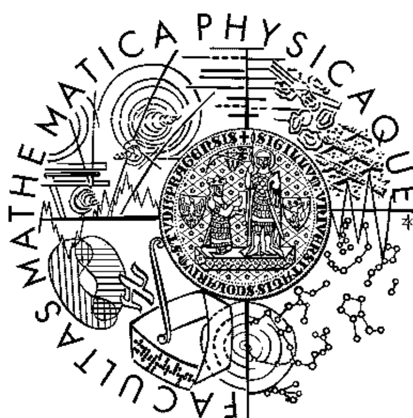
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Jaromír Šatánek

Stroj pro algoritmické obchodování

Katedra softwarového inženýrství

Vedoucí diplomové práce: Mgr. Pavel Ježek

Studijní program: informatika

2009

Poděkování za mnoho podnětných připomínek, rad a ochotu při vedení práce patří Mgr. Pavlu Ježkovi. Mé poděkování rovněž náleží celému oddělení pro elektronické obchodování v zadavatelské společnosti. Zástupci společnosti mi umožnili rychle proniknout do problematiky elektronického obchodování a velkou měrou se podíleli na testování systému.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 14.4.2009 Jaromír Šatánek

Obsah

1 Úvod **10**
1.1 Motivace . . . 10
1.2 Elektronické obchodování . . . 10
1.3 Role brokerských firem . . . 11
1.4 Cíle . . . 13

2 Analýza požadavků **14**
2.1 Cílová platforma . . . 14
2.2 Obecné požadavky na obchodovací platformy . . . 14
2.3 Specifické požadavky a situace v zadavatelské firmě . . . 16

3 FIX protokol **19**
3.1 Příklady komunikace . . . 20
3.2 Market data . . . 20
3.3 FAST protokol . . . 21
3.4 Dostupné implementace protokolu . . . 22
3.4.1 QuickFIX . . . 22
3.4.2 Onixs FIX Engine . . . 22
3.4.3 CameronFIX Universal Server . . . 23
3.4.4 Porovnání funkčnosti knihoven . . . 24
3.4.5 Výkonnost knihoven . . . 24
3.4.6 Shrnutí . . . 28

4 Persistence **29**
4.1 Docílení persistentnosti systému . . . 29
4.2 Úložiště . . . 30
4.3 Serializace . . . 33
4.4 Shrnutí . . . 34

5 Paralelní zpracování 35

5.1 Potřeba paralelního zpracování . . . 35
5.2 Možná řešení . . . 36
5.3 Volba implementace threadpoolu . . . 38

6 Architektura systému a implementace 39

6.1 Možná řešení . . . 39
6.1.1 Prostředník mezi FIX vrstvou a překladovým pluginem . . . 39
6.1.2 Samostatný systém . . . 40
6.2 Základní rozvržení systému . . . 42
6.3 Komponenty systému a jejich funkce . . . 44
6.3.1 ClientInterface . . . 44
6.3.2 MarketInterface . . . 44
6.3.3 MarketDataInterface . . . 46
6.3.4 MarketDataManager . . . 47
6.3.5 RoutingTable . . . 47
6.3.6 MessageManager . . . 48
6.3.7 AlgoEngine . . . 50
6.4 Propojení komponent . . . 51
6.5 Interní reprezentace dat . . . 53
6.5.1 Zprávy . . . 54
6.5.2 Market data . . . 56
6.5.2.1 Raw market data objekty . . . 57
6.5.2.2 Business market data objekty . . . 57

7 Pluginy systému 58

7.1 Komunikace pomocí protokolu FIX . . . 58
7.2 Komunikace mezi více algo systémy . . . 58
7.3 Příjem market dat . . . 59
7.4 Obchodovací algoritmy . . . 59
7.5 Testovací pluginy . . . 60

8 Srovnání s existujícími platformami 61

9 Závěr **64**
9.1 Splnění cílů 64
9.2 Nasazení systému v praxi 65
9.3 Možná vylepšení systému 65
9.3.1 Urychlení procesu obnovy systému 65
9.3.2 Zefektivnění serializace dat 66

A Seznam použitých pojmů **67**

B Seznam použité literatury **69**

C Obsah přiloženého CD **71**

Seznam obrázků

1.1 Vztah brokerské společnosti ke klientům a trhům 11
1.2 Příklad možnosti využití arbitrage 12

2.1 Smart order router 16
2.2 Situace v zadavatelské firmě 17
2.3 Situace s využitím algoritmického stroje 18

3.1 Příklad komunikace zobrazený v monitorovacím systému 20
3.2 Komunikace pomocí protokolu FAST 21
3.3 Business logika systému CameronFIX Universal Server 23
3.4 Test implementací 25
3.5 Průběh testu 26
3.6 Průběh modifikovaného testu 27

4.1 Ukládání obchodovacích zpráv do persistentního úložiště 30
4.2 Průběh testu A 32

5.1 Zpracování obchodovacích zpráv v jednom vlákně 35
5.2 Paralelní zpracování zpráv pomocí front 36
5.3 Příklad synchronizace 38

6.1 Prostředník mezi FIX vrstvou a pluginem 40
6.2 Distribuovaný algo systém 41
6.3 Základní komponenty systému 43
6.4 Architektura systému 45
6.5 Routování zpráv 48
6.6 Reprezentace orderu v úložišti 49
6.7 Propojení komponent zpracovávajících obchodní zprávy 53
6.8 Interní reprezentace zpráv (diagram tříd) 56

9.1 Komponenta RecoveryHelper 66

Seznam tabulek

3.1 Srovnání implementací 24
3.2 Výkonnost knihoven 25

4.1 Průměrný čas potřebný pro uložení jedné položky 31
4.2 Rozptyl a nejhorší čas potřebný pro uložení jedné položky pro test A 33
4.3 Srovnání typů serializací 34
4.4 Srovnání datových úložišť při použití binární serializace 34

8.1 Srovnání obchodovacích platforem 62
8.2 Licence a náklady 63

Název práce: Stroj pro algoritmické obchodování
Autor: Jaromír Šatánek
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: Mgr. Pavel Ježek
e-mail vedoucího: pavel.jezek@mff.cuni.cz

Abstrakt: Na kapitálových trzích se během posledních čtyř let výrazně zvýšilo využití technik algoritmického obchodování. Odhaduje se, že do roku 2010 bude jejich podíl činit více než 50 procent. Cílem práce je navrhnout a vyvinout systém umožňující algoritmické obchodování na burzách cenných papírů. Důraz je kladen na minimální latenci při zpracování obchodních dat, stabilitu systému a jeho modulárnost. Provozovatel musí být schopen snadno modifikovat systém podle aktuálních potřeb (přidat nový algoritmus, spojit systém s novým zdrojem dat z trhu).

Klíčová slova: obchodování, algoritmické, elekronické, akcie

Title: Algorithmic Trading Platform
Author: Jaromír Šatánek
Department: Department of Software Engineering
Supervisor: Mgr. Pavel Ježek
Supervisor’s e-mail address: pavel.jezek@mff.cuni.cz

Abstract: In the last four years it is possible to see a significant increase in the usage of algorithmic trading. It is estimated that by the year 2010 more than 50 percent of trading on the stock markets will be performed by using algorithmic trading systems. The aim of this thesis is to create a system which can enable algorithmic trading on the stock markets. Emphasis is placed on the fast processing of data as well as the stability of the system and its modularity. Users must be able to easily modify and enhance the system according to their own individual needs (e.g. to add new algorithms or connect to another stock exchange).

Keywords: trading, algorithmic, electronic, securities

Kapitola 1

Úvod

1.1 Motivace

Prvotní motivací pro vznik této diplomové práce byla potřeba spolehlivé, flexibilní a škálovatelné obchodní platformy umožňující algoritmické obchodování na straně zadavatelské firmy. Zadavatel se specializuje zejména na oblast rozvojových trhů v regionu střední a východní Evropy. Hlavním rysem těchto trhů je značná heterogennost a často i velmi specifická pravidla obchodování. Existující dostupné systémy se zaměřují na velké vyspělé trhy a jejich adaptace na podmínky zadavatele výrazně převyšuje rámec možností jejich konfigurace. Systém musí být natolik flexibilní, aby umožnil relativně snadnou adaptaci na libovolný trh a směrem k firmě poskytl univerzální standardizované rozhraní pro komunikaci s připojenými trhy.

1.2 Elektronické obchodování

Počátky elektronického obchodování se datují do 70. let 20. století, kdy pokrok v komunikačních technologiích umožnil obchodníkům s cennými papíry zasílat objednávky na burzu vzdáleně. V následujících letech pak již prakticky zmizela potřeba fyzické přítomnosti obchodníka na burze. Výhodami nového přístupu jsou mimo jiné snazší monitoring dění na burze a rychlejší vyřízení objednávek. V průběhu dalších let vznikaly elektronické obchodní platformy na všech významných burzách, což umožnilo nové strategie a přístupy v oblasti obchodování s cennými papíry. S tímto vývojem souvisí potřeba vzniku nových systémů na straně brokerských firem, které by dostatečně využily výhod elektronického obchodování.

Dalším stupněm ve vývoji je využití technik algoritmického obchodování. Na straně obchodníka s cennými papíry je sestavena strategie (algoritmus), která je aplikována ve formě počítačového programu. Algoritmus může být komplexní (založen například na matematickém modelu) a zcela nezávislý na obsluze. Jiným příkladem může být algoritmus, který obchodníkovi tvoří pouze jakéhosi asistenta, který mu pomáhá v realizaci dílčích problémů. Typickým příkladem může být situace, kdy obchodník čeká na konkrétní událost na burze, která vyvolá reakci (například odeslání objednávky či zrušení objednávky). Řešení podobných situací pomocí automatizovaných systémů je jednoduché a ušetří obchodníkovi mnoho práce.

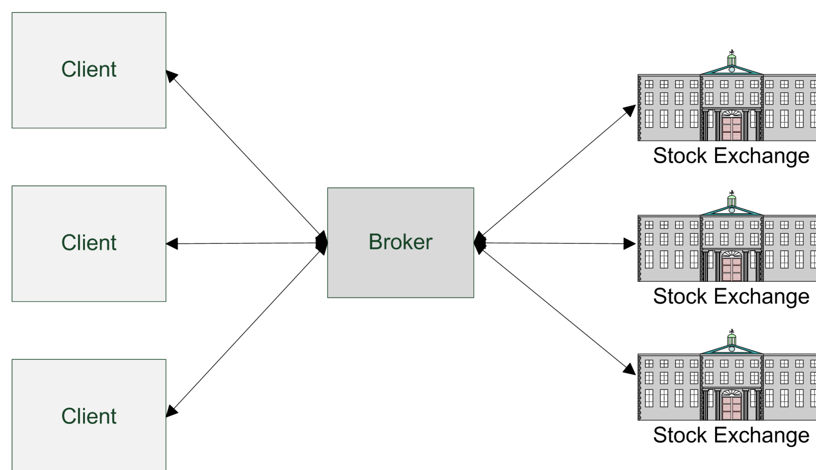
Roli obchodovacích algoritmů lze vidět na analogii s živým makléřem. Úlohou makléře je pro potřeby klienta nakupovat a prodávat cenné papíry (s důrazem na vyřízení daného požadavku za co nejlepších podmínek). Stejnou funkci plní i obchodovací algoritmus. Požadavky od klienta makléř přijímá pomocí telefonu nebo informačního systému, na který je klient napojen. Obchodovací algoritmus je typicky spojen

s klientským systémem, od kterého přijímá požadavky ve formě zpráv. Jak makléř, tak obchodovací algoritmus potřebují znát aktuální stav trhu. Makléř pro tyto potřeby používá specializovaný software (například systém Bloomberg), který zobrazuje všechna potřebná data. Algoritmus je spojen se systémem, který poskytuje aktuální data z burzy (tzv. market data). Kromě komunikace s klientem a znalosti aktuálního stavu trhu je potřeba rozhraní pro komunikaci se samotnou burzou (zasílání objednávek). Makléř má k dispozici specializovaný software (mnohdy spojený se systémem pro sledování stavu trhu), přes který na burzu zasílá objednávky. Obchodovací algoritmus typicky komunikuje s burzovním systémem pomocí zpráv (stejně jako s klientským systémem). Výše popsaný princip je uplatňován u naprosté většiny obchodovacích algoritmů. V závislosti na složitosti zvolené obchodovací strategie může algoritmus používat celou škálu dalších dat (například výstup statistické analýzy obchodovacích dat).

V textu práce se na mnoha místech vyskytují pojmy z oblasti elektronického obchodování (přejaté z angličtiny), jejich seznam lze nalézt v příloze A.

1.3 Role brokerských firem

Zadavatelem práce je brokerská společnost, jejíž úlohou je zprostředkování nákupu a prodeje akcií obchodníkům s cennými papíry. Typickými klienty brokerských společností jsou banky a jiné finanční instituce. Aby mohla firma či instituce obchodovat s cennými papíry, je třeba být členem burzy nebo využívat služeb brokerské společnosti. Členství na burze však mimo jiné vyžaduje mnoho administrativních úkonů a nemalé finanční náklady. Klienti (kdykoli bude dále v textu použito termínu klient, je jím myšlen subjekt, který obchoduje s cennými papíry) využívají služeb brokerů, aby sami nemuseli být členy burzy. Přes jednu brokerskou firmu typicky obchoduje více klientů. Mnoho brokerských společností umožňuje přístup na více než jednu burzu.



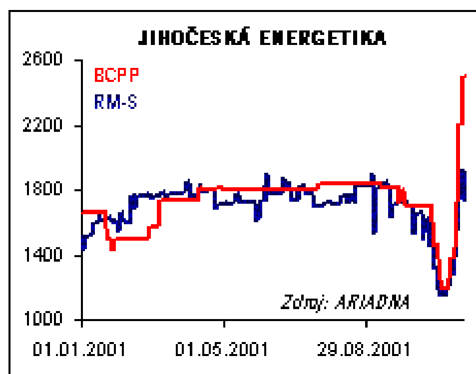
Obrázek 1.1: Vztah brokerské společnosti ke klientům a trhům

Na obrázku 1.1 lze vidět jednoduché schéma znázorňující brokerskou společnost ("Broker" uprostřed), která poskytuje 3 klientům (objekty "Client" vlevo) přístup na 3 burzy cenných papírů (objekty "Stock Exchange" vpravo). Ze schématu vyplývá, že broker je pro klienta burzou a pro burzu klientem. Ve skutečnosti platí, že každá burza může poskytovat svůj vlastní protokol pro komunikaci s klienty. Klient však typicky umí

komunikovat pouze pomocí jednoho protokolu, proto u brokerských společností vznikají systémy, které zastřešují všechny burzy a vytvářejí iluzi jednotného komunikačního rozhraní.

Brokerské firmy mohou klientům nabízet (mimo jiné) služby EDA a DMA. Služba EDA (z anglického Execution Desk Access) znamená, že klientská objednávka je zpracovávána makléřem. Makléř se snaží danou objednávku na trhu vyřídit za co možná nejvýhodnější cenu pro klienta. Při této činnosti může využívat služeb algoritmického stroje, kdy předává k zpracování části objednávky a nastavuje parametry obchodovacího algoritmu. Druhou službou je DMA (Direct Market Access - přímý přístup na trh), kdy jsou objednávky bez jakékoli asistence makléře zasílány na trh. Výhodou DMA jsou nižší náklady pro klienta (klient využívá infrastruktury brokerské firmy pro zasílání objednávek, nikoli však dalších služeb) a možnost objednávku plně kontrolovat. Broker může svoji službu DMA doplnit podporou pro algoritmické obchodování. Příkladem využití algoritmického obchodování může být záruka vyřízení objednávky za cenu VWAP (z anglického "Volume Weighted Average Price" - průměrná cena vážená množstvím zobchodovaného cenného papíru). Pro vyřízení této objednávky je využito speciálního obchodovacího algoritmu, který je navržen tak, aby objednávku dokázal vyřídit právě za tuto cenu. Strategie VWAP může být využívána v případech, kdy je potřeba koupit / prodat velké množství cenného papíru, aniž by došlo k zásadnímu ovlivnění ceny na trhu. Logickým důsledkem umístění kupního orderu s velkou kvantitou na trh je vzrůst ceny akcie, proto je potřeba order rozdělit do menších orderů a vyřídit jej postupně. Známkou kvality vyřízení je průměrná cena v porovnání s cenou VWAP.

Zisky brokerské firmy nemusí být závislé pouze na zprostředkování obchodů třetí straně (typicky však tvoří naprostou většinu profitu). Brokerská firma může využít členství na burzách cenných papírů a sama praktikovat algoritmické obchodování s cennými papíry. Příkladem může být algoritmus Arbitrage, který využívá situací, kdy je stejný cenný papír obchodován na více burzách a při dostatečném rozdílu v cenách na jedné z burz nakupuje, zatímco na druhé stejné množství prodává. Podmínkou pro nasazení tohoto algoritmu jsou dostatečně rychlá Market Data (příjem dat popisujících stav trhu) ze všech burz, na kterých se cenný papír obchoduje.



Obrázek 1.2: Příklad možnosti využití arbitrage

Na obrázku 1.2 lze vidět pohyb ceny akcie společnosti Jihočeská energetika na trzích: Burza cenných papírů Praha a RM Systém. Graf zobrazuje rozdílnost cen vždy na konci obchodovacího dne. Na obrázku lze vidět situace, kdy bylo výhodné použít algoritmus Arbitrage. Jedná se zejména o levou část grafu, kde je možné si všimnout výrazně vyšší ceny, za kterou se obchodovalo v RM systému (algoritmus by kupoval na Burze cenných papírů Praha a prodával v RM systému).

1.4 Cíle

Cílem práce je analyzovat prostředí a požadavky brokerské firmy (zadavatel práce), pomocí těchto nabytých poznatků navrhnout a vytvořit obchodovací platformu, která umožňuje algoritmické obchodování na burzách cenných papírů. Při návrhu a vývoji systému je třeba brát v potaz dílčí cíle (vlastnosti systému):

- **Flexibilita** - Systém musí být snadno modifikovatelný a tudíž aplikovatelný na mnoho problémů (zadání) v oblasti elektronického a zejména algoritmického obchodování. Jedná se především o:
  - Snadné zařazení nových obchodovacích strategií (algoritmů) do systému.
  - Možnost využívat stejný algoritmus pro více burz (i když komunikují pomocí různého protokolu).
  - Možnost spojit systém s novou burzou či zdrojem market dat.
- **Škálovatelnost** - Systém musí umožňovat rozložení zátěže mezi více výpočetních a úložních prostředků. Propojením více instancí systému tak bude možné vytvořit jednotný distribuovaný systém, který je schopen zvládnout úkoly přesahující možnosti každé jednotlivé instance.
- **Nízká latence** - Systém musí zpracovávat data dostatečně rychle, což znamená:
  - Nezpomalit klientské požadavky při průchodu na burzu. Čas zpracování požadavku systémem musí být zanedbatelný ve srovnání s celkovým časem od odeslání požadavku klientem až po jeho zpracování burzou.
  - Dostatečně rychle reagovat na nové události z trhu (příchozí market data obsahující novou informaci).

  Čas potřebný pro zpracování jedné klientské zprávy či zprávy obsahující market data nesmí přesáhnout jednotky milisekund.
- **Stabilita** - Systém musí vykazovat vysokou míru stability a být odolný vůči nevalidním vstupům do systému.

Kapitola 2

Analýza požadavků

2.1 Cílová platforma

Výběr cílové platformy byl zařazen do kapitoly věnované požadavkům na systém, neboť jedním z požadavků zadavatelské firmy byla i technologie použitá při implementaci systému. V úvahu by připadalo více technologií / programovacích jazyků. V užším výběru skončily objektově orientované jazyky C++, Java a C#. Z těchto tří jazyků byl vybrán jazyk C# a platforma .NET, zejména právě z důvodu požadavku ze strany zadavatelské firmy. Pro jazyk C# hovoří rychlost a pohodlnost vývoje zejména ve srovnání s jazykem C++. Zadavatelská firma disponuje vývojáři se znalostmi tohoto jazyka, kteří budou moci v budoucnu převzít údržbu systému a přidávat nové obchodovací algoritmy či konektory. Fakt, že může být platforma .NET využita pro implementaci obchodovacích systémů, dokládá mimo jiné systém Infolect ([2]). Tento systém využívá jako svoji obchodovací platformu burza London Stock Exchange, která je jednou z největších burz na světě.

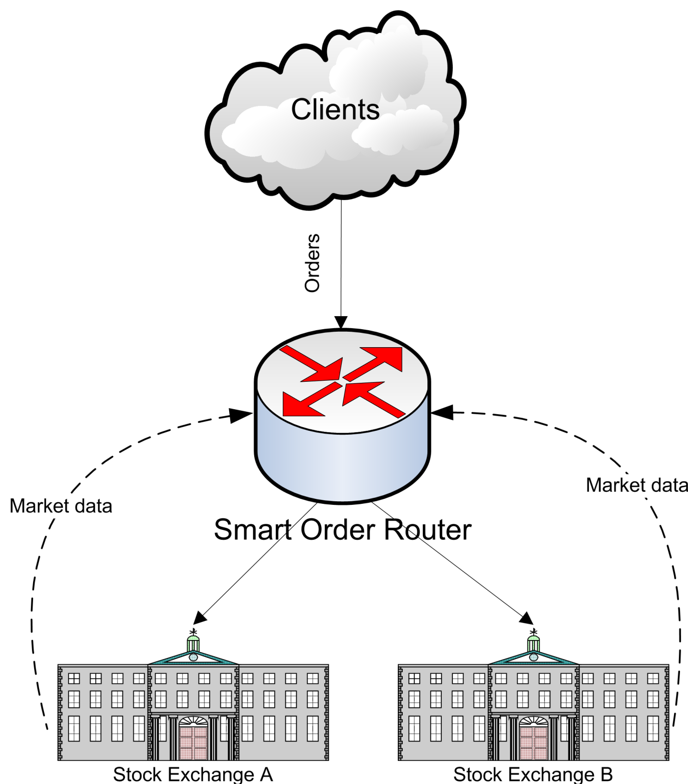
Volba cílové platformy výrazně ovlivňuje další analýzu systému, avšak mnoho z postupů, které byly při vývoji systému uplatněny (docílení persistentnosti systému, komunikace komponent systému, paralelní zpracování požadavků, ...), mohou být stejně vhodně použity i při implementaci systému v jiných objektově orientovaných jazycích.

2.2 Obecné požadavky na obchodovací platformy

Základním požadavkem na každou obchodovací platformu je vysoká spolehlivost. Systém musí být co nejméně náchylný ke vzniku chyb. Každá chyba může mít narozdíl od běžné aplikace (například klientského gui) fatální následky. Jako příklad lze uvést nezaslání informace o provedeném obchodu způsobené výjimkou v kódu zpracovávajícím tuto informaci. V případě, že se klient rozhodne mezitím order zrušit, nemá povinnost obchod, o kterém mu včas nedošla informace, akceptovat. Při větším množství kusů zobchodovaného cenného papíru může vzniklá ztráta nabýt vysoké hodnoty (obchodované objemy se pohybují v řádech milionů Eur). Proto je potřeba minimalizovat riziko vzniku podobného problému. Většina z burz poskytuje svým členům přístup na testovací prostředí, kde je možno obchodovací systém ladit. Důkladné otestování systému výrazně sníží pravděpodobnost vzniku chyby v produkčním prostředí. Systém lze testovat manuálně. Tester však musí být dobře obeznámen se specifiky dané burzy a na základě těchto znalostí vhodně volit

testovací scénáře. Alternativou manuálního testování či vhodným doplněním je automatický testovací systém, který obsahuje testovací scénáře (například ve formě skriptů). Testovací scénáře jsou aplikovány na testovaný systém, ve scénářích jsou specifikovány očekávané reakce systému a na základě toho je vyhodnoceno, zda daný test dopadl podle očekávání. Implementace dostatečně obecného testovacího systému je však mnohdy náročnější než implementace samotné obchodovací platformy. Je potřeba si uvědomit, že testovací systém musí znát aktuální stav trhu, pokud má umožnit kvalitní otestování reakcí. Proto se v praxi často využívá jednorázových testovacích systémů, které slouží k otestování pouze jednoho konkrétního systému.

Dalším požadavkem na obchodovací systémy je persistentnost. Systém musí být připraven na možnost přerušení činnosti (například z důvodu poruchy hardware) v průběhu obchodovacího dne. Systém se musí po opětovném spuštění obnovit do původního stavu tak, aby znal kontext všech orderů, se kterými pracoval před přerušením. S persistentností systému úzce souvisí další důležitý požadavek: rychlost (snaha o minimalizaci času potřebného pro zpracování klientského požadavku a reakcí zaslaných burzou). Ukládání dat do persistentního úložiště (soubor či databáze) je časově mnohem náročnější než operace nad strukturami, které jsou uloženy v paměti. Proto je důležité najít takové řešení persistence, které (výrazně) nezpomalí běh systému. Rychlost systému je do značné míry měřítkem jeho konkurenceschopnosti. V oblasti elektronického obchodování hraje roli každá milisekunda (pro algoritmické obchodování to platí dvojnásob). Důležité jsou rychlé odezvy na události, které se na trhu dějí. Jako příklad lze uvést systém, který využívá faktu, že s některými cennými papíry lze obchodovat na více burzách a klientské ordery zasílá na burzu s nejlepší cenou (tzv. smart order router, obrázek 2.1). Systém umístí order na burzu A ("Stock Exchange A") s aktuálně nejlepší cenou a čeká na případné informace (market data) z burzy B ("Stock Exchange B"). V případě, že se naskytne na burze B výhodnější nabídka, je třeba původní order zrušit a zaslat na burzu s lepší nabídkou. Důležité je vyčkat na potvrzení o zrušení orderu z burzy A, aby nedošlo k zobchodování orderu na obou burzách. Operace zrušení orderu na burze A a zaslání orderu na burzu B musí být co nejrychlejší, aby nabídka na burze B byla ještě aktuální.



Obrázek 2.1: Smart order router

2.3 Specifické požadavky a situace v zadavatelské firmě

Zadavatelská firma poskytuje své služby mnoha klientům. Klientské systémy zasílají své ordery pomocí protokolu FIX (standardizovaný protokol navržený pro výměnu finančních informací, protokolu je věnována kapitola 3). Vstupem algoritmického stroje však nemusí být pouze požadavky od klientů, ale též požadavky zaslané z interních systémů. Příkladem může být systém OMS (Order Management System), který slouží makléřům pro správu klientských orderů. Systém OMS by mohl využívat funkcionality algoritmického stroje (algo systém může být rozšířen o sadu pomocných obchodovacích algoritmů, které asistují makléři; tyto algoritmy mohou řešit například problémy typu "v případě, že se na trhu objeví nabídka prodeje akcie společnosti X za cenu nižší než Y, pošli order, který tuto nabídku zobchoduje"). Většina systémů, které by mohly využívat služeb stroje pro algoritmické obchodování, podporuje protokol FIX. Existují však i systémy, které tento protokol nepodporují. Proto je důležité, aby systém nebyl vázán na vstup v konkrétním formátu. Nezávislost na formátu vstupních dat usnadňuje také propojení více algo systémů v jeden distribuovaný systém (systémy si tak mezi sebou mohou vyměňovat prakticky libovolná data).

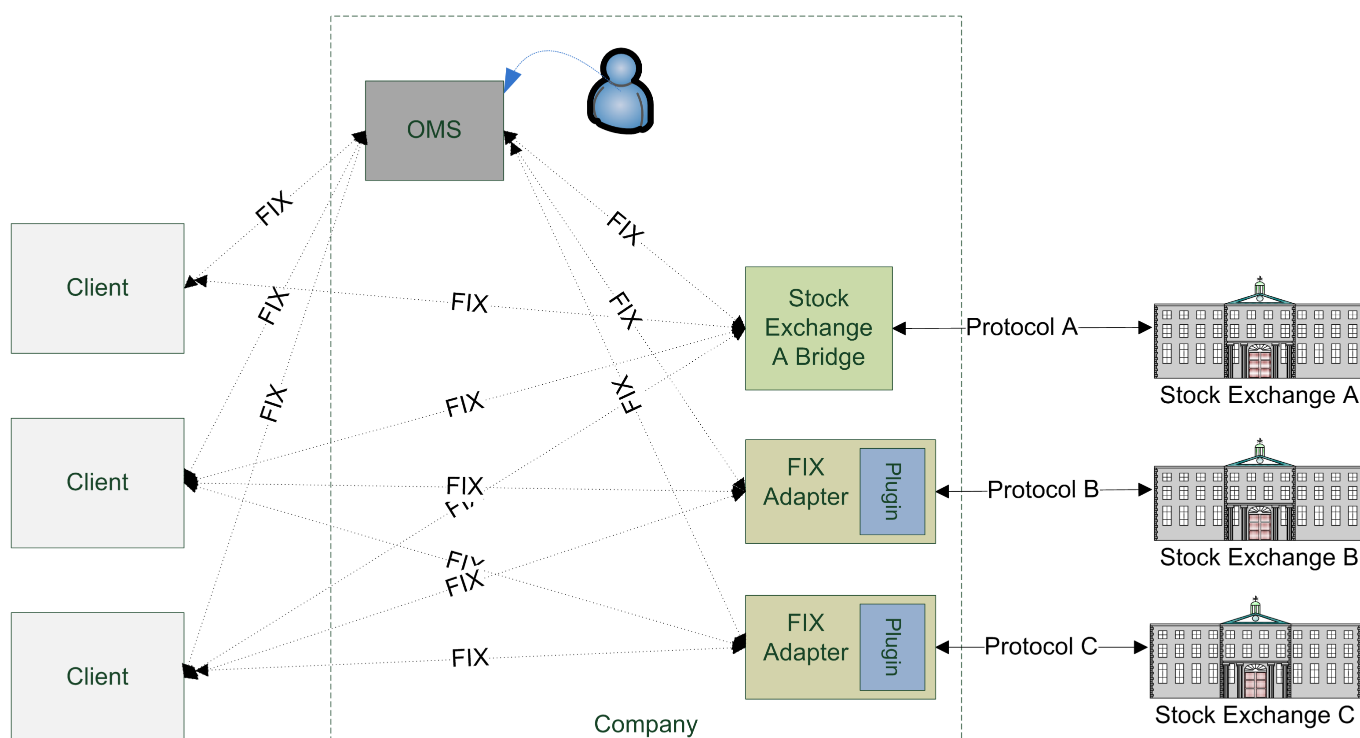
Komunikace s burzami cenných papírů je ještě komplikovanější než komunikace s klientskými systémy. Zadavatelská firma se orientuje na méně rozvinuté trhy, které ve většině případů nepodporují protokol FIX. Každá z těchto burz nabízí pro zasílání orderů vlastní protokol (XML zprávy přes TCP, komunikace pomocí webové služby atd.). Rozhraní pro komunikaci s trhy tedy musí být také obecné a nezávislé na konkrétním protokolu.

Kromě rozhraní pro zasílání orderů poskytují jednotlivé trhy také možnost odběru market dat. Market data jsou však také typicky zasílána pomocí protokolů, které jsou specifické pro každou z burz. Systém tedy

musí být dostatečně modulární, aby dokázal komunikovat s klientskými systémy i trhy (zasílání orderů a příjem market dat) různými druhy protokolů.

Pro analýzu požadavků na algoritmický systém je důležitá současná infrastruktura systémů pro elektronické obchodování v zadavatelské firmě. Zkoumána byla možnost využití stávajících systémů ve spolupráci s algoritmickým strojem. Zadavatelská společnost disponuje systémy, které klientům umožňují zasílat na burzu ordery pomocí protokolu FIX. Klientům je tak vytvořena iluze jednotného komunikačního rozhraní pro všechny burzy, které zadavatelská firma pokrývá. Pro překlad FIX protokolu do formátu, kterému rozumí burza, jsou používány:

- **Bridge systémy** - Systémy určené pro překlad z FIX protokolu do specifického protokolu burzy. Každý z bridge systémů v zadavatelské firmě lze využít právě pro jednu burzu.
- **FIX Adaptéry** - Jedná se o překladové systémy, které narozdíl od systémů typu bridge nabízejí možnost aplikace systému pro více výstupních protokolů (pomocí pluginů).



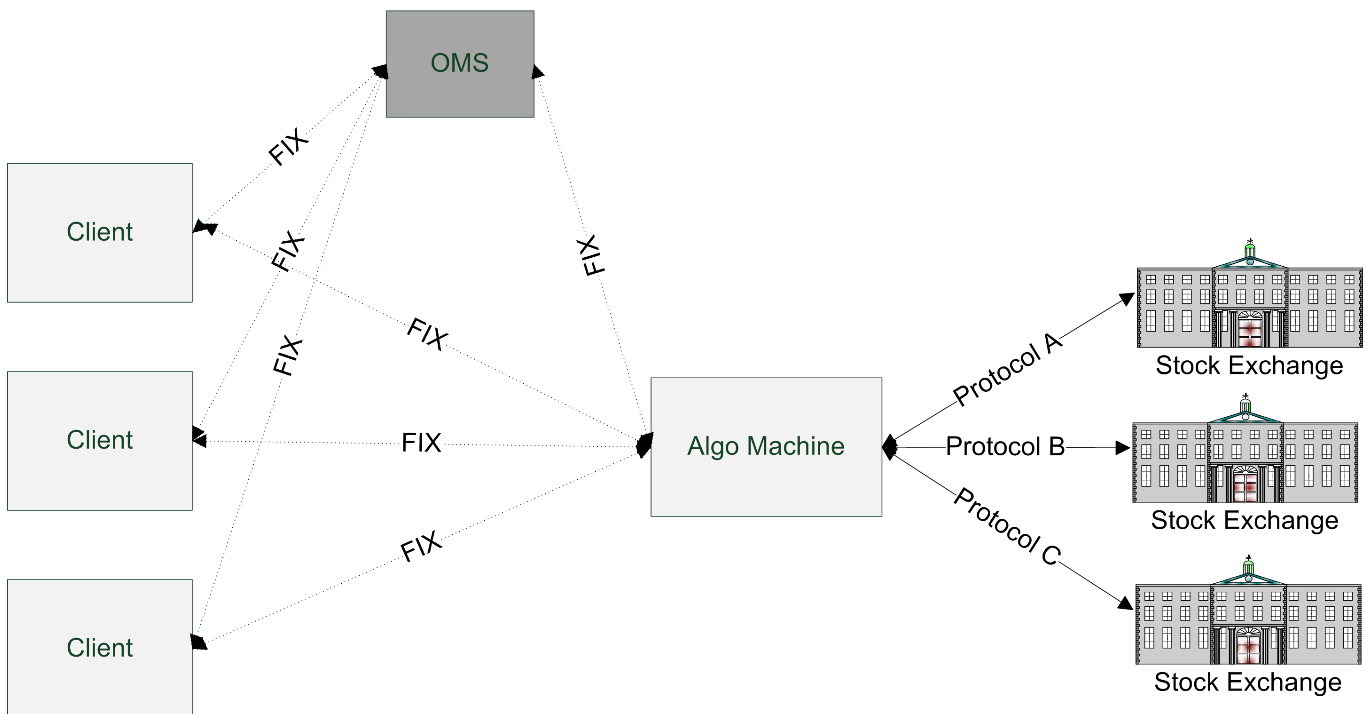
Obrázek 2.2: Situace v zadavatelské firmě

Na obrázku 2.2 lze vidět nástin infrastruktury pro elektronické obchodování v zadavatelské společnosti. Obrázek znázorňuje propojení klientských systémů (objekty "Client" vlevo) se systémy uvnitř firmy (uprostřed obrázku). Dále lze vidět využití systémů typu bridge a FIX Adaptérů pro překlad FIX protokolu. Ani jeden z výše uvedených systémů však není vhodný pro algoritmické obchodování. Hlavními důvody jsou:

- Optimalizace pro překlad zpráv v poměru 1:1. Vstupní zpráva je přeložena do formátu, kterému rozumí burza a zpráva je na burzu odeslána. Algoritmický stroj však potřebuje zasílat zprávy na burzu i bez impulsu od klienta, některé zprávy od klienta naopak mohou "vygenerovat" více zpráv odeslaných na trh.
- Možnost komunikace pouze s jednou burzou. Každý z adaptérů je specializován pro komunikaci s konkrétní burzou. Mnoho algoritmů však využívá faktu, že se vybrané cenné papíry obchodují na více burzách.
- V systémech chybí napojení na zdroj market dat. Obchodovací algoritmy by tak neměly informace o aktuálním stavu trhu.

FIX adaptéry, ačkoli nejsou vhodné pro algoritmické obchodování, jsou lepším řešením než bridge. Nevyžadují totiž opakované implementování vstupní FIX vrstvy a narozdíl od bridge systémů mají potenciál být využity i ve spolupráci s algo systémem (zejména jejich překladové pluginy).

Požadavkem na algo systém nebylo jen umožnit algoritmické obchodování na burzách cenných papírů, ale také nahradit starší překladové systémy a vytvořit jednotnou obchodovací platformu v rámci firmy. Tím se usnadní správa systémů v rámci firmy (systémoví administrátoři budou spravovat pouze jeden typ systému).



Obrázek 2.3: Situace s využitím algoritmického stroje

Obrázek 2.3 znázorňuje situaci v zadavatelské firmě s využitím algo systému. Algo machine uprostřed obrázku reprezentuje množinu instancí algo systému. Každá z těchto instancí má specifické úkoly a dohromady tvoří distribuovaný systém - obchodovací platformu umožňující algoritmické obchodování.

Kapitola 3

FIX protokol

Jak již bylo zmíněno v kapitole 2.3, protokol FIX je využíván klienty zadavatelské firmy pro zasílání orderů na burzu. Z tohoto důvodu bylo potřeba důkladně analyzovat vlastnosti protokolu. Pokud nepočítáme interní systémy zadavatelské firmy, které mohou být s algo systémem propojeny, bude protokol FIX představovat jediný vstup do systému. V některých speciálních případech může protokol FIX být i výstupním formátem systému (pokud burza komunikuje tímto protokolem).

FIX protokol (zkratka pro "Financial Information eXchange", více v [12]) je v současnosti pravděpodobně nejhojněji používaný protokol pro komunikaci mezi brokerskými firmami a jejich klienty. FIX protokol existuje ve více verzích a neustále se vyvíjí. Na vývoji se podílejí velké bankovní a brokerské společnosti. První verze protokolu vznikla v roce 1992, nejnovější verze (5.0) je z roku 2008, dodnes jsou však na mnoha místech využívany starší verze (4.1, 4.2, 4.3, 4.4).

Komunikace přes FIX protokol je typicky realizována nad spolehlivým TCP/IP, avšak protokol je možné použít i nad nespolehlivým komunikačním protokolem (například UDP). Protokol je založen na výměně zpráv mezi komunikujícími stranami. Zprávy lze rozdělit do dvou tříd: tzv. session zprávy a application zprávy. Session zprávy slouží k návázání spojení mezi komunikujícími stranami a k jeho údržbě. Zaručují spolehlivost protokolu: každá zpráva obsahuje sekvenční číslo a pokud jedna z komunikujících stran neobdrží zprávu (obdrží zprávu se sekvenčním číslem N a dále zprávu se sekvenčním číslem N + 2 nebo vyšším), je využito session zprávy `Resend Request`, pomocí níž lze vzdálenému systému specifikovat, které zprávy je nutno přeposlat. Application zprávy slouží k přenosu finančních dat, tedy k zasílání orderů na trh klientem a zasílání informací o orderu trhem. Každá zpráva (platí pro session zprávy i pro application zprávy) je tvořena posloupností `X=Y`, kde X je identifikátorem pole a Y je jeho hodnota. Identifikátorem pole je vždy kladné celé číslo, hodnotou pak libovolný řetězec ASCII znaků, který neobsahuje speciální znak sloužící pro oddělení polí. Pole lze rozdělit do tří základních skupin: hlavička, tělo a tzv. trailer (obsahuje pouze informace o délce zprávy). Hlavička obsahuje specifikaci odesílatele zprávy, adresáta a další údaje, jako je čas odeslání zprávy. V těle zprávy jsou obsažena pole s finančními daty. Zpráva může kromě předem definovaných polí obsahovat i uživatelská pole (pole, jejichž identifikátor dosud nebyl přidělen žádnému z definovaných polí). Komunikující strany si tak mohou zprávy rozšířit o nové informace, což způsobuje větší flexibilitu protokolu. Základní komunikace mezi klientem a burzou probíhá pomocí zpráv: `New-Order Single` (vytvoření orderu klientem), `Order Cancel Request` (zrušení orderu), `Order Cancel/Replace Request` (změna parametrů orderu), `Execution Report` (informace o orderu z burzy, např. potvrzení o přijetí orderu burzou) a `Order Cancel Reject` (zamítnutí zrušení či modifikace orderu burzou). Specifikace protokolu ([16]) obsahuje detailní popis těchto zpráv a ukazuje mechanismus výměny finančních dat mezi klientem a burzou.

3.1 Příklady komunikace

Na obrázku 3.1 lze vidět výsek komunikace mezi klientem a burzou. Klient zasílá zprávu `Order-New Single`, čímž žádá trh o přijetí nového orderu (zpráva obsahuje veškeré parametry orderu: cena, kvantita, typ orderu, atd.). Trh reaguje potvrzením přijetí orderu (pomocí zprávy `Execution Report-New`). V tuto chvíli je již order umístěn na trh a může být zobchodován jiným klientem. Následně klient zasílá požadavek na modifikaci orderu (zasláním zprávy `Order Cancel/Replace Request`, zpráva obsahuje nové parametry orderu). Trh reaguje nejprve zprávou `Execution Report-PendingCancel`, čímž informuje klienta, že je jeho požadavek vyřizován. Trh následně požadavek na modifikaci přijímá, zasílá tedy klientovi zprávu `Execution Report-Replaced`. V tuto chvíli jsou v platnosti nové parametry orderu. Poté dochází k částečnému zobchodování orderu, kdy burza zasílá zprávu `Execution Report-PartiallyFilled`, která obsahuje informace o provedené transakci. Nakonec burza zasílá zprávu `Execution Report-Canceled` a tím daný order ruší (neruší však transakci provedenou pomocí zprávy `Execution Report-PartiallyFilled`).

2008-09-15 11:10:54,664 OrderSingle (15110954648_0_1) BUY 200@51 SIMULATION ROSNPPACNOR9XX_ GOOD_TILL_CANCEL
2008-09-15 11:10:54,664 ExecutionReport NEW (15110954648_0_1) 0@0 (0@0) 200 remaining
OrderCancelReplaceRequest (15110954648_0_1 to 15110954648_0_2) BUY 250@51 SIMULATION ROSNPPACNOR9XX_
2008-09-15 11:10:55,164 ExecutionReport PENDING_CANCELREPLACE (15110954648_0_2) 0@0 (0@0) 200 remaining
2008-09-15 11:10:55,164 ExecutionReport REPLACED (15110954648_0_2) 0@0 (0@0) 250 remaining
2008-09-15 11:10:55,164 ExecutionReport PARTIALLY_FILLED (15110954648_0_2) 249@51 (249@51) 1 remaining
2008-09-15 11:10:55,164 ExecutionReport CANCELED (15110954648_0_2) 0@0 (249@51) 0 remaining

Obrázek 3.1: Příklad komunikace zobrazený v monitorovacím systému

Specifikace protokolu ([16]) obsahuje množství scénářů, na kterých lze pochopit fungování protokolu. Tyto scénáře lze také využít pro otestování libovolného obchodovacího systému, který komunikuje protokolem FIX.

3.2 Market data

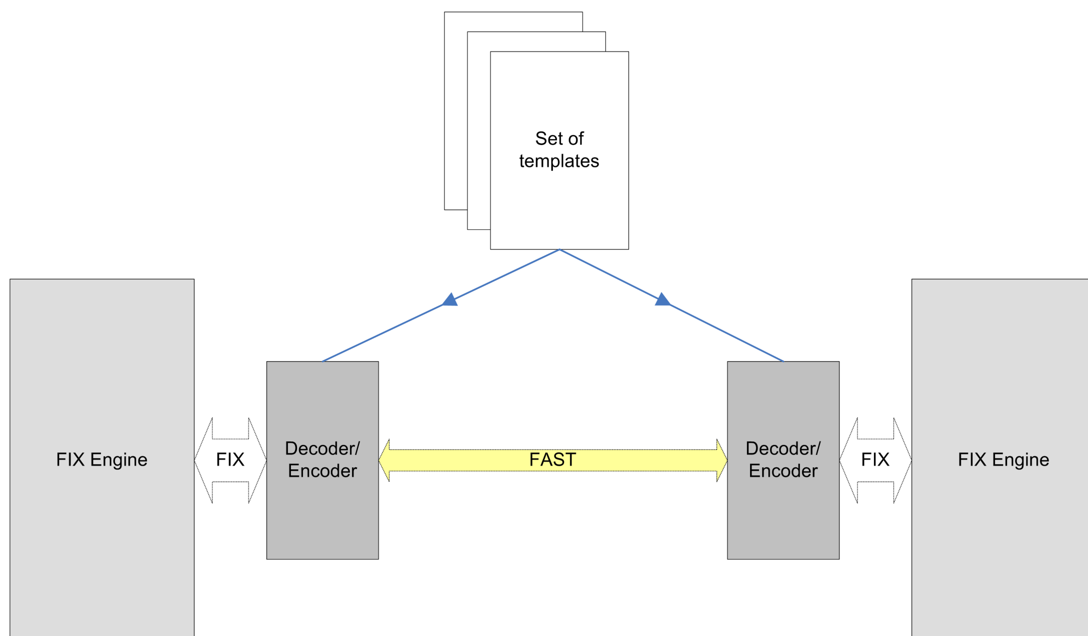
Protokol od verze 4.2 podporuje přenos market dat. Přenos je založen na následujícím principu: Klient pomocí zprávy `Market Data Request` zašle požadavek na trh pro odběr specifických market dat. Trh reaguje zasláním aktuálního stavu na burze a podle typu požadavku případně zasílá aktualizace daného stavu (klient může zažádat o aktuální stav na trhu či o aktuální stav a jeho aktualizace v čase). Data jsou klientovi zasílána pomocí některé z market data zpráv (`Snapshot / Full Refresh` či `Incremental Refresh`). Zpráva od trhu obsahuje jednoznačný identifikátor market dat (pole `MDEntryID`). Pomocí tohoto pole lze market data modifikovat. Trh zašle zprávu se stejným identifikátorem a příznakem, že je třeba data upravit.

Příkladem odebírání market dat z trhu může být situace, kdy klienta zajímá Order Book (seznam orderů, který se na burze aktuálně nachází) konkrétního cenného papíru. Klient zasílá zprávu `Market Data Request`. Jako odpověď obdrží od trhu zprávu `Snapshot / Full Refresh`, která obsahuje aktuální order book pro daný papír. Každému orderu je přiřazen unikátní identifikátor (pole `MDEntryID`). Toto pole se ve zprávě vyskytuje vícekrát (podle počtu orderů). Skupina polí týkající se jednoho orderu je vždy

seskupena vedle sebe a začíná právě polem `MDEntryID`. Dále jsou klientovi zasílány zprávy `Incremental Refresh`, které order book aktualizují. `Incremental Refresh` může také obsahovat více záznamů, pro každý záznam je specifikována operace `Insert`, `Update` či `Delete`. Tímto způsobem je zajištěno, že klientský systém (např. algo) má vždy aktuální informaci o stavu trhu.

3.3 FAST protokol

Protokol FAST (zkratka pro FIX Adapted for Streaming - FIX adaptér pro tok dat) vznikl původně jako optimalizace dat (zpráv) přenášených pomocí protokolu FIX. V průběhu času se stal obecnějším protokolem využitelným pro optimalizace více protokolů. Cílem při návrhu bylo snížit objem přenesených dat. Klíčovou vlastností protokolu je snaha neodesílat data, která již byla odeslána. Při komunikaci pomocí protokolu FAST jsou využívány šablony pro zakódování zprávy na straně odesílatele a dekódování zprávy na straně příjemce (šablonu si mohou komunikující strany "domluvit" při startu komunikace dynamicky, lze ji také nastavit ještě před startem komunikace staticky). Schéma komunikace lze vidět na obrázku 3.2, kdy na obou komunikujících stranách je komponentě, která komunikuje protokolem FIX, předřazen speciální adaptér ("Encoder / Decoder") pro konverzi do / z protokolu FAST.



Obrázek 3.2: Komunikace pomocí protokolu FAST

Pro kompresi jsou mimo jiné využívány následující předpoklady:

- Jednotné pořadí polí ve zprávě - není nutné do zprávy vkládat identifikátory polí.
- Opakující se hodnoty - není nutné do zprávy vkládat pole, které se nezměnilo oproti předchozí zprávě (například odesílatel zprávy).
- Seznam polí přítomných ve zprávě (reprezentován bitovou maskou) - není nutné do zprávy vkládat identifikátory polí.

- Binární serializace - číselné hodnoty ve zprávě jsou zakódovány do binární reprezentace (rozlišují se celočíselné a neceločíselné typy).

Principy využívané v protokolu FAST mohou nalézt využití v mnoha oblastech elektronického obchodování, nejvíce jsou však uplatněny při přenosu market dat protokolem FIX.

3.4 Dostupné implementace protokolu

Na oficiálních stránkách protokolu ([12]) lze nalézt podrobné instrukce potřebné pro implementaci protokolu. Implementace protokolu se všemi jeho možnostmi je však poměrně náročná (zejména otestování korektního chování za všech okolností). Proto je dobré využít řešení, která již byla vyvinuta a otestována v praxi. Z existujících implementací lze jmenovat knihovny QuickFIX ([18]), Onixs FIX Engine ([4]) a systém CameronFIX Universal Server ([19]). Tyto knihovny (systém) byly vybrány jako zástupci různých typů implementace protokolu FIX.

3.4.1 QuickFIX

QuickFIX je open source knihovna (analyzována byla verze 1.12.4), jejíž rozhraní je dostupné pro jazyky C++, Java, jazyky rodiny .NET, Python a Ruby. Knihovna je využívána například obchodovací platformou Marketcetera ([3]). Knihovna obsahuje kompletní implementaci FIX protokolu ve verzích 4.0 až 4.4, chybí podpora pro FAST.

Použití knihovny při implementaci systému, který komunikuje pomocí protokolu FIX, je poměrně přímočaré. Třídě, která zastřešuje spojení se vzdáleným systémem (případně více systémy), je při inicializaci předána cesta ke konfiguračnímu souboru, který obsahuje specifikaci veškerých spojení. Třídy knihovny jsou persistentní, po restartu systému se automaticky obnoví do původního stavu. Persistentnost lze realizovat pomocí souborů či databáze (podporovány jsou MSSQL, MySQL a PostgreSQL), kam jsou zapisovány údaje potřebné pro obnovení. Persistentnost je důležitá zejména pro zachování spolehlivosti protokolu i po restartu systému: systém A odeslal systému B zprávu se sekvenčním číslem N a poté došlo k jeho restartu. Systém B, který měl tuto zprávu přijmout, ji neobdržel a zasílá tedy systému A požadavek na přeposlání zprávy se sekvenčním číslem N, systém A díky persistentnosti může zprávu N odeslat i po restartu.

Knihovna obsahuje speciální třídu pro každý druh zprávy FIX protokolu, což usnadňuje práci s knihovnou. Pole zprávy jsou spjata s konkrétním datovým typem, není tedy potřeba dodatečné přetypování při práci s obsahem zprávy. Pro odeslání zprávy vzdálenému systému stačí korektně vyplnit hlavičku zprávy (vyplnit příjemce a odesilatele zprávy) a zavolat statickou metodu pro odeslání na třídě zastřešující všechna spojení.

3.4.2 Onixs FIX Engine

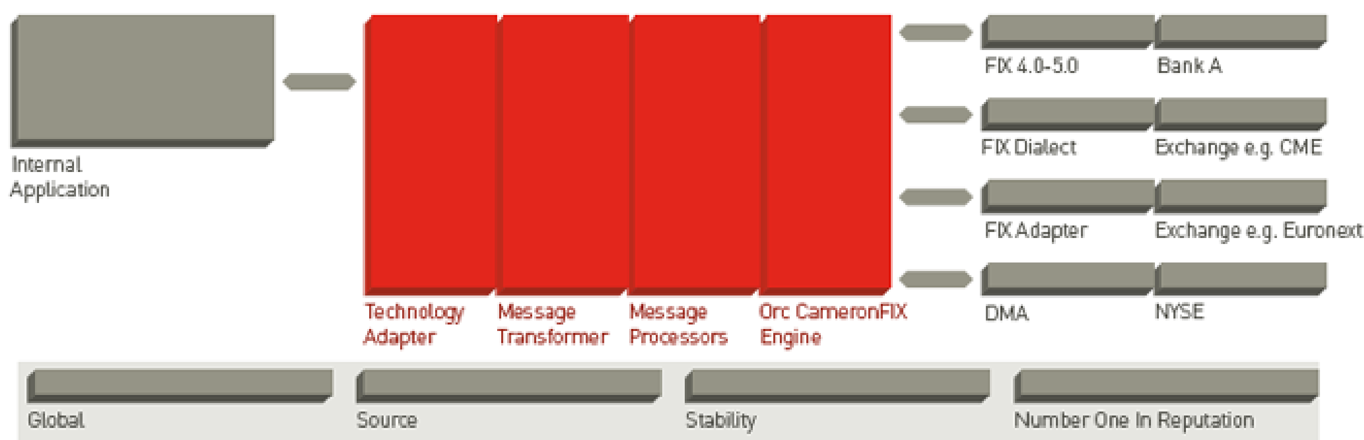
FIX Engine (analyzována byla verze 2.45.1.0) od společnosti Onixs je komerční implementace protokolu. Knihovna je dostupná pro jazyky Java, C++ a jazyky rodiny .NET. Knihovna obsahuje podporu pro protokol ve verzích 4.0 až 5.0 včetně optimalizace FAST. Knihovna je optimalizovaná pro rychlost přenosu obchodních zpráv.

Použití knihovny je podobné jako u knihovny QuickFIX, chybí však implementace specializovaných tříd pro různé typy zpráv. Pro všechny typy zpráv se využívá třídy `Message`. Tento fakt výrazně znepříjemňuje

práci s knihovnou. Vhodné řešení by bylo chybějící třídy doimplementovat, přičemž `Message` by tvořila bázovou třídu. Samozřejmostí je persistentnost knihovny. Persistentnost je založena na ukládání dat do souboru, narozdíl od knihovny QuickFIX chybí podpora pro ukládání do databáze.

3.4.3 CameronFIX Universal Server

CameronFIX Universal Server (analyzována byla verze 6.2r33) je komplexní komerční software vyvíjený společností Orc. Narozdíl od knihoven QuickFIX a Onixs FIX Engine se jedná o hotový systém, který lze upravit pro konkrétní potřeby pomocí zásuvných modulů v jazyce Java. Komunikace s ostatními systémy je založena na protokolu FIX. Protokol je podporován ve verzích 4.0 až 5.0 (včetně optimalizace FAST).



Obrázek 3.3: Business logika systému CameronFIX Universal Server

Na obrázku 3.3[1] lze vidět systém napojený na interní aplikaci ("Internal Application" vlevo) a více externích vzdálených systémů vpravo (mezi nimi i dvě burzy cenných papírů - "Exchange e.g. Euronext" a "Exchange e.g. CME"). Framework systému CameronFIX nabízí velké množství hotových komponent, které lze při implementaci obchodovacího systému využít. Za zmínku stojí například datová struktura `MarketMirror`, která udržuje aktuální stav všech orderů na základě zpráv, které systémem protékají. Další užitečnou strukturu, kterou CameronFIX nabízí, je `PersistedArray`. Tato struktura je persistentním úložištěm FIX zpráv. Spojením struktury `MarketMirror` a struktury `PersistedArray` lze dostat persistentní úložiště orderů (persistentnost je založena na ukládání dat do souboru). Podobné úložiště je zapotřebí k implementaci každého netriviálního obchodovacího systému, který umožňuje algoritmické obchodování (aktuální stav orderu je základní informací pro určení další strategie).

Společnost Orc dále k produktu nabízí adaptéry (překlad interního protokolu burzy na FIX) pro více než 100 burz. Využití každého z adaptérů je však samostatně zpoplatněno. Systému chybí možnost implementace vlastních modulů pro komunikaci s burzami, které nekomunikují pomocí protokolu FIX. Tento fakt je výraznou nevýhodou pro společnosti, které jsou členy menších burz.

[1] Zdroj: `http://www.orcsoftware.com`

3.4.4 Porovnání funkčnosti knihoven

Knihovna QuickFIX je vhodná v situacích, kdy je potřeba vyvinout obchodovací platformu s relativně nízkými náklady. Od knihovny nelze očekávat žádnou přidanou funkcionalitu, pouze implementaci protokolu FIX. Použití knihovny je přímočaré a bezproblémové (o čemž svědčí využití knihovny v mnoha systémech). Výhodou knihovny je také podpora pro více programovacích jazyků a možnost zásahu do zdrojových kódů.

Komerčně vyvíjená knihovna Onixs FIX Engine je optimalizovaná pro vyšší výkonnost při komunikaci pomocí protokolu FIX, což je její největší výhodou oproti knihovně QuickFIX. Nevýhodou knihovny je neexistence specializovaných tříd pro všechny typy zpráv a dostupnost implementace v méně jazycích.

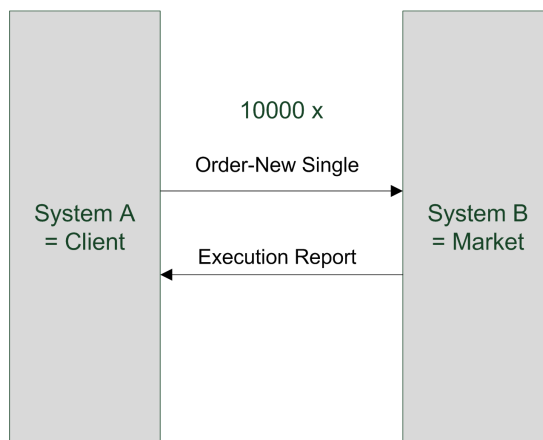
CameronFIX Universal Server nabízí oproti knihovnám QuickFIX a Onixs FIX Engine mnoho přidané funkcionality potřebné pro implementaci obchodovacích platforem. CameronFIX je však nákladné řešení. Další nevýhodou je závislost na jazyce Java a nemožnost dodatečných úprav kódu. Konfigurace systému je díky jeho komplexnosti o třídu náročnější než je tomu u knihoven QuickFIX a Onixs FIX Engine.

Implementace	Podporované OS	Podporované technologie	Typ	Licence
QuickFIX	Windows, Linux, Solaris, FreeBSD a Mac OS X	C++, Java, .NET, Python a Ruby	Jednoduchá knihovna implementující FIX protokol	Open Source
Onixs FIX Engine	Windows, Linux, Solaris, platformy podporující technologii Java	C++, Java, .NET	Jednoduchá knihovna implementující FIX protokol	Komerční
CameronFIX	Platformy podporující technologii Java	Java	Komplexní framework pro vytváření obchodovacích platforem	Komerční

Tabulka 3.1: Srovnání implementací

3.4.5 Výkonnost knihoven

Důležitým aspektem pro analýzu implementací protokolu FIX je také výkonnost konkrétního řešení. Z tohoto důvodu byl proveden se třemi výše uvedenými implementacemi test porovnávající jejich výkonnost. U knihoven QuickFIX a Onixs FIX Engine byly testovány verze pro .NET (dáno výběrem cílové platformy). Jelikož nebyl autory protokolu zveřejněn standardizovaný test, který by různé implementace protokolu porovnával, byl jako vhodný vybrán test, který se svými knihovnami dodává společnost Onixs. Tento test prověřuje, jak rychle dostává klientský systém odpovědi od trhu v závislosti na zvoleném middleware (implementaci protokolu).



Obrázek 3.4: Test implementací

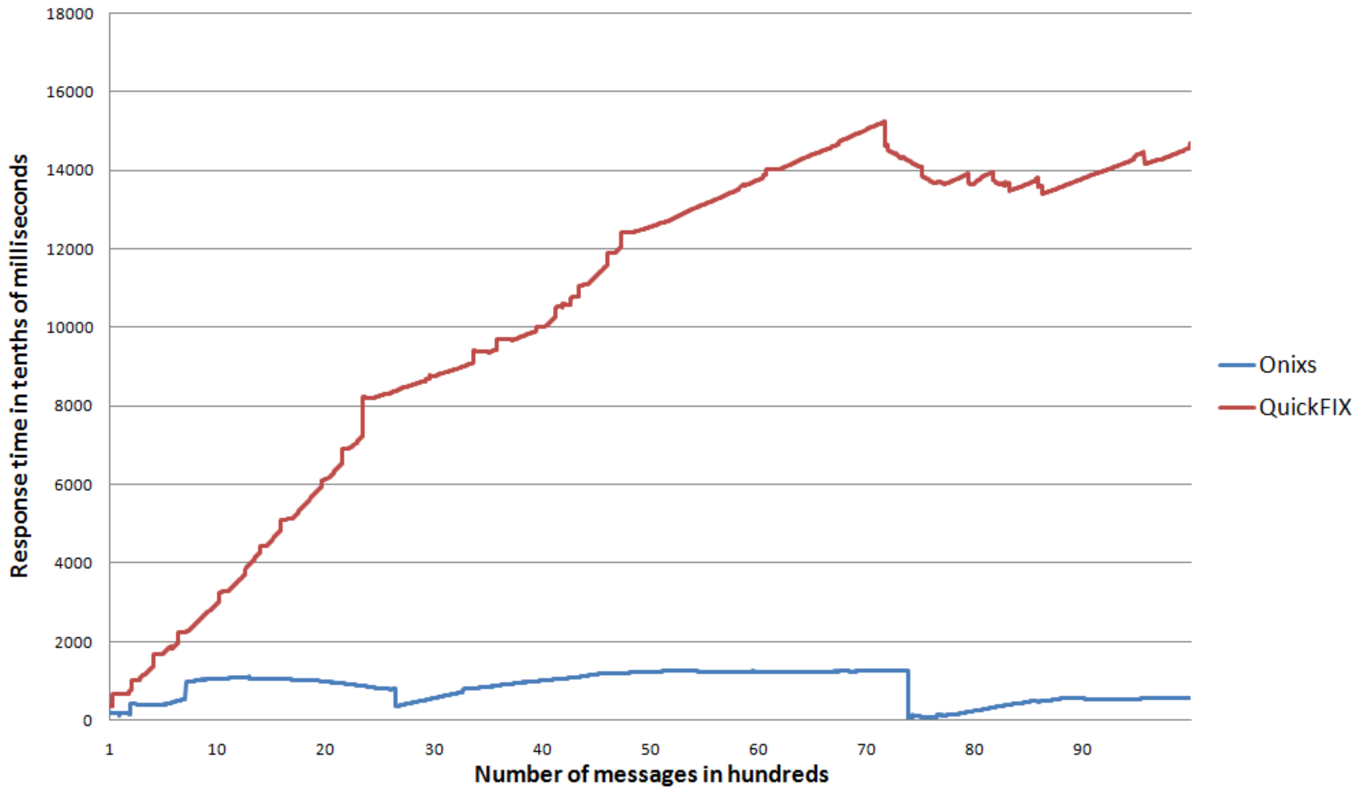
Test probíhá podle následujícího scénáře (obrázek 3.4): Vytvořeny jsou dva systémy (označme A a B, tyto systémy pro jednoduchost běží na stejném stroji) komunikující protokolem FIX. Systém A reprezentující klienta odesílá systému B 10 000 zpráv (počet odeslaných zpráv nehraje významnou roli, důležitý je průměrný čas potřebný pro zpracování jedné zprávy, tzn. za jak dlouho se dostane odpověď klientskému systému). Systém B odpovídá na každou zprávu právě jednou. Jedná se o simulaci vytvoření orderu systémem A a jeho potvrzení systémem B, tedy typický příklad komunikace mezi klientským a trhovým systémem. Měření celkového času testu je ukončeno v okamžiku, kdy systém A obdrží poslední z reakcí, které zasílá systém B. Z výsledků testu v tabulce 3.2 lze vyčíst výrazně lepší výsledky u knihovny Onixs FIX Engine.

Implementace	**Průměrný čas potřebný pro zpracování jedné zprávy**
QuickFIX	0.179ms
Onix FIX Engine	0.04ms
CameronFIX	0.36ms

Tabulka 3.2: Výkonnost knihoven

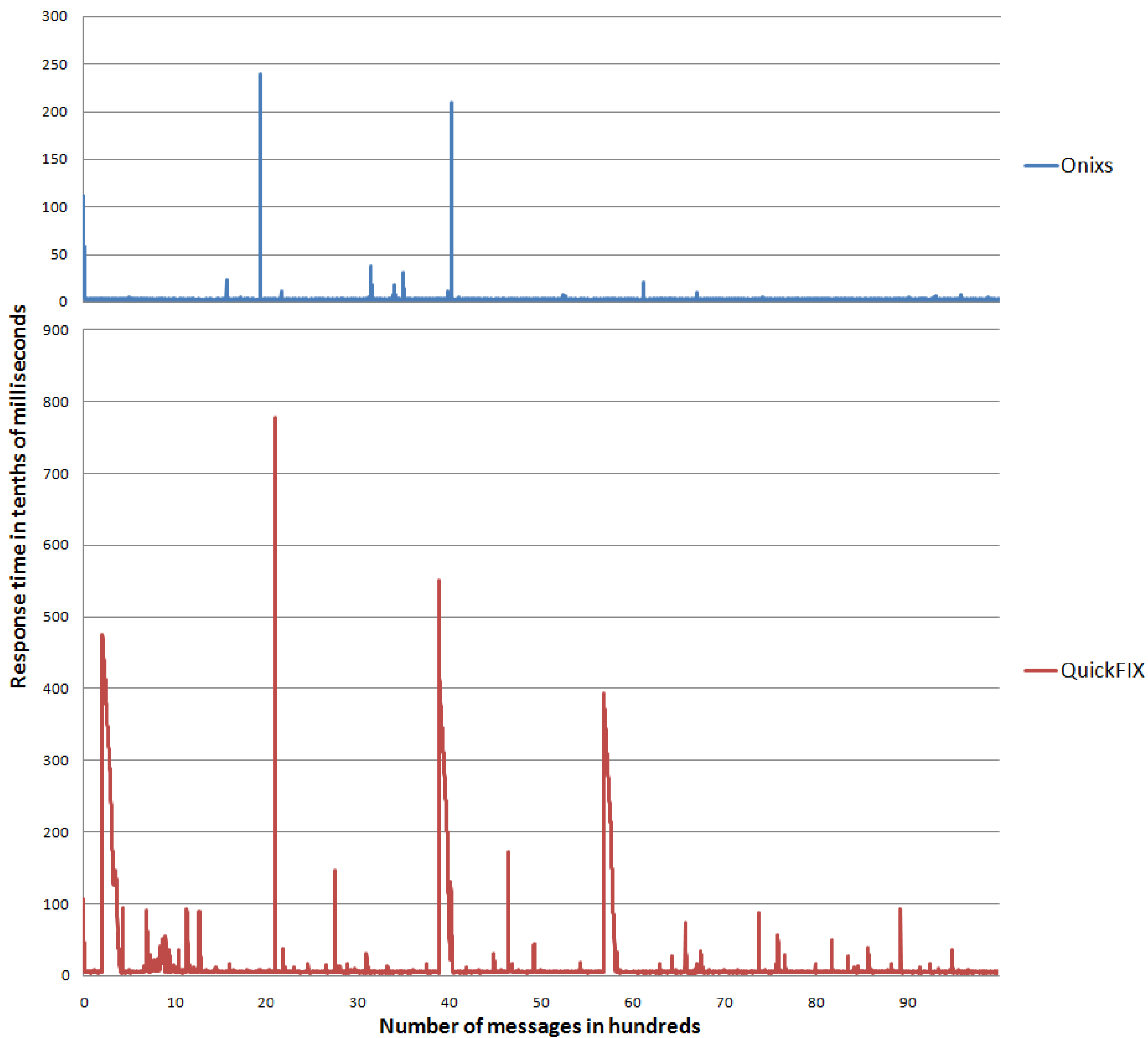
Poznámka: Test systému CameronFIX byl proveden pouze pro srovnání, neboť tento systém nelze využít při implementaci obchodovacího systému založeného na platformě .NET (.NET je cílovou platformou algoritmického systému vyvíjeného v rámci této práce). Systém CameronFIX typicky běží na operačních systémech jako je Solaris na výkonných serverových strojích (test implementací protokolu FIX byl proveden na běžném PC s operačním systémem Windows). Tento fakt mohl ovlivnit výsledky testu v neprospěch tohoto systému.

Při testu knihoven QuickFIX a Onixs FIX Engine byl monitorován také čas, který uplyne mezi odesláním zprávy ze systému A a přijetím odpovědi od systému B v průběhu testu (obrázek 3.5, osa Y znázorňuje čas odezvy v desetinách milisekund, osa X znázorňuje pozici zprávy v rámci testu). Výsledky jsou ovlivněny tím, že při testu jsou odeslány všechny zprávy ze systému A najednou (tzn. mezi odesláním jednotlivých zpráv nejsou žádné časové intervaly). Knihovna Onixs se s tímto faktem vyrovnala o poznání lépe a nejhorší čas odpovědi nepřekročil 200 milisekund. U knihovny QuickFIX čas odpovědi rostl téměř lineárně a v nejhorším případě překročil hranici 1500 milisekund.



Obrázek 3.5: Průběh testu

Provedený test byl zaměřen na maximální zátěž knihoven tím, že odeslal všechny klientské zprávy za sebou bez jakýchkoli prodlev. V praxi se počítá s rovnoměrnějším rozložením zátěže. Při současném objemu zpráv, které jsou v zadavatelské firmě zpracovány, a s ohledem na současné trendy (počet zpráv se zvyšuje) bylo odhadnuto, že i při nejvyšší zátěži systému by neměl počet zpracovaných zpráv překročit stovky zpráv za vteřinu (současné nároky jsou řádově nižší). Proto byl proveden s knihovnami QuickFIX a Onixs FIX Engine modifikovaný test, kdy systém A odešle systému B vždy 2 zprávy a poté se na milisekundu uspí. Zbytek testu probíhá podle nezměněného scénáře. Průběh modifikovaného testu lze vidět na obrázku 3.6. Na obrázku lze vidět, že i knihovna Onixs vždy nezaručuje dobu odezvy blízko naměřeného průměru (nejhorší čas odezvy systému B se pohybuje kolem 20 milisekund). U knihovny QuickFIX lze pozorovat častější odchýlení od průměrného času (nejhorší čas odezvy byl naměřen 80 milisekund). I v tomto modifikovaném testu tedy dosahovala knihovna Onixs výrazně lepších výsledků.



Obrázek 3.6: Průběh modifikovaného testu

Konfigurace stroje, na kterém byly testy prováděny:

- *Procesor: Intel(R) Core(TM)2 Duo CPU E4400 @ 2.00GHz 2.00GHz*
- *Paměť: 4GB*
- *Disk: WDC WD800JD-75MSA3 ATA*
- *Operační systém: Windows Vista Enterprise, 64-bit*

3.4.6 Shrnutí

Pro implementaci algo systému byla zvolena knihovna QuickFIX zejména z důvodu nulových nákladů v porovnání s dobrou použitelností knihovny a v současné chvíli i dostatečnou výkonností. Testy provedené v kapitole 3.4.5 však odhalily nedostatky ve výkonnosti této knihovny. Při návrhu systému proto bylo potřeba brát v potaz možnost pozdějšího nahrazení knihovny QuickFIX jiným řešením (například knihovnou Onixs FIX Engine z důvodu vyšší výkonnosti), tzn. jádro systému nesmí být s touto knihovnou příliš svázáno.

Kapitola 4

Persistence

Jak již bylo popsáno v kapitole 2.2, persistentnost je jedním ze základních prvků každého obchodovacího systému využitelného v praxi. Při návrhu a implementaci algo systému bylo zapotřebí vybrat konkrétní persistentní úložiště a navrhnout co nejoptimálnější způsob, jak persistentnosti docílit.

4.1 Docílení persistentnosti systému

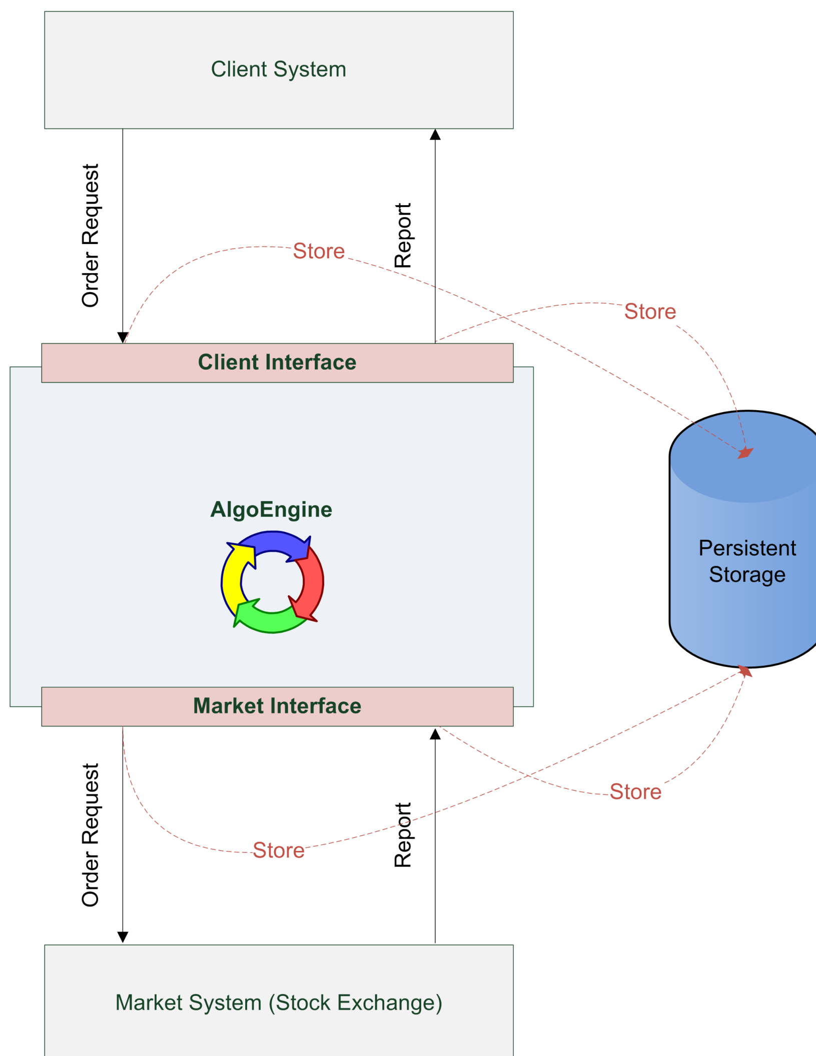
Pro docílení persistentnosti systému existuje více řešení. Pro dosažení optimálního výkonu systému bylo potřeba vybrat co nejvhodnější. Jednou z hlavních podmínek při návrhu systému bylo odstínit od mechanismu persistování samotné algoritmy (tím je implementace obchodovacího algoritmu značně usnadněna). Stav systému je dán stavem orderů, které jsou systémem zpracovávány. Stavem orderu je myšlena kompletní informace, která se k danému orderu vztahuje (základní parametry: cena, kvantita, typ orderu, identifikátor cenného papíru, ...; statistiky: zobchodovaná kvantita, průměrná cena, seznam exekucí - provedených obchodů, ...). Tato data jsou uložena v globálním úložišti, které je dostupné pro všechny obchodovací algoritmy, které jsou v systému zapouzdřeny stejně jako pro jádro systému. Pro docílení persistentnosti systému tedy stačí zaručit persistentnost úložiště orderů.

Pro docílení persistentnosti připadají v úvahu následující řešení:

1. Do persistentního úložiště jsou ukládány veškeré změny všech orderů v systému. Toto řešení vyžaduje neustálé modifikace orderů v úložišti. Po restartu systému jsou načteny ordery z persistentního úložiště do paměťových struktur.

2. Ukládány jsou pouze obchodovací zprávy, které jsou systémem zpracovávány. Stav orderů je z těchto zpráv rekonstruován. Stav úložiště orderů je možné obnovit ze zpráv díky následujícím dvěma faktům:

   - Obchodovací zprávy nesou veškeré informace potřebné pro sestavení aktuálního stavu orderu (order je vždy vytvořen / modifikován na základě příchozí zprávy / zprávy vygenerované algoritmem).
   - Zprávy musí být ukládány do persistentního úložiště ihned při vstupu / opuštění systému (obrázek 4.1). Po restartu systému je díky tomu možno zpracovat zprávy, které do systému vstoupily při minulém běhu, ale ještě nebyly systémem zpracovány (tato vlastnost vylučuje nekorektní chování systému - ztrácení nezpracovaných zpráv).



Obrázek 4.1: Ukládání obchodovacích zpráv do persistentního úložiště

Oba uvedené návrhy mají své výhody a nevýhody, které je potřeba analyzovat v kontextu potřeb algo systému. Zásadním faktem, který je potřeba si uvědomit, je, že i nejrychlejší persistentní úložiště je řádově pomalejší než systémová paměť. Proto je potřeba minimalizovat počet přístupů do persistentního úložiště. Hlavní výhoda druhého řešení spočívá v tom, že systém do persistentního úložiště ukládá pouze obchodovací zprávy a nemusí již ukládat další data. Nevýhodou je pomalejší proces obnovy při startu systému, kdy stav orderů je konstruován, nikoli pouze načten. Důraz je ale kladen především na rychlost systému při standardním zpracování zpráv, restart systému v průběhu obchodovacího dne by měl být naprosto výjimečnou záležitostí. Proto je persistentnost systému založena na druhém řešení.

4.2 Úložiště

Volba typu úložiště je důležitým aspektem pro minimalizaci času potřebného pro uložení jedné položky (tím je snížen čas potřebný pro zpracování jedné zprávy systémem). Při návrhu systému bylo potřeba prozkoumat možnosti dostupných typů úložišť a na základě potřeb systému zvolit nejvhodnější z nich.

Pro vybrání vhodného úložiště byla provedena sada testů s cílem vybrat úložiště, které se pro algoritmický stroj hodí nejvíce. Pro volbu testů je důležitý následující fakt: Uložení zprávy musí probíhat synchronně. Tím je zabráněno situacím, kdy algoritmus vykoná operaci na základě příchozí zprávy, která dosud nebyla uložena do persistentního úložiště a běh systému je ukončen (např. restart systému). V takovém případě po opětovném nastartování systému neexistuje žádný záznam o klientské zprávě, na základě níž mohl algoritmus umístit na trh order. Z tohoto důvodu je důležitý čas potřebný pro uložení právě jedné datové položky. Velikost položky při testování byla zvolena tak, aby odpovídala serializované podobě běžné obchodovací zprávy (cca 400 bytů). Prováděny byly následující testy:

- **A** - 10 000 uložených položek, úložiště na stejném stroji, bez použití transakcí.
- **B** - 10 000 uložených položek, úložiště na jiném stroji, bez použití transakcí.
- **C** - 10 000 uložených položek, úložiště na stejném stroji, ukládání po 500 položkách v transakci.
- **D** - 10 000 uložených položek úložiště na jiném stroji, ukládání po 500 položkách v transakci.

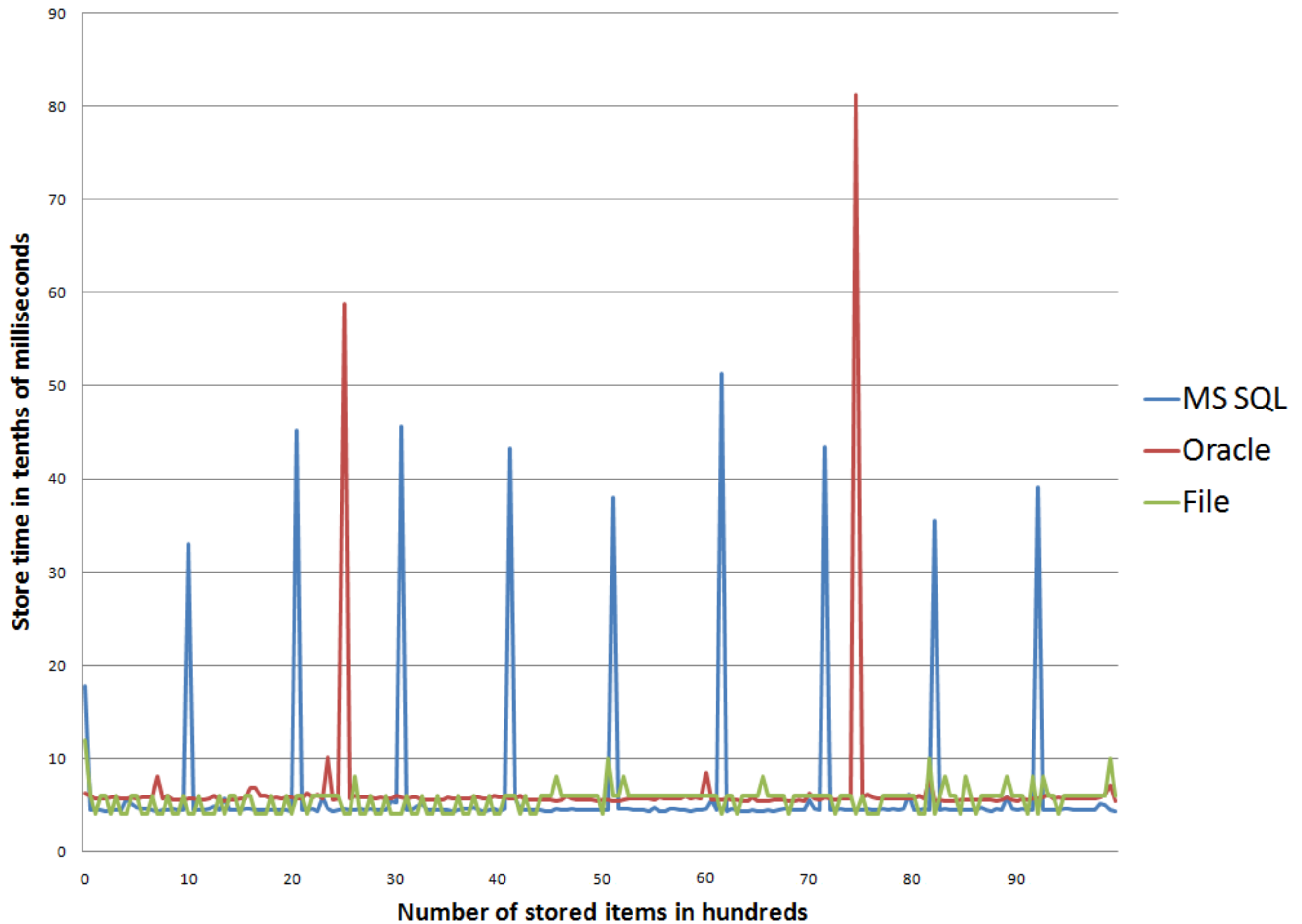
Poznámky k prováděným testům:

1. Účelem prováděných testů nebylo zvolit nejvýkonnější databázový stroj, ale obecně nejvhodnější úložiště využitelné k docílení persistentnosti algo systému. Zvolené databázové systémy od společností Microsoft a Oracle mají pouze demonstrovat výsledky databázových systémů obecně.
2. Testy B a D byly prováděny, neboť databázový systém je typicky nainstalován na speciálním serverovém stroji, nepoběží tedy na stejném stroji jako algo systém.
3. Počet ukládáných položek byl zvolen tak, aby bylo možné korektně analyzovat výsledky testů. Tzn. rozsah ukládaných dat umožní monitorovat případnou závislost času potřebného pro uložení jedné datové položky na inicializaci databázového stroje či na jiných aspektech.
4. Byly prováděny i testy, kdy byly položky ukládány v transakcích. Tento přístup výrazně zvýšil rychlost ukládání, nicméně pro samotný systém nelze použít (důvod je stejný jako v případě asynchronního ukládání).
5. Transakce nad souborovým systémem nejsou typicky podporovány. V testech C a D byly položky ukládány do souboru po 500 kusech najednou.
6. Testy byly prováděny opakovaně pro ověření výsledků.
7. Testy byly prováděny na stroji se stejnou konfigurací jako testy implementací protokolu FIX (kapitola 3.4.5).

Úložiště\Test	**A**	**B**	**C**	**D**
MS SQL Server 2008	0.60ms	0.61ms	0.20ms	0.26ms
Oracle 10g Express Edition	0.60ms	0.66ms	0.23ms	0.37ms
Souborový systém	0.0065ms	-	0.0022ms	-

Tabulka 4.1: Průměrný čas potřebný pro uložení jedné položky

Z výsledků testů v tabulce 4.1 lze vidět, že průměrný čas potřebný pro uložení jedné datové položky je výrazně menší při použití souborového systému jako persistentního úložiště. Tento přístup zaručuje oproti databázovým systémům řádově lepší výsledky.



Obrázek 4.2: Průběh testu A

Pro algo systém je důležitý nejen průměrný čas potřebný pro uložení jedné datové položky, ale také rozptyl od tohoto průměru. Obrázek 4.2 zobrazuje průměrný čas potřebný pro uložení jedné položky (osa Y, čas je uveden v desetinách milisekund, průměrný čas je měřen vždy pro 50 po sobě jdoucích položek) v závislosti na pozici ukládané položky (osa X). Na obrázku lze vidět výrazné odchylky, ke kterým dochází u databázových systémů. Naproti tomu souborový systém se pohybuje rovnoměrně kolem průměrného času (pro souborový systém byl čas na ose Y vynásoben stokrát). V tabulce 4.2 lze vidět rozptyl a nejhorší čas naměřený pro test A. I z tohoto porovnání vychází nejlépe souborový systém.

Souborový systém na prováděných testech dosáhl znatelně lepších výsledků než zvolené databázové systémy. Hlavní výhody databázových systémů (snadná modifikace uložených dat, možnost kladení složitých dotazů atd.) nejsou systémem využívány, proto byl jako vhodné úložiště zvolen souborový systém.

Přestože byl na prováděných testech vyhodnocen jako nejvhodnější souborový systém, je kdykoli možné toto úložiště nahradit. Pro ukládání položek je totiž využito obecného frameworku, který podporuje ukládání jak do souboru, tak do databáze, a je snadno rozšiřitelný o nové typy úložišť.

Úložiště\Naměřená hodnota	**Rozptyl**	**Nejhorší čas**
MS SQL Server 2008	320ms	230ms
Oracle 10g Express Edition	60ms	170ms
Souborový systém	0.0006ms	0.4ms

Tabulka 4.2: Rozptyl a nejhorší čas potřebný pro uložení jedné položky pro test A

4.3 Serializace

Dalším důležitým bodem pro zvýšení výkonnosti systému je rychlost serializace datových položek před ukládáním a také minimalizace velikosti ukládáných dat. Serializace obchodovacích zpráv ovlivňuje také rychlost přenosu zpráv po síti v případě, kdy spolu komunikuje více algo systémů.

Framework .NET podporuje XML ([21]) a binární ([22]) serializaci. XML serializace funguje na jednoduchém principu, kdy jsou do XML vloženy hodnoty veřejných polí třídy. Tento způsob serializace má však několik zásadních nedostatků pro použití v algo systému. Serializované instance tříd jsou sice jednoduše čitelné pro uživatele, ale XML reprezentace je příliš obsáhlá a zpomaluje ukládání do persistentního úložiště. Další nevýhodou je nemožnost serializace specifických dat, která jsou přiložena k obchodovacím zprávám. Specifická data jsou uložena v hašovací tabulce jako dvojice: klíč a hodnota obecného typu `object`. Aby bylo možné tuto strukturu serializovat pomocí XML serializace, bylo by nutné nahradit typ `object` speciálním bázovým typem, který by pomocí atributu `XmlInclude` obsahoval specifikaci všech zděděných tříd, které lze jako specifická data do zprávy uložit. Tento postup je nepřijatelný, protože při implementaci nového algoritmu může vzniknout potřeba vložit do zprávy úplně nový typ dat. V takovém případě by bylo potřeba zasahovat do jádra systému, aby bylo možné tato data správně serializovat (bylo by potřeba upravit obsah atributu `XmlInclude` v bázové třídě, která reprezentuje specifická data).

Alternativou pro XML serializaci je binární serializace. Tento přístup je flexibilnější než XML serializace. Pomocí binární serializace lze serializovat libovolný objekt, který je označen atributem `Serializable`. Po hlubším prozkoumání binární serializace však bylo zjištěno, že serializované obchodovací zprávy jsou příliš veliké na to, jakou informaci nesou. Důsledkem je zpomalení ukládání do persistentního úložiště i pozdější načítání a deserializace. Proto bylo potřeba navrhnout vlastní způsob serializace interní reprezentace zpráv, který informace ze zprávy serializuje rychlým způsobem a velikost serializované podoby dat je úměrná obsaženým informacím. Serializace je založena na převodu zpráv z a do datového typu `string`. Serializace je založena na stejném principu jako převod zprávy do FIX reprezentace, kdy je využit automaticky generovaný kód, který vzniká při překladu systému na základě atributů u položek zprávy (více v kapitole 6.5.1). Zprávu tak lze jednoduše rozšířit o nové datové položky a serializace zpráv bude fungovat korektně bez nutnosti zásahu do kódu jádra systému. Serializace specifických dat zprávy je založena na centrálním úložišti instancí tříd, které serializují jednotlivé datové typy. Obchodovací algoritmus si při inicializaci zaregistruje v tomto úložišti serializační třídy pro specifické datové typy, jejichž instance potřebuje ke zprávě přikládat, a persistence těchto dat i přenos po síti bude fungovat korektně.

V tabulce 4.3 lze vidět, jak ovlivní běh systému použití vlastní serializace oproti použití binární serializace z .NET frameworku. Testy ukázaly násobné urychlení procesu obnovy systému i snížení místa na disku potřebného pro persistování zpráv. Při testování bylo systémem zpracováno 5000 obchodovacích zpráv.

Výsledky testů z tabulky 4.4 lze porovnat s výsledky testů v tabulce 4.1 (použita byla velikost zpráv při použití vlastní serializace). Z porovnání vyplývá, že při použití binární serializace je ukládání do souboru výrazně pomalejší.

Typ serializace	Místo na disku	Proces obnovy systému
Binární	13200KB	1800ms
Vlastní serializace pro standardní položky zprávy, binární serializace pro specifická data	5800KB	1300ms
Vlastní serializace pro celou zprávu	2800KB	800ms

Tabulka 4.3: Srovnání typů serializací

Úložiště\Test	A	B	C	D
MS SQL Server 2008	0.61ms	0.82ms	0.24ms	0.4ms
Oracle 10g Express Edition	0.68ms	0.70ms	0.31ms	0.38ms
Souborový systém	0.017ms	-	0.011ms	-

Tabulka 4.4: Srovnání datových úložišť při použití binární serializace

4.4 Shrnutí

Hlavním hlediskem při návrhu mechanismu, který zajistí persistentnost systému, bylo co nejméně zpomalit běh systému. Po analýze persistentních úložišť a rozebrání možných přístupů pro docílení persistentnosti bylo vybráno jako vhodné řešení postavit persistentnost systému na ukládání obchodovacích zpráv do souboru. Z uložených zpráv lze po nastartování systému obnovit stav všech orderů a tím navázat na předchozí činnost systému.

Kapitola 5

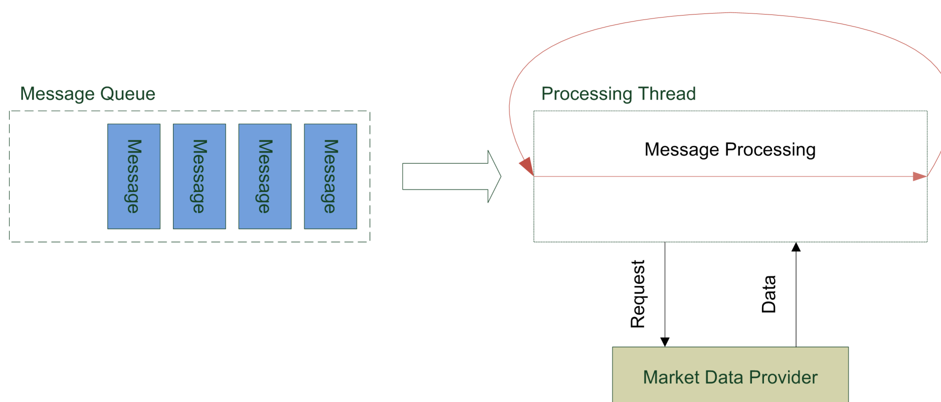
Paralelní zpracování

Pro optimální běh systému bylo potřeba vhodně navrhnout zpracování dat ve více vláknech. Tato kapitola je věnována možným řešením paralelního zpracování obchodovacích dat a rozebírá výhody a nevýhody jednotlivých přístupů.

5.1 Potřeba paralelního zpracování

Základním faktem, který je potřeba si uvědomit, je nemožnost zpracování obchodovacích dat v jednom vlákně. Běh systému v jednom vlákně je nepřípustný zejména pro následující dva důvody:

- Využití pouze části výkonu serverového stroje (za předpokladu, že stroj disponuje více procesorovými jádry).
- Přerušování činnosti systému v důsledku uspání vlákna pro zpracování požadavků. Tato situace by typicky nastávala v případě, kdy by obchodovací algoritmus při zpracování příchozí zprávy potřeboval market data (vlákno by čekalo na vyřízení požadavku a tím by zastavilo běh celého systému).

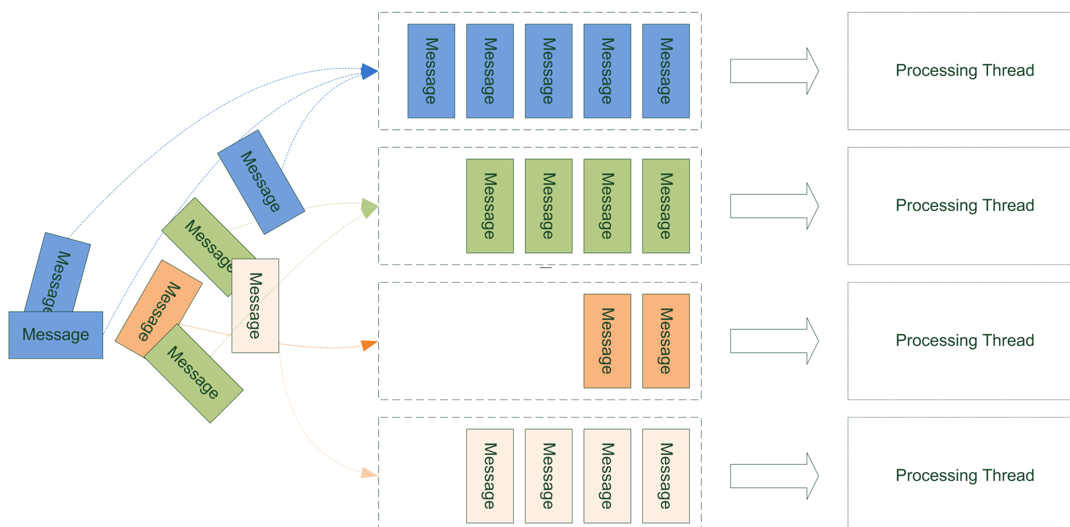


Obrázek 5.1: Zpracování obchodovacích zpráv v jednom vlákně

Obrázek 5.1 znázorňuje vlákno ("Processing Thread" vpravo), které zpracovává veškeré příchozí zprávy do systému. Vlevo je znázorněna fronta ("Message Queue"), kde se hromadí příchozí zprávy (objekty "Message"). Z této fronty jsou zprávy vybírány a zpracovávány, znázorněné požadavky na market data způsobují nečinnost systému, i když jsou ve frontě připraveny další obchodovací zprávy ke zpracování.

5.2 Možná řešení

Prvním možným přístupem bylo vytvořit N front pro příchozí zprávy, kde N je počet procesorových jader stroje, na kterém algo systém běží. Každou z front by obsluhovalo jedno vlákno a zpracovávalo by zprávy z této fronty. Zprávy týkající se jednoho orderu by musely být zařazovány do stejné fronty, aby nedocházelo k jejich předbíhání. Předbíhání zpráv by mohlo mít fatální následky zejména při zpracování zpráv od trhu: v případě, že trh zasílá zprávu o provedení částečného obchodu a následně o zrušení orderu, je velice důležité tuto informaci předat ve stejném pořadí klientskému systému. V opačném případě není klient povinen přijmout obchod, o kterém přišla informace až po zrušení orderu.



Obrázek 5.2: Paralelní zpracování zpráv pomocí front

Obrázek 5.2 znázorňuje rozřazování příchozích obchodovacích zpráv (objekty "Message" v levé části obrázku) do front a následné zpracování (každé frontě odpovídá právě jedno vlákno - "Processing Thread" vpravo). Toto řešení paralelního zpracování rozkládá zátěž mezi všechna jádra procesoru a je vhodné pro situace, kdy zpracování zprávy nevyžaduje synchronní požadavek na market data (popřípadě jinou aktivitu způsobující uspání vlákna). Naprostá většina obchodovacích algoritmů však market data pro korektní rozhodování potřebuje, což je nevýhodou tohoto řešení. Dalším minusem tohoto řešení je, že v případě dlouhých výpočtů spojených se zpracováním konkrétní zprávy by zprávy ve frontě za ní nebyly zpracovány (zpracovány by byly, ale až po delší době, i když by se mezitím jiné fronty vyprázdnily).

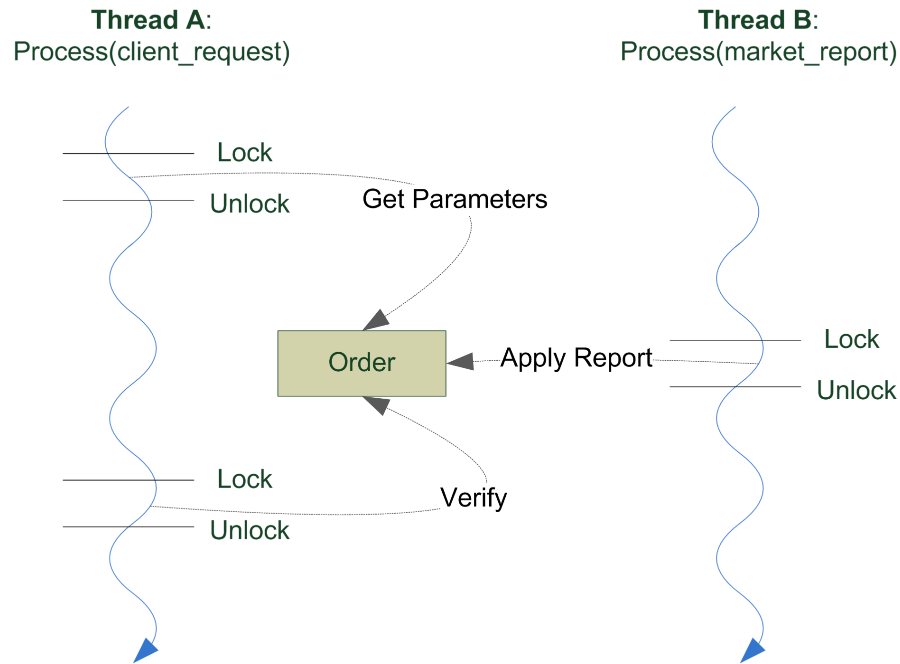
Výše zmíněné nedostatky činí toto řešení nevhodným pro použití v algo systému. Jako vhodnější řešení se jeví využití threadpoolu. Threadpool je komponenta, která přijímá příchozí požadavky, jež zařazuje do fronty, a tyto požadavky přiřazuje ke zpracování volným vláknům. Hlavní výhodou threadpoolu je

nevytváření nového vlákna s každým novým požadavkem (obchodovací zprávou). Vlákna jsou využita opakovaně, což šetří čas při zpracování požadavku. V threadpoolu je typicky vytvořen počet vláken větší než je počet jader procesoru, což umožňuje uspání vlákna, které aktuálně zpracovává požadavek. V takovém případě může operační systém přeplánovat na jiné vlákno z threadpoolu a zpracování dalších zpráv není brzděno.

Při použití threadpoolu je však potřeba zamezit možnosti předbíhání zpráv od trhu. Zamezení předbíhání zpráv je zaručeno pomocí tzv. checkpointů v systému, ve kterých je kontrolováno pořadí zpracování zpráv od trhu. Kontrola pořadí zpráv probíhá před předáním zprávy obchodovacímu algoritmu a před odesláním zprávy klientskému systému. U klientských zpráv není předbíhání kritické. Nepředpokládá se, že by klientský systém zaslal dva požadavky na vytvoření / modifikaci jednoho orderu ve stejný okamžik. V případě, že klientský systém opravdu zašle více požadavků vztažených k jednomu orderu ve stejném okamžiku, algo systém zpracuje pouze první příchozí a ostatní odmítne. Důvodem odmítnutí dalších požadavků je neukončené zpracování prvního požadavku (toto chování odpovídá protokolu FIX, [16]).

Kromě kontroly pořadí zpracovávaných zpráv souvisí s použitím threadpoolu i synchronizace vláken při práci se sdílenými daty. Jedná se zejména o struktury reprezentující jednotlivé ordery. V extrémních situacích může ve stejný okamžik zpracovávat zprávy týkající se jednoho orderu hned několik vláken. Proto je důležité při manipulaci tyto struktury zamykat. Jádro systému bylo navrženo tak, aby obchodovací algoritmy byly odstíněny od používání konkrétního synchronizačního primitiva. Obchodovací algoritmy vždy volají metody jádra systému pro uzamknutí a odemknutí orderu. Tím je docíleno jak synchronizace mezi více obchodovacími algoritmy, tak i synchronizace mezi algoritmy a jádrem samotným.

Možnost paralelního zpracování více zpráv týkajících se stejného orderu má i výhody. Představme si situaci (obrázek 5.3), kdy obchodovací algoritmus obdrží od klientského systému požadavek na modifikaci orderu ("client_request" vlevo zpracovávaný ve vlákně A - "Thread A"). Algoritmus si zjistí aktuální stav orderu (např. zobchodovanou kvantitu) jako vstupní parametry pro komplexní výpočet pro určení další strategie (order je zamknut jen po dobu zjišťování jeho stavu, při výpočtech další strategie je již odemknut). V průběhu výpočtu systém obdrží report z trhu("market_report" vpravo zpracovávaný ve vlákně B - "Thread B"), který mění stav orderu (tedy i vstupní parametry pro výpočet algoritmu). Obchodovací algoritmus může díky paralelnímu zpracování zpráv detekovat změnu stavu orderu a výpočet provést znovu na základě aktualizovaného stavu.



Obrázek 5.3: Příklad synchronizace

Ačkoli použití threadpoolu přináši i některé komplikace (nutnost zamykání sdílených dat, kontrola pořadí zpráv), celkově se jeví jako vhodnější řešení než zpracování příchozích zpráv pomocí více front (nevyužití prostředků systému při pasivním čekání a brzdění dalších zpráv ke zpracování).

5.3 Volba implementace threadpoolu

Ačkoli by bylo možné implementovat v rámci algo systému vlastní threadpool, tato implementace nebyla nutná, neboť v současnosti existuje několik hotových řešení. Jako první byly rozebrány možnosti threadpoolu, který je přímo součástí frameworku .NET. Toto řešení však není vhodné pro spouštění operací, které nepotřebují využívat procesor (např. operace s diskem, čekání na odezvu vzdáleného systému atd., více v [5]). Nevhodné chování threadpoolu z .NET frameworku lze demonstrovat na následujícím příkladu: v threadpoolu je spuštěno N + 1 vláken (N je počet procesorových jader), každé z vláken vypíše svůj identifikátor a poté se na jednu vteřinu uspí. Vlákno s pořadovým číslem N + 1 vypíše svůj identifikátor až po 1 vteřině. Pokud by došlo k přeplánování, mohlo by vlákno s identifikátorem N + 1 přijít na řadu ihned po uspání prvního ze zbývajících vláken. Operace způsobující uspání vlákna však při zpracování obchodovacích zpráv nelze vyloučit (například pasivní čekání na market data). Z tohoto důvodu threadpool z .NET frameworku nemohl být použit.

Jako vhodné řešení byl naopak zvolen Smart Thread Pool ([6]). Jedná se o opensource projekt, který nabízí oproti threadpoolu z .NET frameworku mnoho přidané funkcionality. Pro algo systém je nejdůležitější právě možnost pasivního čekání ve vláknech, které běží v rámci threadpoolu.

Kapitola 6

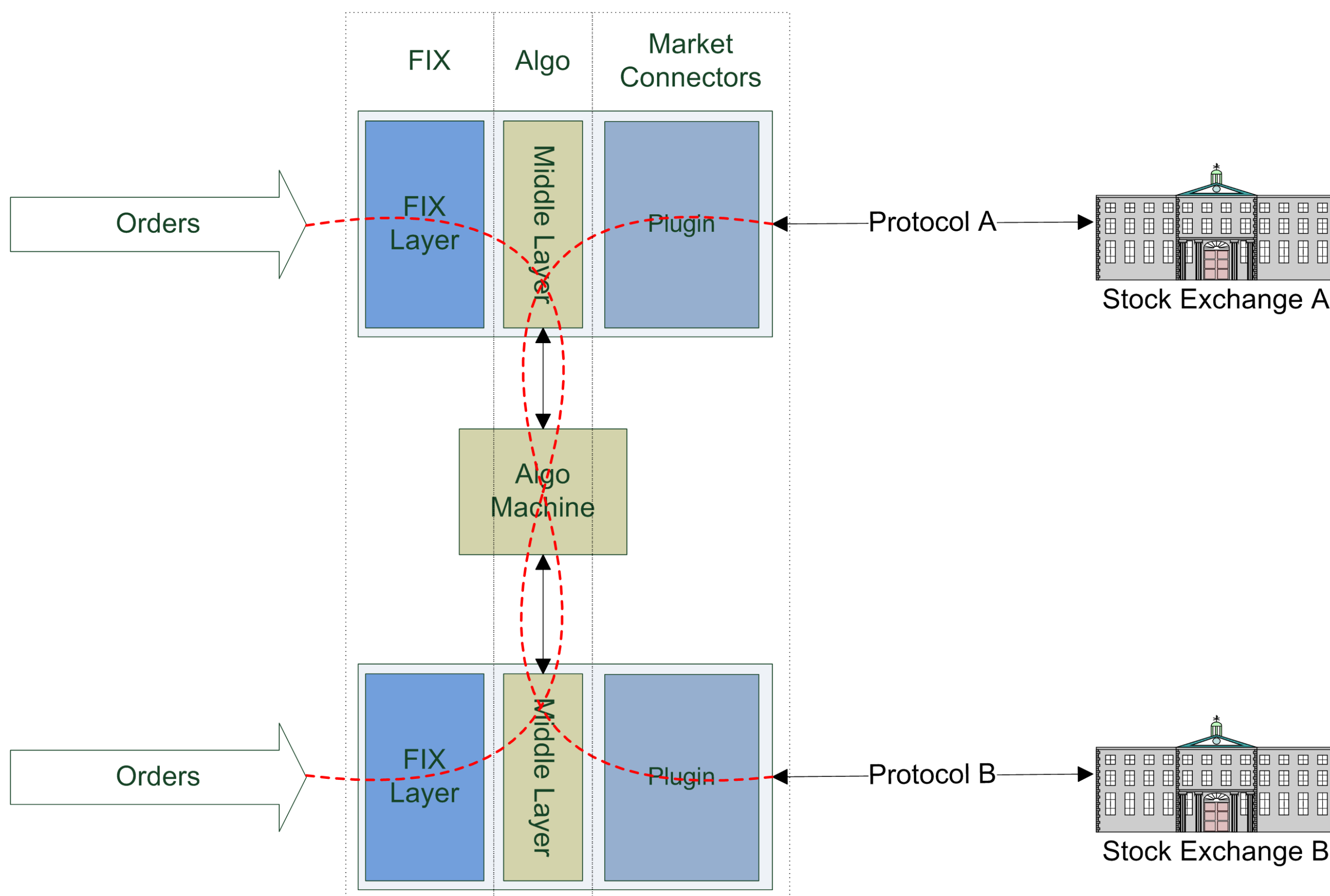
Architektura systému a implementace

6.1 Možná řešení

Při návrhu systému bylo potřeba zohlednit požadavky a celkovou situaci v zadavatelské firmě (zejména existenci FIX adaptérů, které lze využít pro připojení k trhům, více v kapitole 2.3). Při analýze systému bylo navrženo několik přístupů k problému. V dalším textu budou tato řešení popsána a budou rozebrány jejich hlavní přednosti a nedostatky.

6.1.1 Prostředník mezi FIX vrstvou a překladovým pluginem

Prvním možným řešením je úprava existujících FIX adaptérů, jejichž architektura je poměrně jednoduchá. Systém FIX adaptér byl vytvořen stejně jako algo systém pro platformu .NET. Adaptér se skládá z FIX vrstvy, která reprezentuje jádro systému, a z pluginu, který překládá zprávy do protokolu, kterému rozumí burza. Při návrhu adaptéru nebyla zohledněna možnost využití tohoto systému pro jiné než překladové účely. Systém překládá vstupní zprávy v poměru 1 : 1 (pro každou vstupní vytvoří jednu výstupní zprávu). Toto chování neodpovídá potřebám algoritmického systému, který umisťuje na burzu ordery na základě mnohem komplexnějších podmínek (nikoli jen na základě zprávy od klientského systému). Další nevýhodou FIX adaptéru je využívání databáze pro docílení persistentnosti systému, což výrazně zpomaluje zpracování příchozích zpráv (viz. kapitola 4). Naopak výhodou systému je stejný formát vstupu a výstupu jako u algo systému (vstupem jsou požadavky z klientských systémů a reporty z trhů, výstupem jsou přeložené verze těchto zpráv). Z tohoto důvodu byla analyzována možnost využití tohoto systému (nebo jeho části) pro vznik algoritmického stroje. Možným řešením, jak FIX adaptér využít, je vložit mezi FIX vrstvu a překladový plugin speciální komponentu, která by nebyla přímo algoritmickým strojem, ale zprávy z FIX vrstvy by mu zasílala (obrázek 6.1). Algo systém by příchozí zprávy vyhodnocoval a přeposílal do překladových pluginů. Takto by byl systém napojen na všechny adaptéry a měl by dostupné informace o všech zasílaných orderech.



Obrázek 6.1: Prostředník mezi FIX vrstvou a pluginem

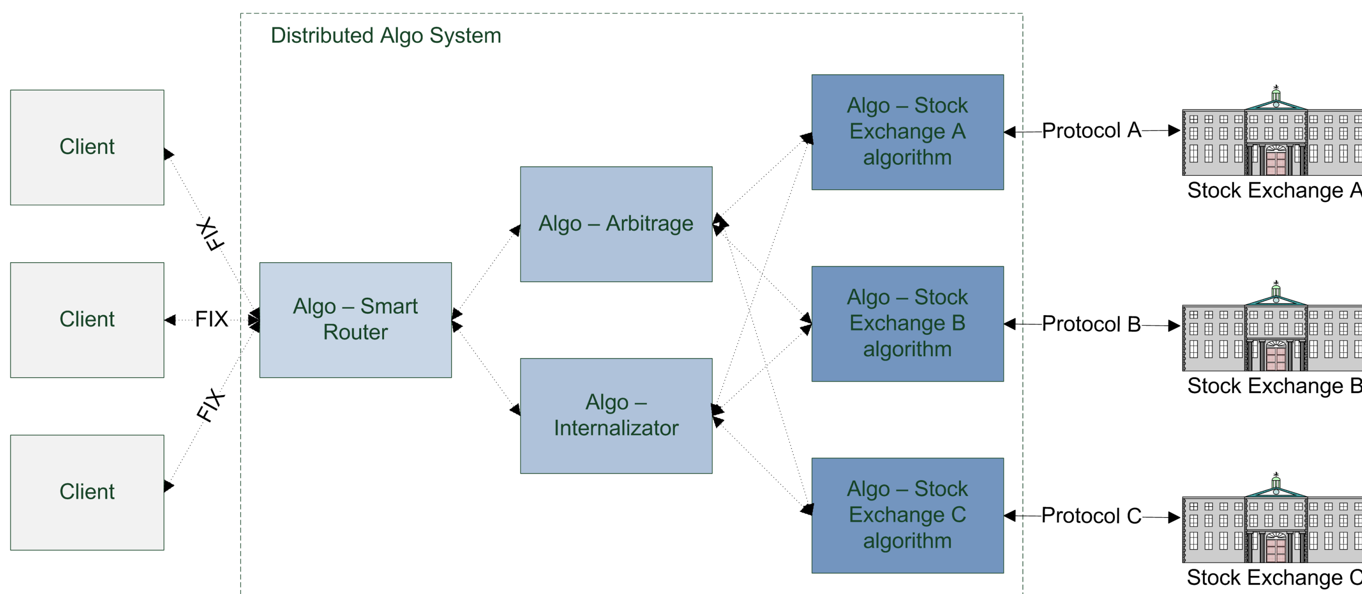
Mezi hlavní výhody tohoto řešení patří využití FIX adaptérů, které by sice musely být upraveny, ale značná část již hotového a používaného systému by byla využita. Naopak nevýhodou je nutnost běhu alga jako samostatného procesu (mimo FIX adaptér). Oproti standardnímu zpracování požadavku by musela být zpráva zaslána (typicky přes síť nebo v rámci jednoho stroje například přes named pipes) z FIX vrstvy algu, které by zprávu zpracovalo a případně zaslalo na některý z překladových pluginů. Tento fakt způsobuje zpomalení systému. Nevýhodou tohoto řešení je také jeho složitost, protože FIX vrstva pro příjem klientských požadavků by v tomto distribuovaném systému stačila pouze jedna a mohla by být přímo součástí algoritmického stroje. V této modifikaci by tedy v algu bylo využito FIX vrstvy jako vstupního bodu do systému. Otázkou však zůstává, kam zařadit překladové pluginy. Jako vhodné řešení se jeví rozšířit funkcionalitu systému o možnost běhu více překladových pluginů najednou (toto FIX adaptér neumožňuje). Toto řešení se již hodně blíží návrhu, který byl použit při implementaci (kapitola 6.1.2).

6.1.2 Samostatný systém

V závěru předchozí kapitoly bylo popsáno řešení, které umožňuje fungování alga jako samostatného systému, do kterého se ve formě pluginů přidávají konektory na různé burzy. Jeho nevýhodou je však přílišná vázanost na protokol FIX (FIX vrstva adaptéru navíc podporuje pouze verzi protokolu 4.1). Vstup do systému může být tvořen pouze zprávami ve formátu FIX, který se nehodí například pro komunikaci více algo

systémů. Tyto systémy si mohou mezi sebou kromě zpráv samotných vyměňovat další komplikovanější informace, pro které není v protokolu podpora. Proto byl jako finální řešení vybrán návrh, který není spjat s konkrétním vstupem ani výstupem systému. Obě rozhraní jsou tedy řešena ve formě pluginů. Výhodou je vyšší obecnost systému a nevázanost na konkrétní protokol, nevýhodou pak menší využití existujícího řešení. Z FIX adaptéru mohou být využity pouze překladové pluginy.

Při použití tohoto návrhu je umožněn běh více algoritmických systémů současně, přičemž každý se specializuje na konkrétní úkoly. Jednotlivé instance algo strojů spolupracují a tvoří komplexní distribuovaný systém. Tím je rozložena zátěž mezi více strojů a je docíleno vyšší výkonnosti systému. Příklad distribuovaného algoritmického stroje lze vidět na obrázku 6.2. Vstupním bodem do systému je tzv. smart router, jehož úkolem je přeposílat ordery na aktuálně nejlepší destinaci (některé cenné papíry se současně obchodují na více burzách). V druhé vrstvě lze vidět Arbitrage algoritmus a internalizátor. Internalizátor je simulátor burzy uvnitř brokerské společnosti. V něm se uskutečňují obchody mezi klienty aniž by jejich ordery byly umístěny na reálný trh. Internalizátor se může rozhodnout, zda order odešle přímo na trh, či zda ho pozdrží a počká na případný protiorder. V poslední vrstvě lze vidět překladové algoritmy, které komunikují pouze s jednou z burz a jsou vyvinuté speciálně pro chování dané burzy.



Obrázek 6.2: Distribuovaný algo systém

Využití algo systému pro překlad z protokolu FIX do protokolu burzy s sebou nese hned několik výhod. Překlad by měl probíhat rychleji než při použití FIX adaptéru a hlavně algo systém může klientovi vytvořit dokonalejší iluzi, že systém burzy komunikuje protokolem FIX. Jako příklad lze uvést simulaci modifikace orderu, i když ji burza nepodporuje. Některé z burz totiž podporují pouze vytvoření a rušení orderu. V takovém případě o tom klientský systém nemusí vůbec vědět a kdykoli pošle požadavek na modifikaci orderu, algo systém zruší původní order a zašle na burzu order s novými parametry.

6.2 Základní rozvržení systému

Na obrázku 6.3 je znázorněno rozvržení systému do hlavních komponent. V horní části obrázku lze vidět komponentu `ClientInterface`, která slouží pro komunikaci s klientskými systémy (příjem požadavků / odesílání reportů). Jelikož je systém nezávislý na vstupním protokolu, tato komponenta sama o sobě neobsahuje žádnou překladovou logiku. Veškeré překlady z / do protokolu, který používá klientský systém, jsou realizovány pomocí speciálních pluginů (klientských konektorů), které jsou v komponentě zapouzdřeny. Jakmile klientský konektor obdrží zprávu od klientského systému (zpráva obsahuje požadavek na vytvoření / modifikaci / zrušení orderu), přeloží tuto zprávu do interní reprezentace a předá ji ke zpracování komponentě `ClientInterface`. Komponenta `ClientInterface` tuto zprávu předá ke zpracování jádru systému.

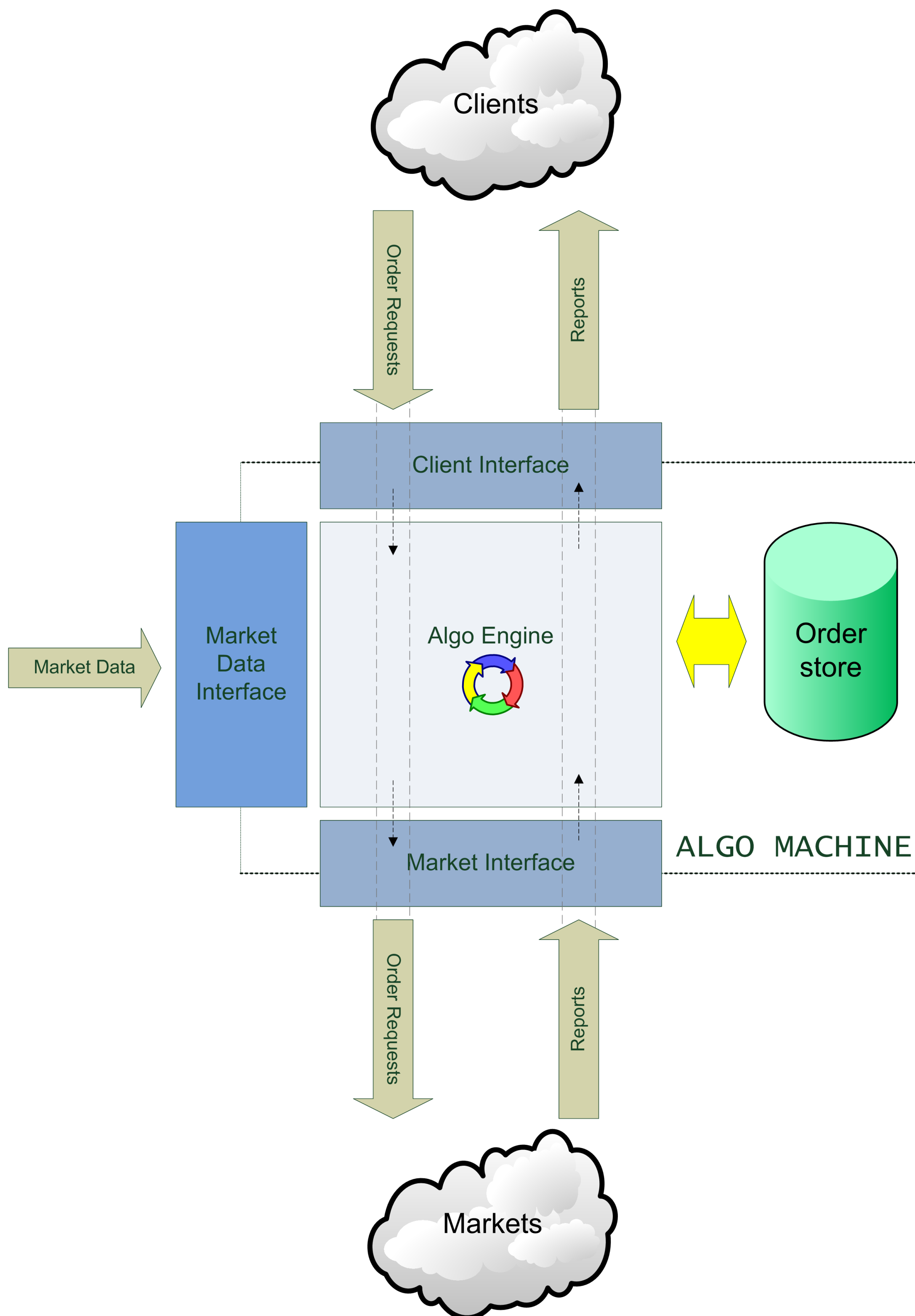
Jádro systému (komponenta `AlgoEngine` uprostřed obrázku 6.3) samo o sobě neobsahuje obchodovací strategii, ta je v systému obsažena přes pluginy reprezentující obchodovací algoritmy. Jádro systému tyto pluginy zapouzdřuje a poskytuje jim podpůrnou funkcionalitu. Po obržení zprávy od klientského systému jádro nejdříve tuto zprávu zkontroluje (zda obsahuje validní data) a následně je zpráva předána ke zpracování obchodovacímu algoritmu. Obchodovací algoritmus reaguje na příchozí klientskou zprávu podle strategie, která je algoritmem implementována. Typicky je zkontrolován aktuální stav na burze (market data) a poté vyhodnocena optimální reakce (například zaslání zprávy pro vytvoření nového orderu na burze). V případě, že algoritmus zasílá na burzu zprávu, je tato zpráva předána prostřednictvím jádra systému komponentě `MarketInterface`, která zaručí její odeslání na burzu.

Komponenta `MarketInterface` (na obrázku 6.3 ve spodní části) funguje na stejném principu jako komponenta `ClientInterface`, zajišťuje však komunikaci s burzovními systémy. Zpráva vygenerovaná obchodovacím algoritmem je přeložena pomocí trhového konektoru (plugin zapouzdřený v komponentě `MarketInterface`) a odeslána na burzu. Reakce z burzy se do systému dostávají také přes trhové konektory, po přeložení do interního formátu jsou předány jádru systému a zpracovány obchodovacím algoritmem. Algoritmus informuje o stavu orderu klientské systémy přes komponentu `ClientInterface`. Tok zpráv je na obrázku 6.3 znázorněn šipkami vedoucími napříč jednotlivými komponentami.

Při zpracování zpráv jádrem systému jsou tyto zprávy aplikovány na globální úložiště orderů (na obrázku 6.3 vpravo). Aplikovat zprávu na úložiště znamená aktualizovat podle příchozí zprávy parametry orderu, kterého se daná zpráva týká. Toto úložiště je dostupné obchodovacím algoritmům.

Obchodovací algoritmy jsou závislé na znalosti aktuálního stavu trhu (příjmu market dat). Market data jsou zprostředkována algoritmům pomocí komponenty `MarketDataInterface`, kterou lze vidět v levé části obrázku 6.3. Tato komponenta funguje na podobných principech jako komponenty pro komunikaci s klientskými a trhovými systémy (tzn. je nezávislá na vstupním formátu market dat, překlad je realizován pomocí speciálních pluginů).

Zvolená architektura odpovídá požadavkům na algo systém (kapitola 2.3). Komunikační rozhraní systému reprezentované komponentami `ClientInterface` (komunikace s klientskými systémy), `MarketInterface` (komunikace s trhovými systémy) a `MarketDataInterface` (příjem market dat) je zcela nezávislé na vstupním formátu dat. Jádro systému však pracuje pouze s jedinou reprezentací dat (jádro si vyměňuje data s komunikačními komponentami v interní reprezentaci, kapitola 6.5), díky čemuž lze implementované algoritmy využívat pro obchodování na libovolné z burz a poskytovat algoritmus libovolnému z klientských systémů (včetně interních systémů v zadavatelské firmě).



Obrázek 6.3: Základní komponenty systému

6.3 Komponenty systému a jejich funkce

V předchozí kapitole byly popsány základní prvky architektury systému, nyní budou jednotlivé komponenty a subkomponenty popsány podrobněji. Detailní schéma znázorňující propojení jednotlivých komponent lze vidět na obrázku 6.4. Toto schéma je detailnější variantou obrázku 6.3, kromě hlavních komponent zobrazuje i subkomponenty a pluginy systému.

6.3.1 ClientInterface

Jedná se o komponentu pro příjem klientských zpráv (na obrázku 6.4 lze vidět v horní části). Účelem komponenty je vytvořit pro jádro systému iluzi, že komunikuje se všemi vzdálenými klientskými systémy stejným způsobem. Tím je umožněno obchodovacím algoritmům přijímat zadání úkolu (úkolem je myšleno plnění klientských požadavků - orderů, např. nákup X akcií společnosti Y za cenu VWAP) v jednotném formátu. Účelem komponenty je také úplně oddělit jádro systému od povinnosti starat se o správné doručení zpráv klientovi. Komunikace od komponenty k jádru systému probíhá podle scénáře: přišla zpráva X od klientského systému Y. V opačném směru pak: pošli zprávu Z systému Y.

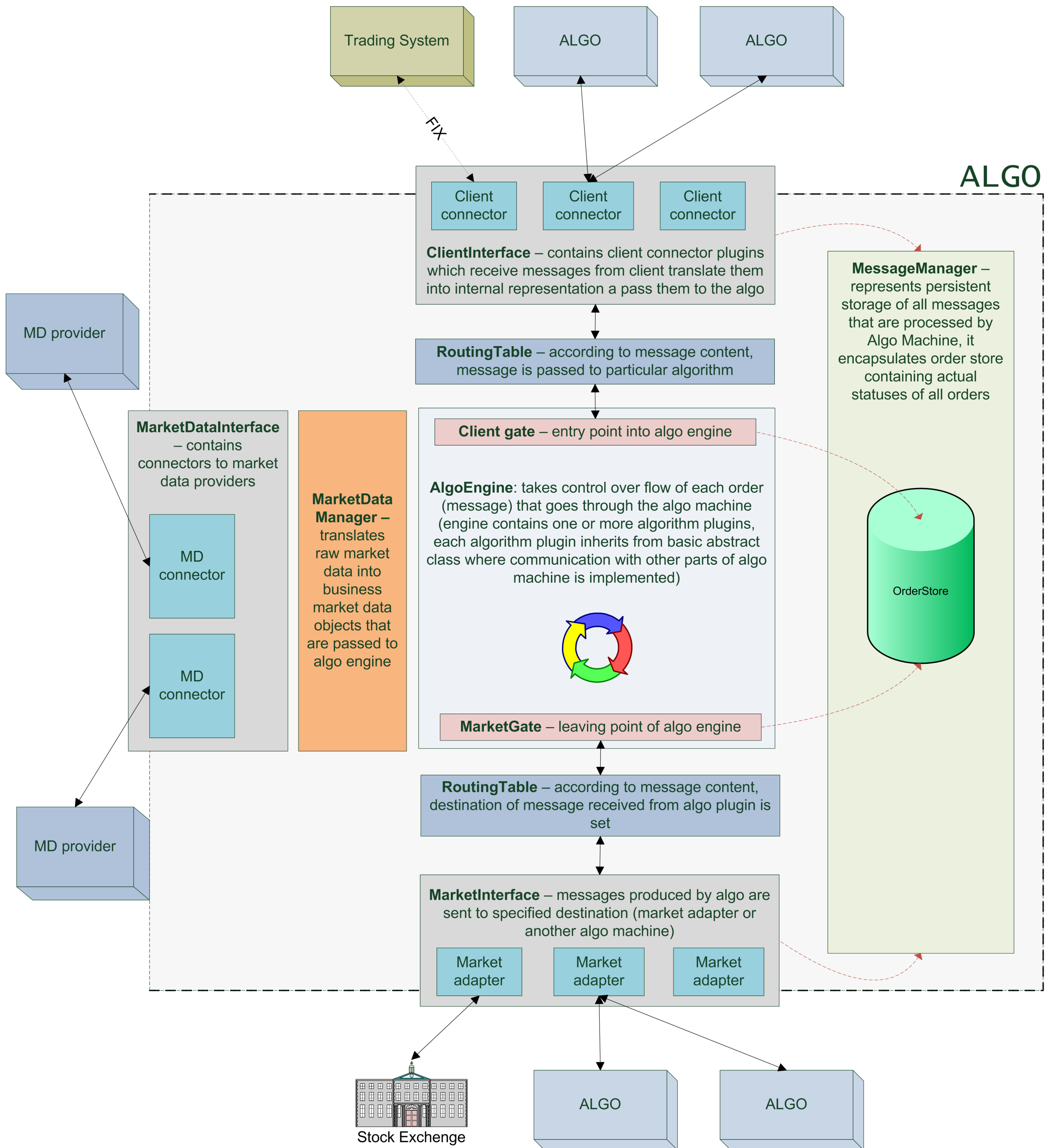
Komponenta byla navržena tak, aby mohla pojmout libovolné množství pluginů sloužících pro komunikaci s klientskými systémy. Každý z pluginů překládá příchozí zprávy do interní reprezentace, se kterou pracuje jádro systému (více v kapitole 6.5.1). Klientské pluginy specifikují seznam systémů, se kterými komunikují, a pro každý z těchto systémů také stav připojení. Plugin může zprostředkovávat komunikaci s mnoha vzdálenými systémy, žádné dva z pluginů však nesmí komunikovat se stejným systémem (v takových situacích není zřejmé, který z pluginů použít pro zasílání zpráv směrem ke klientskému systému).

Ačkoli se o přímou komunikaci s klientskými systémy starají pluginy, společná funkcionalita (funkcionalita využitelná pro všechny pluginy) je zahrnuta přímo v komponentě `ClientInterface`. Příkladem může být bufferování zpráv pro nedostupné klientské systémy. Pro korektní funkcionalitu systému je důležité, aby byly klientským systémům odeslány veškeré zprávy, které jsou jim adresovány. A to i v případě, že jsou klientské systémy dočasně nedostupné. Klienti využívající služeb brokerské společnosti nemusí přijmout obchod, o kterém nebyli informováni. Pokud by došlo ke ztrátě zprávy, která nese informaci o provedeném obchodu, mohlo by to způsobit vysokou finanční ztrátu (obchod v takovém případě přechází na brokerskou firmu).

6.3.2 MarketInterface

Komponenta `MarketInterface` se stará o komunikaci s burzami cenných papírů (na obrázku 6.4 lze vidět v dolní části). Účel komponenty je podobný jako u komponenty `ClientInterface`, tedy vytvořit jádru systému iluzi, že se všemi burzovními systémy lze komunikovat stejným způsobem. Tato komponenta umožňuje obchodovacím algoritmům umisťovat ordery na burzu (kromě vytváření lze ordery obecně také modifikovat a rušit). Až na drobné odlišnosti je funkcionalita komponenty založená na stejných principech jako komponenta `ClientInterface`, pouze zastřešuje komunikaci s burzovními systémy.

Podobně jako komponenta pro komunikaci s klientskými systémy může i komponenta `MarketInterface` pojmout libovolné množství pluginů. Komponenta si vyměňuje data (zprávy) s jádrem systému v interní reprezentaci, úkolem pluginů je tyto zprávy přeložit do specifické podoby (které rozumí burza) a zprávu burzovnímu systému odeslat (úkolem pluginů je samozřejmě i příjem zpráv z trhu a překlad do interní reprezentace).



Obrázek 6.4: Architektura systému

Hlavní odlišností od komponenty `ClientInterface` je chování v případě nedostupnosti vzdáleného systému. Pokud je burzovní systém nedostupný, zpráva je automaticky odmítnuta. Algoritmus, který zprávu vygeneroval, dostane zprávu obsahující informaci o odmítnutí zprávy z důvodu nedostupnosti vzdáleného

systému. Bufferování zpráv v případě nedostupnosti burzovního systému není korektní. U každého požadavku platí, že může být modifikován či zrušen až po potvrzení přijetí burzou. Potvrzení od burzy však v tomto případě nedorazí, dokud není spojení s burzovním systémem obnoveno. V takovém případě by mohl být na burzu umístěn požadavek, který již není aktuální (v případě, že se mezitím výrazně změnila cena obchodovaného cenného papíru, hrozí vysoká ztráta).

6.3.3 MarketDataInterface

Komponenta `MarketDataInterface` dotváří (po komponentách `ClientInterface` a `MarketInterface`) komunikační rozhraní systému. Účel komponenty a principy fungování jsou podobné jako u výše zmíněných rozhraní pro komunikaci s klientskými, respektive trhovými systémy. Komponenta obsahuje libovolné množství market data konektorů (pluginů), které přijímají data z trhů a překládají je do tzv. raw interní reprezentace (více v kapitole 6.5.2.1), čímž jádru systému vytváří iluzi stejného formátu market dat ze všech burz.

Výměna market dat mezi komponentou `MarketDataInterface` a jádrem systému je realizována na principech využitých v protokolu FIX (kapitola 3.2). Jádro systému zasílá požadavek o specifická market data (například seznam orderů na burze pro konkrétní cenný papír), komponenta `MarketDataInterface` vrátí jádru aktuální stav na burze a následně zasílá aktualizace tohoto stavu. Tímto způsobem je zajištěno, že jádro systému má vždy k dispozici aktuální market data. Výměna market dat mezi komponentou `MarketDataInterface` a jádrem systému mohla být založena i na principu použitém například v protokolu CHIXMD. V tomto protokolu není zaslán žádný požadavek a poskytovatel market dat automaticky zasílá veškeré informace o dění na trhu automaticky (na podobném principu je založena celá řada dalších protokolů pro zasílání market dat). Výhodou řešení je rychlejší odezva na požadavek o market data (požadavek je vyřízen uvnitř systému, není potřeba jej zasílat přes market data konektor vzdálenému systému). Naopak nevýhodou je nutnost zpracovávat typicky obrovské množství market dat přímo v rámci algo systému (vede ke zpomalení systému).

Pro některé z burz platí, že poskytují v rámci jednoho komunikačního kanálu jak možnost zasílat ordery, tak přijímat market data (z celkového počtu více než 10 analyzovaných burz byla taková nalezena pouze jedna). V takovém případě by stačila pouze komponenta `MarketInterface`, která by sloužila pro komunikaci s burzou i pro příjem market dat. Pokud pomineme skutečnost, že většina burz poskytuje pro market data speciální komunikační kanál, existují další důvody, proč komponenty oddělit:

- Nemožnost jednoduše odebírat market data z jiného zdroje než je samotná burza (např. systém Bloomberg).
- Algo systém nemusí komunikovat přímo s burzou, ale například s dalším algo systémem, který by musel market data přeposílat.
- Zjednodušení architektury systému (plyne z předchozího bodu).

Při návrhu systému byl brán v potaz plánovaný vznik market data serveru jako jednotného úložiště market dat v zadavatelské firmě. Do komponenty `MarketDataInterface` se tak v budoucnu pouze zapojí jednoduchý plugin, který bude z market data serveru přijímat data v raw podobě a distribuovat tato data dále do systému.

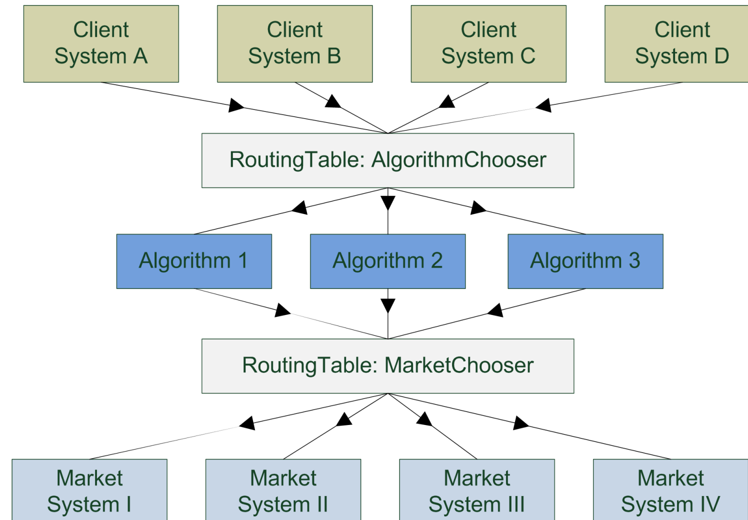
6.3.4 MarketDataManager

Práce s market daty v raw podobě (dvojrozměrné pole atributů popisující market data objekt, více v kapitole 6.5.2.1) je komplikovaná a nevhodná pro použití při implementaci obchodovacího algoritmu. Proto byla mezi jádro systému a komponentu `MarketDataInterface` vložena komponenta `MarketDataManager`, jejímž hlavním úkolem je překlad raw market dat do market data business objektů. Každému typu market dat odpovídá speciální třída, ze které lze pohodlně dostat požadovaná data. Komponenta obsahuje metody pro získání aktuálního stavu trhu (například seznam obchodovaných cenných papírů, aktuální ceny papírů atd.) a dále tzv. subscribe metody, pomocí nichž se algoritmus může zaregistrovat pro příjem aktualizací konkrétního typu market dat.

Komponenta si drží v paměti aktualizované business market data objekty. Po každé aktualizaci market dat jsou business objekty předány algoritmům, které jsou pro daný typ market dat zaregistrovány. Algoritmům jsou vždy předány klony business objektů. Tímto jsou řešeny situace, kdy je aktualizován business objekt, se kterým algoritmus právě pracuje v jiném vlákně. V komponentě se udržuje aktuální stav pro market data, se kterými pracuje alespoň jeden algoritmus. Tím je urychleno vyřízení požadavku o data, která jsou již využívána jiným algoritmem.

6.3.5 RoutingTable

Komponenta tvořící mezičlánek mezi jádrem systému a komponentami `ClientInterface` a `MarketInterface` (routovací komponenty ve vztahu ke klientským systémům, trhovým systémům a algoritmům lze vidět na obrázku 6.5). Protože systém komunikuje s mnoha vzdálenými systémy (jak na klientské, tak na trhové straně), je potřeba pro průchozí zprávy zvolit správně adresáta. Příchozí zprávy od komponenty `ClientInterface` jsou předány do routovací komponenty (`AlgorithmChooser`, na obrázku 6.4 pod komponentou `ClientInterface`), která na základě obsahu zprávy zvolí algoritmus (systém může obsahovat více algoritmů), který zprávu zpracuje. Toto platí pouze pro zprávy vytvářející nový order. Modifikační zprávy jsou předány přímo algoritmu, který daný order zpracovává. Reakce na klientské zprávy, které vygenerovalo jádro systému, jsou automaticky adresovány klientskému systému, který zaslal požadavek na vytvoření daného orderu. Podobně je routovací komponenta vložena mezi jádro systému a komponentu `MarketInterface` (`MarketChooser`, na obrázku 6.4 nad komponentou `MarketInterface`). V tomto případě routovací komponenta určí na základě obsahu zprávy, na který trh bude odeslána. Modifikační zprávy a reakce od trhu jsou routovány automaticky.



Obrázek 6.5: Routování zpráv

Hlavní význam komponent pro routování je usnadnit implementaci obchodovacích algoritmů. Pro samotný obchodovací algoritmus není důležité, kterému systému bude jím vygenerovaná zpráva odeslána. Obchodovací algoritmus do zprávy pouze uvede, na jakou burzu chce zprávu zaslat (routování může být založeno i na jiném obsahu zprávy, například na identifikátoru cenného papíru) a dále je již zpráva zaslána správnému adresátovi podle nastavení routování.

Konfigurace routovacích komponent je obsažena v XML souboru, který obsahuje množinu cílů a pro každý z nich sadu pravidel, které musí zpráva splňovat, aby byla adresována danému cíli (algoritmus / trh). Zpráva je adresována prvnímu cíli, pro který je seznam pravidel splněn.

6.3.6 MessageManager

Komponenta `MessageManager` zastřešuje persistentní úložiště orderů a její hlavní význam spočívá v tom, že odděluje jádro systému od povinnosti starat se o persistentnost. Veškeré zprávy procházející systémem jsou předány komponentě `MessageManager`, která realizuje ukládání zpráv do úložiště a aktualizaci orderů. Komponenta není spjata s konkrétním typem úložiště, může tedy pracovat jak nad databází, tak nad soubory. Po restartu systému je stav všech orderů obnoven ze zpráv, které byly systémem zpracovány v předchozím běhu. Komponenta je tedy persistentní a díky ní i celý systém (persistentnosti systému je věnována kapitola 4).

Stav systému je dán aktuálním stavem orderů, které jsou jím zpracovávány. Stav orderů je důležitý pro všechny obchodovací algoritmy, které v systému fungují. Proto je tato funkcionalita zahrnuta do systému samotného a nemusí ji každý z algoritmů implementovat sám (stav orderů je potřebný i pro jádro systému, lze díky tomu například validovat příchozí zprávy).

Z obrázku 6.4 lze vidět, že komponenta stojí mimo standardní tok zpráv (komponenta se nachází v pravé části schématu). Napojena je na jádro systému a komponenty `ClientInterface` a `MarketInterface`. Po obdržení klientské zprávy ji komponenta `ClientInterface` předá komponentě `MessageManager`, která ji uloží do persistentního úložiště. Zpráva v tomto okamžiku ještě není aplikována na úložiště orderů. Uložení zprávy probíhá synchronně (uložení musí probíhat synchronně, aby nemohlo dojít

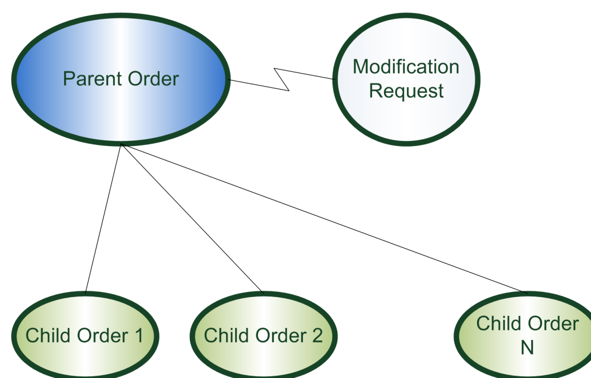
k akci algoritmu na základě dosud neevidované zprávy, více v kapitole 4.2). Po uložení zprávy je zpráva předána další komponentě v systému. Stejný princip funguje pro trhové zprávy přijaté pomocí komponenty `MarketInterface`.

Jakmile se zpráva od klienta dostane do jádra systému, je tato zpráva aplikována na úložiště orderů v rámci komponenty `MessageManager`. Nastat mohou následující situace:

- Ze zprávy `OrderNewSingle` (požadavek o vytvoření nového orderu) je vytvořen nový tzv. parentský order. Order je označen jako parentský, protože na základě něj algoritmus typicky vygeneruje více dílčích orderů, které jsou umístěny na trh (tyto ordery jsou pak označovány jako child ordery).
- Při aplikaci zprávy `OrderNewSingle` na úložiště orderů dojde k výjimce `DuplicateOrderIde-ntifierException` v případě, že klientský identifikátor orderu není unikátní (tzn. již byl systémem přijat požadavek na vytvoření orderu se stejným identifikátorem). V tomto případě je order systémem odmítnut (klientskému systému je odeslána zpráva obsahující důvod odmítnutí).
- Zpráva `OrderReplaceRequest` (požadavek o modifikaci orderu) je aplikována na existující order. Order je přepnut do stavu pending (nevyřešený), tzn. vlastnosti orderu nebyly změněny, pouze je k němu uložen požadavek na modifikaci. V případě, že je požadavek algoritmem přijat, jsou orderu nastaveny nové parametry.

  Podobně pro zprávu `OrderCancelRequest` (požadavek o zrušení orderu).
- Při aplikaci zprávy `OrderReplaceRequest` na úložiště orderů dojde k výjimce `OrderCannotB-eModifiedException`. K této výjimce může dojít z několika důvodů (jedním z nich může být neexistence modifikovaného orderu, chybný obsah modifikační zprávy atd.). V tomto případě je požadavek odmítnut (klientskému systému je odeslána zpráva obsahující důvod odmítnutí).

  Podobně pro zprávu `OrderCancelRequest`.



Obrázek 6.6: Reprezentace orderu v úložišti

Na obrázku 6.6 lze vidět příklad reprezentace orderu v úložišti. Order se nachází v pending stavu, kdy čeká na přijmutí či odmítnutí modifikačního požadavku (požadavek na modifikaci - "Modification Request" je navázán na parentský order, lze vidět v horní části obrázku). Algoritmus vyprodukoval při zpracovávání orderu N child orderů. Vazby mezi parentským orderem a jeho childy jsou v úložišti uchovávány, aby bylo možné pro potřeby algoritmu všechny jeho child ordery dohledat.

Zpracování zpráv od trhu se od zpracování klientských zpráv liší v tom, že pokud dojde při aplikaci zprávy na úložiště orderů k jakékoli chybě (výjimce), je tento fakt daleko závažnější než v případě klientské zprávy (chyba při zpracování klientské zprávy znamená pouze odmítnutí dané zprávy). Výjimka při zpracování zprávy od trhu by při normálním běhu systému neměla nastávat. Pokud se objeví, značí to chybu v systému, popřípadě chybu v trhovém systému, který zprávu vyprodukoval. Chyba je okamžitě reportována (například zasláním emailu) a zpráva není předána algoritmu ke zpracování. Jako příklad této situace lze uvést příjem zprávy, která informuje o provedeném obchodu za neodpovídající cenu. Za předpokladu korektního chování trhových systémů je zřejmé, že došlo k závažné chybě v trhovém konektoru či v algoritmu.

Ve výše uvedeném textu bylo popsáno zpracování zpráv, které byly přijaty z externích systémů. Komponenta `MessageManager` zpracovává podobným způsobem i zprávy, které byly vygenerovány algoritmem.

6.3.7 AlgoEngine

Jádro systému tvoří komponenta `AlgoEngine` (uprostřed obrázku 6.4). Jádro systému samo o sobě neobsahuje žádnou obchodovací strategii, o určení obchodovací strategie se starají obchodovací algoritmy. Algoritmy jsou v jádru systému zapouzdřeny ve formě pluginů, čímž je umožněno využití algo systému pro řešení mnoha problémů v oblasti algoritmického obchodovaní. Jádro systému pojme libovolné množství pluginů (pluginy musí mít unikátní identifikátory pro možnost korektního routování zpráv), což umožňuje běh více obchodovacích algoritmů v rámci jednoho algo systému. Lze tak vytvořit například algo systém specializovaný na arbitrage, kdy je v systému více verzí tohoto algoritmu. Jelikož jádro pracuje výhradně s daty v jediné interní reprezentaci (více v kapitole 6.5), mohou být obchodovací algoritmy využity opakovaně při komunikaci s různými typy burz.

Komponenta `AlgoEngine` využívá služeb ostatních komponent a zprostředkovává je algoritmům, jedná se především o:

- Rozhraní pro příjem klientských požadavků a pro zasílání reportů směrem ke klientským systémům (komponenta `ClientInterface`).
- Rozhraní pro zasílání požadavků na burzu a pro přijímání reportů od trhu (komponenta `MarketInterface`).
- Rozhraní pro příjem market dat (komponenta `MarketDataInterface`, respektive `MarketDataManager`).
- Aktuální stav všech orderů (komponenta `MessageManager`).

Kromě zprostředkování služeb od jiných komponent poskytuje jádro systému obchodovacím algoritmům další funkcionalitu. Pro potřeby algoritmů jsou v jádru systému implementovány třídy, které poskytují algoritmům podpůrnou funkcionalitu pro operace s obchodovacími zprávami a ordery (vytváření modifikačních zpráv pro order, vytváření reportovacích zpráv atd.). Tato funkcionalita by měla výrazně urychlit implementaci nových obchodovacích algoritmů.

Jádro systému podporuje možnost přepnutí orderu do manuálního zpracování (úlohu algoritmu převezme uživatel). I přes maximální úsilí při testování algoritmů nelze úplně vyloučit chybu při zpracování orderu (chyba je systémem zareportována). Proto je pro správce systému důležité mít možnost zasáhnout do činnosti algoritmu. Obchodovací algoritmy mohou podporovat různou úroveň spolupráce s uživatelem. Pro

všechny však platí, že v kritických situacích je potřeba algoritmus od orderů odstavit a vzniklou situaci vyřešit manuálně. Jako příklad lze uvést situaci, kdy algoritmus na trh umisťuje child ordery, jejichž celková kvantita přesahuje kvantitu parentského orderu. V takovém případě hrozí zobchodování většího počtu kusů cenného papíru než bylo v požadavku od klientského systému (hrozí vysoká ztráta). Tato chyba je detekována jádrem systému a je odeslán report vyškolenému administrátorovi. Ten přepne zpracování orderu do manuálního módu, zruší child ordery a klientskému systému zašle zprávu, která ruší parentský order. V čase, kdy administrátor opravuje order, mohl od burzy přijít report o provedeném obchodu (pro dosud nezrušený child order). Algoritmus je však od orderu odstaven a report klientskému systému musí zaslat administrátor. Administrátor má při manuálním zpracování orderu omezené možnosti a povoleny jsou mu pouze operace, které umožní korektní uzavření orderu.

Jádro systému kontroluje vstupní zprávy jak od klientských, tak od trhových systémů pomocí subkomponent `ClientGate` a `MarketGate` (na obrázku 6.4 lze tyto komponenty vidět v horní, respektive spodní části jádra systému - komponenty `AlgoEngine`). Účelem komponent je nevpustit do jádra nevalidní obchodovací zprávy, které nemohou být systémem zpracovány. Validní zprávy jsou aplikovány na úložiště orderů a předány ke zpracování obchodovacímu algoritmu.

Kompletní popis funkcionality jádra systému a návod, jak implementovat obchodovací algoritmus, lze nalézt v uživatelské příručce k systému, [7].

6.4 Propojení komponent

Všechny komponenty systému jsou zapouzdřeny v centrální třídě `AlgoMachine`. Tato třída komponenty propojuje a stará se o korektní komunikaci komponent. Z hlediska architektury systému je zajímavé zejména propojení komponent pro zpracování obchodních zpráv. Z obrázku 6.4 lze vidět posloupnost komponent (`ClientInterface`, `RoutingTable`, `ClientGate`, `AlgoEngine`, `MarketGate`, `RoutingTable` a `MarketInterface`), kterými prochází tok obchodovacích zpráv. Propojení mezi sousedními komponentami je vždy založeno na stejném principu: první komponenta v pořadí zasílá té druhé klientské zprávy a naopak druhá z komponent zasílá té první trhové zprávy (reporty). Na stejném principu je vždy založena komunikace klientského systému / komponenty s trhovým systémem / komponentou. Pokud se díváme na řetězec komponent v systému (obrázek 6.4, řetězec začíná v horní části), lze vidět, že pro každé dvě sousední komponenty platí, že první v pořadí je ve vzájemném vztahu klientskou komponentou, druhá pak komponentou trhovou. Proto byl vymyšlen obecný mechanismus pro propojení libovolného počtu komponent zpracovávajících obchodovací zprávy. Tento mechanismus umožňuje snadné rozšíření systému o další komponenty, které zpracovávají obchodovací zprávy (systém by mohl být rozšířen například o monitorovací komponentu). Mechanismus je založen na rozhraních, které jednotlivé komponenty musí implementovat:

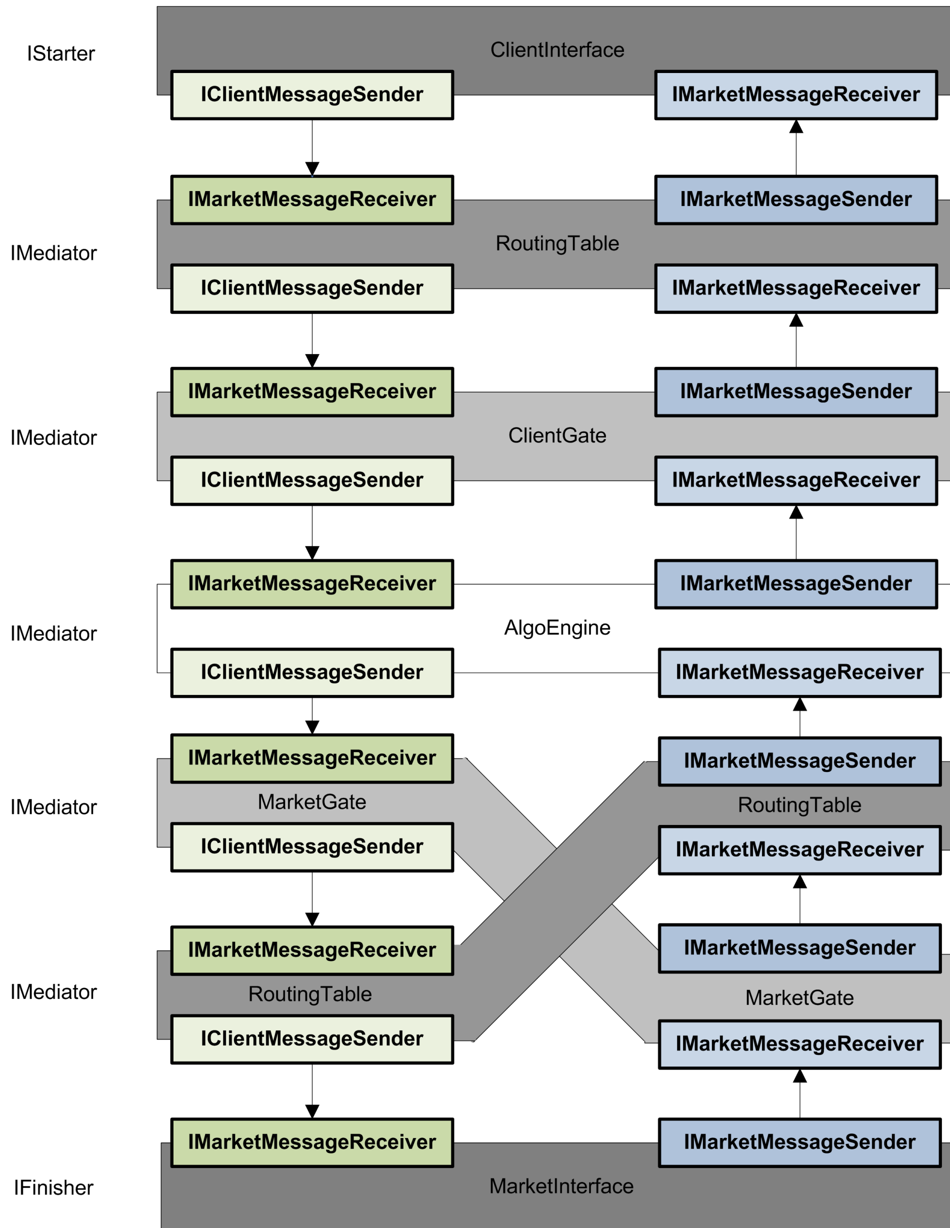
- `IClientMessageSender` - Rozhraní, které implementují komponenty odesílající klientské zprávy.
- `IMarketMessageSender` - Rozhraní, které implementují komponenty odesílající trhové zprávy.
- `IClientMessageReceiver` - Rozhraní, které implementují komponenty přijímající klientské zprávy.
- `IMarketMessageReceiver` - Rozhraní, které implementují komponenty přijímající trhové zprávy.
- `IStarter` - Sjednocení rozhraní `IClientMessageSender` a `IMarketMessageReceiver`. Toto rozhraní implementuje první komponenta v řetězci komponent.

- `IMediator` - Sjednocení rozhraní `IClientMessageSender` , `IMarketMessageReceiver`, `IClientMessageReceiver` a `IMarketMessageSender`. Toto rozhraní implementují komponenty uvnitř řetězce.
- `IFinisher` - Sjednocení rozhraní `IClientMessageReceiver` a `IMarketMessageSender`. Toto rozhraní implementuje poslední komponenta v řetězci komponent.

V následující ukázce kódu lze vidět propojení komponent systému:

```
ComponentConnector.ConnectComponents(
    clientInterfaceManager,
    new List<IMediator>() { algorithmChooser, clientGate,
        algoEngine, marketChooser, marketGate },
    new List<IMediator>() { algorithmChooser, clientGate,
        algoEngine, marketGate, marketChooser },
    marketInterfaceManager);
```

Do systému lze snadno zařadit novou komponentu pro zpracování obchodovacích zpráv přidáním na korektní místo v řetězci. V ukázce kódu lze vidět propojení komponent, které je využito přímo v algo systému: prvním argumentem je komponenta implementující rozhraní `IStarter` (komponenta pro komunikaci s klientskými systémy - `ClientInterface`), posledním argumentem je komponenta implementující rozhraní `IFinisher`(komponenta pro komunikaci s trhovými systémy - `MarketInterface`). Druhý a třetí argument jsou seznamy komponent, které implementují rozhraní `IMediator` (routovací komponenty a komponenty jádra systému). Toto rozdělení do dvou seznamů umožňuje rozdílné pořadí komponent při zpracování klientských a trhových zpráv. Na obrázku 6.7 lze vidět graficky znázorněné propojení jednotlivých komponent pomocí výše uvedených rozhraní, obrázek znázorňuje prohození pořadí komponent `MarketGate` a `RoutingTable`.



Obrázek 6.7: Propojení komponent zpracovávajících obchodní zprávy

6.5 Interní reprezentace dat

V této kapitole budou popsány datové struktury, které jsou využívány v systému. Interní reprezentace dat je důležitá zejména proto, že formát vstupu a výstupu systému může být prakticky libovolný, ale implementované obchodovací algoritmy musí být aplikovatelné pro kterýkoli z nich. Vstup a výstup systému je realizován pomocí konektorů (překladových pluginů), které vstupní data přeloží do interní reprezentace a předají je jádru systému. V této kapitole budou rozebrány výhody a nevýhody zvoleného řešení interní reprezentace dat v porovnání s možnými alternativami.

6.5.1 Zprávy

Zprávy, pomocí nichž systém komunikuje s klienty a trhy, jsou nejpoužívanější strukturou v systému. Proto bylo důležité vhodně zvolit jejich reprezentaci. Při volbě reprezentace bylo potřeba vzít v potaz následující aspekty:

1. Snadná manipulace s daty uloženými ve struktuře (zprávě). Tato vlastnost výrazně usnadní implementaci obchodovacích algoritmů.
2. Snadná rozšiřitelnost struktury o nová data. Zpráva musí být rozšiřitelná o nové datové položky bez nutnosti zásahu do jádra systému (serializace, překlady atd.).
3. Možnost připojit ke zprávě další libovolná data. Tím je umožněno zaslat algoritmům takřka libovolné informace, které mohou obsahovat například specifické parametry, s jakými má být order zpracován. Této funkcionality je využito v případě komunikace více algo systémů, popřípadě při komunikaci s interním systémem (například vzdálené klientské grafické rozhraní pro ovládání algoritmu).
4. Snadný překlad do protokolu FIX. Přestože protokol FIX netvoří jediný vstupní formát systému, navržené datové struktury musí být optimalizované právě pro překlad do tohoto protokolu. Protokol je používán klientskými systémy a je důležité, aby překlad do tohoto protokolu fungoval rychle a spolehlivě.
5. Možnost rychlé serializace a deserializace struktury. Tento aspekt je důležitý zejména pro přenos zpráv po síti (komunikace algo systémů) a ukládání zpráv pro docílení persistentnosti systému.

Jako nejpřímočařejší řešení se na první pohled jeví využití zpráv používaných v protokolu FIX (více v kapitole 3). Protokol je vyvíjen již přes 10 let a je využíván mnoha velkými společnostmi, zprávy tedy odpovídají současným požadavkům elektronického obchodování. Další výhodou tohoto řešení je volně dostupná knihovna QuickFIX (více v kapitole 3.4.1), která obsahuje implementaci všech zpráv používaných protokolem FIX ve verzích 4.0 až 4.4. Body 1 a 5 z výše uvedeného seznamu jsou u tohoto řešení splněny bez problémů. Bod 2 je řešitelný díky tzv. user tagům, které protokol i knihovna samotná podporuje (možnost doplnění zpráv o další datové položky). Bod 4, tedy překlad do protokolu FIX, je na první pohled splněn triviálně. Je však potřeba si uvědomit, že protokol FIX existuje ve více verzích a překlad mezi verzemi nemusí být úplně triviální (zejména překlad z nižší verze protokolu do vyšší). Jako největší problém se jeví bod 3, kdy neexistuje snadný způsob přiložit ke zprávě libovolná další data. Tato data by se musela rozložit do user tagů a navkládat do zprávy.

Od přímého použití FIX zpráv bylo upuštěno pro nesplnění požadavků 3 a 4. Jádro systému a obchodovací algoritmy budou stavěny pouze pro jedinou verzi interní reprezentace dat. Kdyby jádro systému podporovalo protokol FIX ve všech jeho verzích, byla by jeho implementace stejně jako implementace obchodovacích algoritmů daleko komplikovanější. Obchodovací algoritmy by musely ošetřovat sitauce, kdy jim ekvivalentní informace z trhu / od klienta přijde v různých formátech. Proto byl navržen interní formát zpráv vycházející přímo z protokolu FIX, který bude snadno převoditelný do různých verzí protokolu FIX. Tato interní reprezentace sjednocuje verze protokolu FIX (formát interní reprezentace vychází spíše z vyšších verzí protokolu, aby bylo možné zprávy v interním formátu převádět do všech verzí protokolu) a zaručuje jednotný formát dat, který je jádrem systému používán.

Interní obchodovací zprávy obsahují pouze ty položky, které jsou v zadavatelské firmě využívány, doplněny jsou pak o některá další pole, která protokol FIX nepodporuje (například datové položky, kterých bude

využívat monitorovací systém ve firmě). Protože se potřeby firmy v čase mění, bylo potřeba vymyslet způsob, jak umožnit rozšíření datových struktur tak, aby byl tento proces co nejméně komplikovaný, a tím se snížilo riziko vzniku chyb. Jednou z možností bylo zapouzdřit do tříd reprezentujících zprávy hash tabulku (slovník) a každému z polí přiřadit unikátní identifikátor, přes který by bylo možné získat a nastavit hodnotu daného pole. V tomto návrhu by stačila základní třída `Message`, která by obsahovala metody `GetField(int)` a `SetField(int, object)`. Takto pracovat s daty ve zprávě však není pohodlné. Tento přístup vyžaduje neustálé přetypovávání hodnot polí a umožňuje do zpráv vkládat i pole, která by v dané zprávě být neměla. Proto byla vytvořena sada tříd (obrázek 6.8) odpovídajících zprávám popsaných v kapitole 3, které mají pevný počet položek. Výhodou tohoto řešení je pohodlnější práce se zprávami. V následující ukázce kódu lze vidět jednoduché srovnání obou přístupů:

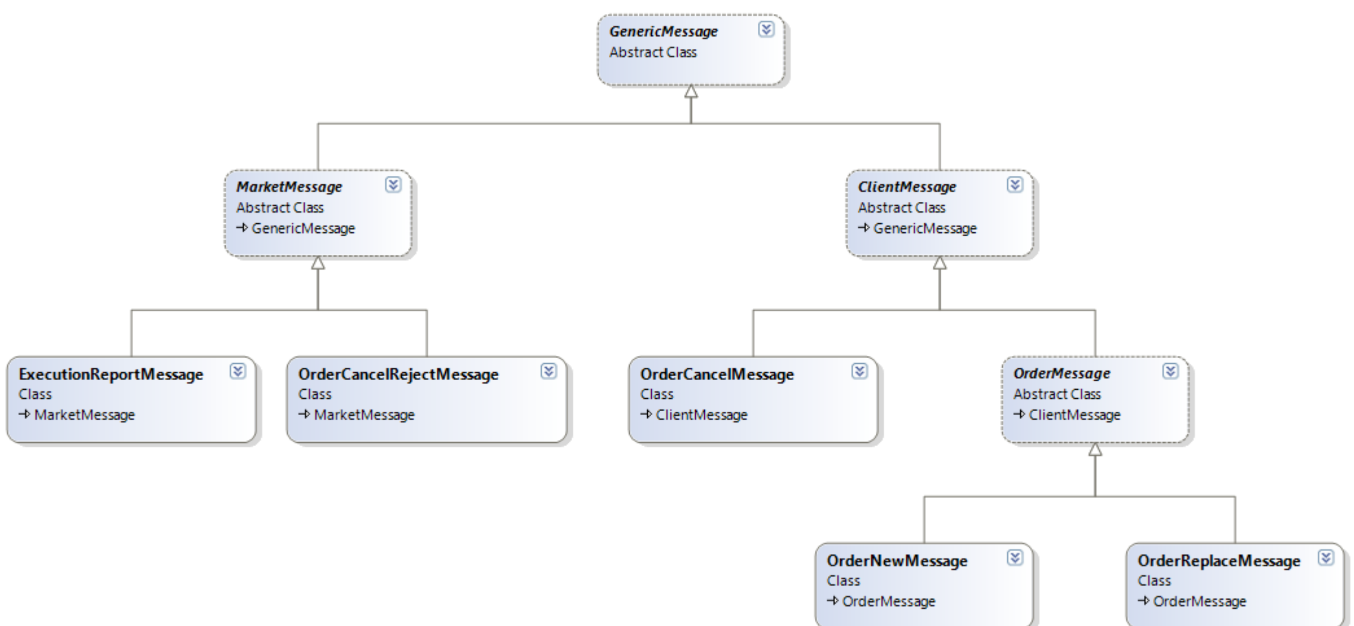
```
// First solution (message = encapsulated dictionary).
int quantity = (int)message.GetField(MessageFields.Quantity);

// Second solution (each data item in message has special property).
int quantity = message.Quantity;
```

Rozšiřitelnost zpráv je řešena pomocí atributů, které jsou jednotlivým polím zprávy přiřazeny. Tyto atributy přiřazují polím jednoznačný identifikátor a informaci, zda pole odpovídá některému z polí FIX zprávy. Úsek kódu třídy odpovídající zprávě `OrderNewMessage`:

```
public class OrderNewMessage : OrderMessage
{
  #region Properties

  [MessageField(Constants.Symbol)]
  [Category(StringConstants.FinancialData)]
  [Description(StringConstants.Symbol)]
  [ReadOnly(true)]
  public string Symbol
  {
    get { return symbol; }
    set { symbol = value; }
  }

  [MessageField(Constants.OrdType)]
  [Category(StringConstants.FinancialData)]
  [Description(StringConstants.Type)]
  [ReadOnly(true)]
  public OrderType Type
  {
    get { return type; }
    set { type = value; }
  }

  [MessageField(Constants.OrdQty)]
  [Category(StringConstants.FinancialData)]
  [Description(StringConstants.Quantity)]
  [ReadOnly(true)]
  public long Quantity
```

```
    {
      get { return quantity; }
      set { quantity = value; }
    }

    #endregion
  }
```

V kódu je důležitý zejména atribut `MessageField`, který přiřazuje poli jednoznačný celočíselný identifikátor (v případě, že je identifikátor kladný, jedná se o identifikátor pole z FIX protokolu). Po překladu knihovny je na základě těchto atributů automaticky vygenerován kód jiné knihovny, která umí provádět se zprávami operace, jako je překlad do protokolu FIX, serializace, deserializace atd. Tento kód je generován pomocí speciální překladové utility využívající reflexe ([20]). Utilita projde všechny neabstraktní třídy dědící od bázové třídy `GenericMessage` a vygeneruje pro ně příslušný kód. V případě, že je potřeba rozšířit některou ze zpráv o novou datovou položku, stačí tuto položku označit atributem a knihovnu přeložit. Překladová utilita automaticky aktualizuje kód tříd pro práci se zprávami. Tzn. překlad do protokolu FIX, přenos zpráv po síti atd. bude nadále fungovat bez potřeby dalších zásahů do kódu systému. Díky automaticky generovanému kódu (pro překlady a serializace) jsou splněny body 2, 4 a 5.



Obrázek 6.8: Interní reprezentace zpráv (diagram tříd)

Bázová třída `GenericMessage` obsahuje strukturu (hašovací tabulku), do které lze uložit libovolná další data, která jsou potřeba ke zprávě přibalit, čímž je splněn bod 3. Toto řešení tedy splňuje všechny podmínky, které byly na interní reprezentaci zpráv kladeny.

6.5.2 Market data

Struktury reprezentující market data jsou v systému využívány podobně často jako struktury reprezentující zprávy. Při návrhu reprezentace market dat však bylo potřeba zohlednit plánovaný vznik market data

serveru jako jednotného úložiště market dat ze všech trhů. Práce s market daty se v obou systémech bude lišit. Market data server bude stahovat market data z burz, ukládat je do databáze a následně je poskytovat dalším systémům. Pro tento systém není důležité, jestli ukládá cenu akcie či její jméno. Nepotřebuje tedy znát význam jednotlivých položek ve strukturách market dat. V algo systému je situace zcela odlišná. Zapouzdřené algoritmy musí znát sémantiku polí v market data strukturách. Proto byly navrženy 2 interní reprezentace market dat: tzv. raw market data objekty a business market data objekty.

6.5.2.1 Raw market data objekty

V této reprezentaci jsou market data tvořena posloupností hodnot otagovaných celočíselným identifikátorem (podobně jako v protokolu FIX, více v kapitole 3). Raw reprezentace není vhodná pro komplikovanější operace s market daty, které provádí algoritmy. Tento formát je však velmi vhodný pro přenos dat po síti. Formát je snadno serializovatelný do pole bytů a navíc je optimalizovaný pro aktualizace market dat. Při aktualizaci například ceny akcie není potřeba zasílat všechny informace, které se akcie týkají, ale stačí pouze odeslat novou cenu. Další výhodou raw formátu je jeho snadná rozšiřitelnost jak o nový typ market dat, tak i o možnost rozšíření stávajících objektů o nové položky.

Raw reprezentace market dat vychází z velké části z potřeb market data serveru (systém sloužící jako úložiště a poskytovatel market dat v zadavatelské firmě). Tato reprezentace byla navržena ve spolupráci se zástupci zadavatelské společnosti, kteří se podíleli na vývoji market data serveru.

6.5.2.2 Business market data objekty

Business market data objekty jsou optimalizovány pro použití při algoritmickém obchodování. Při jejich návrhu byl využit stejný princip jako u interní reprezentace zpráv (více v kapitole 6.5.1). Každý z business objektů má pevný počet datových položek, které jsou označeny atributy. Pomocí těchto atributů je při překladu systému automaticky vygenerován kód pro konverzi raw market dat do business market data objektů.

Kapitola 7

Pluginy systému

Z popisu architektury systému (kapitola 6) vyplývá, že systém nelze používat bez pluginů. Pluginy jsou právě tím, co z obecného algo systému (frameworku) tvoří systém pro řešení konkrétních problémů z oblasti algoritmického obchodování. Proto bylo potřeba navrhnout a implementovat základní sadu pluginů, které umožní nasazení systému v praxi. V příručce k systému ([7]) je věnována samostatná kapitola popisu implementace nových pluginů. Příručka obsahuje detailní popis funkcionality jednotlivých typů pluginů a měla by výrazně urychlit vývoj nových modulů systému.

7.1 Komunikace pomocí protokolu FIX

Z požadavků na systém (kapitola 2.3) vyplývá potřeba komunikace systému pomocí protokolu FIX (kapitola 3). Systém bude komunikovat protokolem FIX zejména s klientskými systémy, přesto byl vyvinut i konektor založený na protokolu FIX pro komunikaci s trhy. To umožní z algo systému vytvořit univerzální systém pro konverzi mezi verzemi protokolu FIX.

FIX-ové konektory jsou založeny na knihovně QuickFix (kapitola 3.4.1), což značně usnadnilo jejich vývoj. Knihovna poskytuje úplnou implementaci protokolu, v konektoru tak stačí implementovat překlad mezi interní reprezentací obchodovacích zpráv a reprezentací v protokolu FIX. Překlad je snadno realizovatelný díky vlastnostem interní reprezentace, u které byl při návrhu kladen důraz na snadnou konverzi do / z protokolu FIX (kapitola 6.5.1).

Konektory poskytují možnost administrace jednotlivých FIX session podobně jako systém CameronFIX (kapitola 3.4.3). Administrátor systému tak má možnost dočasně přerušit komunikaci se vzdáleným systémem (například pro potřebu údržby systému), případně synchronizovat sekvenční čísla v případě poškození dané session.

7.2 Komunikace mezi více algo systémy

Při návrhu systému nebylo cílem vytvořit jedinou centrální aplikaci pro algoritmické obchodování, nýbrž systém, jehož více instancí lze pomocí pluginů nastavit a spojit v jeden distribuovaný systém, kde každá z instancí systému má specifický úkol. Příklad distribuovaného algo systému lze nalézt v kapitole 6.1.2. V kapitole 6.4 byl popsán způsob komunikace mezi komponentami v systému. Tok zpráv od klientského systému

k trhovému je realizován pomocí řetězce komponent, kde komunikace mezi sousedními komponentami je vždy založena na stejném principu. Po propojení více algo systémů je tak tento řetězec pouze prodloužen, přičemž část komponent zpracovává zprávy v rámci jiného procesu (tento proces typicky poběží na jiném stroji). Pro spojení dvou algo systémů stačí propojit komponentu `MarketInterface` z prvního systému s komponentou `ClientInterface` druhého systému. Propojení je docíleno pomocí jednoduchých pluginů, které fungují v rámci výše zmíněných komponent. Tyto pluginy pouze serializují / deserializují interní reprezentaci zpráv a vyměňují si serializovanou podobu dat po síti (případně přes named pipes).

Pro implementaci konektorů zajišťujících výměnu dat v interní reprezentaci byla důležitá volba middleware. Důraz byl opět kladen na co nejmenší zpomalení zpracování zprávy při jejím přenosu mezi více algo systémy. Analyzovány byly technologie .NET remoting ([9]), WCF ([11]) a MSMQ ([10]). Tyto technologie jsou však poměrně komplexní v porovnání s nízkými nároky, které jsou kladeny na middleware použitý při přenosu dat mezi algo systémy. Díky snadné serializaci a deserializaci interní reprezentace zpráv stačí tyto zprávy zasílat přes TCP/IP (případně named pipes) bez žádných dalších operací, které proces zpracování zprávy zpomalují. Proto bylo rozhodnuto pro implementaci vlastního jednoduchého middleware. Implementovaný middleware podporuje pouze přenos serializovaných dat, žádná přidaná funkcionalita nebyla implementována.

7.3 Příjem market dat

Market data jsou systémem přijímána pomocí jednoduchého pluginu, který odebírá data z market data serveru (úložiště market dat v zadavatelské firmě) v interní reprezentaci (kapitola 6.5.2.1). Interní reprezentace market dat podobně jako interní reprezentace obchodovacích zpráv je snadno serializovatelná, proto byl pro komunikaci s market data serverem využit stejný middleware jako pro propojení více instancí algo systému (kapitola 7.2).

7.4 Obchodovací algoritmy

Cílem práce nebylo vyvinout konkrétní obchodovací algoritmus, ale systém, který umožňuje algoritmické obchodování obecně. Proto nebyl jako součást práce k systému implementován plugin realizující konkrétní obchodovací strategii. Implementovány byly tři pluginy, které lze využít jako základ při tvorbě konkrétního obchodovacího algoritmu:

- `SkeletAlgorithm` - Algoritmus obsahuje základní implementaci komunikace s jádrem systému, neobsahuje žádné zpracování příchozích zpráv. Tento "algoritmus" lze použít jako základ prakticky pro všechny typy obchodovacích algoritmů.
- `ThroughFlowAlgorithm` - Rozšířená verze algoritmu `SkeletAlgorithm`, kdy příchozí zprávy jsou jen předány další komponentě v řetězci (tzn. klientské požadavky jsou přímo zasílány na burzu, trhové reporty jsou přímo zasílány klientským systémům). Algoritmus lze využít jako základ pro jednoduché algoritmy, které modifikují zprávy od klienta směrem na trh.
- `DrivenFlowAlgorithm` - Rozšířená verze algoritmu `ThroughFlowAlgorithm`. Algoritmus nabízí základní operace pro filtrování a úpravu průchozích zpráv.

Účelem výše zmíněných algoritmů je usnadnit implementaci nových obchodovacích strategií. Algoritmy poskytují obecnou funkcionalitu, která může být využita pro řešení více problémů.

7.5 Testovací pluginy

Pro dokonalé otestování jádra systému byla vyvinuta speciální sada pluginů. Tyto pluginy se vloží do jednotlivých komponent systému a po jeho nastartování spustí sadu testů, které mají ověřit korektnost chování systému. Jednotlivé pluginy při testování spolupracují a ověřují korektnost chování systému (jedná se o speciální formu unit testingu). Pomocí těchto pluginů lze minimalizovat pravděpodobnost vzniku chyby po zásahu do kódu jádra systému.

Kapitola 8

Srovnání s existujícími platformami

V současnosti existuje více systémů, které podporují nebo jsou přímo určené pro algoritmické obchodování na burzách cenných papírů. Tyto systémy jsou typicky extrémně nákladné a jejich autoři je nejsou ochotni poskytovat v trial verzi. Nainstalován a otestován byl systém Marketcetera ([3]). Z oficiálních zdrojů byly získány informace o systémech Apama Algorithmic Trading Platform ([13]), Orc Liquidator ([14]) a GL Stream for Algorithmic Trading ([15]).

Prvním analyzovaným systémem je open source platforma Marketcetera. Platforma sestává z více komponent. Hlavní součástí je server ORS (Order Routing System), jehož funkcionalitu využívají ostatní komponenty systému pro zasílání a příjem obchodovacích zpráv. Na ORS je napojena komponenta Photon, která představuje uživatelsky přívětivé rozhraní pro zasílání orderů a monitorování dění na trhu. Photon podporuje implementaci jednoduchých obchodovacích strategií pomocí jazyka Ruby. Obchodovací strategie lze vyvíjet přímo pomocí uživatelského rozhraní komponenty Photon. Další komponentou je Strategy Agent. Tato komponenta je obdobou vyvinutého Algo Engine systému. Narozdíl od nástroje Photon je určena pro běh komplexních obchodovacích strategií. Obchodovací algoritmy jsou v komponentě zapouzdřeny ve formě skriptů v jazyce Ruby. Autory systému jsou poskytovány instalační balíčky pro operační systémy Windows i Linux. Systém lze snadno zprovoznit a napojit na trhové i klientské systémy. Komunikační rozhraní systému je založeno na protokolu FIX. Využita je open source knihovna QuickFIX (kapitola 3.4.1).

Zástupcem komerčního software, který je určen pro algoritmické obchodování, je systém Apama Algorithmic Trading Platform. Tato platforma se vyznačuje nástrojem Progress Apama Event Modeler, který umožňuje modelovat obchodovací strategie pomocí grafického rozhraní. Nové obchodovací strategie lze do systému zadávat také přímo pomocí kódu programovacího jazyka. Apama podporuje jazyk vyvinutý přímo touto společností nazvaný Event Programing Language a také jazyk Java. Součástí platformy je nástroj pro testování obchodovacích strategií na historických datech a vestavěná sada základních algoritmů. Komunikační rozhraní systému je založeno na protokolu FIX.

Dalším analyzovaným systémem je platforma Orc Liquidator. Společnost Orc, která tento produkt nabízí, poskytuje nativní konektory pro více než sto burz. Na těchto konektorech je postavena komunikace s burzami. Konektory jsou však zpoplatněny zvlášť, tudíž náklady na provoz systému rostou s každým připojením na novou burzu. Obchodovací algoritmy lze do systému přidat ve formě modulů v jazyce Java.

GL Tactics je systém pro algoritmické obchodování poskytovaný společností Sungard. Tento systém je svou charakteristikou velice podobný platformě Orc Liquidator, kdy společnost poskytuje nativní konektory pro více než sto čtyřicet burz cenných papírů, pomocí nichž systém s trhy komunikuje. Systém obsahuje vestavěné obchodovací strategie například pro Smart Order Routing (zasílání orderů na trh s nejlepší cenou)

nebo VWAP. Obchodovací strategie lze do systému přidávat pomocí skriptů v jazyce Java. Pro dosažení maximálního výkonu lze tyto skripty převést automaticky do kódu jazyka C++.

Porovnání základních charakteristik výše popsaných systémů s platformou Algo Engine vyvinutou v rámci této práce lze nalézt v tabulce 8.1.

Platforma	**Komunikační rozhraní**	**Implementace vlastních algoritmů**	**Specifické vlastnosti**
Marketcetera	Protokol FIX	Skripty v jazyce Ruby	Nulové náklady, vlastní vývojové prostředí
Apama Algorithmic Trading Platform	Protokol FIX, vestavěná podpora pro příjem market dat ze systémů Reuters, ActivFinancial, ...	Skripty v jazyce EPL / Java	Grafické rozhraní pro modelování obchodovacích strategií, možnost testovat obchodovací algoritmy na historických datech, vestavěné obchodovací algoritmy
Orc Liquidator	Protokol FIX pro komunikaci s klientskými systémy, nativní konektory pro komunikaci s trhy	Moduly v jazyce Java	Nativní konektory pro komunikaci s více než 100 burzami
GL Tactics	Protokol FIX pro komunikaci s klientskými systémy, nativní konektory pro komunikaci s trhy	Skripty v jazyce Java	Nativní konektory pro komunikaci s více než 140 burzami, vestavěné obchodovací strategie
Algo Engine	Nezávislé na konkrétním protokolu	Pluginy vytvořené v jazycích rodiny .NET	Nezávislost na komunikačním protokolu s burzovními i klientskými systémy

Tabulka 8.1: Srovnání obchodovacích platforem

Pro finanční instituce, které chtějí provozovat algoritmické obchodování, jsou velice důležité náklady na jednotlivá řešení. Tabulka 8.2 zobrazuje cenové relace, ve kterých se jednotlivé platformy nacházejí. Pro komerční software je v ceně zahrnuta základní podpora systému. Pro systémy Orc Liquidator a GL Tactics nejsou v ceně zahrnuty konektory k burzám. Cena za připojení k jednomu trhu se pohybuje okolo deseti tisíc Euro za rok.

Analyzované platformy nabízejí oproti vyvinutému systému Algo Engine mnoho přidané funkcionality: vlastní nástroje pro vývoj obchodovacích strategií, vestavěné obchodovací algoritmy, ... Vyvinutý Algo Engine systém má však pro zadavatelskou firmu oproti těmto systémům výhodu v nezávislosti na komunikačním protokolu s klientskými i trhovými systémy. Zadavatelská společnost se tak může kdykoli rozhodnout rozšířit portfolio podporovaných burz, aniž by byla závislá na dodavateli obchodovací platformy. Společnosti Orc (systém Orc Liquidator) a Sungard (systém GL Tactics) sice ke svým systémům nabízí implementovanou komunikaci s mnoha trhy, zdaleka však nepokrývají všechny. Další výhodou Algo

Platforma	**Licence**	**Cena licence na jeden rok**
Marketcetera	Open Source	0
Apama Algorithmic Trading Platform	Komerční	100 000 Euro
Orc Liquidator	Komerční	25 000 Euro
GL Tactics	Komerční	50 000 Euro

Tabulka 8.2: Licence a náklady

Engine systému je možnost implementace algoritmů a překladových pluginů v jazyku C# (vývojáři v zadavatelské firmě disponují znalostmi právě tohoto jazyka a platformy .NET). V neposlední řadě pak stojí náklady na systém, které jsou u komerčně vyvíjených systémů velmi vysoké.

Kapitola 9

Závěr

Nejdůležitějším faktorem pro vytvoření algo systému použitelného v praxi bylo důkladně nastudovat problematiku elektronického obchodování a analyzovat prostředí a potřeby zadavatelské firmy (existující infrastrukturu pro elektronické obchodování, komunikační protokoly atd.). Při práci na projektu bylo důležité neustále komunikovat a konzultovat návrhy řešení se zástupci zadavatelské společnosti. Tímto bych chtěl ocenit zejména přístup členů oddělení pro elektronické obchodování, kteří byli vždy ochotni zodpovědět mé otázky a pomoci hlavně v počátečních fázích práce na systému.

9.1 Splnění cílů

Cíle práce byly popsány v kapitole 1.4. Hlavním cílem bylo vytvořit obchodovací platformu, která umožňuje algoritmické obchodování na burzách cenných papírů. Tento cíl byl splněn, o čemž svědčí využití systému v praxi (kapitola 9.2). Kapitola 1.4 obsahuje také seznam dílčích cílů (nároků na systém), které bylo potřeba při práci na systému zohledňovat. Podívejme se, jak se podařilo dílčí cíle naplnit:

- **Flexibilita** - Systém je díky své architektuře velice modulární (komunikační rozhraní je založeno na překladových pluginech), je nezávislý na vstupním formátu dat při komunikaci s klientskými i trhovými systémy. Díky jediné interní reprezentaci obchodovacích dat, se kterou pracuje jádro systému, je možné vyvinuté obchodovací algoritmy využívat pro komunikaci s různými burzami (i když tyto burzy komunikují různým způsobem). Obchodovací algoritmy jsou zapouzdřeny jádrem systému ve formě pluginů, což umožňuje snadné zařazení nového algoritmu do systému.
- **Škálovatelnost** - Díky možnosti propojení více instancí systému lze snadno rozložit celkovou zátěž na systém.
- **Nízká latence** - Latence je výrazně ovlivněna časem potřebným pro docílení persistentnosti systému (ukládání dat, ze kterých lze obnovit stav systému po jeho restartu). Proto byly důkladně analyzovány jednotlivé typy persistentních úložišť a navržen způsob docílení persistentnosti s minimálním počtem přístupů k úložišti.

  Dalším faktorem, který výrazně ovlivňuje rychlost zpracování dat systémem, je paralelní zpracování. Při návrhu systému byly analyzovány různé přístupy k paralelnímu zpracování tak, aby systém fungoval co nejefektivněji.

Na latenci systému má zásadní vliv také jeho architektura. Proto byla při návrhu systému vyloučena možnost využití starších řešení, která jsou v současnosti v zadavatelské firmě využívána. Tyto existující systémy nejsou dostatečně optimalizovány pro nízkou latenci a nebylo proto vhodné je použít při implementaci algo systému.

Celkový čas potřebný pro zpracování jedné obchodovací zprávy systémem lze měřit ve stovkách mikrosekund, což odpovídá současným nárokům zadavatelské firmy. Přesto lze latenci snížit, některé možnosti snížení latence jsou popsány v kapitole 9.3.

- **Stabilita** - Testování systému bylo prováděno ve spolupráci se zaměstnanci zadavatelské firmy (odladění systému zabralo týdny práce). Pro zvýšení stability systému byly vyvinuty speciální testovací pluginy, které testují korektní chování jádra systému. Po zásahu do jádra systému tak stačí spustit systém s testovacími pluginy, které s velkou pravděpodobností vyloučí chyby v jádře systému.

9.2 Nasazení systému v praxi

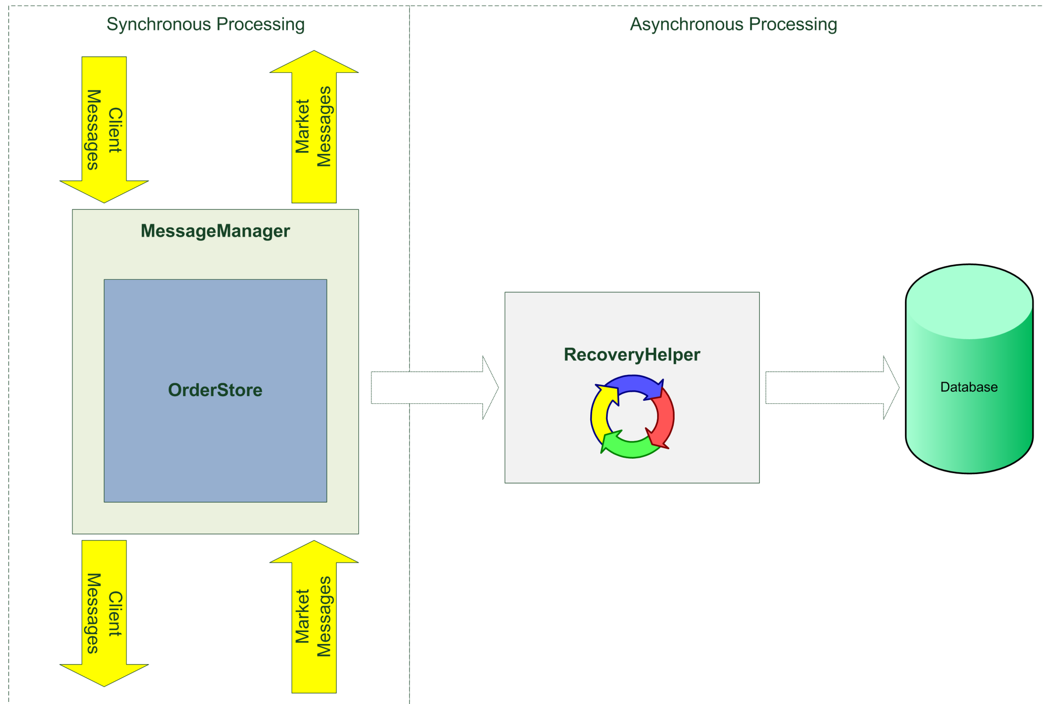
Systém je v současnosti ve fázi příprav pro nasazení do produkčního prostředí. Již několik měsíců je nasazen v testovacím prostředí několika burz. Zadavatelská firma již připravila řadu obchodovacích strategií, které jsou postaveny na vyvinutém algo systému a v nejbližších měsících by měly být nasazeny do praxe. Systém byl již experimentálně spuštěn v produkčním prostředí Burzy cenných papírů Praha. Testována byla obchodovací strategie poskytnutá od externí firmy. Strategie byla vyvinuta na základě analýzy dat z burzy za posledního půl roku s cílem dosáhnout u orderů ceny VWAP. Algo systém v produkčním prostředí zobchodoval akcie v hodnotě několika milionů korun, což je jen zlomek objemu, který by měl být zobchodován denně při plném nasazení. Přesto běh systému v produkčním prostředí prokázal jeho připravenost pro využití v praxi.

9.3 Možná vylepšení systému

Systém v současném stavu odpovídá nárokům zadavatelské firmy, přesto se do budoucnosti počítá s dalšími úpravami a vylepšeními. V této kapitole jsou stručně popsána některá vylepšení, která již nebylo možno v rámci rozsahu práce realizovat, i když by vedla k vyšší kvalitě systému.

9.3.1 Urychlení procesu obnovy systému

V kapitole 4 věnované persistentnosti systému byl popsán způsob procesu obnovení systému, kdy stav orderů je rekonstruován ze zpráv, které byly systémem zpracovány (tento proces může být v závislosti na počtu zpracovaných zpráv poměrně zdlouhavý). Možným vylepšením, které by mohlo být v budoucnosti v systému implementováno, je nasazení speciální komponenty `RecoveryHelper` (obrázek 9.1). Tato komponenta poběží asynchronně ve speciálním vlákně a bude v databázi držet stav úložiště orderů. Při přerušení systému nemusí databáze obsahovat aktuální stav všech orderů, protože modifikování probíhá asynchronně. Stačí si však držet identifikátor poslední zprávy, která stav databáze ovlivnila. Po startu systému se úložiště načte z databáze a aplikují se na něj zprávy, které dosud nebyly komponentou `RecoveryHelper` zpracovány.



Obrázek 9.1: Komponenta RecoveryHelper

9.3.2 Zefektivnění serializace dat

V kapitole 4.3 byl popsán vlastní způsob serializace obchodovacích zpráv (vyloučeno bylo použití jak XML serializace, tak binární serializace, kterou poskytuje .NET framework). Cílem serializace bylo co nejvíce snížit velikost serializované podoby dat a tím snížit čas potřebný pro uložení zprávy do persistentního úložiště. Možným způsobem, jak velikost dat ukládaných do persistentního úložiště ještě snížit, je využití principů použitých v protokolu FAST (více v kapitole 3.3). Do persistentního úložiště by se vždy ukládal pouze takový obsah zprávy, který nese novou informaci. Například u zprávy nesoucí informaci o provedeném obchodu by byly uloženy pouze nové informace (cena a kvantita provedeného obchodu). Při využití tohoto řešení bude možné v budoucnu proces ukládání obchodovacích zpráv ještě zrychlit.

Dodatek A

Seznam použitých pojmů

Většina pojmů, které se v oblasti elektronického obchodování běžně používají, pochází z angličtiny a nemají ustálený ekvivalent v českém jazyce. Proto bylo na mnoha místech použito anglických výrazů namísto uměle přeložených českých formulací. Seznam použitých výrazů společně s popisky:

Algo

Zkrácený výraz pro stroj umožňující algoritmické obchodování na burzách cenných papírů.

Order

Příkaz na prodej či koupi cenného papíru.

OMS

Zkratka pro "Order Management System". Jedná se o systém, který využívají makléři pro správu klientských orderů (objednávek).

DMA

Zkratka pro "Direct Market Access". Jedná se o službu, kterou poskytují brokerské společnosti, kdy jsou klientské ordery přímo zasílány na burzu.

EDA

Zkratka pro "Execution Desk Access". Jedná se o službu, kterou poskytují brokerské společnosti, kdy jsou klientské ordery spravovány makléři. Tato služba se mnohdy označuje jako "Care" - starat se.

Market Data

Data popisující aktuální stav burzy cenných papírů. Market data jsou poskytována naprostou většinou burz a jsou nutným předpokladem pro algoritmické obchodování (bez znalosti aktuálních dat nelze aplikovat prakticky žádnou strategii). Market data jsou typicky zasílána ve formě snapshotu a jeho následných aktualizací v čase.

VWAP

Z anglického Volume-Weighted Average Price, tedy průměrná cena cenného papíru vážená objemem obchodů. Lze vypočíst pomocí vzorce:

$$P_{\mathrm{VWAP}} = \frac{\sum_j P_j \cdot Q_j}{\sum_j Q_j}$$

P_j - cena j-tého obchodu.

Q_j - objem j-tého obchodu.

Arbitrage

Obchodovací algoritmus, který využívá rozdílné ceny cenného papíru, který se obchoduje na více burzách. Tento pojem je také někdy použit pro obchodovací algoritmus, který využívá rozdílnosti cen akcií podobně zaměřených firem (např. Cola a Pepsi) a předpokládá, že tyto ceny se vždy po určitém čase "sjednotí". Obchodovací strategii Arbitrage je věnována kapitola v [8].

Dodatek B

Seznam použité literatury

[1] *Algorithmic Trading*, http://en.wikipedia.org/wiki/Algorithmic_Trading_Platforms.

[2] *Infolect system*, http://www.microsoft.com/uk/getthefacts/lse.mspx.

[3] *The Marketcetera Trading Platform*, http://www.marketcetera.com/.

[4] *FIXForge FIX Engines*, http://www.onixs.biz/FIXEngines/.

[5] David Carmona, *Programming the Thread Pool in the .NET Framework*, 2002, http://msdn.microsoft.com/en-us/library/ms973903.aspx.

[6] *Smart Thread Pool*, http://www.codeplex.com/smartthreadpool.

[7] Jaromír Šatánek, *Algo Engine User Guide*, 2009.

[8] Yogesh Shetty a Samir Jayaswall, *Practical .NET for Financial Markets*, 2006.

[9] *.NET Remoting Overview*, http://msdn.microsoft.com/en-us/library/kwdt6w2k(VS.71).aspx.

[10] *Microsoft Message Queuing*, http://www.microsoft.com/windowsserver2003/technologies/msmq/default.mspx.

[11] *Windows Communication Foundation*, http://msdn.microsoft.com/en-us/netframework/aa663324.aspx.

[12] *FIX Protocol*, http://www.fixprotocol.org.

[13] *Apama Algorithmic Trading Platform*, http://www.progress.com/apama/products/apama_algo/index.ssp.

[14] *Orc Liquidator*, http://www.orcsoftware.com/Solutions/Orc-Trading/Orc-Algorithmic-Trading/.

[15] *GL STREAM FOR ALGORITHMIC TRADING*, http://www.sungard.com/financialsystems/products/glstreamforalgorithmictrading.aspx.

[16] *FINANCIAL INFORMATION EXCHANGE PROTOCOL (FIX) Version 4.2 with Errata 20010501*, 2001.

[17] David Rosenborg, *FAST Specification*, 2006.

[18] *QuickFIX Engine*, http://www.quickfixengine.org/quickfix/doc/html/index.html, 2006.

[19] *CameronFIX Version 6.2 Documentation*, 2008.

[20] Jesse Liberty, *Programming C#: Attributes and Reflection*, http://www.ondotnet.com/pub/a/dotnet/excerpt/prog_csharp_ch18/.

[21] *XML and SOAP Serialization*, http://msdn.microsoft.com/en-us/library/90c86ass(VS.71).aspx.

[22] *Binary Serialization*, http://msdn.microsoft.com/en-us/library/72hyey7b(VS.71).aspx.

Dodatek C

Obsah přiloženého CD

- `bin/algo` - Instalační adresář systému Algo Engine (návod k instalaci je popsán v uživatelské příručce).
- `bin/devel_kit` - Instalační adresář sady nástrojů pro usnadnění vývoje pluginů (návod k instalaci je popsán v uživatelské přiručce).
- `bin/plugins` - Základní sada pluginů pro systém Algo Engine.
- `docs/manual` - Uživatelská přiručka k systému.
- `docs/api` - Dokumentace ke zdrojovým kódům.
- `src/algo` - Zdrojové kódy systému Algo Engine.
- `src/tests` - Zdrojové kódy testů, které byly provedeny v rámci práce.