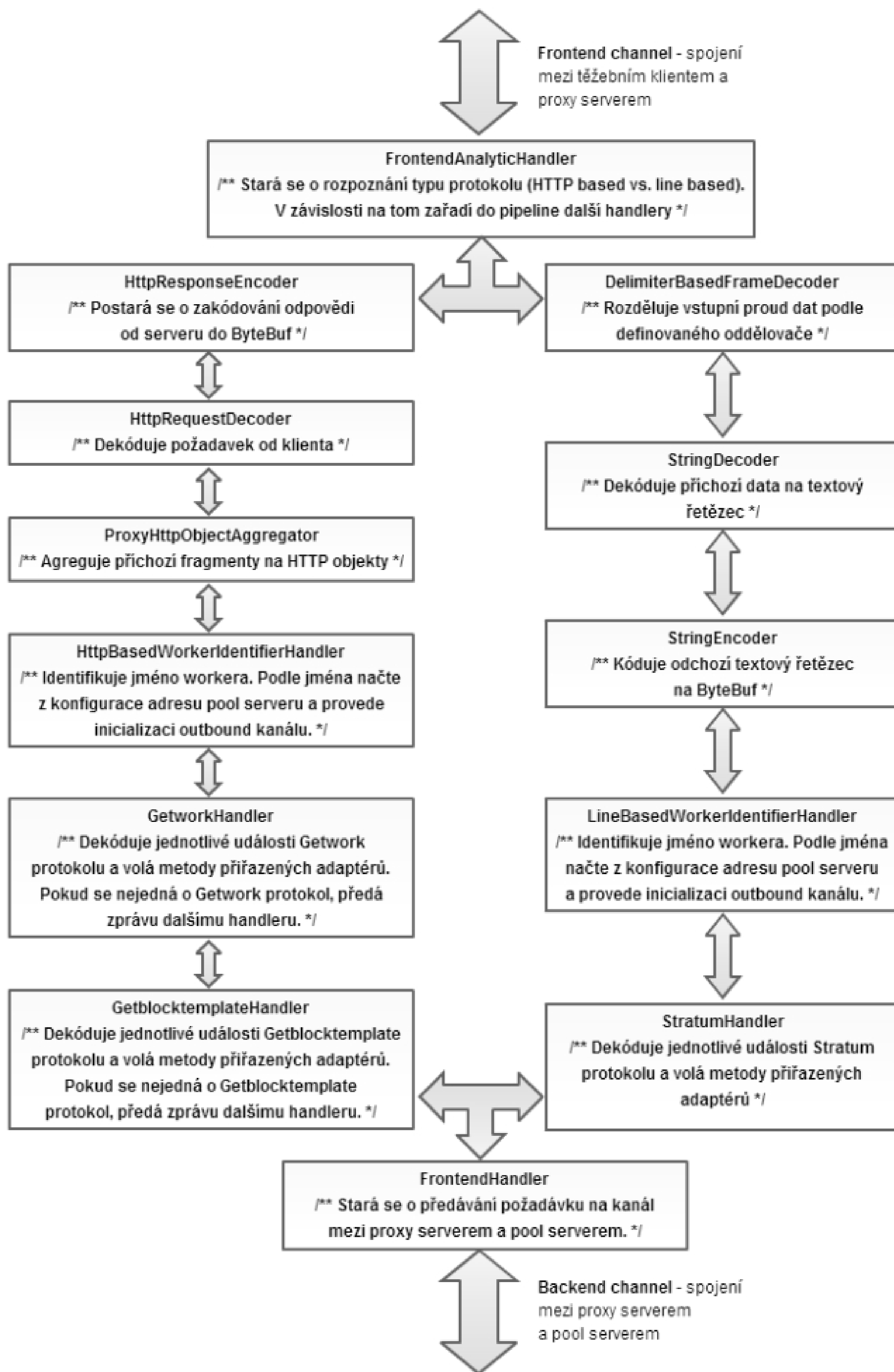
Jako první je ve frontě zařazen handler starající se o rozpoznání typu RPC protokolu. Na základě jeho výstupu dojde k zařazení dalších handlerů pro konkrétní typ protokolu. Ty se nejdříve postarají o převedení zprávy z typu `ByteBuf` na čitelný formát. Dále přijde na řadu handler pro autentizaci těžebního klienta. Pokud tento proces proběhne úspěšně, je zpráva předána handleru určenému k identifikaci události RPC protokolu. Nakonec zpráva putuje do handleru, který ji předá na backendový kanál, kde dojde k jejímu přeposlání na pool server.

Aby mohl proxy server zpracovávat i odpovědi pool serveru, jsou tyto handlery zařazeny i v backendovém kanálu. Jejich instance je tedy sdílená pro oba kanály. Pro tyto účely vyžaduje Netty framework takto používané handlery označit pomocí anotace `@Sharable`.

4.1.2 Identifikace RPC protokolu

Všechny RPC protokoly komunikují pomocí zpráv zasílaných ve formátu JSON. Liší se však v komunikačním kanálu, který pro výměnu těchto zpráv využívají. Podle využívaného protokolu je můžeme rozdělit na dvě základní skupiny. První skupinu tvoří protokoly Getwork a Getblocktemplate (viz kapitoly 2.9.1 a 2.9.2), které komunikují pomocí HTTP protokolu. Protokol Stratum (viz kapitola 2.9.3) využívá ke komunikaci TCP socket, po kterém posílá jednotlivé textové zprávy oddělené znakem konce řádky. Další zpracování zprávy tedy závisí na typu RPC protokolu. U každé zprávy proto musíme nejdříve rozhodnout, o jaký typ protokolu se jedná.

Z toho důvodu je na začátku fronty handlerů umístěn `FrontendAnalyticHandler`. Ten se postará o rozpoznání typu protokolu a na základě toho zařadí do fronty další handlery. Ty se postarají o dekódování zprávy z objektu `ByteBuf` do čitelného formátu. U RPC protokolů založených na HTTP protokolu se jedná o instanci třídy `DefaultFullHttpRequest` pro příchozí požadavky, respektive o instanci třídy `DefaultFullHttpResponse` pro odpovědi.



Obrázek 4.2: Handlery umístěné ve frontě frontendového kanálu

U protokolu Stratum se jedná o objekt datového typu `String`.

Na základě identifikace typu protokolu se do fronty zařadí handlery specifické pro rozpoznaný protokol. Je-li rozpoznán protokol komunikující pomocí HTTP, je ještě potřeba rozlišovat mezi protokoly Getwork a Getblocktemplate. Abych nemusel v aplikaci definovat speciální sadu handlerů pro každý RPC protokol, využívají HTTP protokoly společnou frontu handlerů. Jejich zpracování je totiž, až na rozpoznání události RPC protokolu, naprosto shodné.

Do fronty se tak zařadí handlery `GetworkHandler` a `GetblocktemplateHandler`, které dědí od společného předka `AbstractRpcHttpBasedHandler`. V předkovi je implementována logika, která se postará o určení RPC protokolu. Pokud rozpoznaný protokol nesouhlasí s protokolem handleru, je zpráva předána dalšímu handleru bez jakéhokoliv zpracování. Zpráva je tím pádem vždy zpracována jen jedním handlerem RPC protokolu.

4.1.3 Autentizace uživatele

Těžební klient ve zprávě posílá přihlašovací údaje k pool serveru. Na základě těchto údajů ale nejsem schopný rozpoznat uživatelův účet v systému. Potřebuji proto, aby klient v těle zprávy zasílal autentizační řetězec vygenerovaný ve webovém rozhraní a název workera evidovaný u jeho účtu. Na základě těchto údajů jsem schopný klienta autentizovat a přiřadit mu uživatelský účet v aplikaci. Aby došlo ke korektnímu spojení s pool serverem, tak je nutné tyto údaje ve zprávě změnit na autentizační údaje pool serveru. Ty jsou načteny z databáze, kde si je pro jednotlivé pool servery definuje uživatel.

Knihovna pracuje s rozhraním `Authenticator`, které definuje metodu pro autentizaci klienta a načtení přihlašovacích údajů k pool serveru. Implementace, která načítá data z databáze, je umístěna v `proxy` projektu. Knihovna obsahuje pouze implementaci s testovacími daty.

Identifikace jména a hesla u protokolů, které komunikují pomocí HTTP je celkem snadná. Autentizace klienta probíhá pomocí `basic access authentication`. Jméno a heslo je tudíž přenášeno v hlavičce HTTP zprávy. Pro zjištění jména a hesla je potřeba přečíst hodnotu hlavičky a provést base 64 dekódování. Tím získám řetězec, ve kterém je jméno a heslo oddělené dvojtečkou.

U protokolu Stratum je ale identifikace jména workera mnohem obtížnější. Navázání spojení mezi klientem a pool serverem probíhá následovně:

- Klient zašle zprávu volající metodu `mining.subscribe`.
- Pool server zašle inicializační údaje, aby klient mohl začít počítat.
- Teprve poté zašle klient zprávu s autentizačními údaji.
- Pool server potvrdí, nebo zamítne autentizaci.

Zde narážím na problém. Proxy server by totiž potřeboval první zprávu přeposlat na pool server. Bohužel v tomto okamžiku ještě nezná jméno workera a nezná tím pádem ani URL adresu pool serveru. Proxy server v tomto případě postupuje následovně:

- Po obdržení inicializační zprávy od klienta mu proxy server odešle „fiktivní odpověď“ a původní zprávu si uloží.
- Od klienta obdrží autentizační zprávu, kterou si rovněž uloží.
- V této chvíli zná jméno workera, naváže tedy spojení s pool serverem.
- Po úspěšném spojení s pool serverem mu přepošle první zprávu od workera.
- Odpověď od pool serveru přepošle klientovi, aby začal počítat reálný blok.
- Následně proxy server přepošle na pool sever také druhou (autentizační) zprávu, aby došlo ke správné asociaci mezi pool serverem a workerem.
- Dále již komunikace probíhá standardně, proxy server tedy překládá veškeré požadavky mezi klientem a pool serverem.

Kromě takto získaného jména a hesla potřebuji u každého těžebního klienta znát i další informace. Z tohoto důvodu se pro každého autentizovaného klienta vytvoří instance objektu `WorkerInternalName`. Ten obsahuje číslo portu a adresu aktuálně využívaného pool serveru. Kromě těchto údajů uchovává také instanci objektu `WorkerInternalName`. V něm je zapouzdřeno jméno těžebního klienta, jeho identifikační řetězec, identifikátor účtu, název těžební skupiny a aktuálně těžená měna.

4.1.4 Handlery RPC protokolů

U každé zprávy dojde k identifikaci RPC protokolu a podle toho se rozhodne, který handler ji bude zpracovávat. Ke každému protokolu náleží právě jeden handler. Jedná se o handlery `GetworkHandler`, `GetblocktemplateHandler` a `StratumHandler`. Z každé příchozí zprávy se nejdříve přečte IP adresa a číslo portu. Následně dojde k parsování těla zprávy. JSON zpráva přenášená ve formě textového řetězce se převede do objektové reprezentace, se kterou dále pracuji. K této transformaci využívám knihovny `FasterXML/jackson`. Ta na základě mé rešerše [17] vykazovala z dostupných knihoven nejlepšího výkonu. Rychlé zpracování zprávy je totiž pro účely mého systému velmi důležité.

Následně jsou analyzovány jednotlivé atributy JSON zprávy. Na jejich základě se rozhodne o události RPC protokolu a jejích parametrech. Handler má k dispozici sadu adaptérů. Po dokončení analýzy přijaté zprávy zavolá na každém z adaptérů příslušnou metodu, které předá načtená data. Handler se stará i o počítání doby odpovědi pro každý požadavek. Tento časový údaj je také předáván do adaptérů.

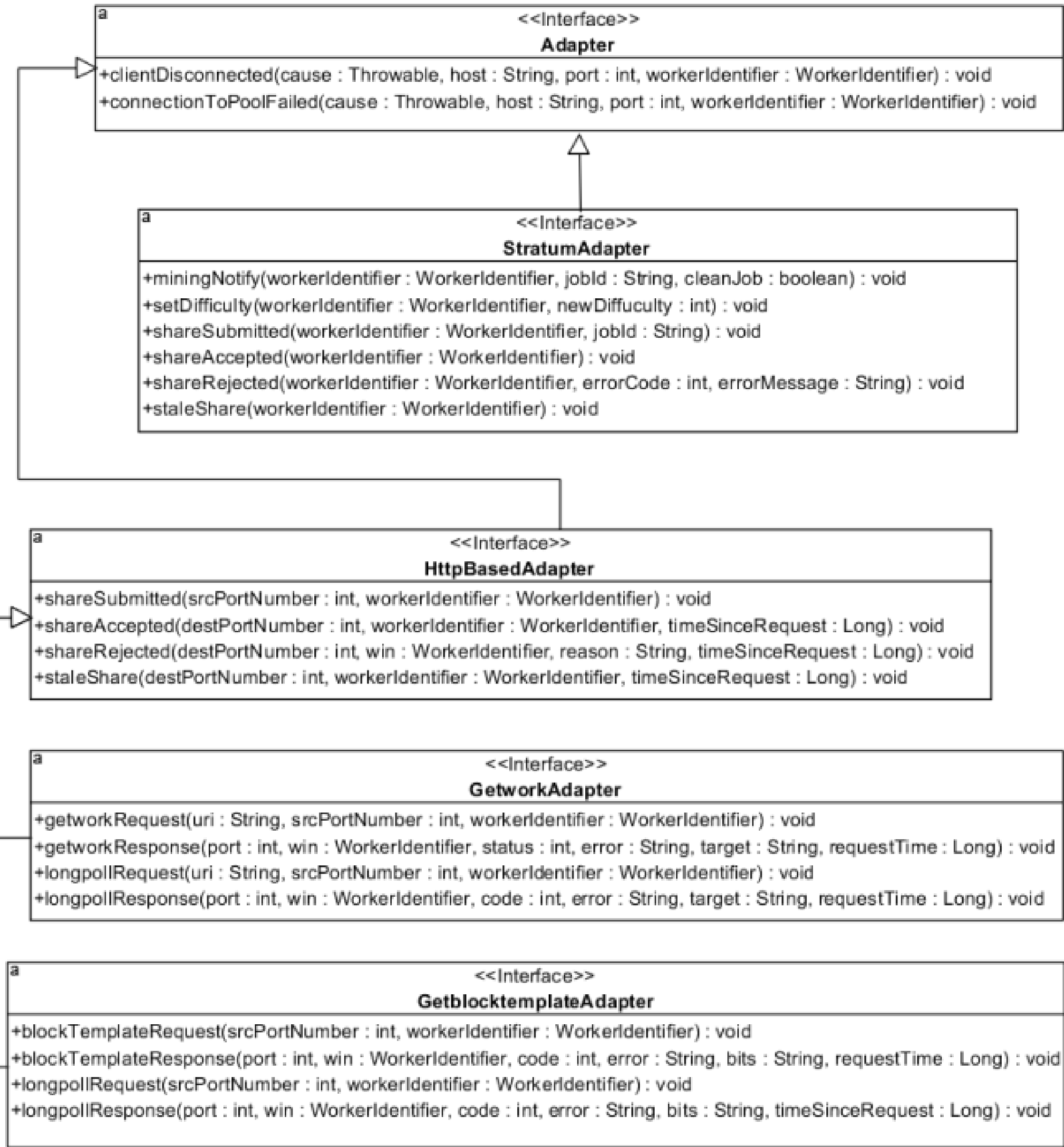
Cílem adaptérů je abstrahovat jednotlivé události RPC protokolů. Pro každý protokol existuje interface, který tyto události definuje. Hierarchie rozhraní je zobrazena na obrázku 4.3.

Pro jeden protokol je možné definovat i více adaptérů. Operace prováděné v adaptérech musí být velmi rychlé. V případě jejich vyšší časové náročnosti by docházelo ke zpožďování komunikace mezi klientem a pool serverem. Z tohoto důvodu používá jejich implementace v `proxy` modulu asynchronní zápis do databáze.

Knihovna pro abstrakci RPC protokolů obsahuje pro každý z protokolů dvě výchozí implementace adaptérů. Ty opět slouží pouze k ověření funkčnosti této knihovny. První adaptér slouží k prostému výpisu zaznamenaných událostí do konzole. Druhý slouží k zachytávání statistických ukazatelů těžby, které si ukládá do paměti.

4.1.5 Přepojování klienta mezi pool servery

Abych mohl realizovat přepojování klientů mezi pool servery, musím si v aplikaci ukládat všechna vytvořená spojení. Pro každého klienta musím také ukládat adresu pool serveru, ke kterému je aktuálně připojen. Kromě toho potřebuji uchovávat všechny požadavky na přepojení klienta. Abych mohl řešit problém popisovaný v ka-



Obrázek 4.3: Hierarchie rozhraní adaptérů pro abstrakci RPC protokolů

pitole 3.2.3, musím si u každého klienta ukládat identifikátor posledního těženého bloku.

Ukládání těchto údajů závisí na konkrétním použití knihovny. Proto pro úložiště definuji rozhraní `Storage`. V tom se nachází metody pro čtení i zápis ukládaných dat. Knihovna obsahuje základní implementaci, která vše ukládá do operační paměti.

Vzhledem k tomu, že u mého systému počítám s možností jeho distribuce mezi více serverů, je potřeba tomu přizpůsobit i ukládání stavových dat aplikace. Může se totiž stát, že by se těžební klient v průběhu těžby připojoval k různým proxy serverům. Některé z těchto údajů tak musí být dostupné na všech serverech současně. Takto vyčleněné údaje bude proto potřeba ukládat do databáze. Data zapsaná do databáze se automaticky replikují mezi všechny servery, na kterých aplikace běží.

K tomuto účelu se v `proxy` modulu nachází implementace rozhraní `Storage`, která vybrané údaje ukládá do databáze. Mezi data ukládaná do databáze patří informace o pool serveru, ke kterému je těžební klient aktuálně připojen, informace o době těžby aktuální měny pro každého klienta a identifikátor posledního těženého bloku s vazbou na pool server. Ostatní stavové informace bude stačit ukládat pouze do operační paměti. Jsou totiž potřeba pouze na tom serveru, na kterém požadavek od klienta vznikl.

Kromě úložiště definuji v knihovně ještě rozhraní `ConnectionManager` pro správu spojení. To obsahuje metody pro vytvoření nového požadavku na přepojení klienta, okamžité přepojení klienta a přepojení na alternativní pool server v případě výpadku spojení. Jeho implementace se opět nachází v modulu `proxy`.

4.2 Úložiště nasbíraných dat

Jak jsem již zdůvodnil v kapitole 3.2.2, používám v aplikaci databázi Cassandra. Pro komunikaci mezi aplikací a databází používám `DataStax Java Driver` [18]. Přemýšlel jsem o využití některého z existujících frameworků pro mapování aplikačního modelu na databázová data. Tím by však vznikla poměrně vysoká režie a výsledné databázové dotazy bych neměl plně pod kontrolou. Rychlost zápisu a čtení dat je však pro efektivní chod mého systému klíčová. Rozhodl jsem se proto, že se pokusím mapovací vrstvu pojmout co nejefektivněji. Vytvořil jsem proto sadu servisních tříd, které se starají o mapování dat. S daty poté manipuluji pomocí dotazovacího jazyka CQL. Databázové dotazy tak mohu maximálně optimalizovat

a dosáhnu mnohem lepšího výkonu než při použití frameworku.

4.2.1 Verzování databáze

Po navržení aplikačního modelu je nutné vytvořit požadovanou strukturu v databázi. V průběhu vývoje aplikace se musí změny provedené v modelu synchronizovat s databází. Vzhledem k rozhodnutí, že nebudu pro mapování dat využívat žádný framework, si budu muset i tyto operace řešit sám. Rozhodl jsem se pro osvědčenou metodu, kdy spolu se změnami v modelu bude v příslušném `commitu` umístěna také migrace, která provede všechny potřebné změny v databázi. Tyto migrace budou psány v jazyce CQL.

Takto vytvářené migrace budou pojmenovány podle čísla verze. Číslo verze může také obsahovat název prostředí. Mohu tak definovat migraci s testovacími daty, která se provede pouze, pokud aplikace poběží ve vývojovém módu. Aktuální číslo verze bude uloženo v databázi.

Skript pro inicializaci databáze je definován ve třídě `DefaultDbInitializer`. Na začátku si z adresáře s migracemi načte všechny CQL soubory. Podle definovaného algoritmu seřadí čísla verzí od nejstarší po nejnovější. Čísla verzí mohou obsahovat i čísla podverzí (např.: 3, 3.1, 3.5.2, 3.5.2.10). Poté se pokusí načíst číslo aktuální verze z databáze. Pokud není nalezen žádný záznam (požadovaná databáze nebo tabulka neexistuje, popřípadě neobsahuje žádné záznamy), tak tuto situaci považuje za první spuštění inicializace databáze a budou provedeny všechny dostupné migrace. Když je nalezeno číslo aktuální verze, tak postupně provede všechny migrace novějších verzí. Nakonec se do databáze zapíše číslo verze poslední aplikované migrace.

V CQL skriptech je nutné používat různé hodnoty, které jsou závislé na aktuálním prostředí, ve kterém aplikace běží (například název databáze). Aby tyto údaje nebyly zadány přímo v migračních skriptech, zavedl jsem možnost používat proměnné. Skripty používají jednoduchý šablonovací systém, ve kterém je možné využívat definované proměnné.

4.2.2 Model aplikace

Třídy definující persistentní model aplikace se nachází v modulu `core`. Struktura těchto tříd odpovídá jednotlivým tabulkám v databázi. V každé třídě jsou definovány

jednotlivé položky a metody pro získání a nastavení jejich hodnot. Vzhledem k tomu, že při testování aplikace (více v kapitole 4.2.4) potřebuji mezi sebou porovnávat různé instance modelových tříd, jsou v každé třídě definovány metody `hashCode` a `equals`. Tyto metody se v jazyce Java starají o správné porovnávání objektů mezi sebou. Jinak modelové třídy neobsahují žádnou logiku.

V modulu zajišťující REST API, který popisuji v kapitole 4.4.1, jsou tyto třídy automaticky serializovány do JSON formátu. V některých případech ale není žádoucí, aby se serializovaly všechny položky entity. V takových případech využívám třídní anotace `JsonIgnoreProperties`. V té se definují proměnné, které se mají při serializaci objektu přeskakovat.

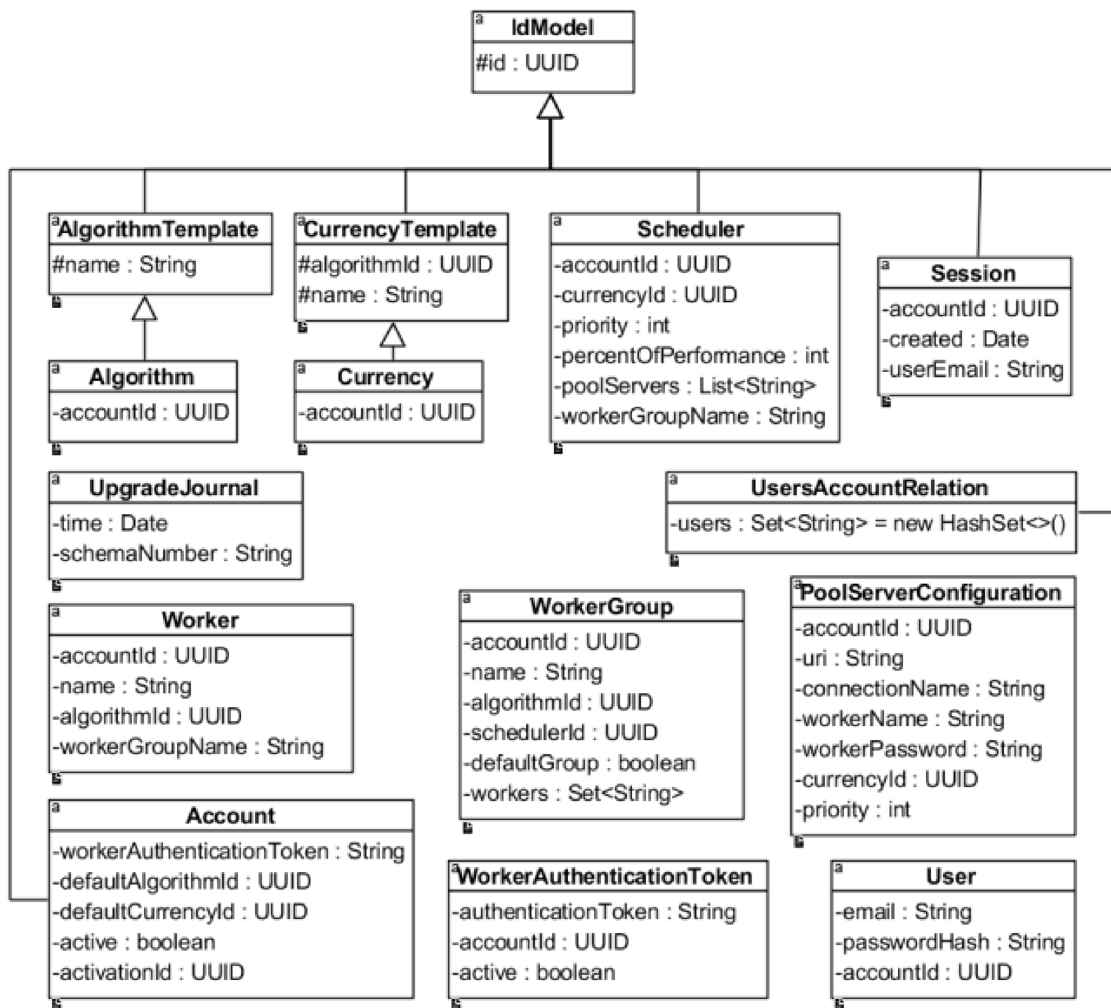
Jak jsem definoval v kapitole 3.2.2, data jsou logicky rozdělena ve třech databázích (`keyspaces`). Toto rozdělení se odráží i v aplikačním modelu. Modelové třídy jsou tudíž rozděleny ve třech balíčcích. Strukturu všech tříd v jednotlivých balíčcích znázorňuji pomocí class diagramů na obrázcích 4.4, 4.5 a 4.6.

4.2.3 Mapování dat

Mým cílem bylo vytvořit mapování tak, aby se již v servisní vrstvě co nejvíce minimalizovala závislost na konkrétní databázové struktuře. Chtěl jsem mít mapování zapouzdřené na jednom místě a izolované od zbytku aplikace. To v budoucnu usnadní veškeré úpravy v modelu dat a eliminuje to vzniky chyb. Kromě aplikačního modelu a servisních tříd pro samotnou manipulaci s databázovými daty, definuji v aplikaci navíc ještě jednu vrstvu určenou k mapování databázové struktury na model aplikace. Definují se tam názvy tabulek, názvy jednotlivých datových atributů a primární klíče tabulek.

Každá mapovací třída implementuje rozhraní `Mapper` a dědí od abstraktní třídy `AbstractMapper`, která je generická. Rozhraní definuje následující metody:

- `String getTableName()` - Vrací název tabulky v databázi.
- `Optional<T> mapObjectFromDbRow(Row row)` - Vytvoří databázový objekt z databázového výsledku.
- `Map<String, Object> getObjectColumnsWithValues()` - Vrací mapu s názvy datových položek tabulky a jejich hodnotami.



Obrázek 4.4: Struktura modelových tříd pro ukládání konfiguračních dat

IdModel
#id : UUID

StatisticsModel
-accountId : UUID
-workerName : String
-datetime : Date
-processed : boolean
-poolServerUri : String

JobDifficulty
-newDifficulty : String

JobNotification
-jobId : String
-cleanJob : boolean

StatisticsWithProtocolName
-protocolName : String

HttpLongpollRequest

HttpWorkRequest

StatisticsWithMessage
-message : String

ConnectionToPoolServerError

ConnectionToWorkerError

ProtocolErrorMessage
-code : int

StatisticsWithResponseTi...
-responseTime : int

SubmittedShare
-jobId : String

AcceptedShare

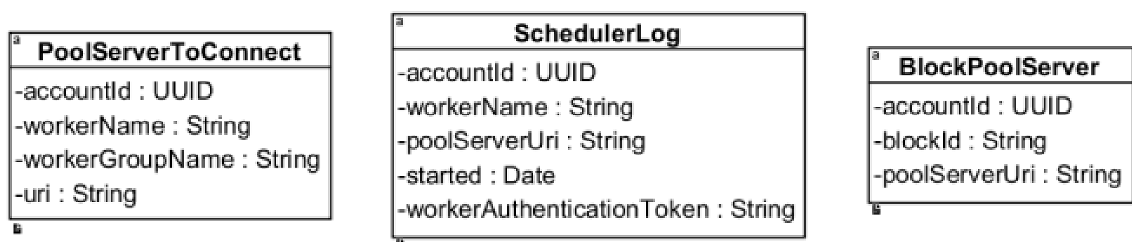
HttpLongpollResponse
-statusCode : int
-difficulty : String

HttpWorkResponse
-statusCode : int
-difficulty : String

RejectedShare
-errorCode : int
-errorMessage : String

StaleShare

Obrázek 4.5: Struktura modelových tříd pro ukládání statistických ukazatelů



Obrázek 4.6: Struktura modelových tříd pro ukládání stavových dat aplikace

- `String[] getPrimaryKeyNames()` - Vrací názvy atributů, které tvoří primární klíč záznamu.
- `Object[] getPrimaryKeyValues()` - Vrátí hodnoty atributů primárního klíče.
- `Object getColumns()` - Vrátí objekt definující strukturu databázové tabulky.

Této mapovací vrstvy využívají servisní třídy. Za pomoci mapovací vrstvy mohou manipulovat s daty, aniž by musely přímo používat názvy datových atributů tabulky. Každá servisní třída implementuje metodu, která vrací instanci rozhraní `mapper` pro konkrétní modelovou třídu.

Servisní třídy definují základní CRUD operace. U nich implementuji podporu pro vlastnosti databáze Cassandra. Těmi jsou možnost vložení záznamu pouze, pokud již neexistuje, možnost nastavení TTL nebo nastavení stupně konzistence dat. Umožňují také asynchronní zápis do databáze. Pro databázové dotazy používám `prepared statements`, pomocí kterých se samotný dotaz pošle do databáze pouze poprvé. Při jeho dalším volání se do databáze přenesou pouze parametry, které se do dotazu vkládají.

4.2.4 Testování servisní vrstvy

Mapovací vrstva mi zajišťuje vcelku pěknou abstrakci databázové struktury. Její hlavní nevýhodou je ale nutnost ruční synchronizace mapovacích tříd se strukturou databáze. Hlavně při úpravách aplikačního modelu hrozí, že se zapomene na databázovou migraci. Abych předešel vzniku takovýchto chyb, testuji všechny mapovací třídy pomocí integračních testů. Pro každou servisní třídu je napsán vlastní test, který na testovacích datech provede všechny CRUD operace. Tím se ověří bezchybná funkčnost mapování dat mezi databází a aplikačním modelem. Pokud servisní třída obsahuje i jiné metody určené pro konkrétní modelový objekt, píši testy i pro ně. Vzhledem k modulárnímu návrhu systému to považuji za velkou výhodu. Pokud projdou všechny testy správně, mohu předpokládat, že modul `core` poskytuje pro správná vstupní data správný výstup.

4.3 Proxy server

Modul `proxy` v sobě využívá dalších částí systému a to modulu `core` pro komunikaci s databází a modulu `rpclib` pro abstrakci RPC protokolů. Z jedné strany se k němu připojuje těžební klient, na druhé straně komunikuje s cílovým pool serverem. V `proxy` modulu je implementována veškerá aplikační logika, která se stará o ukládání a vyhodnocování statistických údajů těžby a správné přepojování těžebního klienta mezi jednotlivými pool servery. V rámci proxy serveru se spouští i plánovač (`scheduler`). Ten se stará o vyhodnocování efektivity těžby a o periodické přepojování klienta na základě procentuálního rozdělení výkonu. Na straně proxy serveru implementuji všechny potřebné rozhraní poskytované knihovnou pro abstrakci RPC protokolů.

Jsou zde implementovány jednotlivé adaptéry, které zpracovávají události RPC protokolů. V nich dochází k zápisu vybraných statistických údajů do databáze. Z výkonnostního hlediska musí být operace v adaptérech velmi rychlé. Využívám zde proto možnosti databáze Cassandra zapisovat data do databáze asynchronně. V tomto případě aplikace pošle dotaz do databáze a dále nečeká na jeho provedení a návrat výsledku. Může se tedy teoreticky stát, že se zápis nezdaří a aplikace o tom nebude informována. Vzhledem k velkému počtu dat, které se budou takto zaznamenávat, můžeme tuto skutečnost ignorovat. Jak popisuji v kapitole 3.2.4, nastavuji těmto záznamům hodnotu TTL. Po její expiraci dojde k automatickému smazání na straně databáze.

Hodnotu TTL definuji společně s dalším nastavením v konfiguračním souboru. Ten v aplikaci za pomoci frameworku Spring načítám a v kódu používám jednotlivé hodnoty jako proměnné. Díky tomu půjde aplikace snadno ladit a mohu používat různé konfigurace v závislosti na běhovém prostředí.

V proxy serveru dále implementuji rozhraní pro autentizaci. To zaslané přihlašovací údaje porovnává s údaji uživatelského účtu v databázi. Pokud je autentizace úspěšná, dojde k uložení identifikátoru uživatelského účtu a údajů o těžební skupině klienta do objektu `WorkerInternalName`. Tyto údaje jsou potřeba při dalším zpracovávání požadavků a uchováváním v paměti ušetřím zbytečné databázové dotazy.

Další implementovanou částí je rozhraní `Storage`. To ukládá část dat do databáze a část do operační pamětí (více o tomto rozdělení v kapitole 4.1.5). Úložiště dat počítá s tím, že k němu bude současně přistupovat několik procesů. Metody pro

zápis a čtení dat jsou proto synchronizované.

Z knihovny `rpclib` v `proxy` modulu implementuji ještě rozhraní `ReconnecApi` pro přepojování klientů. Jeho implementace se stará o vytvoření požadavku na přepojení klienta za pomoci rozhraní `ConnectionManager`, jehož logika se také nachází v proxy serveru.

4.3.1 Výpadky spojení

V případě výpadku spojení s cílovým pool serverem se má proxy server postarat o okamžité přepojení na alternativní pool server. Z tohoto důvodu se v proxy serveru nachází implementace rozhraní `ConnectionErrorListener`, které je definované v knihovně `rpclib`. Ke své činnosti potřebuje mít přístup k adaptérům RPC protokolů, správci spojení a servisním třídám načítajícím data z databáze. V případě selhání spojení je v `rpclib` zavolána metoda `connectionFailed`. Té jsou předány veškeré informace o těžebním klientovi a pool serveru. Nejdříve se rozhodne, jestli se jedná o přerušení spojení mezi klientem a proxy serverem, nebo výpadek mezi proxy serverem a pool serverem. V obou případech se zavolá příslušná metoda na adaptérech. V nich dojde k zaznamenání informace do databáze podobně jako při ostatních událostech RPC protokolů.

Pokud se jedná o výpadek spojení k pool serveru, postará se dále `ConnectionErrorListener` o přepojení na alternativní pool server. Z databáze se nejdříve musí načíst informace o těžební skupině, ve které je worker umístěn. Podle pool serveru známého z informací o výpadku spojení se zjistí aktuálně těžená měna. Proxy server se snaží o dodržení uživatelem stanoveného rozdělení výkonu mezi jednotlivé měny. Proto se nejdříve pokusí nalézt alternativní server, který si uživatel definoval pro aktuálně těženou měnu.

Pokud není nalezen žádný alternativní server, dojde k přepnutí těžby na jinou měnu. Měny jsou definovány v rámci těžební skupiny a jsou jim přiřazeny priority. Dojde tedy k přepojení na měnu, která je podle definovaných priorit další na řadě. Pokud je aktuálně těžená měna poslední v pořadí, dojde k přepnutí zpět na první měnu. Střídání měn tedy funguje na principu cyklického zásobníku.

Pokud má uživatel definovanou pouze jednu měnu bez alternativních pool serverů, bude pokus o připojení na aktuální pool server opakován. Toto řešení není dokonalé a spoléhá na správnou konfiguraci ze strany uživatele. Do budoucna bych chtěl systém obohatit o globální definici pool serverů. Pokud by uživatel neměl defi-

novaný žádný alternativní pool server, na který by se mohl klient připojit, došlo by k připojení na jeden z globálních pool serverů. Odměna z globálních pool serverů by se poté dělila mezi uživatele, kteří na nich těžili.

4.3.2 Plánovač

Plánovač má v systému dva základní úkoly. Stará se o procentuální rozdělení výkonu mezi jednotlivé měny a řeší analýzu probíhající těžby a vyhodnocování její efektivity. Jeho výstupem může být vytvoření žádosti o přepojení klienta na jiný pool server.

V aktuální verzi aplikace je plánovač implementován v rámci `proxy` modulu. Zde má k dispozici všechny potřebné části ke své činnosti. Do budoucna uvažuji o jeho přestěhování do samostatného modulu aplikace. Mohl by se poté spouštěn na jiném serveru, čím bych dosáhl možnosti lepšího škálování výkonu systému.

Spouštění plánovače probíhá periodicky v pevně definovaném intervalu. K periodickému spouštění využívám frameworku Spring. Ve třídě `SchedulerConfigurer`, která implementuje rozhraní `SchedulingConfigurer`, nastavuji spouštění plánovače v intervalu definovaném v konfiguračním souboru. Pravidelně je volána metoda `execute` třídy `SchedulerExecutor`. Odtud se postupně zavolají metody definované v rozhraní `Scheduler`. To obsahuje dvě metody: `reconnectIneffectiveConnection(SchedulerLog schedulerLog)` a `handlePerformanceSplitting(SchedulerLog schedulerLog)`. Ty implementuji ve třídě `DefaultScheduler`. Obě mají návratový typ `boolean`. Vrací hodnotu `true`, pokud vytvořily požadavek na přepojení klienta. V opačném případě je vrácena hodnota `false`.

Pravidelně prováděný skript si z databáze nejprve načte informace o všech aktivních klientech. Pro každého z nich poté postupně zavolá metody definované v rozhraní `Scheduler`. Nejdříve je volána metoda pro vyhodnocení efektivity těžby. Pokud v rámci této metody nedojde ke vzniku žádosti o přepojení, zavolá se metoda určená k rozdělení výkonu.

4.3.3 Výpočet efektivity spojení

Prvně jmenovaná metoda se postará o analýzu nasbíraných statických ukazatelů probíhající těžby. Na jejich základě spočítá skóre. Pokud takto spočítané skóre nepřesáhne mezní práh, je těžba považována za neefektivní a dojde k vytvoření

požadavku na přepojení klienta. Skóre počítám z celkového počtu potvrzených, zamítnutých a prošlých výsledků výpočtů. V anglickém názvosloví se jedná o hodnoty `accepted share`, `rejected share` a `stale share`. Každé z hodnot přiřazuji jinou váhu. Nejvyšší přidělenou váhu má samozřejmě schválený výsledek. Naopak nejnižší nastavenou váhu má prošlý výsledek. Počet výskytů jednotlivých hodnot roznásobím definovanou váhou a sečtu je. Pokud mi výsledný součet vyjde záporně, považuji těžbu za neefektivní.

K tomuto výpočtu jsem dospěl úvahami podloženými testováním systému v různých modelových situacích (více v kapitole 4.5). Nejnižší váhu jsem přiřadil situaci, kdy pool server na výpočet pošle odpověď, že je výpočet již zastaralý. To v praxi znamená, že výsledek výpočtu zaslal na pool server již někdo dříve. K těmto situacím může docházet relativně často, a pokud se občas povede zaslat výpočet s ověřenou odpovědí, tak těžbu nemůžeme považovat za neefektivní. Vyšší váhu si však vysloužila odpověď o zamítnutém výpočtu. Pod touto odpovědí se totiž může skrývat velké množství chyb ve výpočtu. Největší koeficient má přiřazena odpověď potvrzující obdržení správného výpočtu. Její přijetí tedy do jisté míry kompenzuje ostatní negativní odpovědi. Výsledné přidělené váhy jsou následující:

- potvrzení výsledku (`accepted share`): 1,5
- zamítnutí výsledku (`rejected share`): 0,9
- prošlý výsledek (`stale share`): 0,7

Původně jsem chtěl do výsledného skóre zahrnout i počet výpadků spojení. Nakonec jsem se ale rozhodl, že bude efektivnější klienta při výpadku rovnou přepojit na jiný pool server. Z tohoto důvodu by počet výpadků spojení ve výsledném skóre neměl žádný význam.

Tato část systému je poměrně klíčová pro výslednou efektivitu těžby. Je tedy možné, že se bude v budoucnu ještě ladit na základě získaných zkušeností z dlouhodobé reálné těžby. Na základě těchto zkušeností se mohou upravit jednotlivé váhy, popřípadě i samotný výpočet skóre. Vždy se ale bude jednat o jistý kompromis, který nemusí vyhovovat každému. V budoucnu možná vznikne několik scénářů pro výpočet skóre, mezi kterými si bude vybírat sám uživatel. K těmto závěrům ale potřebuji zkušenosti z dlouhodobého provozu systému, které v současné době ještě nemám.

4.3.4 Rozdělování výkonu

Uživatel má možnost si rozdělit výkon svého těžebního stroje mezi více měn. Každé měně přiřadí určitou část výkonu definovanou v procentech. Ke splnění tohoto požadavku potřebuji jeden základní předpoklad: znát dobu, po kterou klient těží aktuální měnu. Toho jsem docílil tak, že při každém přepojení těžebního klienta vytvořím záznam ve stavových datech aplikace. Kromě identifikačních údajů o klientovi a cílovém pool serveru si uložím hlavně čas začátku těžby. Tento záznam vytvářím pouze, pokud dojde k přepojení na jinou měnu. Pokud je klient v rámci výpadku spojení přepojen na alternativní sever, který ovšem těží stejnou měnu, nemá to na rozdělení výkonu žádný vliv.

V konfiguračním souboru si definuji časové kvantum, které považuji za 100% výkonu. Od této jednotky se poté odvíjí doba těžby jednotlivých měn. Časová jednotka musí být dostatečně dlouhá. V opačném případě by docházelo k příliš častému přepojování klienta, které by mělo za následek sníženou efektivitu těžby. Jako základní jednotku jsem zvolil hodnotu 100 minut. Do budoucna uvažuji o tom, že by se nejednalo o konstantu, ale každý uživatel by si tuto hodnotu mohl sám nastavit. Systém by definoval pouze rozmezí maximální a minimální hodnoty.

V metodě `handlePerformanceSplitting` se nejdříve z databáze načtou informace o těžební skupině a jejích měnách. Zjistí se aktuálně těžená měna a její procentuální podíl. Z toho se spočítá maximální doba v minutách, po kterou může těžba probíhat. Dále se z databáze načte, kdy klient začal tuto měnu těžit a spočítá se doba probíhající těžby. Doba těžby se následně porovná se spočítanou maximální dobou. Pokud došlo k překročení maximální délky těžby, je vytvořen požadavek na přepojení klienta.

Při takto jednoduchém porovnání obou časů by mohlo dojít k tomu, že doba těžby sice ještě nepřesáhla maximální povolenou dobu, ale už se jí velmi blíží. K přepojení podle přesného výpočtu by tedy mělo dojít v řádu několika desítek sekund. K periodickému spouštění plánovače ale dochází jednou za 5 minut. K přepojení by tedy došlo až při jeho dalším spuštění a uživatelovo rozdělení výkonu by nebylo příliš přesné. Z tohoto důvodu definuji v konfiguraci hodnotu diference o hodnotě 100 sekund. Reálná doba těžby je tak prodloužena o tuto diferenci. Pokud by přepojení klienta mělo nastat během následujících 100 sekund, přepojí se klient okamžitě. Tím dojde ke zpřesnění rozdělení výkonu.

Nabízí se otázka, proč diferenci nezvolit jako polovinu intervalu spouštění

plánovače. Poté by ovšem celkem často docházelo k situaci, kdy by se měna těžila po kratší dobu, než bylo definováno. Někteří uživatelé by tento jev mohli vnímat negativně. Do budoucna bude asi ideální přemístit tuto konstantu také do uživatelem konfigurovatelné hodnoty.

Při přepojení klienta na jinou měnu je záznam v databázi s časem začátku těžby přepsán novou hodnotou. V databázi tak nedochází k hromadění nepotřebných záznamů, ale každý aktivní klient má pouze jeden záznam. Aby nedocházelo k uchovávání záznamů již neaktivních těžebních klientů, je u každého záznamu nastavena hodnota TTL. Ta musí být větší než časové kvantum pro rozdělování výkonu.

4.4 Webové rozhraní pro konfiguraci systému

Abych uživatelům umožnil snadné nastavení celého systému, vytvořil jsem webové rozhraní. To jsem postavil na architektuře REST API, čemuž odpovídá i logické uspořádání v projektu. Server, který poskytuje REST API, implementuji v modulu `rest`. Webový kontejner s javascriptovým klientem se nalézá v modulu `web`. Oba kontejnery se poté spouští společně v rámci webového serveru Tomcat. Webový klient se serverem poskytujícím REST API komunikuje pomocí AJAXových požadavků po šifrovaném kanálu pomocí HTTPS protokolu.

Při návrhu REST API jsem využil české služby Apiary. Pomocí této služby jsem navrhl všechny metody včetně jejich parametrů a návratových hodnot. Služba poté z těchto dat automaticky generuje dokumentaci [19]. Hlavním přínosem ale je, že Apiary umožňuje simulovat REST server. Poskytuje webovou adresu, na které je možné volat všechny definované metody REST API. Odpovědí na toto volání budou data v JSON formátu, které jsou definovány v dokumentaci API. Při použití této služby je možné vyvíjet webového klienta naprosto odděleně od samotného REST serveru. Po navržení API v této službě jsem se tedy pustil do vývoje webového klienta, kterého jsem za pomoci této služby testoval. Poté jsem se pustil do implementace REST serveru, přičemž jsem postupoval podle dokumentace v Apiary.

4.4.1 REST API

REST server jsem implementoval za použití frameworku Spring, konkrétně jeho části `spring-webmvc`. Vzhledem k nutnosti používat databázi, využívá tato část aplikace

modulu `core`. API je rozvrženo podle jednotlivých modelových entit. Jako první parametr URL adresy je použit název entity. Na této adrese jsou poté definovány všechny metody vztahující se k dané entitě. Tomuto rozdělení odpovídá i návrh REST serveru. Každé modelové entitě používané v API odpovídá jeden `controller`, ve kterém jsou definované všechny metody pracující s danou entitou.

Každá třída reprezentující `controller` je označena anotací `RestController`. Webová adresa, na kterou bude třída namapována, se určí pomocí anotace `RequestMapping`. Stejná anotace se používá i pro jednotlivé funkce třídy. Tím se definuje typ HTTP metody a URL adresa včetně parametrů. Pomocí vstupních proměnných funkce je umožněn přístup ke všem parametrům REST metody, tělu HTTP požadavku i jeho hlavičkám. Návratová hodnota funkce je definována pomocí generické třídy `ResponseEntity`. Ta udává návratový HTTP kód a data. Ta jsou automaticky serializována do JSON formátu a poslána v těle odpovědi.

Při implementaci REST API jsem musel řešit autentizaci uživatele. Podobně jako u klasické webové aplikace je uživatel nejdříve vyzván k přihlášení do aplikace a přihlašovací údaje jsou zaslány na server. Ten je ověří s údaji v databázi. Pokud je autentizace úspěšná, tak server vygeneruje bezpečnostní řetězec (token), který vrátí v těle odpovědi klientovi. Tímto bezpečnostním klíčem se poté uživatel musí prokazovat [20]. Každý dotaz mířící na server musí tento klíč obsahovat ve své hlavičce. Klíč je na straně serveru ověřován a na jeho základě se zjišťuje uživatelský účet. Kontroluje se také, jestli se uživatel dotazuje na data, která patří k jeho účtu.

Vygenerovaný bezpečnostní klíč si server uloží do databáze spolu s vazbou na uživatelský účet a časem vytvoření. Z bezpečnostních důvodů jsem chtěl zajistit automatické odhlášení uživatele v případě jeho nečinnosti. Vytvářenému záznamu v databázi proto nastavuji hodnotu TTL. Při každém ověření uživatelského požadavku je tento záznam aktualizován. Tím dojde k prodloužení času jeho životnosti na původní hodnotu TTL. Pokud doba mezi dvěma požadavky přesáhne stanovenou hodnotu, bude záznam automaticky odstraněn a uživatel odhlášen.

Kromě tohoto bezpečnostního opatření jsem chtěl také omezit následky případné krádeže uživatelského klíče. Pokud by zůstal bezpečnostní kód po celou dobu aktivity uživatele stejný, získal by potencionální útočník v případě jeho krádeže plný přístup k aplikaci. Z tohoto důvodu řeším automatické přegenerování klíče. Při ověření požadavku dojde také ke kontrole stáří bezpečnostního klíče. Pokud jeho stáří přesáhne stanovenou hranici, dojde k vygenerování nového klíče. Z tohoto důvodu se bezpečnostní řetězec posílá také v hlavičce každé odpovědi serveru. Kli-

ent si po přijetí odpovědi svůj klíč aktualizuje a ten dále používá.

Po přegenerování klíče je také nutné zneplatnit starý klíč. Může se však stát, že klient pošle několik požadavků v rychlém sledu za sebou. V takovém případě by při zpracování prvního z požadavků došlo k vygenerování nového klíče a smazání toho starého. Ostatní požadavky by ale ve své hlavičce obsahovaly ještě starý klíč, který by v době jejich zpracování již neexistoval. Tato situace by skončila odhlášením uživatele z aplikace. Proto klíč nemažu přímo, ale provedu jeho úpravu s nastaveným TTL na hodnotu 5 sekund. To je doba dostatečně dlouhá pro vyřízení všech nahromaděných požadavků a po jejím uplynutí se záznam automaticky smaže.

4.4.2 Webová aplikace

Pro tvorbu webového klienta jsem využil javascriptového frameworku React. Aplikace je tedy kompletně napsána v Javascriptu a HTML. Pro design aplikace používám CSS framework Semantic UI.

Framework React přichází s trochu odlišným přístupem ke stavbě aplikace. Výsledná stránka se sestavuje z jednotlivých komponent, které fungují samostatně a uchovávají si svůj vnitřní stav. Framework využívá technologii virtuálního objektu `DOM`, díky kterému dosahuje vysokého výkonu. Při tvorbě komponent se používá `JSX` syntaxe. Ta přináší možnost kombinovat HTML zápis s javascriptovým kódem. Pomocí kompilátoru se poté HTML elementy převedou na DOM objekty zapsané v Javascritpu.

Po načtení indexového HTML souboru je vše v režii Javascriptu. Při přechodu mezi jednotlivými požadavky nedochází k novému načtení stránky v prohlížeči. Překreslování stránky probíhá na základě změny vnitřních stavů aplikace. Jediná komunikace se serverem probíhá pomocí AJAXových požadavků na REST API. K routování používám plugin, který zajišťuje změnu adresy v prohlížeči.

Prohlížeč si nejprve stáhne soubor `index.html`, ve kterém je nalinkovaný soubor `App.js`. Ten již obsahuje React komponentu. Framework automaticky zavolá metodu `render` na této komponentě. V ní se na základě aktuální URL rozhodne, jaký obsah se bude vykreslovat. Nejsou-li v URL žádné parametry, vytvoří se komponenta reprezentující domácí stránku. Pokud URL obsahuje další parametry, bude se vykreslovat část aplikace vyžadující autentizaci. Z tohoto důvodu se z `cookie` zjistí, jestli je uživatel přihlášen. Do `cookie` si totiž aplikace ukládá bezpečnostní klíč vygenerovaný serverem po autentizaci. Pokud takový záznam neexistuje, vy-

kreslí se přihlašovací formulář. V ostatních případech si aplikace za pomoci routeru zjistí název komponenty reprezentující aktuální cestu v URL a tu vykreslí. Veškerá další logika se řeší v konkrétních komponentách.

Kromě React komponent, ze kterých se sestavují jednotlivé stránky, obsahuje aplikace ještě servisní vrstvu pro komunikaci s REST API. V této vrstvě je abstrahováno REST API a je zde obsažena logika pro autentizaci požadavků posílaných na server. Do hlavičky každého požadavku se automaticky vkládá autentizační klíč. Po přijetí odpovědi se klíč zaslaný serverem uloží do `cookie`. Pokud server zašle odpověď s hlavičkou `401 unauthorized`, dojde k odstranění hodnoty z `cookie`. Uživateli je následně zobrazen formulář pro přihlášení.

4.5 Testování systému

Jednou z obtíží, se kterou jsem se při vývoji systému potýkal, bylo jeho testování. Nejdříve jsem se zaregistroval na několika pool serverech. Vybíral jsem při tom měny, které mají co nejnižší obtížnost. K dispozici jsem totiž měl pouze běžný notebook, takže i těžba měn s relativně nízkou obtížností byla velmi zdlouhavá. Průběžné testování systému za pomoci reálné těžby se tak ukázalo jako velmi neefektivní.

4.5.1 Testovací prostředí

Rozhodl jsem se proto pro vytvoření emulátoru pool serveru. K jeho tvorbě jsem podobně jako při tvorbě knihovny pro abstrakci RPC protokolů využil síťový framework Netty.

Emulátor definuje pro každý ze tří RPC protokolů číslo portu, na kterém poslouchá. Pro každý protokol existuje v emulátoru zvláštní handler. Ten přijatou zprávu zpracuje a určí příslušnou událost RPC protokolu. Po dekódování zprávy vytvoří fiktivní odpověď, kterou pošle klientovi zpět. Klientovi je zaslán blok k ověření s velmi nízkou složitostí. Těžební klient je tak schopný relativně rychle spočítat výsledek, který emulátoru pošle ke kontrole. Tím dosáhnu dostatečně vysoké frekvence výměny zpráv mezi klientem a pool serverem, kterou potřebuji k testování mého systému. Emulátor se také stará o periodické generování nových bloků k ověření. Jako odpověď na zaslaný výpočet se klientovi vrací náhodně zvolená z přípustných hodnot.

Díky této pomůcce jsem byl schopný relativně jednoduše testovat všechny části

systému v průběhu jeho vývoje.

4.5.2 Ověření funkčnosti systému

Po dokončení implementace systému a otestování jeho jednotlivých částí pomocí výše popisovaného emulátoru, přišlo na řadu ověření funkčnosti při reálné těžbě.

K tomuto účelu jsem si za pomoci webového rozhraní vytvořil uživatelský účet. Zaregistroval jsem se také na několika pool serverech těžící měny s nízkou obtížností. Tyto pool servery jsem následně nakonfiguroval ve webovém rozhraní mého systému. Vytvořil jsem těžební skupinu obsahující celkem tři různé měny. Pro každou z měn jsem definoval primární i záložní pool server.

První testovací scénář měl za úkol otestovat procentuální rozdělování výkonu mezi různé měny a současně ověřit přepojení na alternativní pool server v případě výpadku spojení. Abych mohl monitorovat, jaké operace na proxy serveru probíhají, vytvořil jsem za tímto účelem další sadu adaptérů. Ty veškeré probíhající operace zaznamenávaly do logu. Abych nasimuloval výpadky spojení mezi těžebním klientem a pool serverem, vytvořil jsem si další jednoduchý proxy server. Pomocí něj jsem přemostil spojení k cílovému pool serveru. Když nastalo přepojení klienta na tento pool server, proxy server přemosťující připojení jsem pozastavil. Poté jsem sledoval, jestli dojde k přepojení na alternativní pool server.

Proxy server korektně prováděl periodické přepojování mezi různými měnami. Vzhledem k principu přepojování, který popisuji v kapitole 4.3.4, nedochází k naprosto přesnému rozdělení výkonu. Při dlouhodobém běhu se ale systém k uživatelsky definovanému rozdělení velmi blíží. Přepojování na alternativní pool servery probíhalo korektně a bez časové prodlevy. Pokud by těžba probíhala bez proxy serveru pro optimalizaci, neměl by těžební stroj po dobu výpadku spojení co počítat. V případě použití proxy dojde k velmi rychlému přepojení na jiný server. Efektivitu těžby tak sníží pouze zahození výsledku aktuálně počítaného bloku, protože jej není kam zaslat.

Zbývalo tak ověřit vyhodnocování efektivity těžby. Zařadil jsem proto do těžební skupiny měnu s vyšší obtížností. U té logicky nedocházelo k potvrzení zaslaných výsledků příliš často. Ve srovnání s ostatními měnami ve skupině byla tato těžba neefektivní. Proxy server tak správně těžbu této měny přeskakoval. V praxi by k této situaci mohlo dojít, pokud by se náhle zvýšila konkurence u těžené měny. K síti by se připojil stroj s vyšším výkonem, který by bloky ověřoval rychleji a uživatelův

klient by tím pádem nedosáhl žádného ověřeného bloku. V případě těžby bez proxy by klient počítal bloky bez dosažení jakékoliv odměny do té doby, kdy by si toho jeho majitel nevšiml a těženou měnu ručně nezměnil. V případě využití proxy je tato situace automaticky detekována a takto vyhodnocená měna je přeskakována. Tím se minimalizuje výpočetní doba těžebního stroje, při které je minimální šance na získání odměny.

5 Závěr

Výsledkem této práce je systém zvyšující efektivitu těžby kryptoměn. Uživatelům nabízí možnost definovat si několik těžebních skupin, díky kterým může jeden těžební klient těžit různé měny. Výkon mezi měnami lze procentuálně rozdělovat. Tato konfigurace je uživatelům zpřístupněna pomocí webového rozhraní. Kromě tohoto vylepšení přináší systém oproti přímé těžbě automatické přepojování na alternativní pool server v případě výpadku spojení. Při těžbě je analyzována její efektivita. Je-li detekováno neefektivní spojení, dojde k okamžitému přepojení na další definovanou měnu. Tyto vlastnosti přináší uživateli vyšší komfort a hlavně větší efektivitu celé těžby.

Díky modulárnímu návrhu systému vznikla při jeho realizaci knihovna pro abstrakci RPC protokolů. Ta je navržena tak, že ji lze použít naprosto samostatně. Je ji tak možné využít i v jiných projektech pracujících s RPC protokoly pro těžbu kryptoměn.

Jedním z výstupů této práce je také ucelený teoretický základ fungování kryptoměn sepsaný v českém jazyce.

V budoucnu bych systém rád doplnil o uchovávání dlouhodobých statistik těžby. Takto nasbírané statistiky bych uživateli zobrazoval ve webovém rozhraní. Kromě vyšší efektivity těžby by tak uživatel používáním systému získal také kompletní přehled o činnosti svých těžebních klientů. Měl by k dispozici statistiky pro všechny využívané pool servery na jednom místě.

Literatura

[1] Transactions. NAKAMOTO, Satoshi. *Bitcoin: A Peer-to-Peer Electronic Cash System* [online]. s. 2 [cit. 2016-04-05]. Dostupné z: https://bitcoin.org/bitcoin.pdf

[2] *Nonce* [online]. 27. 2. 2011, 19. 3. 2015 [cit. 2016-04-10]. Dostupné z: https://en.bitcoin.it/wiki/Nonce

[3] *Controlled supply* [online]. 2. 1. 2011, 25. 4. 2016 [cit. 2016-04-26]. Dostupné z: https://en.bitcoin.it/wiki/Controlled_Currency_Supply

[4] *Pooled mining* [online]. 30. 4. 2011, 8. 12. 2015 [cit. 2016-04-11]. Dostupné z: https://en.bitcoin.it/wiki/Mining_Pool

[5] Blockchain. DECKER, Christian a Roger WATTENHOFER. *Information Propagation in the Bitcoin Network* [online]. s. 3 [cit. 2016-04-05]. Dostupné z: http://www.tik.ee.ethz.ch/file/49318d3f56c1d525aabf7fda78b23fc0/P2P2013_041.pdf

[6] *Genesis block* [online]. 16. 12. 2010, 5. 11. 2015 [cit. 2016-04-15]. Dostupné z: https://en.bitcoin.it/wiki/Genesis_block

[7] *Double-spending* [online]. 18. 5. 2012, 11. 4. 2015 [cit. 2016-04-10]. Dostupné z: https://en.bitcoin.it/wiki/Double-spending

[8] DAVIS, Joshua *The Crypto-Currency* [online]. The New Yorker, 10. 10. 2011 [cit. 2014-05-10]. Dostupné z: http://www.newyorker.com/reporting/2011/10/10/111010fa_fact_davis

[9] *FAQ* [online]. 24. 12. 2012, 27. 3. 2014 [cit. 2016-04-05]. Dostupné z: https://en.bitcoin.it/wiki/FAQ#How_long_will_it_take_to_generate_all_the_coins.3F

[10] *Getwork* [online]. 7. 5. 2011, 1. 4. 2015 [cit. 2016-04-04]. Dostupné z: https://en.bitcoin.it/wiki/Getwork

[11] *Getblocktemplate* [online]. 17. 9. 2012, 2. 11. 2015 [cit. 2016-04-11]. Dostupné z: https://en.bitcoin.it/wiki/Getblocktemplate

[12] *BIP 0001* [online]. 19. 9. 2011, 29. 12. 2015 [cit. 2016-04-09]. Dostupné z: https://en.bitcoin.it/wiki/BIP_0001

[13] *Stratum* [online]. 7. 5. 2011, 19. 3. 2016 [cit. 2016-04-06]. Dostupné z: http://mining.bitcoin.cz/stratum-mining

[14] *Apache Cassandra NoSQL Performance Benchmarks* [online]. [cit. 2016-04-28]. Dostupné z: http://www.planetcassandra.org/nosql-performance-benchmarks/

[15] *Cassandra CQL* [online]. [cit. 2016-04-28]. Dostupné z: http://www.guru99.com/cassandra-cql.html#1

[16] *On Channel Handlers and Channel Options* [online]. [cit. 2016-04-28]. Dostupné z: http://seeallhearall.blogspot.cz/2012/06/netty-tutorial-part-15-on-channel.html

[17] *Top 7 Open-Source JSON-Binding Providers Available Today* [online]. 4. 4. 2014 [cit. 2016-04-10]. Dostupné z: http://www.developer.com/lang/jscript/top-7-open-source-json-binding-providers-available-today.html

[18] *Datastax Java Driver for Apache Cassandra* [online]. [cit. 2016-04-28]. Dostupné z: https://github.com/datastax/java-driver

[19] *Fast-track your API Design* [online]. [cit. 2016-04-28]. Dostupné z: https://apiary.io/how-it-works

[20] *REST and Stateless Session IDs* [online]. [cit. 2016-04-28]. Dostupné z: http://appsandsecurity.blogspot.cz/2011/04/rest-and-stateless-session-ids.html

A Přiložené DVD

K práci je přiloženo DVD s následujícím obsahem:

- buchacek-dp-2016.pdf - Soubor s elektronickou verzí práce.
- pool.ec - Adresář obsahující všechny zdrojové kódy vytvořeného systému.