

Sem vložte zadání Vaší práce.

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
KATEDRA POČÍTAČOVÝCH SYSTÉMŮ



Diplomová práce

Simulátor akciové burzy

Bc. Jan Jůna

Vedoucí práce: Mgr. Jan Starý, PhD.

5. ledna 2015

Poděkování

Tímto chci poděkovat panu Mgr. Janu Starému za jeho pomoc při mých prvních krůčcích na kapitálových trzích, za tolik drahocenný čas, který mi věnoval a za inspirativní pohled na svět investičních příležitostí, který jsem si z této práce odnesl. Dále děkuji svým rodičům a přátelům, s jejichž podporou jsem byl schopen práci dokončit.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 5. ledna 2015

.....

České vysoké učení technické v Praze

Fakulta informačních technologií

© 2015 Jan Jůna. Všechna práva vyhrazena.

Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Jůna, Jan. *Simulátor akciové burzy*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

Práce se zabývá tvorbou základního burzovního systému. S využitím algoritmu pro spárování burzovních příkazů budujeme obchodní portál umožňující zadávat příkazy pro nákup a prodej akcií.

Klíčová slova akciová burza, burzovní systém, trading, realtime aplikace, node.js, angular.js, mongodb, socket.io

Abstract

This work deals with a rudimentary stock exchange simulator. Using an order matching algorithm, we build a market portal which accepts basic buy and sell orders.

Keywords stock market, stock market system, trading, realtime application, node.js, angular.js, mongodb, socket.io

Obsah

Úvod	1
1 Cíl práce a motivace	3
1.1 Motivace	3
1.2 Akciová burza a burzovní systémy	3
1.3 Zjednodušení systému	4
1.4 Rozdělení práce na systému	5
1.5 Terminologie a pojmy použité v této práci	5
2 Analýza	7
2.1 Součásti systému	7
2.2 Požadavky	17
3 Návrh	19
3.1 Použité technologie	19
3.2 Obecné části návrhu	26
3.3 Market	27
3.4 Broker	32
3.5 Obchodní platforma	35
3.6 Služba poskytující historická data	38
3.7 Automatický obchodní systém	41
4 Realizace	43
4.1 Market	43
4.2 Broker	48
4.3 Obchodní platforma	50
4.4 Služba poskytující historická data	52
4.5 Automatický obchodní systém	54
5 Deployment	57

5.1	Konfigurace služeb	57
5.2	Kompilace a spuštění workerů	58
5.3	Instalace použitých technologií	58
5.4	Inicializace databází	59
5.5	Instalace závislostí	60
5.6	Spuštění Node.JS aplikací	60
6	Monitoring a ladění	63
6.1	Monitoring Node.JS aplikace	63
6.2	Ladění	63
7	Testování - dodelam v prubehu nedele	65
7.1	Testované prostředí	65
8	Škálování	67
8.1	Oddělení náročných úloh	67
8.2	Cluster modul	68
8.3	Spuštění více aplikací pod load balancerem	69
	Závěr	71
	Literatura	73
A	Seznam použitých zkratk	77
B	Terminologie a použité pojmy	79
C	Screeny služeb v systému	81
D	Obsah příloženého DVD	83

Seznam obrázků

2.1	Struktura burzovního systému	8
2.2	Příkazy zapsané v obchodní knize	11
2.3	Obchodní kniha a kumulativní součty	11
2.4	Obchodní kniha a zobchodovatelný objem s imbalancí	12
2.5	Párování obchodních příkazů v obchodní knize	13
2.6	Svícový graf zobrazující historický vývoj ceny a objemu	16
2.7	Význam jednotlivých částí svíček ve svícovém grafu.	16
3.1	Srovnání asynchronního a synchronního zpracování	21
3.2	Funkcionalita hlavního threadu v technologii Node.JS	22
3.3	Detailní pohled na strukturu systému	27
3.4	Stavový diagram obchodního příkazu	29
3.5	Burzovní cyklus	30
3.6	Průběh přihlášení uživatele do obchodní platformy	37
3.7	Dotaz MapReduce nad MongoDB databází	40
4.1	Seznam připojených workerů ve webovém rozhraní marketu	46
4.2	Detail akcie ve webovém rozhraní marketu	48
4.3	Dashboard klientské aplikace	50
4.4	Atuální data akcií na burze	52
4.5	Grafy vývoje ceny a volume v aplikaci	53
4.6	Komunikace s REST API služby poskytující historická data	54
5.1	Spuštění služby v produkčním prostředí a příkaz forever	61
6.1	Prostředí modulu pro ladění kódu v Node.JS	64
C.1	Výpis uživatelových držených akcií a nevyřízených příkazů	81
C.2	Historie klientových obchodních příkazů v obchodní platformě	81
C.3	Detail akcie zobrazený v obchodní platformě	82
C.4	Dashboard automatického obchodního systému	82

Seznam tabulek

2.1	Seznam nejpoužívanějších obchodních příkazů.	9
3.1	Události, na kterých Market API naslouchá.	33
3.2	REST API endpointy pro klientskou aplikaci.	36

Úvod

V posledních deseti letech dvacátého století došlo k rozmachu kapitálových trhů po celém světě. To bylo způsobeno především díky zavedení internetu a dalších technických inovací do burzovních systémů, čímž, mimo jiné, došlo k výraznému snížení poplatků účtovaných za použití takového systému a do burzovního světa tak začali pronikat i první drobní investoři.

Od té doby uplynulo již více než dvacet let a burzovní svět prošel ještě mnohými změnami. Stejně tak se udály i světové krize, jejichž síla se ještě umocnila právě díky globalizaci a propojení jednotlivých burz.

Na druhou stranu však propojení a modernizace vytvořily nové příležitosti pro menší investory a položily základ automatickému obchodování, které na dnešních burzách naprosto převládá. Podle [1] je průměrná doba držení akcií 22 sekund a průměrným investorem je nyní počítač. Protože se však obchoduje s kapitálem často až v astronomické výši, je důležité takové automatické obchodní systémy otestovat v nejrůznějších situacích, kterými se trh může ubírat.

V této práci si proto kladu za cíl vytvořit základ jednoduchého simulátoru akciové burzy, která bude klientům umožňovat nákup a prodej akcií vypsanych na burze. Práce se pak může dále rozšiřovat za účelem vytvoření komplexního burzovního simulátoru, s jehož pomocí lze testovat chování nejrůznějších obchodních strategií v modelových situacích, jako jsou například světové krize nebo období nízké volatility.

Cíl práce a motivace

1.1 Motivace

Akciová burza a obecně burza cenných papírů je místo, kde mohou firmy získat potřebný kapitál pro další rozvoj tím, že nabídnou veřejnosti svůj podíl v podobě *akcií* (neboli cenných papírů). Drobní investoři, velké investiční banky a další zájemci pak tyto akcie kupují s vírou v jejich rostoucí cenu a tím i ve zhodnocení vloženého majetku. Burza však v sobě skrývá i mnohá rizika v podobě ztráty investovaného jmění, a proto je dobré znát základní složky a procesy v takovém systému, které vývoj ceny ovlivňují.

Mou hlavní motivací pro vytvoření simulátoru akciové burzy je záliba v kapitálových trzích. Spolu s touto prací se chci zorientovat v burzovním světě a odhalit možnosti, které se zde skrývají.

1.2 Akciová burza a burzovní systémy

Cílem této práce je vytvořit základní verzi funkčního burzovního systému, který bude schopen od klientů přijímat obchodní příkazy na nákup a prodej vystavených akcií. Tyto příkazy budou shromážděny na jednom místě, kde budou zpracovány pomocí standardizovaného *order matching* algoritmu a následně budou klientům přerozděleny zobchodované akcie. Burzovní systém se bude skládat z těchto tří služeb, kterým se v práci budu postupně blíže věnovat:

- **Market** - centrální bod systému, kde dochází ke shromáždění a vyřízení obchodních příkazů.
- **Broker** - prostředník mezi burzou a klienty, kterým za poplatek poskytuje možnost nakupovat a prodávat cenné papíry.
- **Obchodní platforma** - webové nebo desktopové prostředí, kde mohou uživatelé spravovat svůj obchodní účet, zadávat obchodní příkazy, zobrazovat si historii svých aktivit a nebo testovat obchodní strategie.

Mimo tyto části bude práce obsahovat službu zajišťující sběr a poskytování historických dat. Jejím úkolem bude uchovávat záznamy o historickém vývoji cen a *volume* (zobchodovaném objemu) akcií, a přes veřejné API pak tyto data poskytovat v agregované podobě třetím stranám. V reálném světě ji lze přirovnat například ke službám Yahoo Finance¹, Google Finance² a nebo z těch komerčních například k IQFeed³. V této práci bude použita především pro kreslení grafů zobrazujících pohyb ceny, *volatility* (kolísavosti ceny akcie) a zobchodovaném objemu daného aktiva. V reálném světě mají však ještě jiné využití a to sice poskytování historických dat pro *backtesting* (historické testování) automatických obchodních strategií.

Další součástí bude jednoduchý automatický obchodní systém, který použiji jednak pro zátěžové testování burzovního systému a pak také pro generování nákupních a prodejních příkazů za účelem zvýšení *likvidity* (obchodovanosti). Roboti, kteří takto obchodují na burzách jsou komplexní algoritmy, jejichž vývoj a následné testování trvá i několik let. V mé práci se proto omezím na jednoduchý algoritmus, jehož účelem nebude profitovat na akciovém trhu, ale generovat smysluplné obchodní příkazy pro již zmíněné zvýšení likvidity.

1.3 Zjednodušení systému

V této práci se omezím pouze na základní podobu burzovního systému. Reálné burzy, jako jsou například americké New York Stock Exchange⁴, NASDAQ Stock Market⁵ nebo londýnská London Stock Exchange⁶, na které se v této práci budu často odkazovat, jsou rozsáhlé systémy obsahující každá kolem 3000 akcií⁷ a dalších obchodovatelných instrumentů v podobě komodit, ETF a derivátů (akciových indexů, futures, options atd.). Obchodní systém vytvořený v této práci bude pro zjednodušení umožňovat pouze obchodování jednotlivých akcií.

Kromě obrovského počtu obchodovatelných instrumentů poskytují reálné burzy také mnoho nástrojů pro práci s obchodními příkazy. Jak bude vysvětleno v kapitole 2, jsou jimi různé typy BUY a SELL příkazů, jako například: *Limit order*, *Market order*, *Cancel order*, *Trailing stop*, *Take profit* a další. S přibývajícím počtem a variací příkazů však roste také složitost vnitřních procesů zajišťujících správné zpracování příkazů, a proto se v práci omezím pouze na Limit order a Cancel order příkazy s možností částečného vyhodnocení, které budou moci klienti použít k zadávání a rušení objednávek v systému.

¹<http://finance.yahoo.com/>

²<http://www.google.com/finance>

³<http://www.iqfeed.net>

⁴<http://www.nyse.com>

⁵<http://www.nasdaq.com>

⁶<http://www.londonstockexchange.com>

⁷<http://www.nasdaq.com/screening/company-list.aspx>

Při využívání služeb reálných burz a brokerských aplikací jsou uživatelům účtovány poplatky podle konkrétní cenové politiky daného subjektu. Protože si ale v práci kladu za cíl vytvořit pouze jednouchou podobu akciové burzy, nebudu v práci tyto poplatky dále rozebírat.

1.4 Rozdělení práce na systému

V mé práci se zaměřím především na návrh a architekturu celé aplikace. Na systému jsem spolupracoval s Bc. Ondřejem Fremuntem, který ve své diplomové práci[2] navrhl databázové schéma marketu včetně *stored procedur* (databázových funkcí) zajišťujících správnou procesní logiku nad uloženými daty a dále samostatnou aplikaci (v dalším textu ji budu nazývat *worker*), jejímž úkolem je správné párování obchodních příkazů. Funkčnost těchto částí zde budu popisovat, pro bližší informace se však vždy budu odkazovat na výše zmíněnou diplomovou práci.

1.5 Terminologie a pojmy použité v této práci

Při psaní této práce budu používat finanční terminologii a některé termíny, které jsou běžné v burzovních systémech. Seznam těchto pojmů spolu s jejich významem bude na konci textu v příloze B.

TODO - dopsat pokračování - J.Muller ..

Analýza

Tato kapitola bude sloužit jako prvotní návrh aplikace. Budou zde uvedeny základní informace o obchodních systémech, jejich vnitřní struktuře a procesech, které běží na pozadí. Bude zde popsán *order matching algoritmus* zajišťující správné párování obchodních příkazů a způsob jejich následného zobchodování. Poté bude následovat seznam funkčních a nefunkčních požadavků tvořený systém a v další kapitole se již budu věnovat návrhu aplikace v konkrétních technologiích.

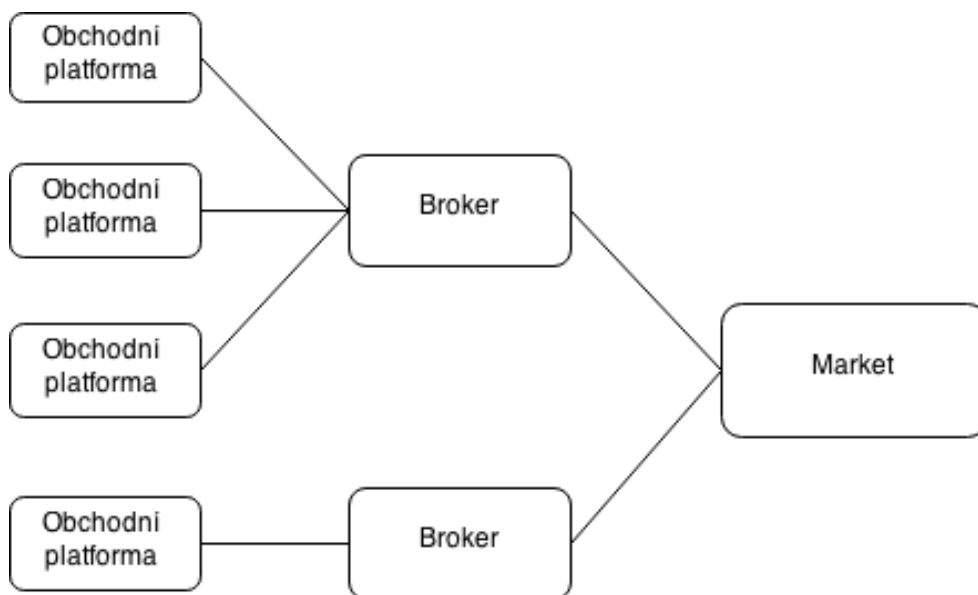
2.1 Součásti systému

Jak už bylo řečeno výše, burzovní systém se skládá ze 3 hlavních částí: *Marketu*, *Brokera* a *Obchodní platformy*. Zároveň platí, že k marketu může být v jednu chvíli připojeno více brokerů, a obdobně pak jeden broker může mít více uživatelů, kterým poskytuje svou obchodní platformu pro zadávání příkazů. V reálném světě je toto ještě složitější. Nejen že může mít klient vedený účet u více brokerů, jeden broker je ale často napojený na více marketů, aby svým klientům poskytl co největší portfolio možných instrumentů, do kterých se může investovat. V mé práci se však omezím na základní strukturu, zachycenou na obrázku 2.1, kde broker využívá služby pouze jednoho marketu a obdobně pro klienty.

Kromě prvků na obrázku zde vystupují i další strany. V České republice je to například Centrální depozitář cenných papírů⁸ (CDCP), ve Spojených státech amerických pak U.S. Securities and Exchange Commission⁹, což jsou organizace sledující pohyby na burzách a působící v systému ve formě kontrolora. Těmito prvky se v této práci nebudu dále zabývat a zaměřím se především na ty, co jsou na obrázku 2.1.

⁸<http://www.cdcp.cz/index.php/cz/>

⁹<http://www.sec.gov/index.htm>



Obrázek 2.1: Burzovní systém se skládá ze tří hlavních částí (marketu, brokera a obchodní platformy).

2.1.1 Obchodní příkazy

Obchodní příkazy mohou být různé, podle možností dané burzy (viz seznam obchodních příkazů na londýnské LSE [3] nebo americké NYSE [4]). Nejznámější a nejčastěji používané příkazy podle [5] jsou znázorněny v tabulce 2.1.1.

Kromě uvedených příkazů však existuje další spousta variací. Například podle aktuální části obchodního období (opening, main season, closing) to jsou *Market "On-the-Open" Order* (MOO) nebo *Market "At-The-Close" Order* (MOC) a další.

Obchodní příkazy mohou být navíc rozšířeny i brokerem, viz například seznam poskytovaných příkazů u brokera Interactive Brokers uvedený v [6]. Je zde například *Trailing Stop*, což je obdoba Stop Orderu s nastaveným chováním, kdy se nastavená hranice Stop Orderu zvyšuje, pokud cena jde nahoru. V opačném případě se Stop Order nemění.

Poslední variací, kterou zde chci rozebrat je částečné a úplné zobchodování. Při zadávání obchodního příkazu můžeme chtít, aby se vyplnil celý nebo vůbec. V takovém případě musí systém dokázat aplikovat pravidlo, že se v jednom cyklu musí nakoupit nebo prodat veškeré množství akcií z daného příkazu. V mé práci se však budu zabývat pouze příkazy s částečným zobchodováním, kde se obchodní příkaz může vyřizovat po částech.

Název	Popis chování
Limit Order	Jedná se o příkaz se stanovenou nejhorší cenou, za kterou může být proveden. Pokud se cena podkladového aktiva dostane na hodnotu rovnou nebo lepší než je nastavená cena a zároveň existuje protistrana, příkaz bude proveden.
Market Order	Oproti Limit order zde není nastavená pevná cena a příkaz je tak proveden za aktuální cenu aktiva na trhu.
Stop Order	Typ příkazu, který čeká v systému mimo obchodní knihu, dokud cena podkladového aktiva není horší nebo rovna nastavené ceně v příkazu. Jakmile se tak stane, příkaz je přidán do obchodní knihy a dále je s ním nakládáno jako s Market Order. Typicky se tento příkaz používá pro realizaci Stop-Lossu.
Stop Limit Order	Příkaz opět čeká mimo obchodní knihu dokud cena aktiva není horší nebo rovna zadané ceně. Jakmile se cena dostane na tuto hranici, příkaz je přidán do obchodní knihy a je s ním nakládáno jako s Limit Order.

Tabulka 2.1: Seznam nepoužívanějších obchodních příkazů.

2.1.2 Market

Centrálním bodem systému se stává market (neboli burza), kterému jsou od napojených brokerů posílány obchodní příkazy. Market provádí tzv. *Continuous Trading*[7], což je proces, jehož přesná podoba se může drobně lišit podle konkrétní burzy, v zásadě jsou zde však tyto fáze:

1. **Pre-Opening Phase** - První fáze, ve které jsou nashromážděny obchodní příkazy do tzv. *Obchodní knihy*. Během této fáze je možné příkazy také zrušit, protože v této fázi zatím neprobíhá žádné jejich další zpracování.
2. **Opening Auction** - Zde se projde obchodní kniha a za pomoci order matching algoritmu (popsaného dále v textu) dojde ke spárování nákupních a prodejních příkazů spolu s vypočítáním tzv. *strike price* (střetávací cenu nabídky a poptávky).
3. **Main Trading Session** - V této části jsou spárované obchodní příkazy z druhé fáze zpracovány za vypočítanou otevírací cenu a informace o jejich zobchodování je odeslána brokerům. Nespárované příkazy zůstávají v obchodní knize pro další zpracování. Během této doby mohou brokeri zadávat nové obchodní příkazy a celý proces, tj. spočítání strike price,

2. ANALÝZA

párování obchodních příkazů pomocí order matching algoritmu, jejich následné zobchodování a notifikování brokerů se kontinuálně opakuje až do konce této fáze.

4. **Pre-closing Phase** - V této části dojde k zastavení zpracování příkazů nashromážděných v obchodní knize.
5. **Closing Auction** - Z obchodní knihy je pomocí order matching algoritmu spočítána *Closing price* (zavírací cena obchodního dne). Pokud se v obchodní knize obchodní příkazy nepřekrývají (bude vysvětleno dále), vezme se poslední obchodovaná cena jako Closing price.
6. **Trading-at-last Phase** - V této fázi mohou být zadány a zpracovány příkazy pouze za stanovenou Closing price.
7. **After Hours Trading** - Po ukončení obchodního období (fáze 1 až 6) již nejdou zadat obchodní příkazy a jsou povolené pouze dotazy na stav příkazů a akcií.

Tento process se opakuje každý obchodní den a přesné časy jednotlivých fází se liší podle konkrétní burzy, například na americké NASDAQ se kromě vypsání svátků obchoduje každý všední den a *Main trading session* trvá v čase od 9:30 ráno do 4:00 odpoledne místního času, viz [8].

2.1.3 Market a zpracování obchodních příkazů

Podle 2.1.2 prochází market během jednoho dne několika fázemi, kdy probíhá obchodování a příjem příkazů. Tyto příkazy jsou zapsány do obchodní knihy, kde čekají na další zpracování. Příklad obchodní knihy s čekajícími příkazy je na obrázku 2.2. Velký důraz se zde klade na rychlost a spolehlivost, přijetí příkazu musí být proto možné i když burza provádí další procesy, jako je například zpracování obchodní knihy.

2.1.3.1 Sestavení kumulativních součtů

Pro další zpracování je potřeba v obchodní knize vytvořit tzv. kumulativní součty, kdy se seskupí příkazy stejného typu a ceny a vše se seřadí podle ceny. U poptávajících příkazů se pak sečte sestupně požadované množství a analogicky u nabízejících příkazů se nabízené množství sečte vzestupně. Kumulativní součty jsou již naznačeny na obrázku 2.3.

Smysl tohoto řazení je, že pokud (podle obrázku) jsme ochotni koupit 2 akcie za cenu \$535, pak je koupíme i za cenu nižší. Proto je za \$520 již poptáváno celkem 9 akcií (předchozí 2 za cenu \$535 spolu s poptávkou na 7 akcií za \$520) a tak dále. Obdobně funguje i nabídka, kde akcie nabízíme za stanovenou cenu a vyšší.

Poptávka (BUY)		Nabídka (SELL)		
Id	Množství	Cena	Množství	Id
1	5	\$490		
2	8	\$500	8	3
4	15	\$505	12	5
7	3	\$510	1	6
8	7	\$520	3	9
10	2	\$535	6	11
		\$560	15	12

Obrázek 2.2: Příkazy zapsané v obchodní knize.

Poptávka (BUY)		Nabídka (SELL)		
Součet	Množství	Cena	Množství	Součet
40	5	\$490	0	0
35	8	\$500	8	8
27	15	\$505	12	20
12	3	\$510	1	21
9	7	\$520	3	24
2	2	\$535	6	30
0	0	\$560	15	45

Obrázek 2.3: Příklad seřazení příkazů v obchodní knize do kumulativních součtů.

2.1.3.2 Order matching algoritmus

Po sestavení kumulativních součtů srovnáme poptávané a nabízené množství a vypočítáme maximální *volume* (zobchodovatelný objem) a *imbalance* (převís objemu nabídky nad poptávkou nebo obráceně). Výpočet probíhá pomocí těchto vzorců:

$$\begin{aligned}
 \text{Zobch. objem} &= \text{Min}(\text{Nabídka}, \text{Poptávka}) \\
 \text{Imbalance} &= \text{Abs}(\text{Nabídka} - \text{Poptávka})
 \end{aligned}$$

Výsledek řazení s vypočítaným maximálním zobchodovatelným objemem a imbalance je na obrázku 2.4.

2. ANALÝZA

Cena	Poptávané množství	Nabízené množství	Zobchodovatelný objem	Imbalance
\$490	40	0	0	40
\$500	35	8	8 (\$4000)	27
\$505	27	20	20 (\$10100)	7
\$510	12	21	12 (\$6120)	9
\$520	9	24	9 (\$4680)	15
\$535	2	30	2 (\$1070)	28
\$560	0	45	0	45

Obrázek 2.4: Vypočítaný zobchodovatelný objem a imbalance v obchodní knize.

Následuje průchod seznamu, jehož úkolem je vybrat strike price, za kterou se zobchoduje největší objem akcií. Pro tento výběr se používá několik kritérií, které jsou následující:

1. Pokud v tabulce není žádný záznam, kde by zobchodovatelný objem byl větší než 0, neproběhne žádný obchod a strike price bude cena z předchozího cyklu.
2. Pokud v tabulce existují řádky u kterých může obchod proběhnout, vezme se ten, kde je největší zobchodovatelné množství.
3. Pokud je více řádků, které mají stejné maximální zobchodovatelné množství, porovná se jejich imbalance a vybere se ten s nejnižší imbalance.
4. Pokud existuje více řádků s maximálním zobchodovatelným objemem a nejmenší imbalance, použijeme čtvrté kritérium pro výběr a sice směr imbalance. U těchto řádků porovnáme počet nabízených a poptávaných akcií. Pokud je u všech více poptávaných akcií než nabízených, algoritmus zvolí vyšší možnou cenu. V opačném případě (větší nabídka než poptávka) zvolí se nižší cena.
5. Pokud mají řádky opačný směr imbalance, vybere z vyhovujících ty s nejmenší a největší cenou a ty zprůměruje. Protože burza má přesně stanovenou *tick size* (granularitu ceny), zprůměrovaná cena se zaokrouhlí směrem k ceně z předchozího cyklu.

Pomocí tohoto algoritmu se vypočítala strike price představující cenu následujícího obchodu a objem udávající počet akcií, které budou zobchodovány. Z příkladu na obrázku 2.4 by se podle kritérií výše vybral třetí řádek (vyznačený červeným písmem), strike price by byla \$505 a objem by byl 20 akcií.

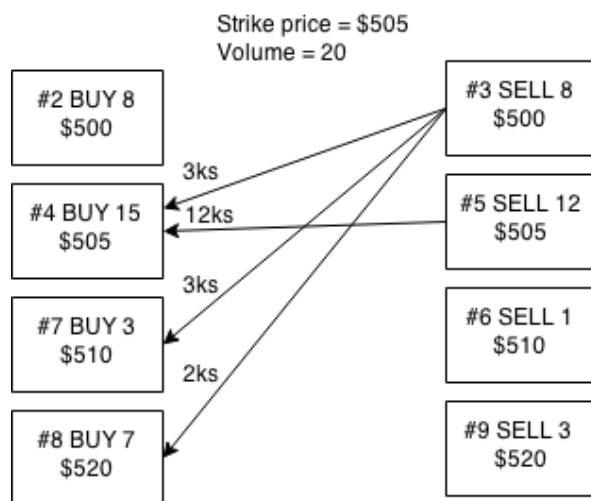
2.1.3.3 Párování a zobchodování obchodních příkazů

Před dokončením cyklu je potřeba spárovat obchodní příkazy a odeslat brokerům notifikaci o jejich vyhotovení. Párování příkazů se provádí tak, že se vezmou postupně nákupní příkazy od nejvyšší ceny a k nim se přiřkládají prodejní příkazy s nejnižší cenou v odpovídajícím množství. Toto se opakuje, dokud nezobchodujeme objem spočítaný v předchozím kroku. Zároveň tím zajistíme, že nabídky s nejlepší cenou budou vždy upřednostněny.

Pokud se při párování za danou cenu může vzít více příkazů, upřednostní se ten, který je v systému nejdelší dobu.

Podle tohoto postupu se může stát, že na jeden nákupní příkaz se složí více prodejních příkazů a obráceně. Možné je i to, že se příkaz uspokojí jen z části. V takovém případě se odešle brokerovi informace o vyřízené části a zbytek příkazu zůstane v obchodní knize do příštího cyklu. Těmto příkazům, které mohou být vyřízeny jen částečně se říká příkazy s částečným zobchodováním a byly již probrány v sekci 2.1.1.

Příklad párování obchodních příkazů z předchozího příkladu je na obrázku 2.5 (některé příkazy, u kterých nedojde k párování, byly schovány pro úsporu místa).



Obrázek 2.5: Párování obchodních příkazů v obchodní knize.

Po spárování se odešle notifikace o provedených obchodech patřičným brokerům a tím zpracování jednoho burzovního cyklu končí.

2.1.4 Broker

Broker je prostředníkem mezi svými klienty a burzou, kde probíhají veškeré obchody. Svým klientům poskytuje obchodní platformu, kde mohou zadávat a spravovat obchodní příkazy spolu se svým účtem. S jeho pomocí na

cílové klienty nejsou kladeny takové požadavky a regule, jako jsou na samotné brokery. Jedná se především o zajištění *solventnosti* a neustálý dohled státních úřadů, které by jinak klienti museli podstoupit, kdyby chtěli obchodovat na burze přímo.

2.1.4.1 Komunikace s burzou a FIX protokol

Klíčovou součástí systému brokera je komunikace s burzou, která probíhá nejčastěji pomocí Financial Information Exchange (FIX) protokolu. Historie tohoto protokolu sahá do roku 1992, kdy byl představen jako nadstavba TCP protokolu s maximálním důrazem na co nejmenší přenášený objem. Pro svou otevřenou specifikaci a nízký objem přenášených dat (tím pádem i rychlost) si získal velkou oblibu u finančních institucí. Mnoho z těchto společností také přispívá k jeho vývoji a úpravám pro současné potřeby, viz [10].

FIX protokol obsahuje tři základní skupiny příkazů, jimiž jsou:

- **Administrační příkazy** - příkazy používané pro správu spojení mezi uzly, jako například *heartbeat*.
- **Aplikační příkazy** - příkazy používané pro obchodování (*New Order*, *Cancel Order*, *Order status* atd.).
- **Bezpečnostní příkazy** - jsou zde především příkazy poskytující autentizaci (*Login* a *Logout* příkazy).

Jeden příkaz se pak sestává z dvojic klíč-hodnota oddělených ASCII znakem #001, kde klíč je reprezentován jako číslo, pro minimalizaci přenášeného objemu. Příkaz obsahuje tři části:

- **Header** - obsahuje informace o spojení, verzi protokolu, typu a délce zprávy a podobně.
- **Body** - část nesoucí vlastní informace o příkazu, jako je jeho typ a na tom závislé další informace (cena, množství, podkladové aktivum apod.).
- **Footer** - obsahuje kontrolní součet zprávy pro ověření integrity.

Obchodní příkaz zakódovaný a poslaný FIX protokolem pak vypadá například takto (Netisknutelný ASCII znak #001 byl nahrazen za znak |):

```
8=FIX4.2|9=0132|35=D|57=EXEC|34=2|49=BROKER|56=SB10|  
52=20100315-17:45:20|55=AAPL|40=2|38=200|21=2|  
11=old0|60=20100315-17:45:20|54=1|44=109.90|10=097
```

V příkladu výše je zakódován Limit Order příkaz (pomocí dvojice 40=2) na nákup 200 akcií (38=200) společnosti Apple Inc. (55=AAPL) za cenu \$109.90 (44=109.90). Dekódování a přehledné zobrazení zprávy je možné na adrese

<http://fixparser.targetcompid.com/> a bližší informace k FIX protokolu jsou například v manuálu americké burzy NASDAQ [11].

2.1.5 Obchodní platforma

Obchodní platformou se myslí uživatelský portál, kde mohou klienti brokera zadávat obchodní příkazy na nákup a prodej akcií, komodit a dalších nabízených aktiv. Jedná se o desktopové, online a v poslední době také mobilní aplikace poskytované brokerem, ale i nezávisle třetími stranami.

Příkladem z reálného světa mohou být desktopové MetaTrader¹⁰, Amibroker¹¹ nebo online poskytovaný WebTrader¹² od Interactive Brokers a spousta dalších.

Komunikace s brokerem probíhá různě podle konkrétního brokera, typicky se však jedná o nějakou formu REST API, RPC a nebo FIX protokolu vysvětleného výše v 2.1.4.1.

2.1.5.1 Zobrazení burzovních dat v grafu

Klíčovou službou poskytovanou bezesporu každou obchodní platformou je zobrazení historického vývoje ceny a zobchodovaného množství v grafu. Je zde několik typů grafů, z nichž se zaměřím především na ten nejjednodušší lineární graf a poté, pro burzovní systémy tolik typický, svícový graf.

Svícové grafy a jejich zobrazení ceny bude důležité pro pochopení textu v další kapitole, proto se jim teď budu krátce věnovat.

Osa X na grafu stejně tak, jako u lineárních grafů vyobrazuje čas a na ose Y je pomocí sloupců vidět zrealizovaný objem akcií. Kromě toho jsou v grafu zakresleny ještě svíčky, které vyobrazují cenu a její změnu ve zvoleném timefram¹³. Příklad svícového grafu je na obrázku 2.6.

Každá svíce představuje nastavený *timeframe* (časový úsek), například 1 minutu, 5 minut, den, týden, atd. a je složena ze 4 cen, kterými jsou:

- Open (O) - otevírací cena v timefram¹³
- High (H)- nejvyšší cena v timefram¹³
- Low (L)- nejnižší cena v timefram¹³
- Close (C)- uzavírací cena v timefram¹³

Svíčky jsou pak ještě černou a bílou (nebo také červenou a zelenou) barvou rozlišeny na klesající (cena Open je výše jak cena Close) a stoupající (cena Open je níže než cena Close). Zobrazení těchto čtyř cen na svíčkách je znázorněno na obrázku 2.7.

¹⁰<http://www.metaquotes.net/en/metatrader4>

¹¹<http://amibroker.com/>

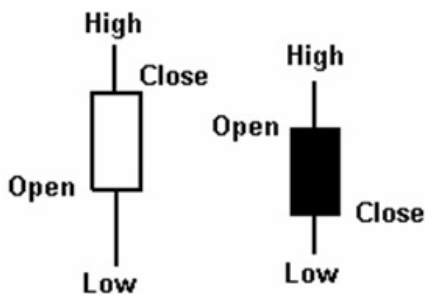
¹²<http://gre.wgw.interactivebrokers.com/webtrader/servlet/login>

¹³Časový úsek.

2. ANALÝZA



Obrázek 2.6: Svícový graf akcie AMG zobrazující historický vývoj ceny a zobchodovaného objemu. [dostupný z [12]]



Obrázek 2.7: Ceny u stoupající (bílé) a klesající (černé) svíčky na svícovém grafu. [dostupný z [12]]

Na lineárních grafech je oproti svícovým zobrazen pouze zobchodovaný objem a Close cena v daném timeframu.

V práci budu pro obchodní platformu tvořit také náhledy historických dat v těchto dvou grafech a jak bude dále popsáno, budu řešit také agregaci dat do čtveřice Open, High, Low a Close cen v daném timeframu.

2.2 Požadavky

Podle popisu systému výše jsem vytvořil následující seznamy funkčních a nefunkčních požadavků, na které budu při vývoji klást důraz.

2.2.1 Funkční požadavky

Funkční požadavky jsou nároky na funkčnost systému a říkají, jaké funkce systém musí mít. Z těch hlavních jsou to především tyto:

- **Vytvoření záznamu akcie na burze** - Systém musí umožnit přidání nové akcie na burze, se kterou lze následně obchodovat.
- **Emise akcií** - Systém musí umožnit *emisi* (prvotní vypsání) nových akcií.
- **Registrace brokera** - Systém musí v aplikaci marketu umět registrovat brokera, který se následně pomocí přihlašovacích údajů může připojit k burze.
- **Registrace a přihlášení uživatele** - Systém musí umožnit registraci u aplikace brokera a následné přihlášení uživatelům do obchodní platformy.
- **Úprava účtu klienta** - Systém musí umožnit uživatelům upravovat jejich účty a měnit hesla.
- **Zadávání obchodních příkazů** - Systém musí umožnit přihlášeným uživatelům zadávat obchodní příkazy na nákup a prodej akcií a příkaz na zrušení přechozího příkazu.
- **Zobrazení stavu akcií** - Systém musí umět zobrazit aktuální stav akcií. Stavem akcií se myslí jejich aktuální cena, zobchodovaný objem a jejich vývoj v grafu.
- **Historie obchodů** - Systém musí umět zobrazit historii klientem provedených obchodů a odeslaných příkazů.

2.2.2 Nefunkční požadavky

Nefunkční požadavky říkají, jaké musí mít systém vlastnosti. Nefunkčními požadavky pro tvořený burzovní systém jsou:

- **Komunikace mezi prvky v systému** - Prvky v systému si musí umět vyměňovat zprávy pomocí komunikačního rozhraní. Tato komunikace musí být spolehlivá a nesmí docházet ke ztrátě dat.

2. ANALÝZA

- **Příjem obchodních příkazů** - Systém musí umět od klientů přijmout a uložit obchodní příkazy.
- **Zpracování obchodních příkazů** - Systém musí umět zpracovat nahromážděné obchodní příkazy a následně odeslat vyrozumění klientům.
- **Anonymizace obchodních stran** - Poptávající strana by po provedení obchodu neměla znát protistranu, která se podílela na obchodu a obráceně.
- **Spolehlivost a perzistence** - Systém musí být perzistentní - jednou přijatý a potvrzený příkaz musí být v systému i po případném pádu. Obchodní příkazy se tak nesmí nikam ztratit. Všechny operace s penězi musí být spolehlivé a nesmí se stát, že by například některé úpravy klientského peněžního účtu proběhly s chybou a klientům se tak chybně strhli nebo naopak přičetli finance.
- **Rychlost** - Na cestě při vyřizování příkazu nesmí dojít k žádnému zdržení, klientský systém i broker nesmí příkazy nijak pozastavovat.
- **Stabilita a zotavení po chybě** - Burzovní systém musí být stabilní a neměl by padat. Při případném pádu se musí obnovit do stavu před pádem.
- **Škálovatelnost** - Při větším počtu obchodovaných akciových titulů může docházet k nárůstu času potřebného pro jejich zpracování a systém by tak měl být škálovatelný pro případné rozložení zátěže mezi více uzlů.
- **Bezpečnost** - Posledním zde zmíněným avšak neméně důležitým požadavkem je bezpečnost. Aplikace pracuje s penězi svých uživatelů a neměla by tak v žádném případě umožnit neoprávněnou modifikaci uživatelských údajů nebo únik citlivých informací.

Návrh

V této kapitole se budu zabývat nejprve technologiemi, které jsem zvolil pro stavbu systému a také důvodem, který mě k jejich výběru vedl. Poté bude následovat návrh jednotlivých stavebních bloků a popis jejich funkčnosti.

3.1 Použité technologie

Použité technologie jsem volil tak, aby poskytovaly potřebnou funkcionalitu pro vytvářenou aplikaci a byly dostupné zdarma pro nekomerční účely. Ne všechny však svými možnostmi byly úplně dostatečné, přidal jsem proto jejich krátké zhodnocení také do závěru této práce.

Jak bude vidět v následující sekci 3.2.3, jednotlivé prvky v systému jsem rozdělil na Frontend a Backend. U každé části uvedu použité technologie a důvod jejich výběru.

3.1.1 Frontend

Pro realizaci Frontendu jsem zvolil JavaScriptový framework AngularJS¹⁴, pomocí kterého budu vytvářet webové rozhraní u jednotlivých služeb v systému.

Na Frontendu dále použiji grafickou knihovnu HighCharts¹⁵, která se svým modulem HighStock¹⁶ umí vytvářet burzovní grafy. Protože chci, aby byla aplikace použitelná i po UX stránce, využiji pro obchodní platformu již vytvořenou designovou šablonu AdminLTE¹⁷ od Almsaeed Studio. Hlavní zmíněné technologie dále rozepíši podrobněji.

¹⁴<http://angularjs.org/>

¹⁵<http://www.highcharts.com>

¹⁶<http://www.highcharts.com/products/highstock>

¹⁷<http://almsaeedstudio.com/AdminLTE/>

3.1.1.1 AngularJS

AngularJS je JavaScriptový framework určený pro tvorbu webových Single Page Aplikací (SPA) a za dobu svého vývoje si získal majoritní postavení na poli webových frontend frameworků. Využiji jej proto při realizaci obchodní platformy pro cílové uživatele obchodující na burze, a také k vytvoření rozhraní u každé služby v systému, jehož úkolem bude přehledné zobrazení aktuálního stavu služby.

Použití této technologie a rozdělení aplikace na frontend a backend pozitivně přispěje k snížení objemu dat přenášných mezi uživateli a serverem. Při prvním načtení se uživateli do prohlížeče stáhne celá aplikace, její dynamika a obecné prvky, jako je například menu, layout nebo styly a následně se na pozadí provádí už jen dotazy na serverovou API k načtení konkrétních dat pro danou stránku. Nedochází tak ke zbytečnému stahování stejného obsahu při načtení nových stránek, jak by tomu bylo u klasických webových aplikací.

3.1.1.2 HighCharts a HighStock

Při výběru webového nástroje schopného generovat grafy jsem kladl hlavní důraz na to, aby knihovna uměla vytvářet svícové grafy (popsány v sekci 2.1.5.1), které jsou tolik typické pro burzovní systémy. Protože bude služba poskytující historická data agregovat záznamy po jedné sekundě, měla by knihovna být rychlá při práci s velkým počtem záznamů a umožňovat jejich další agregaci po minutách, hodinách, dnech atd.

Knihovna HighStock má kromě výše zmíněných funkcí také spoustu dalších volitelných nastavení (pro zobrazení popisků, spojení různých typů grafů do jednoho a podobně), a proto jsem jej vybral a použil v mé práci pro kreslení grafů.

3.1.2 Backend

Pro psaní backendu aplikace jsem vybral relativně mladou technologii Node.JS¹⁸, která umožňuje psát serverové aplikace v jazyce JavaScript. Tuto technologii popíšu blíže v jedné z následujících sekcí, protože *event driven architektura* (architektura řízená událostmi), kterou přináší, má některé výhody pro vytvářený systém.. Kromě Node.JS použiji také webový server Nginx¹⁹, který bude v práci vystupovat jako proxy server a v kapitole 8 popíši jeho použití ve formě load balanceru.

¹⁸<http://www.nodejs.cz/>

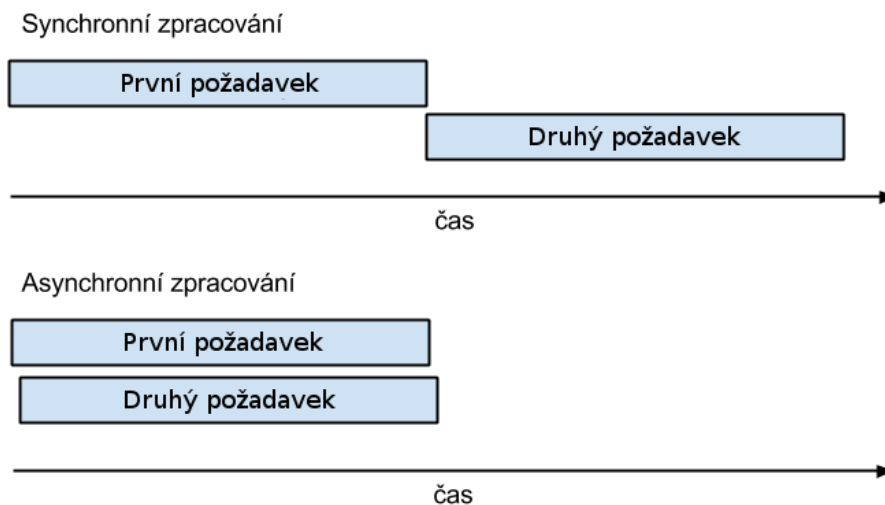
¹⁹<http://nginx.org/>

3.1.2.1 Node.JS

Node.JS je relativně mladá technologie, vyvíjená od roku 2009. První verze vyšla teprve v roce 2011 a od této doby si získává na velké oblibě hlavně použitím všeobecně známého programovacího jazyku JavaScript, pomocí kterého lze nyní psát internetové aplikace na serveru. Hlavní výhodou této technologie je událostmi řízená architektura umožňující asynchronní programování pro efektivní využití zdrojů v počítači.

Pro zpracování JavaScriptu byl použit emulátor V8²⁰ vyvinutý společností Google vzatý z internetového prohlížeče Google Chrome. Protože emulátor obsahoval mnoho zbytečností, byly některé části odebrány, refaktorovány a některé naopak přidány. Z těch zásadních změn je zde odebrání funkcí pro práci s DOM elementy v HTML stránce a naopak přidání knihovny pro práci se souborovým systémem a síťovou vrstvou. Pomocí toho lze v Node.JS pracovat se síťovými protokoly a vytvářet nad nimi například webový server, což bude i obsahem této práce. Mnoho modulů této technologie je psáno v jazyce C/C++ a zkompilováno přímo do strojového kódu, čímž Node.JS získal na rychlosti a tím pak i na oblibě.

Základní ideou v této technologii je asynchronní zpracování požadavků na *I/O operace* (vstupně/výstupní operace), jak je naznačeno na obrázku 3.1.



Obrázek 3.1: Srovnání synchronního a asynchronního zpracování dvou I/O operací.

Obrázek obsahuje srovnání synchronního a asynchronního přístupu na dvou I/O operacích, kterými může být práce s databází, čtení ze souborů, komu-

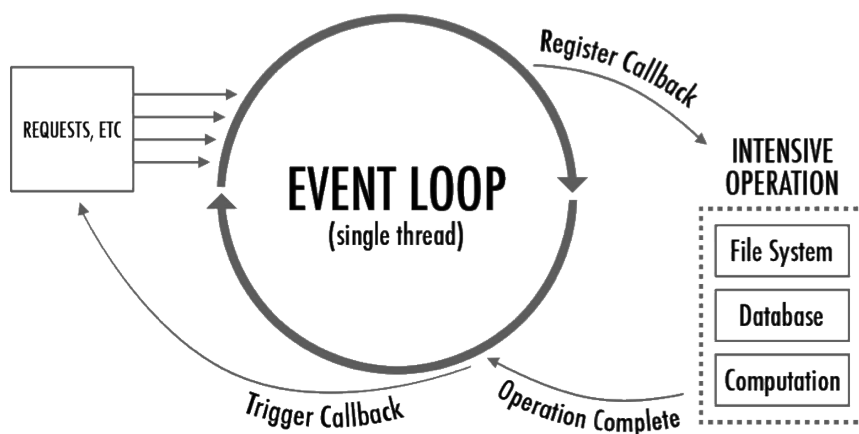
²⁰<https://code.google.com/p/v8/>

3. NÁVRH

nikace po síti atd. U synchronního modelu se operace zpracovávají za sebou a každá nová I/O operace se tak zpracovává až když je předchozí dokončena. Při zpracování více takových operací dochází k mnoha drobným čekáním, což se snaží odbourat právě asynchronní model se svým *I/O non-blocking* zpracováním. Pokud jsou na sobě dvě I/O operace nezávislé, dají se zpracovat asynchronně tím, že se obě odešlou najednou a ve zpracování se bude pokračovat až budou obě vyřízeny. Nedochází zde proto k takovému plýtvání časem jako u synchronního modelu a naopak jsou zde lépe využita I/O zařízení.

Například HTTP požadavek na webovou stránku se typicky skládá z mnoha I/O operací a pomocí asynchronního modelu tak lze ušetřit čas při jejich zpracovávání a tím zvýšit počet zpracovaných požadavků za jednu sekundu. U event driven architektury se navíc nevytváří pro každý požadavek nové vlákno/proces (které stráví mnoho času čekáním, viz výše) a tím se ušetří systémové prostředky, protože se nevytváří vždy nové prostředí pro běh vlákna, ale dojde pouze k vytvoření události a tím i zavolání obslužné funkce, což opět přispělo k snížení času potřebného na zpracování jednoho požadavku.

Příkladem nasazení technologie Node.JS do ostrého provozu a následným pozitivním zvýšením výkonu může být například služba PayPal²¹, která použitím této technologie snížila čas zpracování jednoho požadavku o 35% a tím zvýšila počet zpracovaných požadavků až na dvojnásobek oproti staré verzi aplikace psané v Javě, viz [13].



Obrázek 3.2: Zobrazení hlavního threadu, který zpracovává seznam eventů v event loop. [Zdroj: [14]]

Na obrázku 3.2 je zobrazeno požadavků uvnitř technologie Node.JS. Je zde jeden hlavní thread, který obsahuje seznam eventů (příchozí HTTP požadavky, výsledky I/O operací apod.), které postupně prochází a zpracovává

²¹<https://www.paypal.com>

jejich *callback funkce* (funkce zavolaná při dokončení operace). Při volání asynchronní I/O operace se práce předá vedlejšímu threadu, který při dokončení úkolu vloží výsledek operace spolu s callback funkcí do seznamu eventů a hlavní thread pak pomocí *event loop* (periodického procházení eventů) převezme výsledek a zavolá callback funkci, která jej zpracuje.

Z obázku vyplývá také to, že hlavní zpracování probíhá pouze v jednom vlákně, což je cena za jednoduchost event driven architektury. Při zpracování více vláken by systém neúměrně narostl na složitosti kvůli potřebě zamykání sdílených prostředků a synchronizaci, a proto je z důvodu jednoduchosti hlavní zpracování řízeno pouze jedním jádrem. Otázkou škálování na více vláken se budu zabývat v kapitole 8.

Využití pouze jednoho hlavního vlákna však přináší ještě jeden důležitý fakt, a to sice, že při zpracování tzv. *CPU intensive task* (úkolů náročných na výpočet) nemůže hlavní vlákno zpracovávat jiné události. Příkladem takového úkolu v burzovním systému je order matching algoritmus popsany 2.5. Při jeho zpracování by došlo k zablokování hlavního threadu a nebylo by možné zpracovávat jiné požadavky (přijmutí obchodního příkazu, odeslání notifikace brokerovi, atd.) a proto budou tyto úkoly delegovány na externí službu (workery) z diplomové práce Ondřeje Fremunta [2].

Node.JS je open source technologie s mnoha rozšířeními dostupnými na stránce NPM²², z nichž některé budou použity i v této práci. Pro burzovní systém bude Node.JS představovat vnitřnosti aplikačního serveru a bude fungovat jako hlavní řídicí prvek burzy rozdělující práci jednotlivým workerům.

3.1.2.2 Express

Pro tvorbu webového administračního rozhraní a zpracování HTTP požadavků využiji framework Express²³, který bude použit pro tvorbu REST API rozhraní, přes které bude proudit komunikace s dalšími částmi systému.

3.1.2.3 Nginx

Při škálování aplikací psaných v technologii Node.JS se typicky spustí více instancí na různých portech, které se zastřeší pomocí *load balanceru*, který mezi ně rozděljuje jednotlivé příchozí požadavky. Pro základní verzi burzovního systému použiji technologii Nginx pouze jako *proxy*, která umožní přistupovat na Node.JS aplikaci spuštěnou na nějakém portu jednoduše pomocí doménového jména. V kapitole 8 pak popíši, jak se dá Nginx nastavit tak, aby v systému vystupoval jako load balancer.

²²<https://www.npmjs.com/>

²³<http://expressjs.com/>

3.1.3 Databáze

V práci O. Fremunta [2] byla marketu vytvořena databázová vrstva v technologii PostgreSQL²⁴. Ta bude sloužit pro ukládání a práci s obchodními příkazy a bude proto použita jako hlavní databázová vrstva marketu. Pro ukládání dat v systému brokera, automatických obchodních systémů a služby poskytující historická data jsem vybral databázi MongoDB²⁵ a pro uložení session připojených uživatelů k serverům brokera se bude starat key-value databáze Redis²⁶.

3.1.3.1 PostgreSQL

PostgreSQL je open-source objektově relační databáze[15] poskytovaná s licencí MIT. Jedná se o databázi s dlouhou historií a rozsáhlým množstvím podporovaných funkcí, z nichž pro systém marketu budou využity především stored procedures zajišťující práci s obchodními příkazy.

Další informace o výběru a realizaci databázové vrstvy marketu jsou v již zmíněné diplomové práci [2].

3.1.3.2 MongoDB

MongoDB je dokumentově orientovaná databáze, která, oproti tabulkám u relačních databází, uchovává záznamy v tzv. kolekcích složených z jednotlivých dokumentů. Dokumenty obsahující data uložená ve formátu BSON²⁷ nemají pevnou strukturu, což pozitivně přispívá k rozšiřitelnosti systému, protože není potřeba upravovat schéma databáze při změně ukládaných dat.

Klíčovou vlastností MongoDB databáze rychlost a výkon, o což se stará především podpora shardingu na více fyzických strojů a agregačního příkazu map-reduce, který bude vysvětlen dále v textu. Ke zvýšení výkonu přispívá také možnost zanoření dokumentů, kdy za cenu duplikace dat není potřeba tvořit komplexní dotazy využívající relace, jako tomu je u relačních databází. Protože MongoDB relace neumí, je potřeba tento typ příkazu řešit na aplikační vstvě, což může snížit výkon, viz [16], a tak je použití MongoDB databáze výhodné v aplikacích, kde se používají především jednoduché nerelační dotazy.

Protože budu aplikační server jednotlivých služeb psát v technologii Node.JS, kde se dají data cachovat a nebude tak potřeba v databázi volat složité relační příkazy, využiji MongoDB jako databázovou vrstvu pro aplikaci brokera, automatického obchodního systému a služby poskytující historická data.

²⁴<http://www.postgresql.org/>

²⁵<http://www.mongodb.org/>

²⁶<http://redis.io/>

²⁷<http://bsonspec.org/>

3.1.3.3 Redis

Třetí a poslední databází využitou v burzovním systému bude key-value databáze Redis, která byla původně navržena jako in-memory databáze²⁸, nyní však obsahuje možnosti periodického ukládání dat na disk a tím i částečnou perzistenci dat. Jedná se o velmi rychlé a jednoduché úložiště pracující pouze s dvojicí klíč-hodnota, kde klíčem může být například session ID a hodnotou budou informace o uživateli, což bude využito pro ukládání uživatelských session v systému brokerů. Více o Redis databázi spolu se srovnáním výkonnosti oproti dalším úložištím je v [17].

V Node.JS aplikacích se databáze Redis používá také při škálování systému, kdy se spustí více aplikačních serverů a na pozadí se spojí pomocí Redis databáze pro společnou komunikaci a ukládání sdílených dat.

3.1.4 Komunikace

Pro komunikaci mezi obchodní platformou a brokerem bude využita technologie *AJAX* (Asynchronous JavaScript and XML) spolu s *XMLHttpRequest* požadavky fungujícími na protokolu HTTP/HTTPS. Protože v době psaní této práce neexistuje pro technologii Node.JS spolehlivá knihovna realizující FIX protokol popsáný v 2.1.4.1 a jeho tvorba spolu s navrhovanou aplikací by přesahovala rozsah této práce, použijí technologii Socket.IO pro propojení dalších prvků v systému, kterými jsou broker, burza, roboti a služba poskytující historická data. Obě služby nyní popíšu blíže.

3.1.4.1 AJAX a XMLHttpRequest

Technologie AJAX je spojení webových technologií, jako je dynamické HTML, CSS styly, DOM model jazyk JavaScript a XMLHttpRequest rozhraní pro vytvoření dynamických a interaktivních stránek. Pomocí toho lze vytvořit webovou stránku, která bude na pozadí asynchronně komunikovat se serverem a nebude tak potřeba refresh stránky při každém požadavku.

3.1.4.2 Socket.IO

Obchodní platforma bude obsahovat notifikace, které klienta upozorní, pokud se stane nějaká akce na burze. Akcí může být vyexpirování nebo vyřízení příkazu. Pro realizaci těchto notifikací využijí technologii Socket.io²⁹, která umožňuje vytvořit *Push notifikace* mezi serverem a klientem. Podle [18] je takovýto způsob notifikací úspornější na posílané požadavky (zprávy se posílají pouze ve chvíli, kdy je v systému notifikace), ale zato je systém náročnější na CPU kvůli velkému počtu neustále připojených uživatelů.

²⁸Databáze pracující s daty pouze v RAM paměti.

²⁹<http://socket.io/>

Socket.IO umožňuje navázat takovéto spojení, kde server může odesílat klientům notifikace zakódované do formátu JSON. Ten má velmi blízko k programovacímu jazyku JavaScript, který bude použit pro tvorbu aplikace a není zde proto třeba žádného speciálního překladače pro rozparsování posílané zprávy. Další pozitivní vlastností na Socket.IO je, že umí navázat spojení s klienty více způsoby podle možností prohlížeče, z nichž automaticky podle své interní heuristiky zvolí vždy tu nejlepší variantu a je tak kompatibilní i se staršími verzemi prohlížečů. Kromě propojení webového prohlížeče lze využít Socket.IO i k propojení dvou a více Node.JS serverů nebo aplikací psaných v jiných jazycích díky, u kterých je dostupná Socket.IO knihovna. V této práci jej proto použiji pro realizaci komunikační vrstvi zajišťující posílání Push notifikací mezi serverem a klientem.

3.2 Obecné části návrhu

Protože jsou všechny služby systému tvořeny v technologii Node.JS a programovacím jazyce JavaScript, budou mít některé společné znaky, které popíšu v této sekci.

3.2.1 Použité datové typy

Základní datové typy v jazyce JavaScript jsou *String*, *Number* a *Boolean*. Pro práci s *tickery* (zkratkami akcií) budu používat typ *String*. U identifikátorů položek v databázi, obchodovaných objemů a cen použiji typ *Number*, u kterého dochází k automatické konverzi mezi celočíselným typem a typem s plovoucí desetinnou čárkou. Zároveň je dostatečně veliký, aby v sobě dokázal obsáhnout všechny používaná typy používaných čísel a proto jej použiji při práci s čísly v tvořeném systému.

Ceny akcií mají v burzovních systémech pevně stanovenou tick size, což je nejmenší možný přírůstek ceny. Většinou se jedná o setiny nebo tisíce jednotky a v práci této práci bude použita tick size 0.001. Typ *Number* má pohyblivou desetinnou čárku s rozsahem větším, než je rozsah tick size a tak jej považuji za dostatečný k práci s cenami akcií.

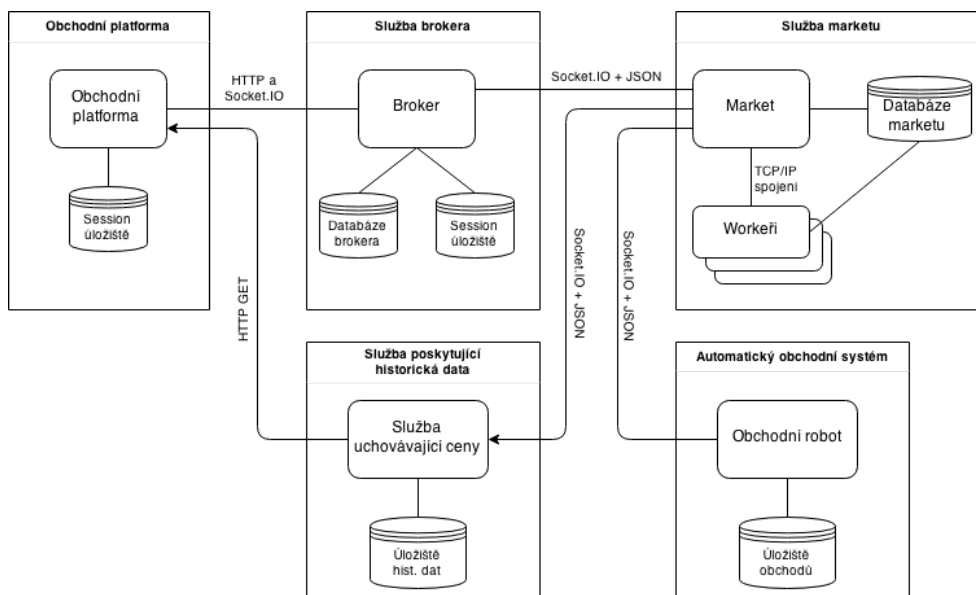
3.2.2 Webové rozhraní

Pro přehledné zobrazení aktuálního stavu bude každá služba obsahovat webové rozhraní realizováno jako SPA aplikaci dotazující se na REST API pro získání informací o aktuálním stavu.

Pro zvýšení bezpečnosti budou takto vytvořené webové aplikace komunikovat přes proxy server realizovaný v technologii Nginx s nastaveným HTTPS protokolem.

3.2.3 Služby v systému

Na obrázku 3.3 jsou zobrazeny jednotlivé služby vystupující v systému, kterým se budu v rámci této kapitoli blíže věnovat.



Obrázek 3.3: Zobrazení detailní struktury navrhovaného systému.

Systém obsahuje pět základních služeb, kterými jsou:

- **Market** - Centrální místo pro zpracování obchodních příkazů.
- **Broker** - Služba zprostředkovávající klientům obchodování na burze.
- **Obchodní platforma** - Klientská aplikace pro zadání obchodních příkazů.
- **Jednoduchý automatický obchodní systém** - Automatický robot obchodující na burze.

3.3 Market

Klíčovými otázkami, které jsem řešil při návrhu marketu, je způsob použití databázové vrstvy, možné stavy obchodních příkazů, synchronizace kritických procesů, realizace burzovního cyklu a v neposlední řadě také použití workerů. Návrh těchto částí nyní popíšu blíže.

3.3.1 Databáze a stored procedury

V systému marketu bude z práce O. Fremunta [2] využito databázové schéma spolu se stored procedurami navrženými pro databázi PostgreSQL ve verzi

3. NÁVRH

9.2 a vyšší. Stored procedury budou sloužit k práci s obchodními příkazy a jejich použití zajistí transakční zpracování, čímž se zamezí možnosti, že by zpracování příkazu z nějakého důvodu proběhlo jen částečně (například, že by došlo k zobchodování příkazu, ale neupravilo se vlastnictví akcií v databázi).

V následujícím příkladu jsou zobrazeny dva typy volání stored procedur v databázi PostgreSQL:

```
SELECT expire_order(123) as expired;
-- 1. prikaz zavola proceduru expire_order s parametrem 123
-- a vysledek volani bere jako elementarni datovy typ

SELECT * FROM get_unnotified_trades();
-- 2. prikaz vola proceduru get_unnotified_trades, ktera vraci strukturovana
-- tabulkova data a je proto volana jako pri cteni z tabulky
```

V prvním případě se očekává jednoduchá návratová hodnota (číslo, boolean, string, atd.) a v druhém případě lze pracovat se složitější strukturou v podobě tabulkových dat.

Při práci s daty v databázi bude market využívat pouze již zmíněné stored procedury navržené v diplomové práci [2].

3.3.2 Obchodní příkaz

Systém marketu bude podporovat dva typy příkazů:

- **Limit Order** - Poptávající a nabízející příkaz
- **Cancel Order** - Příkaz pro zrušení v minulosti zadaného příkazu.

Příkaz pro nákup a prodej akcií (*Limit Order*) se bude skládat z unikátního identifikátoru, ceny, množství a id akcie, která je poptávána/nabízena.

U příkazu pro zrušení (*Cancel Order*) bude jediným údajem identifikátor příkazu, který se má zrušit.

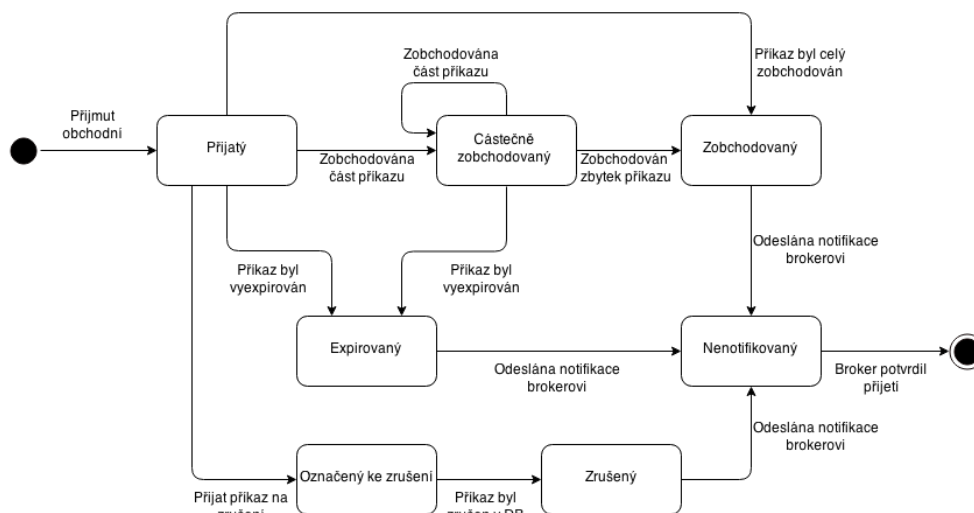
3.3.3 Stavy obchodních příkazů

Obchodní příkaz typu *Limit Order* nabývá v systému marketu během svého života (od jeho přijetí až do potvrzení notifikace o vyřízení brokerem) různých stavů, kterými jsou:

- **Přijatý** - Stav nového příkazu bezprostředně při přijetí burzou.
- **Částečně zobchodovaný** - Pokud se příkaz zobchoduje, avšak realizovaný objem bude menší než objem uvedený v obchodním příkazu.
- **Zobchodovaný** - Pokud se příkaz zobchoduje celý.
- **Označený ke zrušení** - Pokud broker odešle pokyn ke zrušení burzou v minulosti přijatého příkazu.

- **Zrušený** - Pokud burza zruší příkaz označený ke zrušení.
- **Expirovaný** - Pokud burza automaticky zruší příkaz, který je v systému déle, jak nastavený čas.
- **Nenotifikovaný** - Pokud se čeká na brokera po potvrzení odeslané notifikace.
- **Dokončený** - Pokud broker potvrdí příjem notifikace o zrušení, expiraci nebo vyřízení příkazu.

Na obrázku 3.4 je zobrazen stavový diagram obchodního příkazu a přechody mezi popsanými stavy.



Obrázek 3.4: Stavový diagram obchodního příkazu.

Při přijetí *Cancel Order* se k rušenému příkazu přidá pouze příznak na zrušení a k fyzickému zrušení příkazu dojde později, což bude popsáno v dalších sekcích.

3.3.4 Práce s obchodními příkazy

V systému marketu jsou tyto základní procesy pracující se seznamem obchodních příkazů v databázi:

- **Přijmutí příkazu** - Uloží příkaz do databáze a odešle brokerovi potvrzení o přijetí.
- **Zobchodování příkazu** - Zapiše do databáze provedený obchod, u obchodních příkazů podílejících se na obchodu upraví objem, který je potřeba ještě zobchodovat a odešle brokerům notifikaci.

- **Označení příkazu ke zrušení** - Přidá k obchodnímu příkazu příznak představující žádost o zrušení a potvrdí brokerovi přijetí požadavku.
- **Vyexpirování příkazu** - Vyexpiruje příkazy starší jak pevně stanovená doba a odešle brokerům notifikaci.
- **Zrušení příkazu** - Zruší příkazy s příznakem ke zrušení a odešle brokerům notifikaci.
- **Označení obchodu/příkazu za dokončený** - Pokud broker potvrdí přijetí notifikace, označí v databázi příkaz za dokončený (v případě že notifikace byla o zrušení nebo expiraci) a nebo jednotlivý obchod za notifikovaný (pokud notifikace byla o provedení obchodu).

Ne všechny tyto procesy mohou pracovat zároveň a tak je potřeba synchronizace kritických míst, o čemž pojednává následující sekce.

3.3.5 Burzovní cyklus a synchronizace kritických procesů

Aby se obchodní příkaz nedostal do nevalidního stavu, kterým by bylo například jeho zrušení a zobchodování zároveň, obsahuje market cyklus provádějící sekvenčně jednotlivé kritické procesy, jako je vyobrazeno na obrázku 3.5.



Obrázek 3.5: Kritické procesy v burzovním cyklu.

Při zobchodování obchodních příkazů dojde k stáhnutí všech příkazů z databáze a jejich lokálnímu zpracování workerem. V tuto chvíli je potřeba neprovádět nad obchodními příkazy v databázi žádné další úpravy, jako je jejich rušení nebo expirace. Stejně tak při rušení příkazu nesmí dojít k jeho expiraci a obráceně.

Ostatní procesy, jako je přijetí obchodního příkazu, přidání příznaku ke zrušení nebo označení za notifikovaný lze provádět kdykoliv během burzovního cyklu.

3.3.6 Workeri

Jak bylo řečeno v 3.1.2.1, hlavní zpracování aplikace psané v technologii Node.JS běží pouze v jednom vlákně. Zdlouhavé úlohy náročné na výpočet tak mohou blokovat ostatní úlohy a tím dochází ke snížení výkonu.

Řešením tohoto problému je delegování náročných úloh na externí programy, jejichž výpočet neblokuje hlavní vlákně a při jejich dokončení dojde k vrácení výsledku do hlavního vlákna a vytvoření události podobně jako u jiných I/O operací při asynchronním zpracování.

Pro tyto potřeby bude využit worker navržený v práci [2], jehož úkolem bude na jeden požadavek zpracovat obchodní příkazy vždy pro jednu konkrétní akcii. Těchto workerů bude spuštěno více a market s nimi automaticky přes TCP/IP protokol naváže spojení, které bude udržovat po celou dobu běhu systému. Pro zmenšení přenášeného objemu dat mezi marketem, workery a databází bude každý worker napojen přímo na databázi a od marketu obdrží pouze číselné id akcie, kterou má zpracovat. Z databáze si načte obchodní příkazy pro zpracování, pomocí order matching algoritmu spočítá nejlepší strike price a spáruje obchodní příkazy od nabídky a poptávky. Výsledné obchody uloží do databáze a marketu odešle ve formátu JSON informaci o výsledné ceně, zobchodovaném objemu a seznamu provedených obchodů. Návrh komunikace mezi marketem a jedním workerem je na následujícím příkladu:

```
>> Market zasle workeru Id akcie pro zpracovani obchodnich prikazu:
123

<< Worker zasle marketu informace o provedenem obchodu a jednotlivé obchody:
{
  "info": {
    "stockId": 123,      // id akcie
    "price": 490.010,    // nova cena akcie
    "totalAmount": 27,   // zobchodovane mnozstvi
    "bid": 489.960,      // nova nejlepsi nakupni cena
    "ask": 490.010,      // nova nejlepsi prodejni cena
    "strikeTime": "2015-01-04 12:08:16.369181+01" // cas vypoctu
  },
  "trades": [{
    "buyOrderId": 956,    // sparovane obchodni prikazy
                        // Id poptavaciho prikazu
    "sellOrderId": 957,   // Id nabizeciho prikazu
    "buyerId": 3,         // Broker ktery poptavku vytvoril
    "sellerId": 3,        // Broker ktery nabicku vytvoril
    "amount": 9           // Zobchodovany pocet mezi prikazy
  },
  ...                    // dalsi provedene obchody
]
}
```

Market si bude vést informace o napojených workerech a bude jim rozdělovat práci na jednotlivých akciích. Pro dosažení rovnoměrného rozvržení zátěže mezi všechny workery jim bude práce přidělována na základě prioritního plánování, kdy se workeru, jehož výpočet běžel nedávno, přiřadí nižší priorita než workeru, který běžel před ním a pro výpočet jedné akcie se vždy použije worker s nejvyšší prioritou.

Připojení marketu a workeru bude probíhat tak, že worker při svém startu vytvoří TCP server, na který se market připojí. Aby se předešlo pevné konfiguraci,

3. NÁVRH

uraci adres jednotlivých workerů, budou workeri při svém startu dynamicky zapisovat svou adresu a port do databáze, odkud si je market bude periodicky v pevně stanoveném intervalu číst a napojovat se na nové workery. Tím se zajistí automatické napojení na všechny workery.

Pokud se nepodaří s workerem navázat komunikace nebo dojde k výpadku spojení, zkusí se market připojit po čase znova a opakuje to dokud není překročen maximální počet pokusů o navázání spojení. Poté je adresa workera vymazána jak z interního seznamu marketu, tak i z databáze.

3.3.7 Market API

Komunikace s připojenými brokery bude probíhat pomocí API vytvořené v technologii Socket.IO (viz popis v 3.1.4.2). Ta umožňuje oboustranou komunikaci v podobě emitování událostí, které druhá strana zachytí a zpracuje. Přenášené zprávy budou zakódovány ve formátu JSON, což umožní pohodlné zpracování na obou stranách komunikace, protože se jedná o přirozený formát jazyka JavaScript, ve kterém budou jednotlivé části systému tvořeny.

Při komunikaci mezi marketem a brokerem budou použity události popsané v tabulce 3.1.

Při připojení brokera k API marketu (události `connection` a `reconnect`) má broker za úkol provést nejprve přihlášení pomocí události `authenticate`. Po úspěšném přihlášení může teprve využívat ostatní klientské události popsané v tabulce.

Z druhé strany bude market rozesílat notifikace o úpravě stavu objednávek (události `orderProcessed`, `orderCancelled` a `orderExpired`) a po každém burzovním cyklu bude pomocí události `updatedStockList` rozesílat informace o aktuálním stavu všech akcií na burze. Tím se zajistí, aby měli brokeri vždy nejaktuálnější informace o akciích, které pak mohou zobrazit svým klientům.

3.4 Broker

Broker bude představovat jednoduchý mezičlánek mezi připojenými klienty a marketem. Svým klientům, bude pomocí vytvořeného REST API umožňovat zadávání obchodní příkazů, které po validaci odešle marketu.

3.4.1 Databáze brokera

Pro uložení uživatelských účtů, obchodních příkazů a historie transakcí bude použita MongoDB databáze popsaná v 3.1.3.2. Hlavními kolekcemi v databázi brokera budou:

- **Clients** - Obsahuje uživatelské informace, jako je email, jméno, heslo, aktuální výše účtu.

Název	Emitent	Popis události
connection	Socket.IO	Událost emitovaná při připojení klienta.
reconnect	Socket.IO	Při obnově přerušeného spojení.
authenticate	Klient	Vstupem jsou přihlašovací údaje klienta a výstupem výsledek přihlášení.
setOrder	Klient	Událost emitována klientem pro zadání obchodního příkazu.
cancelOrder	Klient	Událost sloužící pro zrušení obchodního příkazu.
getMarketStatus	Klient	Událost pro zjištění stavu marketu.
getStockInfo	Klient	Událost pro načtení aktuálního stavu všech akcií vedených na burze.
updatedStockList	Market	Událost nesoucí data o aktuálním stavu akcií emitována marketem po každém burzovním cyklu.
orderProcessed	Market	Notifikace o zpracovaném příkazu.
orderCancelled	Market	Notifikace o zrušeném příkazu.
orderExpired	Market	Notifikace o expirovaném příkazu.
disconnect	Socket.IO	Interní událost emitovaná při ukončení spojení.
error	Socket.IO	Interní událost emitovaná při chybě.

Tabulka 3.1: Události, na kterých Market API naslouchá.

- **ClientStocks** - Obsahuje klientovi držené akcie, jejich ticker, množství, pořizovací cenu a datum nákupu.
- **Orders** - Obsahuje seznam obchodních příkazů svých klientů.

3.4.2 Práce s obchodním příkazem

Při přijetí obchodního příkazu z klientské aplikace bude v kolekci **Orders** v MongoDB databázi vytvořen záznam s těmito položkami:

- **_id (ObjectId)** - Interní id obchodního příkazu u brokera.
- **orderId (Int32)** - Id příkazu přidělené marketem při přijetí.

3. NÁVRH

- **client (ObjectId)** - Interní id klienta zadávajícího příkaz.
- **date (Date)** - Datum podání příkazu.
- **ticker (String)** - Ticker akcie.
- **price (Double)** - Požadovaná cena v limitním příkazu.
- **amount (Int32)** - Počet požadovaných nebo nabízených akcií.
- **filledAmount (Int32)** - Udává kolik akcií z příkazu bylo již zobchodováno.
- **type (Int32)** - Číselný typ příkazu (0 - sell, 1 - buy).
- **expired (Date)** - Příznak, zda byl příkaz vyexpirován.
- **cancelOrder (Date)** - Příznak, zda byl na příkaz zavolán cancel order.
- **cancelled (Date)** - Příznak, zda byl příkaz zrušen burzou.

Při odeslání příkazu marketu obdrží broker číselný identifikátor příkazu `orderId`, pod kterým jej market bude dále vést. Při dalších dotazech nebo v notifikacích pak bude použito toto id pro další práci s příkazem.

Při přijetí marketem odeslané notifikaci o zobchodování příkazu, broker navýší hodnotu `filledAmount` o zobchodovaný objem a pokud se jedná o nabízející příkaz, dojde také k navýšení finančních prostředků klienta v kolekci `Clients`. Následně dojde k vytvoření záznamu o vlastnictví akcií klientem do kolekce `ClientStocks`, který bude obsahovat tyto údaje:

- **__id (ObjectId)** - Interní id záznamu u brokera.
- **orderId (Int32)** - Id obchodního příkazu přiděleného marketem, z kterého vlastnictví vzešlo.
- **ticker (String)** - Ticker akcie.
- **amount (Int32)** - Aktuální množství akcií.
- **originalAmount (Int32)** - Původní množství akcií
- **date (Date)** - Datum vytvoření záznamu.
- **price (Double)** - Cena za kterou byly akcie pořízeny.
- **client (ObjectId)** - Vlastník akcií.

Při podání nabízejícího příkazu se nejprve ověří, zda klient má v kolekci `ClientStocks` nabízené množství akcií. Pokud ano, sníží se množství `amount` u tolika záznamů a o takové množství, aby to vyrovnalo nabízené množství a vytvoří se nabízející příkaz.

Pokud klient podá poptávající příkaz, ověří se, zda má dostatek finančních prostředků a pokud ano, dojde k jejich strhnutí z peněžního účtu a poté k vytvoření a odeslání příkazu marketu.

3.4.3 Komunikace s marketem

Komunikace s marketem bude probíhat přes Socket.IO protokol a předávané zprávy budou ve formátu JSON. Market API sloužící pro komunikaci mezi brokerem a marketem bylo popsáno v 3.3.7 a příklad navržené komunikace je v následující ukázce:

```
>> Broker emituje udalost setOrder s temito daty:
{
  type: 1,           // typ prikazu 1 - BUY, 0 - SELL
  stockId: 1,        // Id akcie
  price: 124.78,      // cena limitního prikazu
  amount: 5           // mnozství
}

<< Market ulozi prikaz a vrati jeho unikatni Id v databazi
{
  id: '13'           // Id nove vytvoreneho prikazu
}
```

V případě, že při ukládání dojde k chybě, vrátí market místo `id` položku `error` obsahující informace o chybě.

3.4.4 Klientská REST API a CORS

Komunikace obchodní platformy spolu s brokerem bude probíhat přes REST API vytvořené v Node.JS frameworku Express. Klienti budou moci využít *endpointy* definované v tabulce 3.2.

Pro přístup k API se klienti budou muset nejprve přihlásit pomocí endpointu `login`, který vrátí autorizační token. Ten bude posílán při dalších požadavcích v HTTP hlavicce `Authorization` a bude tak sloužit k ověření identity uživatele při jednotlivých požadavcích.

3.5 Obchodní platforma

Obchodní platforma pro koncové uživatele bude aplikace, kde lze pohodlně zadávat obchodní příkazy, zobrazovat stav účtu a historii transakcí. Klientská aplikace bude postavena jako webová stránka, která bude s brokerem komunikovat přes uveřejněné REST API rozhraní popsané v 3.4.4.

3. NÁVRH

Název	Typ	Popis
login	POST	Provede přihlášení klienta k API.
registration	POST	Registruje nového klienta.
news	GET	Vrátí světové události získané z externích zdrojů.
account-info	GET,POST	Vrátí nebo upraví informace o klientovi.
account-info/password	POST	Upraví heslo klienta.
stocks	GET	Vrátí seznam všech akcií.
stocks/:ticker	GET	Vrátí detail jedné akcie podle tickeru.
my-stocks	GET	Vrátí seznam klientem držných akcií.
my-orders	GET	Vrátí seznam nevyřízených obchodních příkazů.
order/buy	POST	Vytvoří poptávající obchodní příkaz.
order/sell	POST	Vytvoří nabízející obchodní příkaz.
order/cancel	POST	Vytvoří příkaz na zrušení předcházejícího obchodního příkazu.

Tabulka 3.2: REST API endpointy pro klientskou aplikaci.

3.5.1 SPA aplikace

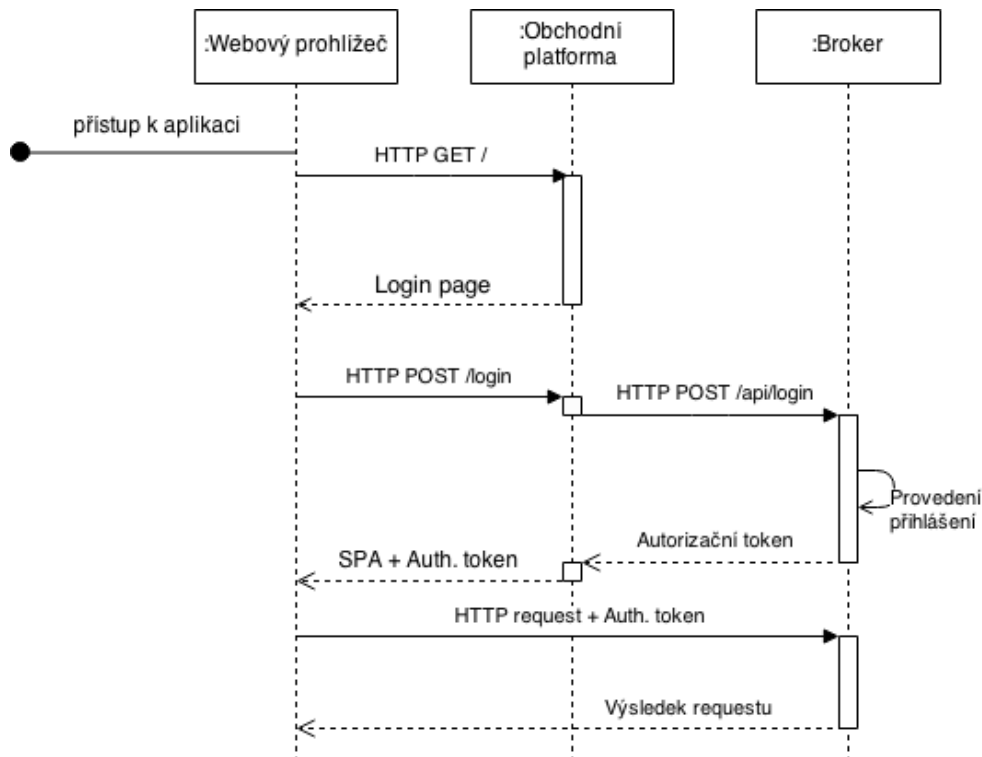
Pro realizaci obchodní platformy bude použita technologie Angular.JS a webová stránka bude mít formu SPA (Single Page Application) aplikace popsané v 3.1.1.1.

3.5.1.1 Přihlášení uživatele

Protože se u SPA aplikací stahuje celý web při prvním klientském požadavku a následně dochází už jen k asynchronní komunikaci na pozadí za účelem dotažení potřebných dat, došlo by v burzovním systému k stažení celé obchodní platformy i při prvním požadavku u nepřihlášených uživatelů.

To by mohlo být potenciální bezpečnostní riziko v podobě odhalení struk-

tury aplikace a REST API brokera, v obchodním systému proto udělám kontrolu při, zda je uživatel přihlášený a pokud není, pošle se mu pouze jednoduchá přihlašovací stránka. Přihlášení uživatele pak bude probíhat přes server poskytující obchodní platformu a je zobrazeno na obrázku 3.6.



Obrázek 3.6: Průběh uživatelského přihlášení do obchodní platformy.

Při prvním přístupu k obchodní platformě se uživatel dostane na přihlašovací stránku, kde zadá email a heslo a odešle ho na webový server obchodní platformy. Ta přepošle požadavek na klientské API brokera, kde dojde k ověření přihlašovacích údajů, přihlášení a následné vrácení autorizačního klíče. Ten se spolu se *sessionId* nastaví na webserveru obchodní platformy do Redis databáze a spolu se SPA představující obchodní platformu se odešle uživateli do prohlížeče. Následná komunikace obchodní platformy již probíhá přímo s brokerem za použití obdrženého autorizačního klíče.

3.5.2 Rozvržení stránek aplikace

Aplikace bude obsahovat tyto stránky:

- **Dashboard** - Základní sekce s přehledem všech akcí, aktuálně držných pozic a zadaných příkazů řekajících k vyřízení.

- **Seznam akcií** - Seznam všech akcií s možností zadání nákupního příkazu pro konkrétní akcii.
- **Detail akcie** - stránka s detailními údaji o akcii a grafem zobrazující vývoj ceny a zobchodovaného množství dané akcie.
- **Moje akcie** - Seznam aktuálně držených akcií a zadaných příkazů s možností prodeje akcie a zrušení zatím nevyřízeného příkazu.
- **Můj účet** - Stránka s možností úpravy uživatelských informací a nastavení nového hesla.
- **Historie** - Seznam všech proběhlých transakcí a zadaných příkazů.

Obsah jednotlivých stránek se bude načítat z REST API brokera pomocí AJAX požadavků a přechod mezi nimi zajistí modul `$route`³⁰ technologie Angular.JS tak, že vznikne jednoduchá single page aplikace.

3.5.3 Realtime Socket.IO API

Kromě HTTP požadavků bude klientská aplikace využívat také komunikaci pomocí technologie Socket.IO a bude napojena na *realtime* API brokera. Výhodou této technologie je možnost posílání zpráv mezi klientem a serverem, aniž by o ně musel klient nejprve požádat, jak je popsáno v 3.1.4.2. Pomocí této komunikace budou přenášeny notifikace událostí, které se staly na burze. Příkladem může být vyexpirování nebo zobchodování obchodního příkazu. V takovém případě si broker zjistí, zda je klient, o jehož příkaz se jedná, připojen k Socket.IO API a pokud ano, odešle mu notifikaci.

3.6 Služba poskytující historická data

Pro realizaci grafů bude potřeba ukládat historická data vývoje ceny a zobchodovaného objemu a proto bude vytvořena služba zaznamenávající stavy jednotlivých akcií.

3.6.1 Příjem aktuálních dat

Pro periodické načítání aktuálních dat ze systému marketu bude použit stejný způsob připojení jako je u brokera a to sice přes Socket.IO API vytvořené marketem a popsané v sekci 3.3.7.

Pro potřeby záznamu aktuálních dat bude služba naslouchat na event `updatedStockList`, který je emitován po každém burzovním cyklu a nese v sobě informace o jednotlivých akciích.

³⁰[https://docs.angularjs.org/api/ngRoute/service/\\$protect/T1/textdollarroute](https://docs.angularjs.org/api/ngRoute/service/$protect/T1/textdollarroute)

3.6.2 Ukládání dat a zajištění perzistence

Pro zajištění perzistence budou data ukládána do MongoDB databáze do kolekce `StockHistory`, která bude u každého záznamu obsahovat tyto údaje:

- **`_id (ObjectID)`** - Interní id záznamu v MongoDB databázi.
- **`ticker (String)`** - Ticker akcie.
- **`id (Int32)`** - Id akcie vedené na burze.
- **`price (Double)`** - Cena v daný okamžik.
- **`time (Double)`** - Počet milisekund od 1.1.1970 udávající čas vytvoření záznamu.
- **`volume (Int32)`** - Zobchodované množství akcií v daný okamžik.

3.6.3 Cachování agregovaných dat a vytvoření OHLC cen

Pro snížení zátěže na databázi se budou záznamy také na serveru agregovat v nastaveném intervalu a poté ukládat do interní *cache*, odkud se budou načítat při klientském požadavku.

Agregace dat bude probíhat tak, že se sečte zobchodovaný objem záznamů spadajících do stejného intervalu, ze všech cen v intervalu se vybere nejnížší (Low) a nejvyšší (High) hodnota a podle času záznamů se vybere otevírací (Open) a zavírací (Close) cena.

Přidání nového záznamu do agregované cache realizované polem bude probíhat pomocí následujícího pseudokódu:

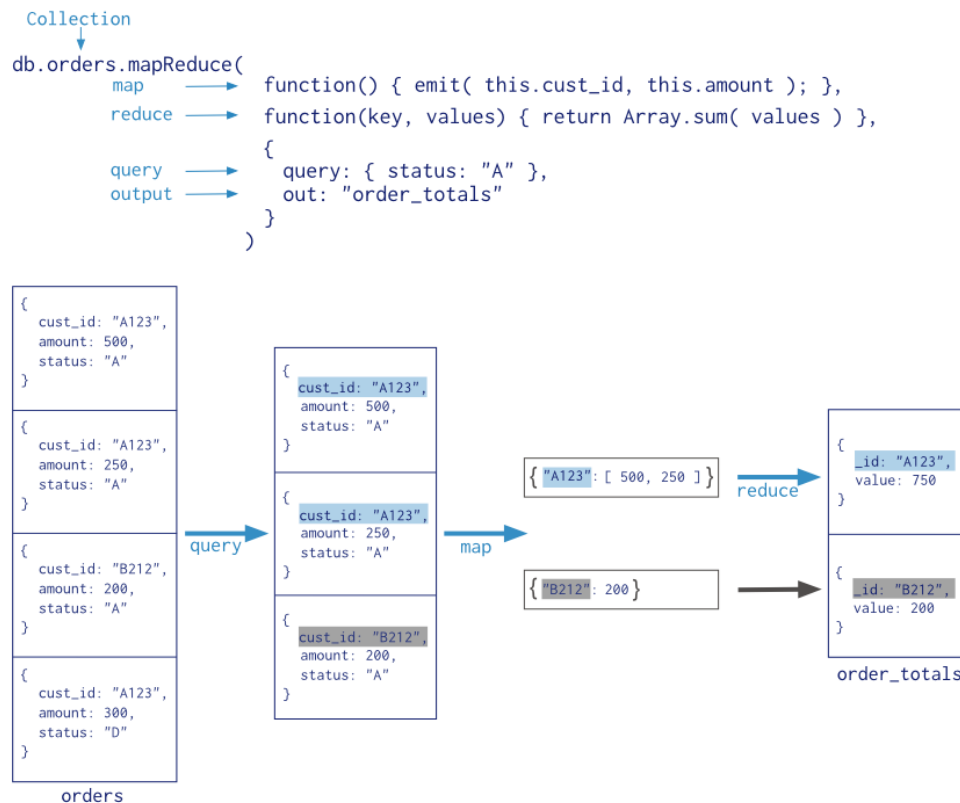
```
last = cache[cache.length - 1]; // poslední prvek v cache
if (isInterval(last.time, newItem.time)) {

    // nový záznam je ve stejném intervalu jako poslední prvek, a proto se
    // agreguje
    last.high = Max(newItem.price, last.high);
    last.low  = Min(newItem.price, last.low);
    last.close = last.price;
    last.volume += newItem.volume;
} else {

    // nový záznam je z nového intervalu, a proto se vloží jako nový prvek
    cache.push({
        volume: newItem.volume,
        open:   newItem.price,
        high:   newItem.price,
        low:    newItem.price,
        close:  newItem.price,
        time:   newItem.time
    });
}
```

3.6.4 Obnovení cache při pádu služby pomocí MapReduce

V případě pádu aplikace dojde k vymazání cache a tím i ke ztrátě agregovaných dat. Služba bude proto při svém startu provádět agregaci a načtení historických záznamů z MongoDB databáze zpět do interní cache pomocí MapReduce³¹ dotazu popsáno v dokumentaci [19], jehož popis je na obrázku 3.7.



Obrázek 3.7: Průběh dotazu MapReduce v MongoDB databázi. [dostupný z [19]]

Dotaz MapReduce má tyto 3 fáze:

1. **Query** - Selekcce dat pro další zpracování.
2. **Map** - Přípravení dat a jejich rozdělení do skupin podle klíče (na obrázku je klíčem atribut `cust_id`).
3. **Reduce** - Redukce dat ve skupinách (na obrázku dochází k sečtení všech `amount` hodnot jednotlivých položek).

³¹<http://docs.mongodb.org/manual/core/map-reduce/>

Při agregaci historických dat se budou vybírat pouze záznamy mladší jak nastavená doba, rozdělí se do skupin podle svého tickeru a intervalu a následná redukce sečte zobchodované objemy a vytvoří OHLC ceny pro jednotlivé intervaly. Tím dojde k obnovení cache do podoby před pádem a dále se bude pokračovat v přidávání dalších záznamů, jak bylo popsáno v předchozí sekci.

3.7 Automatický obchodní systém

Pro testování a zvýšení likvidity v burzovním systému bude sloužit jednoduchý robot, nebo také automatický obchodní systém (AOS), který bude připojen k marketu a podle zadané strategie bude odesílat reálné obchodní příkazy. Účelem této služby nebude dosažení zisku, ale půjde spíše o generování provozu pro možnosti zátěžového testování systému.

Návrh systému je z velké části stejný jako u služby brokera, a proto se v této sekci zaměřím už jen na specifické části, které jsou zde podstatné.

3.7.1 Vnitřní cyklus

Po připojení k marketu, stejně jak tomu bylo u brokera nebo služby poskytující historická data, se v nastaveném intervalu bude spouštět algoritmus pro vygenerování poptávajícího obchodních příkazů. V případě, že robot vlastní nějaké akcie, spustí se také nabízející příkaz a vše se odešle marketu.

3.7.2 Obchodovaná strategie

Obchodovaná strategie bude velmi jednoduchá a bude fungovat na bázi pseudo náhodnosti realizované v jazyce JavaScript pomocí funkce `Math.random()`. Při tvorbě poptávajícího příkazu u jedné akcie bude algoritmus procházet těmito fázemi:

1. Náhodné stanovení požadovaného množství. Minimální množství je 0, maximální pak pevná hranice nastavená v konfiguraci.
2. Náhodné stanovení procentuálního odchýlení od aktuální ceny akcie a následný výpočet poptávané ceny.
3. Odeslání obchodního příkazu marketu.

Pro generování nabídky bude algoritmus fungovat obdobně, přibude zde akorát kontrola, zda je nabízené množství akcií doopravdy vlastněno a zda na nej není již odeslán nabízející příkaz.

Realizace

V této kapitole popíšu stěžejní části systému, které jsem řešil při implementaci jednotlivých služeb v burzovním systému.

4.1 Market

4.1.1 Rozvržení marketu

Systém marketu obsahuje 5 hlavních tříd zajišťujících hlavní funkčnost výše navržené struktury. Jsou jimi:

- **OrderBook** - Třída zajišťující práci s obchodními příkazy, jejich ukládání, zpracování seznamu obchodů a vytvoření notifikací, expirace a rušení příkazů a další.
- **Messenger** - Třída spravující notifikace pro brokery.
- **MarketCore** - Jádru marketu, které se stará o počáteční inicializaci a následné provádění burzovního cyklu popsaného v 3.3.5.
- **MarketApi** - Třída definující rozhraní pro Socket.IO API využívané připojenými brokery.
- **WorkerManager** - Modul zajišťující připojení všech workerů a jejich správu.

Mimo výše zmíněné je zde ještě třída **StockList** poskytující aktuální seznam všech akcií a **WebRestApi** která definuje endpointy pro REST API webového rozhraní marketu.

4.1.2 Použití databázové vrstvy

Pro přístup k databázi marketu jsem použil modul *node-postgres*³², který při použití naváže *pool* spojení s PostgreSQL databází. Při dotazu do databáze se

³²<https://github.com/brianc/node-postgres>

4. REALIZACE

pak použije to spojení, které je zrovna volné. Mimo to, zmíněný modul umí také parametrizovat SQL dotazy, čímž se zabrání neoprávněnému proniknutí do databáze pomocí *SQL injection*. Příklad zjednodušené podoby použitého kódu pro volání databázové procedury je na následujícím příkladu:

```
// funkce zajistující volání SQL dotazu
DB.ps = function(ps, params, done) {

    // funkce query očekává v parametrech buď undefined a nebo array
    if (typeof params !== "undefined" && Object.prototype.toString.call(params)
        !== "[object Array]")
        params = [params];

    self.client.query(ps, params, done);
};

// vytvoří v databázi poptávající příkaz na zadanou akci akci
DB.ps("SELECT bid($1, $2, $3, $4) as inserted;", [data.broker, data.stockId, data.
    amount, data.price], function(err, res) {
    console.log("Výsledek dotazu: ", res);
});
```

Bližší popis realizace databázové třídy marketu je opět v práci [2].

4.1.3 Spuštění a inicializace marketu

Při spuštění systému dochází k inicializaci vnitřních struktur marketu. Načítá se zde asynchronně seznam všech akcií, seznam neodeslaných notifikací, viz 4.1.6, seznam registrovaných workerů a jejich následné připojení do systému a v neposlední řadě pak také spuštění Socket.IO API pro připojení brokerů a webový server spolu s REST API realizující webové rozhraní marketu. Průběh inicializace systému marketu je na následující ukázce:

```
[18:26:37.509] DB: Připojuji se k databázi postgres://market:market@localhost
:5433/market
[18:26:37.524] DB: Úspěšně připojeno k DB
[18:26:37.524] APP: Spouštím market server.
[18:26:37.671] WorkerServer: Nacítám seznam workerů
[18:26:37.672] MarketCore: Probiha init jádra marketu
[18:26:37.674] OrderBook: Inituji seznam nenotifikovaných zrušených příkazů
[18:26:37.675] StockList: Nacítám seznam akcií z DB
[18:26:37.675] OrderBook: Inituji seznam nenotifikovaných expirovaných příkazů
[18:26:37.681] OrderBook: Inituji seznam nenotifikovaných provedených obchodů
[18:26:37.681] APP: Spouštím HTTP server na portu 5555
[18:26:37.682] APP: Express server byl spuštěn
[18:26:37.687] MarketCore: Inicializace marketu byla dokončena - spouštím burzovní
cyklus
```

4.1.4 Burzovní cyklus

O realizaci continuous trading popsaného v 2.1.2 se stará metoda `MarketCore.tick`, která pomocí knihovny `async` spustí synchronně kritické procesy popsané v 3.3.4 a po jejich ukončení naplánuje další běh cyklu pomocí funkce `setTimeout`.

```
self.tick = function() {

    async.series([
        self.printTickStartMessage,

        OrderBook.processCancelOrders,
        OrderBook.processExpiredOrders,
```

```

        self.processOrders,
        MarketApi.sendBrokerStockList,

        self.printTickStopMessage,
    ], function(err) {
        if (err)
            return self.handleMarketError(err);

        // naplanuje pristi burzovni cyklus
        setTimeout(self.tick, CONFIG.orderMatchingInterval);
    });
};

```

Během jednoho cyklu se nejprve pomocí metody `OrderBook.processCancelOrders` načtou (procedura `get_uncanceled_orders`) obchodní příkazy, které byly označeny ke zrušení. Ty se postupně zruší pomocí procedury `cancel_order(order_id)` a brokerům se odešlou příslušné notifikace.

Poté se obdobně načtou příkazy pro expiraci (`get_old_active_orders(ttl)`), postupně se zruší pomocí procedury `expire_order(id)` a brokerům se odešlou notifikace.

Třetí funkcí v burzovním cyklu je `MarketCore.processOrders`, která si načte seznam akcií a spolu se seznamem aktivních workerů jej předá třídě `OrderMatching`, jejíž úkolem bude spustit workery pro zpracování jednotlivých akcií pomocí následujícího kódu:

```
async.mapLimit(stockList, workers.length, self.processStock, done);
```

Funkce `async.mapLimit` zavolá asynchronně funkci `processStock` pro zpracování každé akcie ze seznamu `stockList`. Při zpracování jedné akcie se nejprve vybere první worker ze seznamu aktivních workerů v poli `workers`, odešle se mu Id akcie pro zpracování a po přijetí odpovědi se výsledek uloží do pole výsledků a worker se vrátí na konec pole `workers`. Pomocí této práce se seznamem workerů jsem realizoval load balancing pro rozložení zátěže mezi všechny workery.

V jednu chvíli však bude zpracovááno maximálně tolik akcií, kolik je aktuálně v systému aktivních workerů (`workers.length`), při vyšším počtu by nevystačili workeri a systém by skončil chybou. Když zpracování jedné akcie doběhne, knihovna `async` automaticky spustí zpracování další tak, aby byly workeri vždy optimálně využiti.

Při dokončení zpracování všech akcií se zavolá callback funkce `done`, která uloží nové nové ceny akcií a jejich zobchodovaný objem a následně odešle brokerům notifikace o proběhlých obchodech.

Poslední zajímavou funkcí v burzovním cyklu je `MarketApi.sendBrokerStockList`, která odešle připojeným brokerům aktuální informace o akciích. Seznam akcií se tedy odesílá při každém burzovním ticku, aby brokeři mohli svým klientům zobrazit vždy tu nejaktuálnější cenu akcie.

4.1.5 Správa workerů

V initační fázi aplikace dojde k načtení seznamu workerů z databáze pomocí stored procedury `get_strikers()`. Tento seznam obsahuje dvojice IP a PORT, na kterých workeri čekají na spojení. Záznam o svém umístění vkládá každým worker při startu, a market tak nepotřebuje pevnou konfiguraci workerů.

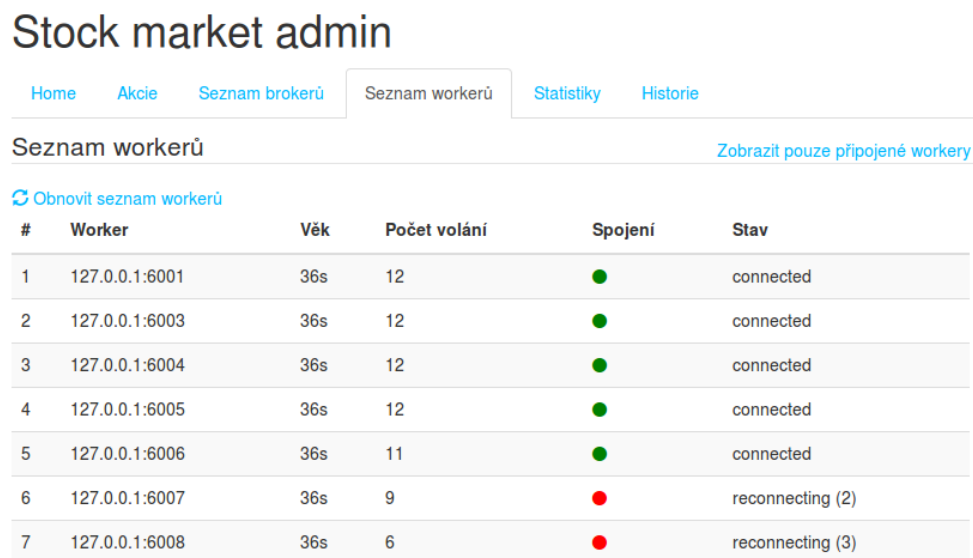
Pro dodatečné škálování za běhu lze spustit načtení workerů z databáze i z webového rozhraní aplikace v sekci **Seznam workerů**.

4.1.5.1 Připojení workerů do systému

Při načtení workerů z databáze si market nové workery přidá do interního seznamu a zkusí se ke každému připojit pomocí TCP/IP socketu. Pokud je připojení úspěšné, přidá workera i do seznamu aktivních workerů. V opačném případě zvýší proměnnou `connectAttempts` představující počet nepovedených pokusů o spojení a naplánuje další připojení. Pokud připojení selže vícekrát, vyřadí jej market ze seznamu workerů a poté i z databáze.

Pokud připojený worker ukončí spojení nebo v socketu dojde k chybě, market jej odstraní ze seznamu aktivních workerů a dále postupuje jako v případě chybného pokusu o připojení (naplánuje *reconnect* a po nastaveném počtu pokusů jej odstraní).

Seznam workerů viditelný ve webovém rozhraní v sekci Seznam workerů je vidět na obrázku 4.1.



Stock market admin					
Home Akcie Seznam brokerů Seznam workerů Statistiky Historie					
Seznam workerů Zobrazit pouze připojené workery					
Obnovit seznam workerů					
#	Worker	Věk	Počet volání	Spojení	Stav
1	127.0.0.1:6001	36s	12	●	connected
2	127.0.0.1:6003	36s	12	●	connected
3	127.0.0.1:6004	36s	12	●	connected
4	127.0.0.1:6005	36s	12	●	connected
5	127.0.0.1:6006	36s	11	●	connected
6	127.0.0.1:6007	36s	9	●	reconnecting (2)
7	127.0.0.1:6008	36s	6	●	reconnecting (3)

Obrázek 4.1: Seznam připojených workerů ve webovém rozhraní marketu.

4.1.5.2 Použití workerů a rozložení zátěže

TODO - smazat ? Jak již bylo řečeno v sekci XYZ, při použití workerů se volá funkce `async.mapLimit`, která asynchronně zpracovává na burze vedené akcie. Maximálně však zpracovává tolik akcií, kolik je aktivních workerů.

Při zpracování jedné akcie se vyjme první prvek z pole aktivních workerů a do naslouchajícího socketu se pošle Id akcie. Při vrácení dat se worker opět vrátí do seznamu aktivních workerů, aby byl připraven pro zpracování dalších akcií. Při vracení se však worker vkládá na konec seznamu, čímž je zajištěno optimální využití všech aktivních workerů.

4.1.6 Messenger a komunikace s brokerem

Při expiraci, zrušení a nebo zobchodování obchodního příkazu je přes třídu **Messenger** odeslána brokerovi notifikace o události. Pokud broker neodpoví do pevně stanovené doby a nebo není připojen, vloží se notifikace do fronty notifikací příslušného brokera a pokud je připojený, odešle se mu po čase znova. Pokud broker připojený není, čeká se na jeho připojení a pak je mu odeslána celá fronta.

Pokud broker potvrdí přijetí, je notifikace označena za vyřízenou pomocí procedury `set_order_completion_notified` nebo v případě, že jde o notifikaci o zobchodovaném příkazu, použije se `set_trade_buyer_notified` nebo `set_trade_seller_notified` podle toho, která strana obchodu potvrdila přijetí. Pokud je notifikace ve frontě, odstraní se i odtud.

V případě pádu serveru dojde k vymazání fronty notifikací a je tak potřeba ji při následném spuštění obnovit z databáze do stavu před pádem. Z toho důvodu se ve třídě **OrderBook** při počáteční inicializaci načtou nenotifikované zrušené, expirované a zobchodované obchodní příkazy (pomocí stored procedur `get_unnotified_canceled_orders`, `get_unnotified_expired_orders` a `get_unnotified_trades`) a následně se odešlou přes třídu **Messenger** znova.

4.1.7 Webové rozhraní

Pro službu marketu bylo vytvořeno také webové rozhraní, které slouží k zobrazení aktuálního stavu služby. Obsahuje informace o připojených workerech, brokerech a akciích vedených na burze. Obrázek 4.2 zobrazuje webové rozhraní spolu s náhledem na detail akcie vedené v marketu.

4. REALIZACE

Stock market admin

[Home](#) [Akcie](#) [Seznam brokerů](#) [Seznam workerů](#)

Detail společnosti Imaginary Company (IMCO)

Kód:	IMCO	Počet akcií:	300
Cena:	\$406.37	Změna:	0.00%
Akce:	emise editace		

Graf vývoje ceny

[zobrazit](#) / [skrýt](#)

Seznam příkazů

[zobrazit](#) / [skrýt ID záznamů](#)

Poptávka				Nabídka					
#		Součet	Množství	Cena	Množství	Součet	Zob. Objem	Previs	
0		73	0	\$389.00	1	1	1	72	
1		73	26	\$399.83	0	1	1	72	
2		47	25	\$401.24	0	1	1	46	
3		22	19	\$406.05	0	1	1	21	
4		3	0	\$406.37	10	11	3	8	
5		3	3	\$407.88	0	11	3	8	
6		0	0	\$408.41	57	68	0	68	
7		0	0	\$409.67	5	73	0	73	
8		0	0	\$410.86	75	148	0	148	
9		0	0	\$416.61	5	153	0	153	
10		0	0	\$421.47	8	161	0	161	

Obrázek 4.2: Detail akcie ve webovém rozhraní marketu.

4.2 Broker

4.2.1 Databáze

Při realizaci databázové vrstvy brokera jsem použil MongoDB databázi ve verzi 2.4.6. Její použití z aplikace zajistil modul *mongoose*³³, který pomocí definované struktury vytvořil databázový objekt pro používané kolekce. Příklad definování kolekce sloužící k uchování uživatelských informací ve službě brokera je na následujícím příkladu:

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var ClientSchema = new Schema({
  name: String,
  email: String,
  salt: String,
  lastUsedKey: {type: String, default: null},
  hash: String,
  password: String,
  accountBalance: {type: Number, default: 0},
  updated: {type: Date, default: Date.now},
```

³³<http://mongoosejs.com/>


```

    regDate: {type: Date, default: Date.now}
  });

  ClientSchema.statics.incBalance = function(userId, amount, cb) {
    mongoose.model('Client').update({_id: userId}, {$inc: {accountBalance: amount}}, cb);
  };

  ClientSchema.statics.decBalance = function(userId, amount, cb) {
    mongoose.model('Client').update({_id: userId}, {$dec: {accountBalance: amount}}, cb);
  };

  module.exports = mongoose.model('Client', ClientSchema);

```

4.2.1.1 Transakce v MongoDB

V databázi MongoDB chybí podpora relací, což je cena za vysoký výkon, který je u této databáze prioritou. V systému brokera a automatických obchodních systémů je však transakční zpracování nutností a tak se tato databáze ukázala jako nedostatečná pro tento typ aplikací. Může se tak stát, že v aplikaci brokera dojde pouze k částečnému zpracování příkazu a není zde tedy dodržen nefunkční požadavek na spolehlivost a perzistenci uvedený v 2.2.2.

Pro vyřešení tohoto problému by se dala použít databáze TokuMX³⁴, která je optimalizovaným klonem MongoDB s možností ACID transakcí. V této práci jsem se však touto možností již z časových důvodů nezabýval.

4.2.2 Komunikace s marketem

Komunikace s API trhu popsané v 3.3.7 probíhá pomocí následujícího kódu, který zajistí také reconnect v případě ztráty spojení.

```

this.connectToStockServer = function(conf) {
  var addr = conf.addr + ":" + conf.port;
  log.message("Connecting to stock server " + addr).yellow);

  addr = "http://" + addr;
  socket = SocketIO.connect(
    addr, {
      'force new connection': true,
      'reconnection delay': 100,
      'reconnection limit': Infinity,
      'max reconnection attempts': Infinity,
    }
  );

  socket.on('updatedStockList', BROKER.updatedStockList);
  socket.on('orderProcessed', BROKER.orderProcessed);
  socket.on('orderCancelled', BROKER.orderCancelled);
  socket.on('orderExpired', BROKER.orderExpired);
  socket.on('connect', self.onConnected);
  socket.on('disconnect', self.onDisconnected);
};

```

³⁴<http://www.tokutek.com/tokumx-for-mongodb/>

4. REALIZACE

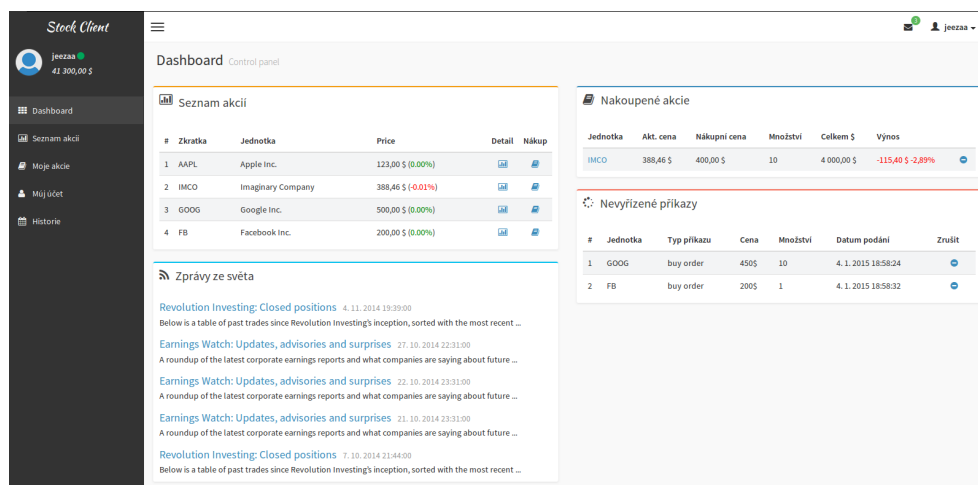
4.2.3 API pro obchodní platformu

4.2.3.1 REST API

4.2.3.2 Realtime Socket.IO API

4.3 Obchodní platforma

Při realizaci obchodní platformy pro koncové uživatele jsem využil grafickou šablonu AdminLTE, kterou jsem upravil a vytvořil v ní jednotlivé stránky, jako je například základní dashboard aplikace (zobrazený na obrázku 4.3), sekce s vypisem klientových akcí a zadaných příkazů nebo dalších částí, jejichž náhledy jsou v příloze C.



Obrázek 4.3: Dashboard klientské aplikace.

4.3.1 Oddělení sekce pro přihlášené uživatele

Pro rozdělení aplikace na sekci pro přihlášené a nepřihlášené uživatele jsem definoval jednotlivé adresy, jejich obslužné funkce a poté middleware funkce `authOnly` a `unAuthOnly`, které jsou zavolány před vlastním zpracováním requestu a mají za úkol přesměrovat uživatele na přihlašovací stránku v případě, že není přihlášený nebo opačně do aplikace, pokud uživatel přihlášený je a je v sekci pro nepřihlášené. Toto zajišťuje následující kód:

```
var Login = {
  authOnly: function (req, res, next){
    if(!req.isAuthenticated())
      return res.redirect("/login");
    next();
  },
  unAuthOnly: function (req, res, next){
    if(req.isAuthenticated())
      return res.redirect("/");
    next();
  }
}
```

```

    }
}

// sekce pro neprihlasene
app.get('/login', Login.unAuthOnly, client.login );
app.get('/registration', Login.unAuthOnly, client.registration );
app.post('/login', Login.unAuthOnly, client.doLogin );

// sekce pro prihlasene
app.get('/logout', Login.authOnly, client.logout );
app.get('/', Login.authOnly, client.index);
app.get('*', Login.authOnly, client.index );

```

4.3.2 Nastavení CORS a autorizačního tokenu pro přístup k API brokera

Protože klientskou aplikaci představuje webová stránka využívající REST API z jiné domény, je potřeba v aplikaci nejprve povolit CORS requesty. Spolu s tím nastavují také autorizační token `Authorization` předávaný v HTTP hlavičce requestů při komunikaci s API. Konfigurace requestů v modulu `$http`, který je použit pro komunikaci s API vypadá takto:

```

// Nastaveni autorizacniho tokenu pro pristup k API
$httpProvider.defaults.headers.common["Authorization"] = CONF.key;

// Komunikace je ve formatu JSON
$httpProvider.defaults.headers.common["Content-type"] = "application/json";

// Povoleni CORS requestu
$httpProvider.defaults.useXDomain = true;
delete $httpProvider.defaults.headers.common['X-Requested-With'];

```

Proměnná `CONF.key` obsahující autorizační token se získá při prvním načtení stránky ze session, kde byla uložena po přihlášení do obchodní platformy, jehož průběh je popsán v 3.5.1.1.

4.3.3 Spuštění obchodní platformy pod proxy serverem

Obchodní platforma, ale i další služby v systému, se spouští na nastaveném portu, jehož konfigurace je v souboru `config/public.js`. Pro přístup k aplikaci pomocí doménového byl použit Nginx v roli proxy serveru s touto konfigurací:

```

server {
    listen    443;

    ssl      on;
    ssl_certificate    /etc/ssl/your_domain_name.pem; (or bundle.crt)
    ssl_certificate_key    /etc/ssl/your_domain_name.key;

    server_name client.stare.cz;

    location / {
        proxy_pass    http://localhost:5000;
        proxy_redirect off;

        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
    }
}

```

4. REALIZACE

```
        proxy_set_header Host $host ;
        proxy_set_header X-Real-IP $remote_addr ;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for ;
        proxy_set_header X-Forwarded-Proto https;
    }
}
```

U dalších webových služeb, které byly vytvořeny v této práci, je konfigurace obdobná.

4.4 Služba poskytující historická data

Pro potřeby zobrazení grafů jsem vytvořil službu, která zaznamenává ceny po každém ticku³⁵ marketu a ty poté v agregované formě poskytuje klientům přes veřejné REST API.

4.4.1 Příjem aktuálních dat

Služba je napojena na market podobně jako broker, naslouchá však pouze na jeden event `updatedStockList`, který je emitován marketem a obsahuje v sobě údaje o aktuální ceně a zobchodovaném množství u všech akcií. Příklad těchto dat je na obrázku 4.4.

```
1  {
2      error: null,
3      stockList:
4      [{
5          id: 1,
6          company_id: 1,
7          issued: '2013-12-31T23:00:00.000Z',
8          shares: 500000,
9          ticker: 'VRCT',
10         name: 'Vercotti A',
11         strikeTime: 1419984758936,
12         price: 137.440,
13         bid: 137.430,
14         ask: 137.420,
15         volume: 1300,
16     },
17     ... // data dalsich akcií
18 }
```

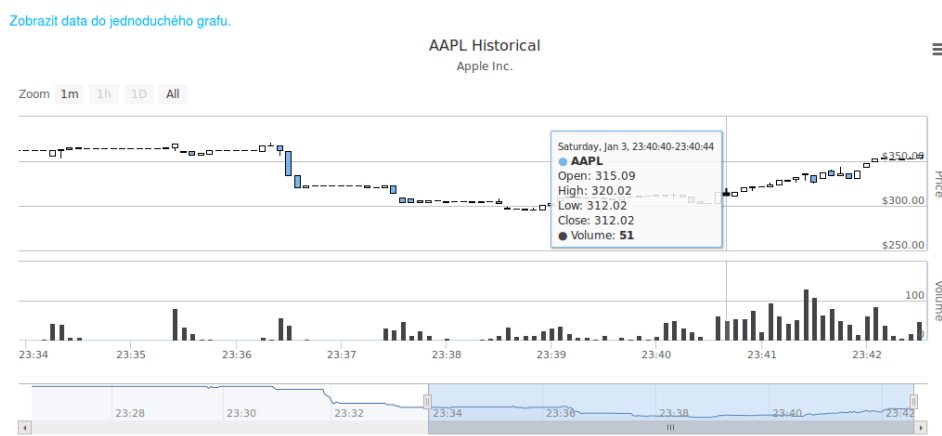
Obrázek 4.4: Marketem emitovaný JSON s aktuálními daty všech akcií.

³⁵Jeden tick představuje jeden burzovní cyklus.

4.4.2 Ukládání dat a interní cache

Přijatá data jsou uložena do MongoDB databáze, která slouží jako perzistentní vrstva aplikace. Aby nedocházelo k zbytečnému čtení a agregaci dat z databáze při každém klientském požadavku na historická data, vloží se data také do interní cache, která je realizována polem obsahujícím záznamy agregované po jedné sekundě. Každý záznam obsahuje celkový zobchodovaný objem v daném intervalu a pak čtveřici OHCL cen (open, high, close, low), které jsou popsány v 2.1.5.1. Přidání záznamu do pole probíhá takto:

kteřé jsou pak zobrazeny v podobě svícových grafů, viz obrázek 4.5.



Obrázek 4.5: Graf v aplikaci pro zobrazení historického vývoje ceny a volume akcie.

4.4.3 Agregace pomocí MapReduce

TODO - pridať ukážku map reduce ?

4.4.4 REST API s podporou CORS

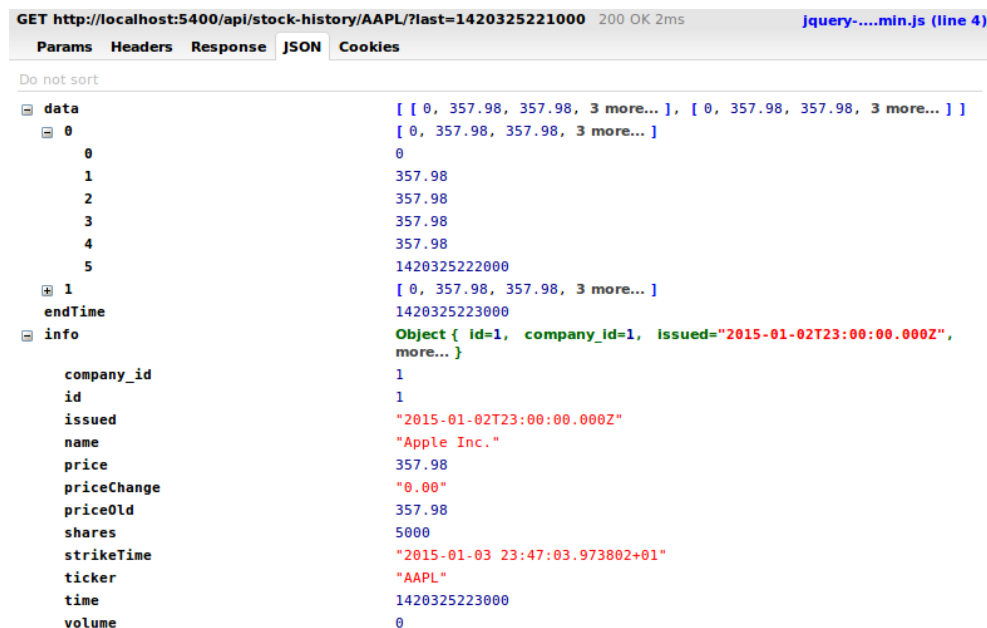
Data služby jsou veřejně poskytovány pomocí REST API. Pro načtení historických dat slouží endpoint `/api/stock-history/:ticker/`, který má jeden povinný parametr a to sice ticker akcie, pro kterou má vrátit historická data.

Aby se zamezilo načítání stejných dat, má ještě nepovinný GET parametr `?last=`, kterým se dá určit čas, od kterého se mají záznamy vypisovat. Typická se tak první request volá bez parametru `last` a další již s nastaveným `last` na poslední přijatý čas z předchozího requestu.

Příklad komunikace s REST API je na následujícím obrázku 4.6.

Protože REST API může běžet na jiné doméně, než cílový klienti, kteří budou data načítat, je potřeba povolit tzv. Cross-origin resource sharing (CORS),

4. REALIZACE



Obrázek 4.6: Komunikace s REST API služby poskytující historická data a formát vrácených dat.

pro podporu AJAX dotazů na cizí doménu. Defaultně je totiž přístup ke zdrojům pomocí AJAX requestů cizí doménu z bezpečnostních důvodů zakázán, viz [20]

4.5 Automatický obchodní systém

Pro potřeby testování a zvýšení likvidity v systému byl vytvořen také jednoduchý automatický obchodní systém (AOS), jehož účelem je generovat nákupní a prodejní příkazy.

Hlavními třídami jsou zde:

- **RobotUI** - Třída zpracovávající obchodní strategii.
- **MarketClient** - Třída zajišťující práci s marketem.
- **RobotApi** - REST API pro potřeby webového rozhraní.

4.5.1 Konfigurace

Konfigurace robota se provádí v souboru `config/public.js` a kromě přístupů k marketu a databázi obsahuje i další konfigurační proměnné ovlivňující chování robota, viz následující příklad:

```
robotConf.ui = {
  interval: 1000,      // intervalu zpracovani v milisekundach
  delay: 5000,         // zpozdeni prace robota po startu v milisekundach
  stocks: ["AAPL"],    // zadane spolecnosti, ktere bude robot sledovat
  buy_limit: 500,      // maximalni pocet akcií poptavanych v jednu chvili
  price_range: 5,      // maximalni odchylka od aktualni ceny v procentech
};
```

4.5.2 Cyklus robota

Zpracování robota probíhá ve třídě `RobotUI` periodicky jednou za nastavený počet milisekund, což jsem v jazyce JavaScript realizoval pomocí funkce `setTimeout` a celý kód vypadá takto:

```
function robotProcess() {
  // pockame, dokud nebudeme pripojeni k marketu
  if (!serverStatus.get("authenticated") || ROBOT_TURN_ON == false)
    return setTimeout(robotProcess, ROBOT_PROCESS_INTERVAL);

  // provedeme asynchrone buy a sell zpracovani
  async.parallel([
    sellProcess,
    buyProcess
  ], function(err, done) {
    setTimeout(robotProcess, ROBOT_PROCESS_INTERVAL);
  });
}
```

4.5.3 Obchodní algoritmus

TODO - ? pridať kod?

Deployment

V této kapitole se budu zabývat spuštěním jednotlivých služeb. Vývoj aplikace byl z velké části prováděn na operačním systému Linux, a proto i návod na spuštění bude směřován pro toto prostředí. Aplikace psané v technologii Node.JS jsou však multiplatformní, a pro jejich spuštění na jiných operačních systémech by tak neměly být potřeba žádné větší úpravy.

5.1 Konfigurace služeb

Konfigurační soubor `config/public.js` u každé služby obsahuje nastavení pro *development* a *production* prostředí, jsou zde nastaveny přístupy k databázím a porty, na kterých jsou služby spuštěny. Na těchto portech běží také webové rozhraní jednotlivých služeb. Defaultní nastavení portů pro produkční prostředí jsou:

- **Klient:** 5000
- **Broker:** 5100
- **Market:** 5555
- **Obchodní roboti:** 5200 a 5201
- **Agregační služba:** 5400

Pro pohodlné spuštění služeb jsem vytvořil několik scriptů, které budou popsány dále. Scripty jsou umístěny ve složce `tools/` a jejich hlavním úkolem je nahrání databáze, provedení základní konfigurace, spuštění a zastavení služeb v produkčním nebo development prostředí.

5.2 Kompilace a spuštění workerů

Pro spuštění workerů z práce [2] je nejprve potřeba kód workeru v adresáři **Striker/C++/** zkompileovat příkazem **make**. Tím se vytvoří spustitelný soubor **striker**, který očekává tři parametry. Prvním je adresa, na které worker bude naslouchat (využití tohoto parametru má význam především u systémů s více síťovými rozhranními), druhým je port, na kterém služba poběží a třetím je cesta ke konfiguračnímu souboru **striker.conf**, který je umístěn ve stejné složce jako zdrojové soubory. Kompilace, spolu se spuštěním workeru, je zobrazena na následujícím příkladu (výpis byl na některých místech zkrácen).

```
# kompilace workeru ve slozce Striker/C++/
$ make
c++ -Wall -I/usr/local/include -I/usr/local/pgsql/include -c Striker.cpp
c++ -Wall -I/usr/local/include -I/usr/local/pgsql/include -c StrikerConfig.cpp
c++ -Wall -I/usr/local/include -I/usr/local/pgsql/include -c TcpServer.cpp
...
c++ -Wall -o mockmarket MockMarket.cpp
c++ -Wall -o decimal TestDecimal.cpp Decimal.o decDouble.o decQuad.o decContext.o

# spusteni workeru na portu 6001 se zadanou konfiguraci
$ ./striker 127.0.0.1 6001 striker.conf
toBool: true -> 1
toBool: false -> 0
...
DATABASE_PASSWORD = market
MARKET_ADDRESS = 127.0.0.1
MARKET_CHECK = 1
TICK_SIZE = 0.001
VERBOSE = 0
Striker listens for Market on IP address = 127.0.0.1, port = 6001
```

Kompilaci workeru také zajistí vytvořený shell script **tools/install**, jehož funkcionality bude rozebrána dále v 5.5.

5.3 Instalace použitých technologií

Pro nasazení burzovního systému jsou potřeba tyto technologie:

- **Node.JS** verze 0.10.35
- **NPM manager** pro instalaci dalších modulů ve verzi 1.4.28
- **PostgreSQL** verze 9.2
- **MongoDB** verze 2.4.6
- **Redis** verze 2.4.16

Instalace těchto technologií pod operačním systémem Debian bude obsahem dalších sekcí. Celý postup byl také zapsán do skriptu **tools/install_prerequisites** a lze jej tak použít pro pohodlnou instalaci všech služeb.

5.3.1 Node.JS a NPM manager

Pro instalaci Node.JS a NPM lze použít tento postup:

```
# instalace prerekvizit
sudo apt-get install curl

# pridani repozitare pro instalator apt-get
sudo curl -sL https://deb.nodesource.com/setup | bash -

# instalace node.js spolu s NPM
sudo apt-get install -y nodejs
```

5.3.2 MongoDB

Pro instalaci jednotlivých technologií lze použít tento postup:

```
# pridani klice pro repozitar
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv 7F0CEB10

# pridani repozitare do apt-sources
echo 'deb http://downloads-distro.mongodb.org/repo/debian-sysvinit dist 10gen' |
sudo tee /etc/apt/sources.list.d/mongodb.list

# stazeni seznamu z apt-sources
sudo apt-get update

# instalace MongoDB
sudo apt-get install -y mongodb-org
```

5.3.3 Redis

Instalace Redis databáze se udělá pomocí následujícího kódu:

```
sudo apt-get update
sudo apt-get install build-essential
sudo apt-get install tcl8.5
wget http://download.redis.io/releases/redis-2.8.9.tar.gz
tar xzf redis-2.8.9.tar.gz
cd redis-2.8.9
make
make test
sudo make install
cd utils
sudo ./install_server.sh
```

5.3.4 PostgreSQL

PostgreSQL lze nainstalovat jednoduše použitím **apt-get** manager:

```
sudo apt-get install postgresql postgresql-client
```

5.4 Inicializace databází

Služba pro poskytování historických dat, broker a automatický obchodní systém využívají ke svému běhu databázi MongoDB, kde se jednotlivé databáze a kolekce vytvoří při jejich prvním použití. Není zde proto potřeba žádné počáteční inicializace.

U brokera a obchodní platformy se využívá databáze Redis, jejíž použití také nepotřebuje žádnou počáteční inicializaci.

Pro službu marketu je použita databáze PostgreSQL, jejíž schéma navržené v práci [2] je uloženo v souboru `Strike/Postgres/market.sql`.

Pro nahrání čisté verze databáze marketu a promazání databází ostatních služeb jsem vytvořil script `tools/init_database`, který v prvním parametru očekává IP adresu naslouchající PostgreSQL databáze a v druhém pak její port. Inicializace databáze se udělá například takto `tools/init_database 127.0.0.1 5432`.

Při prvním spuštění služby marketu je potřeba nastavit přístupy jednotlivých služeb, jako je například broker nebo služba zaznamenávající historická data. Vytvořil jsem proto také script `tools/config_database`, který očekává stejné parametry, jako v předchozím případě a jeho úkolem je nastavit defaultní přístupové údaje, které mají nastavené i jednotlivé služby ve svých konfiguracích.

5.5 Instalace závislostí

Před spuštěním služeb psaných v technologii Node.JS je potřeba nainstalovat použité moduly. K tomu slouží `npm` manažer, který podle seznamu závislostí uvedených v souboru `package.json` stáhne potřebné moduly do složky `./node_modules`, odkud jsou pak dále použity aplikací. Pro instalaci závislostí jsem vytvořil shell script `install` volající `npm install` na jednotlivých službách, a pro jeho spuštění tak stačí zavolat příkaz `./tools/install`.

Kromě instalace závislostí pro Node.JS aplikace script zajistí také automatickou kompilaci workera a jeho zkopírování spolu s konfiguračním souborem do složky `tools/`, odkud jej pak využívají scripty pro spuštění rozebrané v následující sekci.

5.6 Spuštění Node.JS aplikací

Ve složkách jednotlivých služeb je soubor `app.js`, což je hlavní soubor aplikace a jeho spuštění se provede pomocí příkazu `node app.js`. To jej spustí s defaultní konfigurací pro development prostředí. Spuštění aplikace v produkčním (a obdobně i jiném) prostředí definovaném v konfiguračním souboru se provede příkazem `NODE_ENV=production node app.js`.

Pro usnadnění vývoje jsem pro spuštění aplikace použil modul `nodemon` nainstalovaný pomocí `sudo npm install nodemon -g`, kde parametr `-g` zajistí instalaci modulu jako globální služby. Tu pak lze použít příkazem `nodemon app.js`, což zajistí *livereload* chování při úpravě zdrojového kódu aplikace.

V ostrém provozu je potřeba, aby se aplikace při pádu znovu nastartovala, což ani jeden z výše zmíněných příkazů nezajistí. Pro spuštění služeb

v produkčním prostředí jsem proto použil modul `forever`, který se opět nainstaluje pomocí příkazu `sudo npm install forever -g` a zajistí případný restart služby po jejím pádu. Na obrázku 5.1 je zobrazeno spuštění služby spolu se základní funkcionalitou modulu `forever`. V obrázku je také vidět, že se při spuštění služby vytvořil log soubor `/home/cenda/.forever/qmim.log` obsahující výstupy z běžící aplikace.

```
# spustí službu marketu s produkční konfigurací
$ NODE_ENV=production forever start Market/app.js
info:   Forever processing file: Market/app.js

# vypíše běžící služby
$ forever list
info:   Forever processes running
data:   uid command          script      forever pid  logfile                                uptime
data:   [0] qmim /usr/local/bin/node Market/app.js 28031  28036 /home/cenda/.forever/qmim.log 0:0:0:6.997

# restartuje službu s id 0
$ forever restart 0
info:   Forever restarted process(es):
data:   uid command          script      forever pid  logfile                                uptime
data:   [0] qmim /usr/local/bin/node Market/app.js 28031  28036 /home/cenda/.forever/qmim.log 0:0:0:23.81

# restartuje všechny běžící služby
$ forever restartall
info:   Forever restarted processes:
data:   uid command          script      forever pid  logfile                                uptime
data:   [0] qmim /usr/local/bin/node Market/app.js 28031  28068 /home/cenda/.forever/qmim.log 0:0:0:8.973

# zastaví službu s id 0
$ forever stop 0

# zastaví všechny běžící služby
$ forever stopall
```

Obrázek 5.1: Funkcionalita příkazu `forever` a spuštění jedné služby v produkčním prostředí.

Pro rychlé spuštění všech služeb burzovního systému pomocí příkazu `forever` a pěti workerů na portech 6001 až 6005 slouží scripty `tools/start_development` (pro development prostředí) a `tools/start_production` (pro produkční prostředí) a celá instalace spolu se spuštěním služeb v produkčním prostředí na jednom počítači se udělá pomocí příkazů:

```
# smazani databazi a nahrani noveho schematu
./init_database 127.0.0.1 5432
# konfigurace pristupovych udaju sluzeb k marketu
./config_database 127.0.0.1 5432
# kompilace workeru, instalace zavislosti
./install
# spusteni peti workeru a vseh sluzeb v systemu
./start_production
```

A po jejich provedení by měly být dostupné webová rozhraní služeb na portech uvedených v sekci 5.1.

Monitoring a ladění

V této kapitole popíši možnosti monitoringu a ladění aplikací vytvořených v technologii Node.JS, které jsem využil při vývoji burzovního systému. Monitorovací služba pak bude použita také v další kapitole 7 pro účely měření při zátěžovém testování.

6.1 Monitoring Node.JS aplikace

Z monitorovacích nástrojů jsem vybral službu Nodetime³⁶, která umožňuje měřit výkon aplikace, využití systémových prostředků, počty požadavků na databázi, vytížení Socket.IO spojení a další metriky, které jsou odesílány na externí servery služby Nodetime. Tam jsou pak zobrazeny v přehledné podobě pomocí grafů a tabulek.

Službu jsem zvolil především pro velké množství měřených metrik a pak také pro její snadné použití v Node.JS aplikacích, jak ukazuje následující měřicí kód:

```
require('nodetime').profile({
  accountKey: ' - klic prideleny sluzbou nodetime - ',
  appName: 'Nazev Aplikace'
});
```

6.2 Ladění

Pro Node.JS aplikace existuje mnoho nástrojů, kterými lze kód debugovat a ladit. Ze seznamu v diskuzní vláknu³⁷ na webu Stack Overflow jsem pro tyto účely vybral službu node-inspector³⁸.

³⁶<https://nodetime.com/apps>

³⁷<http://stackoverflow.com/questions/1911015/how-to-debug-node-js-applications>

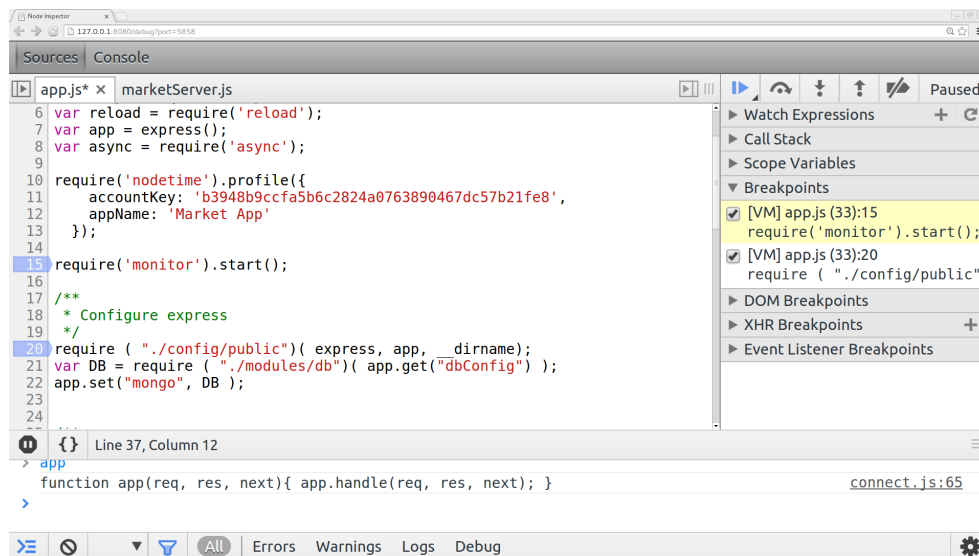
³⁸<https://github.com/node-inspector/node-inspector>

6. MONITORING A LADĚNÍ

Před její použitím je potřeba nejprve spustit samotnou aplikaci s příkazem `node -debug[=port] app.js` nebo `node -debug-brk[=port] app.js`, který na zadaném portu (defaultně 5858) spustí debugovací rozhraní a poté spustí samotnou aplikaci. Druhý příkaz pak navíc ještě pozastaví aplikaci hned při startu a její zpracování lze pak krokovat nebo jinak ovládat v debuggeru.

Modul `node-inspector` se nainstaluje pomocí `npm install node-inspector -g`, kde příkaz `-g` zajistí instalaci modulu jako služby. Tu pak lze nainstalovat příkazem `node-inspector`. Služba je pak přístupná z webového prohlížeče Google Chrome na adrese `http://127.0.0.1:8080/debug?port=5858`, kde parametr `port` říká, kde běží debugovací rozhraní.

Na obrázku C.4 je zobrazeno rozhraní debuggeru během debugování aplikace `marketu`.



Obrázek 6.1: Rozhraní pro debugování kódu v technologii Node.JS.

Testování - dodelam v prubehu nedele

V této kapitole se zaměřím na testování vytvořeného obchodního systému z pohledu rychlosti a výkonu. Pro tyto účely jsem vytvořil tři testovací scénáře, ve kterých budu zkoumat rychlost zpracování obchodních příkazů v závislosti na jejich množství (průběh zpracování příkazu je popsáno v 2.1.3), rychlost komunikace navržené v 3.3.7 a ještě něco **TODO**.

Pro testování využiji službu nodetime popsanou v sekci 6 a automatické obchodní systémy, které byly vytvořeny spolu s obchodním systémem a poslouží nyní pro generování zátěže v systému.

7.1 Testované prostředí

Služby, u kterých bude probíhat měření, poběží na počítači s touto konfigurací:

- CPU:
- RAM:
- Síťová karta:

A

7.1.1 Testovací scénář

7.1.1.1 Scénář 1 - rychlost zpracování obchodních příkazů v závislosti na jejich množství

7.1.1.2 Scénář 2 - roundtrip spojení marketu s brokerem

7.1.1.3 Scénář 3 - ????

7.1.2 Výsledky měření

7.1.3 Závěr měření a návrhy optimalizace

Škálování

Tato kapitola je pojata spíše jako teoretická příručka pro škálování aplikací psaných v technologii Node.JS. Zpracování těchto aplikací probíhá v jednom vlákně a nemusí tak být využito plného potenciálu vícejádrového počítače, na kterém systém běží. V dalším textu se proto budu věnovat způsobům škálování, kterými je delegování náročných úloh na externí thready nebo spuštění celé aplikace pod více vlákna a jejich následná synchronizace. U těchto způsobů škálování popíši jejich konkrétní použití na burzovní aplikaci, která je obsahem této práce.

8.1 Oddělení náročných úloh

Základní metodou optimalizace a škálování systému psaného v technologii Node.JS je oddělení úloh náročných na CPU na externí workery nebo *thready*, jak to ostatně bylo provedeno i v této práci. Pro výpočet zdlouhavých výpočtů, které by jinak blokovaly hlavní vlákno, byl využit externí program psaný v jazyce C++, s kterým systém komunikuje přes TCP/IP protokol. Lze však použít i jinou formu komunikace.

Kromě externího programu, Node.JS umožňuje využít i tzv. *WebWorkers* popsaných v specifikaci [21]. Jedná se o API umožňující vytvořit výpočetní vlákna, kterým lze přidělit práci samostatnou práci. Specifikace je zaměřena primárně na prostředí jazyka JavaScript ve webovém prohlížeči, s příchodem Node.JS však lze využít *webWorkers* i na straně serveru. Konkrétní implementací pro Node.JS je například knihovna *node-webworker-threads*³⁹, která umožňuje vytvořit *pool* pomocných threadů, pod kterými lze spouštět definovanou funkci s libovolným množstvím parametrů. Příklad využití tohoto modulu zobrazuje následující kód:

³⁹<https://github.com/audreyt/node-webworker-threads>

8. ŠKÁLOVÁNÍ

```
var cpuLen = require('os').cpus().length;
var Threads = require('webworker-threads');
var ThreadPools = Threads.createPool(cpuLen);

function threadFunc(a, b){
    return a + b;
}

ThreadPools.all.eval(threadFunc);
ThreadPools.any.eval('threadFunc(1, 2)', function(err, res){
    console.log(res); // vysledek bude cislo 3
});
```

Funkce `Threads.createPool` vytvoří zadaný počet výpočetních vláken, na kterých se pomocí `ThreadPools.all.eval` zaregistruje funkce `threadFunction`. Pomocí `ThreadPools.any.eval` se registrovaná funkce použije s danými parametry na jednom z volných vláken. Pokud není žádné vlákno zrovna volné, knihovna jej dá do fronty a zpracuje až se některé vlákno uvolní.

Použití `webworkerů` je vhodné, pokud jsou předávané parametry co nejmenší. V případě burzovní aplikace by se jednalo o obsáhlé pole s velkým množstvím obchodních příkazů a je tedy jednodušší, když si externí worker získává obchodní příkazy přímo z databáze, čímž market neztrácí zbytečně čas serializací vstupních parametrů a přeposílání dat, které jej nezajímají.

Práce workeru tvoří základní kameny burzovního systému, a proto jsem pro účely této práce využil worker napsaný v jazyce C++ a zkompileovaný přímo do strojového kódu, což přispělo k zvýšení výkonu celého systému.

8.2 Cluster modul

Další možností škálování je tzv. *forknutí* rodičovského procesu pomocí Cluster modulu⁴⁰. Spuštěním takové aplikace se tak vytvoří více procesů, které mají přístup ke stejným zdrojům. Typicky to může být aplikace sdílející síťový port, na kterém odchyťává a zpracovává klientské požadavky.

V burzovním systému by se cluster modul dal použít na všech třech částích systému (klient, broker i market). Na klientovi je jeho použití snadné, protože se nemusí řešit komunikace mezi jednotlivými procesy, jedná se totiž o jednoduchou webovou stránku posílající statický obsah. U brokera a marketu již je ale potřeba určitá úroveň synchronizace. U brokera by se muselo vyřešit zamykání klientského účtu při strhávání financí, aby nešlo překročit možnosti účtu odesláním několika příkazů ve stejnou chvíli. Při návrhu marketu by se zas musela vyřešit otázka, zda rozdělit aplikaci podle zpracovávaných akcií, že by každý proces obsluhoval jen podmnožinu všech akciových titulů a nebo zda rozdělit zpracování podle funkce, že by jeden proces řídil burzovní cyklus, další pak komunikovaly s brokery apod.

⁴⁰<http://nodejs.org/api/cluster.html>

Z důvodu velké náročnosti na návrh a implementaci jsem tuto možnost škálování v tvořeném systému vynechal a jedná se tak o možné pokračování této práce.

8.3 Spuštění více aplikací pod load balancerem

Poslední zde rozebranou technikou škálování je spuštění více aplikací běžících na různých portech (popřípadě různých počítačích a IP adresách) a jejich následné zastřešení pomocí load balanceru. Ten lze realizovat například v technologii Nginx, která pak bude rozhrázovat zátěž na jednotlivé aplikace.

V tomto případě je opět potřeba vyřešit otázku rozdělení aplikace popsané v předchozí sekci a pak také vytvoření společného komunikačního rozhraní, které lze realizovat pomocí key-value databáze Redis a Node.JS modulu `socket.io-redis`⁴¹, přes který pak mohou jednotlivé aplikace komunikovat.

Rozdílem oproti škálovací technice z předchozí sekce je, že se dá tímto způsobem spustit aplikace na více počítačích a jedná se tak o další z možných kroků zvýšení výkonu systému.

Nastavení load balanceru v technologii Nginx, jehož konfigurace je popsána v [22], je ukázáno v následujícím zjednodušeném příkladu:

```
http {
    upstream nodejs_app {
        // umístění služeb, mezi které se bude rozdelovat zatez
        server 127.0.0.1:61337;
        server 127.0.0.1:61338;
        server 127.0.0.1:61339;
        keepalive 64;
    }

    server {
        listen 80;

        server_name nodejs-app.tld;

        location / {
            proxy_redirect    off;
            proxy_set_header  X-Real-IP $remote_addr;
            proxy_set_header  X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header  X-Forwarded-Proto $scheme;
            proxy_set_header  Host $http_host;
            proxy_set_header  Connection "upgrade";
            proxy_http_version 1.1;

            // použití nastaveného load balanceru
            proxy_pass         http://nodejs_app;
        }
    }
}
```

Na příkladu jsou v sekci `upstream` definovány adresy tří aplikačních serverů běžících na adrese 127.0.0.1 a portech 61337, 61338 a 61339, mezi které se budou rozdělovat přicházející požadavky. Při konfiguraci technologie Nginx lze uvést mnoho dalších nastavení, jejichž použití se pozitivně projeví na výkonu aplikace. Detailnější konfigurace pak je například v [23].

⁴¹<https://github.com/Automattic/socket.io-redis>

Závěr

V této práci jsem se věnoval burzovním systémům, z nichž jsem se zaměřil především na akciové burzy. V kapitole 1 jsem popsal důvod, proč jsem si toto téma vybral a nastínil jsem, co je cílem této práce.

V kapitole 2 jsem provedl rešerši burzovních systémů a popsal jejich základní části spolu s hlavními procesy, které takové systémy obsahují. Kapitola je psána jako jednoduchý úvod do problematiky a neměla by tak dělat problém na pochopení i burzy neznalým čtenářům.

Kapitola 3 je již více zaměřena na navrhovaný burzovní systém, jehož vytvoření je cílem této práce. V této kapitole jsem se nejprve věnoval výběru technologií, poté jsem zvolil několik klíčových oblastí návrhu a ty popsal podrobněji.

Následující kapitola 4 obsahuje opět vybrané příklady důležitých částí z realizace burzovního systému tvořeného v této práci.

Při vývoji a pro nasazení systému jsem zvolil některé nástroje, jejichž popis spolu s příklady použití obsahuje kapitola 5 a na otázky monitoringu a ladění jsem sepsal jednoduchý úvod do problematiky v kapitole 6.

V kapitole 7 se věnuji zátěžovému testování klíčových částí burzovního systému, kde jsem vytvořil tři testovací scénáře, na těch jsem systém otestoval a v závěru kapitoly shrnul výsledky měření.

Poslední kapitolou 8 je návrh na vylepšení systému v podobě ověřených postupů pro škálování systémů postavených na zvolených technologiích. Možností je zde více a v práci jsem nestihl realizovat všechny. Tato kapitola by měla dát proto spíše směr, kterým by se vývoj aplikace měl ubírat v budoucnu a čtenář by po jejím přečtení měl získat přehled o dalších možnostech, které použité technologie obsahují.

Práce pro mne byla výzvou, protože obsahovala spoustu nové terminologie a věcí na pochopení jak z burzovního světa, tak i z technologií, které jsem schválně volil tak, aby pro mne byly nové a já se tak naučil něco nového i v tomto směru. Při psaní tohoto závěru proto musím říci, že mne práce posunula v mnoha oblastech. Nejdůležitější však je, že mi dala i rozhled o možnos-

ZÁVĚR

tech, které v sobě burza ukrývá a spolu s tím i cíl v dalších letech mého života. S ukončením této práce tedy nekončím s tématem burzovních trhů a vydávám se za hlubším poznáním v oblasti investování, krátkodobých obchodů a to nejdůležitější na konec, algoritnického obchodování.

Literatura

- [1] Farrow, P.: How long does the average share holding last? Just 22 seconds. Jan 2012. Dostupné z: <http://www.telegraph.co.uk/finance/personalfinance/investing/9021946/How-long-does-the-average-share-holding-last-Just-22-seconds.html>
- [2] Fremunt, O.: *Obchody na kapitálové burze a jejich databázový backend*. Diplomová práce, Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2014.
- [3] London Stock Exchange: *MIT201 - Guide to the Trading System*. Dostupné z: <http://www.londonstockexchange.com/products-and-services/trading-services/guide-to-new-trading-system.pdf>
- [4] New York Stock Exchange: *NYSE Order Types*. Dostupné z: https://www.nyse.com/publicdocs/nyse/markets/nyse/NYSE_Order_Types.pdf
- [5] NYSE: *NYSE Order Type Usage [online]*. [cit. 2014-12-13]. Dostupné z: https://www.nyse.com/publicdocs/nyse/markets/nyse/NYSE_Order_Type_Usage.pdf
- [6] Interactive Brokers: *Interactive Brokers Order Types [online]*. [cit. 2014-12-13]. Dostupné z: <http://www.interactivebrokers.com/download/newMark/PDFs/orderTypes.pdf>
- [7] Euronext: *Continuous Trading Process [online]*. [cit. 2014-12-13]. Dostupné z: <https://www.euronext.com/trading/continuous-trading-process>
- [8] NASDAQ: *NASDAQ Trading Schedule [online]*. [cit. 2014-12-13]. Dostupné z: <http://www.nasdaq.com/about/trading-schedule.aspx>

- [9] NYSE: *NYSE Listing fees [online]*. [cit. 2014-12-43]. Dostupné z: <https://usequities.nyx.com/listings/fees>
- [10] Community, F. T.: What is FIX? 2014. Dostupné z: <http://www.fixtradingcommunity.org/pg/main/what-is-fix>
- [11] NASDAQ: *FIX for Orders Programming Specification [online]*. [cit. 2014-12-14]. Dostupné z: https://www.sec.gov/rules/other/nasdaqllcf1a4_5/e_nasdaqfix.pdf
- [12] education.howthemarketworks.com: How do candlesticks imply volume? December 2014. Dostupné z: <http://education.howthemarketworks.com/stocks/intermediate/chartpatterns/how-do-candlesticks-imply-volume-why-is-that-critically-important/>
- [13] Harrell, J.: Node.js at PayPal. November 2013. Dostupné z: <https://www.paypal-engineering.com/2013/11/22/node-js-at-paypal/>
- [14] Lehmann, M.: Full-Stack JavaScript with Node.js. May 2014. Dostupné z: <http://www.slideshare.net/lehmanic/fullstack-sava-script-with-nodejs>
- [15] *Introduction to Object-Relational Database Development [online]*. [cit. 2014-12-31]. Dostupné z: http://infolab.usc.edu/csci585/Spring2010/den_ar/ordb.pdf
- [16] Hadjigeorgiou, C.: *RDBMS vs NoSQL: Performance and Scaling Comparison*. Diplomová práce, The University of Edinburgh, 2013.
- [17] *Solving Big Data Challenges for Enterprise Application Performance Management [online]*. [cit. 2014-12-29]. Dostupné z: http://vldb.org/pvldb/vol15/p1724_tilmanrabl_vldb2012.pdf
- [18] Delft University of Technology, E. Bozdag, A. Mesbah, A. van Deursen: *A Comparison of Push and Pull Techniques for AJAX [online]*. [cit. 2014-12-17]. Dostupné z: <http://swel1.tudelft.nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2007-016.pdf>
- [19] MongoDB, I.: Map-Reduce - MongoDB. November 2014. Dostupné z: <http://docs.mongodb.org/manual/core/map-reduce/>
- [20] Internet Engineering Task Force (IETF), A. B.: The Web Origin Concept. 2011. Dostupné z: <https://tools.ietf.org/html/rfc6454>
- [21] W3C: Web Workers. May 2012. Dostupné z: <http://www.w3.org/TR/workers/>

- [22] Nginx, I.: Using nginx as HTTP load balancer. May 2014. Dostupné z: http://nginx.org/en/docs/http/load_balancing.html
- [23] Dimson, T.: Hardening node.js for production part 2: using nginx to avoid node.js load. July 2012. Dostupné z: <http://blog.argteam.com/coding/hardening-node-js-for-production-part-2-using-nginx-to-avoid-node-js-load/>

Seznam použitých zkratek

AJAX Asynchronous JavaScript and XML

API Application programming interface

JSON JavaScript Object Notation

REST Representational state transfer

CORS Cross-Origin resource sharing

SPA Single-page application

XML Extensible markup language

BSON Binary JSON

CRUD Create Replace Update Delete operace

ETF Exchange-traded fund

CDCP Centrální depozitář cenných papírů

LSE London Stock Exchange

NYSE The New York Stock Exchange

MOC Market On Close

MOO Market On Open

FIX Financial Information eXchange

NASDAQ National Association of Securities Dealers Automated Quotations

RPC Remote procedure call

DOM Document Object Model

A. SEZNAM POUŽITÝCH ZKRATEK

HTML HyperText Markup Language

NPM Node package manager

CSS Cascading Style Sheets

CPU Central processing unit

HTTP Hypertext Transfer Protocol

I/O Input/output

OHLC Open High Low Close

ACID Atomicity, Consistency, Isolation, Durability

TCP Transmission Control Protocol

IP Internet Protocol

RBDMS Relational database management system

Terminologie a použité pojmy

Akcie Cenný papír představující podíl ve firmě.

Backtesting Testování automatického obchodního systému na historických datech.

Broker Prostředník mezi burzou a klienty, kterým za poplatek dává možnost obchodování na burze.

Callback funkce Funkce zavolaná při vyřízení asynchronní operace.

Cancel order Obchodní příkaz pro zrušení v minulosti zadaného příkazu.

Closing price Uzavírací cena ke konci obchodního dne.

Continuous trading Kontinuální obchodování - proces zpracovávající příchozí příkazy do burzovního systému na kontinuální bázi během celé obchodní doby.

CPU intensive task Úkol ve zpracování programu, který je náročný na výpočet.

Emise akcií Prvotní výpis akcií na burze.

Event driven architektura Architektura orientovaná na události, kde při dokončení úloh dojde k vytvoření události a jejímu zpracování obslužnou callback funkcí.

Event loop Cyklus v technologii Node.JS který prochází frontu eventů a zpracovává callback funkce vyřízených asynchronních operací.

Event Událost definována svým popisem a callback funkcí.

Heartbeat Příkaz sloužící pro "oživení" spojení.

I/O non-blocking Zpracování I/O operací prováděné mimo hlavní thread, takže nedochází k blokaci zpracování v programu.

I/O operace Vstupně/Výstupní operace (například čtení a zápis do souboru, komunikace po síti apod.).

Imbalance Převís nabídky nad poptávkou nebo naopak.

Likvidita Míra obchodovatelnosti na burze .

Limit order Obchodní příkaz s pevně nastavenou cenou.

Load balancer Nástroj pro rozhazování požadavků mezi více uzlů v systému.

Main trading session Hlavní obchodní fáze během obchodního dne.

Market Centrální bod burzovního systému, kde dochází k shromáždění obchodních příkazů od jednotlivých brokerů a jejich následné zobchodování.

Obchodní kniha Seznam rozdělený na nabízející a poptávající příkazy a seřazený podle ceny.

Obchodní platforma Klientské rozhraní sloužící pro zadání obchodních příkazů a jejich správu.

Order matching algoritmus - Algoritmus pro nalezení strike price a spárování obchodních příkazů.

Push notifikace Zpráva poslaná klientovi serverem, aniž by o ni musel klient nejprve požádat.

Stored procedura Uživateli definovaná funkce v databázi.

Strike price Cena, kde se střetává poptávka s nabídkou.

Take profit Typ obchodního příkazu v burzovních systémech.

Tick size Granularita ceny - pevně stanovený minimální přírůstek ceny podkladového instrumentu.

Timeframe Časový úsek.

Trailing stop Typ obchodního příkazu využívaný při obchodování.

Volatilita Velikost rozpětí ceny, v které se akcie pohybuje.

Volume Zobchodovatelný objem - množství akcií, které lze zobchodovat.

Worker V této práci je pojem worker používán pro označení programu navrženého v [2], který slouží pro zpracování obchodních příkazů.

XMLHttpRequest Rozhraní pro tvorba asynchronních HTTP požadavků ve webovém prohlížeči.

Screeny služeb v systému

Moje akcie Nakoupené akcie a nevyřízené příkazy

Nakoupené akcie

#	Jednotka	Aktuální cena	Nákupní cena	Množství	Cena celkem	Výnos	Akce
1	IMCO	383,42 \$	400,00 \$	10	4 000,00 \$	-165,80 \$ (-4,14%)	i pdf +

Příkazy čekající na vyřízení

#	Jednotka	Typ příkazu	Cena	Množství	Cena celkem	Datum podání příkazu	
1	GOOG	buy order	4505	10	4 500,00 \$	4. 1. 2015 18:58:24	pdf +
2	FB	buy order	2005	1	200,00 \$	4. 1. 2015 18:58:32	pdf +

Obrázek C.1: Výpis uživatelových držných akcií a nevyřízených příkazů v obchodní platformě.

Historie příkazů

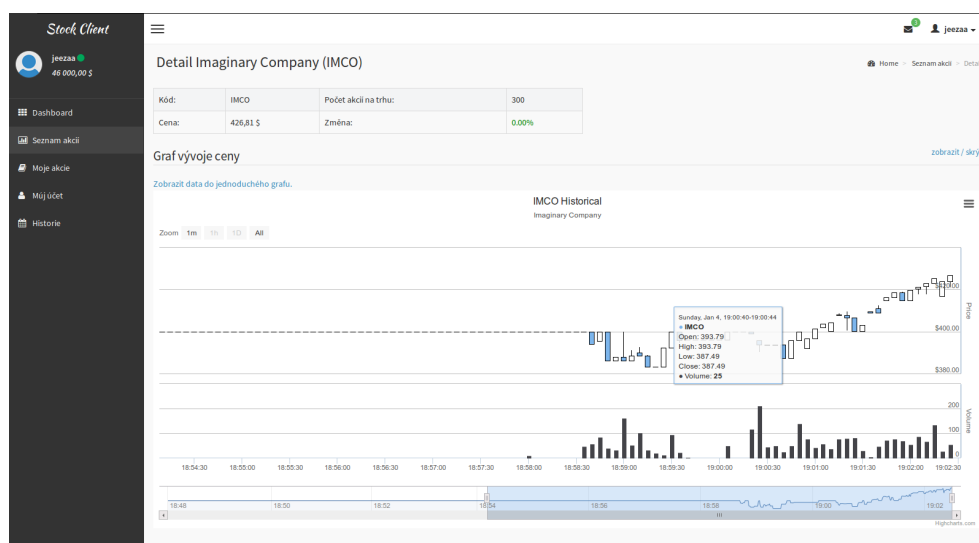
#	Příkaz	Jméno	Množství	Cena za jednotku	Celková cena	Datum
1	BUY ORDER	FB	1	200,00 \$	200,00 \$	4. 1. 2015 18:58:32
2	BUY ORDER	GOOG	10	450,00 \$	4 500,00 \$	4. 1. 2015 18:58:24
3	BUY EXECUTED	IMCO	10	400,00 \$	4 000,00 \$	4. 1. 2015 18:58:04
4	BUY ORDER	IMCO	10	400,00 \$	4 000,00 \$	4. 1. 2015 18:57:41
5	BUY EXPIRED	AAPL	4	123,00 \$	492,00 \$	4. 1. 2015 18:57:06
6	BUY CANCELLED	AAPL	1	123,00 \$	123,00 \$	4. 1. 2015 18:56:05
7	BUY ORDER	AAPL	4	123,00 \$	492,00 \$	4. 1. 2015 18:55:57
8	BUY ORDER	AAPL	1	123,00 \$	123,00 \$	4. 1. 2015 18:55:50

Zobrazuji 1 až 8 z celkem 8 záznamů

[← Předchozí](#) [1](#) [Další →](#)

Obrázek C.2: Historie klientových obchodních příkazů v obchodní platformě.

C. SCREENY SLUŽEB V SYSTÉMU



Obrázek C.3: Detail akcie zobrazený v obchodní platformě

Admin - Robot Broker 01

Úvod	Přehled akcií	Sledované akcie	Vlastněné akcie	Nevyřízené příkazy	Historie	Zastavit robota
------	---------------	-----------------	-----------------	--------------------	----------	-----------------

Základní údaje

Název:	Robot Broker 01	Port:	5200
Prostředí:	robot1	Stav robota:	zapnutý
Doba startu:	2015-01-04 18:53:50.659	Uptime:	11m 27s
Množství vlastněných akcií:	50	Množství nevyřízených příkazů:	69

Burzovní údaje

Market:	127.0.0.1:5555	Spojení:	aktivní
Broker name:	Robot01	Secret key:	robot01_password

Nastavení robota

Delay startu robota:	5000ms	Zpracování v intervalu:	1000ms
Limit na počet poptávaných akcií:	500ks	Maximální odchýlení ceny:	5%
Sledované akcie:	AAPL, GOOG, FB, IMCO		

Obrázek C.4: Dashboard automatického obchodního systému.

Obsah přiloženého DVD

readme.txt	stručný popis obsahu DVD
src	
stock	zdrojové kódy implementace
install.txt	instalační příručka
Broker	zdrojové kódy brokera
Client	zdrojové kódy obchodní platformy
Market	zdrojové kódy marketu
RobotBroker	zdrojové kódy aut. obchodního systému
ChartService	zdrojové kódy služby pro agregaci dat
Strike	dokumentace a zdrojové kódy k práci O. Fremunta [2]
tools	nástroje na ovládání systému
thesis	zdrojová forma práce ve formátu L ^A T _E X
text	text práce
DP_Juna_Jan_2015.pdf	text práce ve formátu PDF
DP_Juna_Jan_2015.ps	text práce ve formátu PS