VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

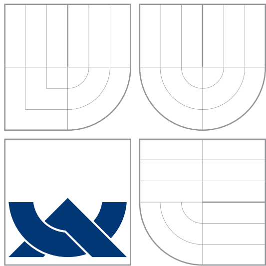
SYSTÉM PRO ZASÍLÁNÍ TEXTOVÝCH ZPRÁV SERVEROVÁ ČÁST

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

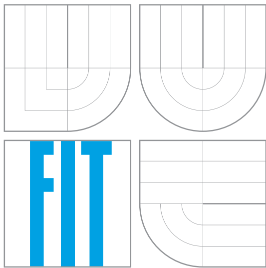
AUTOR PRÁCE
AUTHOR

MAREK GACH

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

SYSTÉM PRO ZASÍLÁNÍ TEXTOVÝCH ZPRÁV SERVEROVÁ ČÁST

INSTANT MESSENGING SYSTEM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — MAREK GACH
AUTHOR

VEDOUCÍ PRÁCE — Ing. JAROSLAV RÁB
SUPERVISOR

BRNO 2007

Zadání bakalářské práce/5823/2006/xgachm00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2006/2007

Zadání bakalářské práce

Řešitel: **Gach Marek**

Obor: Informační technologie

Téma: **Systém pro zasílání textových zpráv - serverová část**

Kategorie: Počítačové sítě

Pokyny:

1. Seznamte se s postupy navrhování aplikačních síťových protokolů, aplikací typu klient-server a peer-to-peer a bezpečností síťové komunikace.
2. <!--StartFragment -->Navrhněte architekturu systému pro zasílaní zpráv s ohledem na bezpečnost komunikace, přenos zpráv a souborů, klienti využívající privátní adresy, vyhledávání v archivu zpráv. Navrhněte aplikační síťový protokol, který bude vyhovovat požadavkům na systém.
3. Zvolte vhodné implementační prostředí a navržený model implementujte. Zaměřte se na implemantaci serverové části.
4. Proveďte testování systému a zhodnoďte dosažené výsledky.
5. Diskutujte další možnost pokračování projektu.

Literatura:

- W. R. Stevens: Unix network programming (3. ed), Volume 1, Addison-Wesley, 2004.

Při obhajobě semestrální části projektu je požadováno:

- Body 1. až 2.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním paměťovém médiu (disketa, CD-ROM), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Ráb Jaroslav, Ing.**, UIFS FIT VUT

Datum zadání: 1. listopadu 2006

Datum odevzdání: 15. května 2007

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

doc. Ing. Jaroslav Zendulka, CSc.
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. **Pan**

   Jméno a příjmení: **Marek Gach**
   Id studenta: 84396
   Bytem: Sídliště ONV 667/10, 737 01 Český Těšín
   Narozen: 18. 07. 1985, Český Těšín
   (dále jen "autor")

a

2. **Vysoké učení technické v Brně**

   Fakulta informačních technologií
   se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
   jejímž jménem jedná na základě písemného pověření děkanem fakulty:

   ....

   (dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
   bakalářská práce

Název VŠKP: Systém pro zasílání textových zpráv - serverová část
Vedoucí/školitel VŠKP: Ráb Jaroslav, Ing.
Ústav: Ústav informačních systémů
Datum obhajoby VŠKP: ....

VŠKP odevzdal autor nabyvateli v:
tištěné formě počet exemplářů: 1
elektronické formě počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:
   - ☐ ihned po uzavření této smlouvy
   - ☐ 1 rok po uzavření této smlouvy
   - ☐ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☐ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: ....

....	....
Nabyvatel	Autor

Abstrakt

Dnešní svět serverových aplikací je velmi dynamicky se rozvíjejícím odvětvím IT. S nástupem vícejádrových procesorů dobře navrhnuté paralelní aplikace získávají na výkonu. Tato práce se snaží nastínit základy vytváření a synchronizace vícevláknových aplikací. Tyto principy se pokouším aplikovat při tvorbě paralelního jádra univerzálního serveru. Nad tímto je poté definována sémantika komunikačního protokolu pro zasílání textových zpráv.

Klíčová slova

wxWidgets, ODBC, XML, klient, server, WxWidgets, C++, MySql, ODBC, thread pool, protokol

Abstract

Present world of server applications is dynamically developing branch of IT. With multicore CPU appearance well designed multithreading applications gain additional performance. This thesis is trying to sketch out some basis of multithreaded application creation and synchronisation. Usage of this principles will help me to develop general-purpose parallel server core. Whole semantics of communication protocol for message sending is based on that core.

Keywords

wxWidgets, ODBC, XML, client, server, WxWidgets, C++, MySql, ODBC, thread pool, protocol

Citace

Marek Gach: Systém pro zasílání textových zpráv
serverová část, bakalářská práce, Brno, FIT VUT v Brně, 2007

Systém pro zasílání textových zpráv
serverová část

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing.Jaroslava Rába. Další informace, rady a připomínky mi poskytl spolupracovník Jan Fiedor. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . .
Marek Gach
15. května 2007

Poděkování

Děkuji tímto vedoucímu své bakalářské práce panu Ing.Jaroslavu Rábovi za cenné rady a připomínky, které přispěly ke zkvalitnění tohoto díla. Poděkování patří také Janu Fiedorovi za výbornou spolupráci a tvorbu klientské části tohoto projektu.

© Marek Gach, 2007.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **3**

2 Teoretický rozbor **4**
- 2.1 Koncepce serverových aplikací . . . 4
  - 2.1.1 Architektura klient–server . . . 4
  - 2.1.2 Třívrstvá architektura klient–server . . . 6
  - 2.1.3 Přístup k datové vrstvě pomocí ODBC . . . 7
- 2.2 Principy paralelních systémů . . . 9
  - 2.2.1 Paralelní programování . . . 9
  - 2.2.2 Rychlost paralelních řešení . . . 10
  - 2.2.3 Modely paralelního programování . . . 10
  - 2.2.4 Synchronizační techniky . . . 11
- 2.3 Paralelní exekuce . . . 12
  - 2.3.1 Možné přístupy . . . 12
  - 2.3.2 Thread pool . . . 13

3 Návrh řešení **16**
- 3.1 Formulace úlohy . . . 16
- 3.2 Určení částí systému . . . 17
  - 3.2.1 Jádro . . . 17
  - 3.2.2 Protokol . . . 18
  - 3.2.3 Elementy datové vrstvy . . . 19

4 Realizace **20**
- 4.1 Implementační prostředí . . . 20
- 4.2 Uživatelem definované funkce . . . 20
  - 4.2.1 Thread pool . . . 20
  - 4.2.2 Vykonávání uživatelských funkcí . . . 23
- 4.3 Operace se schránkami . . . 25
  - 4.3.1 Správa spojení . . . 25
  - 4.3.2 Registr připojených uživatelů . . . 26
  - 4.3.3 Posílání a příjem . . . 27
  - 4.3.4 Struktura jádra . . . 27

5 Výstavba komunikačního protokolu nad jádrem **29**
- 5.1 Protokol . . . 29
  - 5.1.1 Konkrétní protokol . . . 29
- 5.2 Databáze . . . 31

5.2.1 Implementace databázových operací 31
5.2.2 Struktura tabulek 32
5.3 Posílání souborů 33
5.4 Kódování 35
5.5 Zprávy 35
5.5.1 Historie zpráv 36
5.6 Šifrování 36

6 Závěr a zhodnocení **37**
6.1 Směr dalšího vývoje 37
6.2 Závěr 37

Kapitola 1

Úvod

Cílem této práce je navrhnout systém pro zasílání textových zpráv a implementovat k tomuto systému serverovou aplikaci.

Měla by být umožněna komunikace pomocí textových zpráv, zpětné zobrazení všech odeslaných a přijatých zpráv pomocí historie a přenos souborů mezi uživateli za jakékoliv situace. Také by se nemělo zapomínat na bezpečnost komunikace.

Následující kapitola má za úkol přiblížit teoretický základ řešeného problému. Seznamuje se základními principy architektury klient–server a paralelní exekuce. Kapitola *Návrh řešení* formuluje zadání úlohy a nastiňuje možná řešení. V následující kapitole nazvané *Realizace* je popsán samotný způsob implementace serverového jádra. Nad tímto jádrem je v kapitole *Výstavba komunikačního protokolu nad jádrem* popsána implementace protokolu a částí s ním spojených. Poslední kapitola obsahuje shrnutí celé práce a možné kroky dalšího vývoje.

Kapitola 2

Teoretický rozbor

2.1 Koncepce serverových aplikací

Existuje mnoho přístupů k tvorbě klient–server aplikací. Následující text přibližuje některé architektury a techniky právě spojené s tvorbou klient–server aplikací. Vybrat vhodný přístup řešení daného problému je občas velmi obtížné. A většina výsledných řešení je kompromisem mezi přístupy popsanými níže.

2.1.1 Architektura klient–server

Tento typ síťové architektury je jedním z nejrozšířenějších. Skládá se ze dvou základních částí: klientské aplikace a serveru. Proto se tato koncepce často nazývá *dvouvrstvou architekturou*. Obě tyto komponenty mohou vzájemně komunikovat prostřednictvím vytvořeného síťového spojení. Klientská aplikace odesílá dotaz (*GET DATA*), ten je zpracován serverem a odpověď (*SEND DATA*) je poté ve většině případů zobrazena klientem v nějakém typu uživatelského rozhraní. Náznak takovéto jednoduché komunikace je nastíněn na obrázku 2.1. Jelikož aplikace typu klient–server

Obrázek 2.1: Příklad komunikace klient–server

jsou velice rozšířeny, můžeme je v dnešni době použít téměř ve všech odvětvích informačních technologií. Proto zde jen lehce nastíňuji přehled všech možných uplatnění. S ohledem na vykonávané činnosti a poskytovaná data, rozdělujeme servery do těchto pěti hlavních kategorií (podle [8]):

- **webový server**
- **aplikační server**
- **souborový server**
- **terminálový server**

- **e-mailový server**

U těchto typů je zásadní rozdíl ve vykonávané činnosti. Klíčový je ale fakt, že základní architektura a princip komunikace je ve všech těchto případech podobný, ne-li stejný.

V některých případech je žádoucí aby serverová aplikace měla informace o uživatelích, kteří využívají jejich služeb. Servery lze rozdělit z pohledu uchovávání stavu relace do dvou kategorií:

- **bezstavové**
- **stavové**

U stavové komunikace je kladen velký důraz na uložení aktualního stavu relace. V tomto případe je většinou každé aktivní spojení representováno seancí. Seance je unikátní v rámci daného serveru. Seance může být realizována trvalým spojením nebo přímo seanční vrstvou protokolu (např. FTP nebo telnet). V protokolech, které formálně nedefinují seanční vrstvu (např. UDP) nebo protokolech, u kterých nemá seance dlouhého trvání (např. HTTP), se často používají cookies. Cookies jsou ale značně nevýhodné. Umožňují uživateli připojit se jen k jednomu server a nejsou ideálním řešením z hlediska sít'ové bezpečnosti.

Dalším přístupem, který je hojně využíván, je ukládáni a manipulace seance na straně serveru. K zajištění unikátnosti slouží jedinečný identifikátor (většinou číselná konstanta nebo alfanumerický řetězec), kterým se přistupuje k datům seance. Tyto data jsou uložena v interní paměti (dvouvrstvá architektura) nebo v databázi či sdíleném souboru (vícevrstvá architektura). Tento přístup následně umožňuje ukládat různé informace, ke kterým lze poté přistupovat při následujícím požadavku uživatele. Ukládání informace o seanci do databáze má výhodu při dynamickém rozkládání výkonu, kde všechny servery, které mají přístup k databází mohou obsloužit klienta vlastnícího danou seanci.

Bezstavová komunikace je v dnešní době, kdy se klade velký důraz na bezpečnost, spíše na ústupu. Stále lze ale nalézt situace, kdy seance není potřebná. Příkladem bezstavového protokolu je protokol HTTP. Ale i v tomto protokolu lze aplikovat seance pomocí jazyka PHP nebo cookies.

Aplikace typu klient–server jsou flexibilním a hlavně jednoduchým způsobem centralizované sít'ové komunikace. Tento přístup je ideální již z několika důvodů:

- Všechna data jsou uloženy na serveru. Toto umožňuje lepší kontrolu nad jejich bezpečností. Server sám kontroluje přístupová práva uživatelů pro přístup k datům. Z tohoto pramení nevýhoda vyššího zatížení serveru způsobeného nutností spravování datového zdroje vyskytujicího se přímo v programu samotném.
- Architektura klient–server je více flexibilní z pohledu přístupu k datům, jelikož všechny data jsou centralizována. Proto toto řešení je v tomto ohledu lepší nežli koncepce P2P. Jak již bylo řečeno výše, z hlediska zatížení správou dat, je na tom P2P přístup lépe, protože takovéto operace jsou distribuovány.

Jako každá technologie, i přístup klient–server má pár nevýhod. Tyto nevýhody jsou ale jen minoritní a jsou kompenzovány velkým množstvím předností. Všechny nevýhody je možné shrnout do těchto dvou bodů:

- Největším z problémů centralizovaného serveru je fakt, že při velkém množství simultánně připojených klientů může docházet kromě vysokého zatížením serveru k problému zahlcení. Tento problém lze řešit použitim P2P komunikace nebo distribucí výkonu na více serverů.
- Klient–server přístup není do takové míry robustní jako komunikace P2P. V případě selhání nebo výpadku serveru je jakákoliv komunikace nemožná. Proto je možným řešením tvorba klienta poskytujicího možnost P2P i klient–server komunikace.

2.1.2 Třívrstvá architektura klient–server

V softwarovém inženýrství termínem vícevrstvé architektury nazýváme architekturu klient–server, ve které jsou data zpracována více než jedním softwarovým agentem. Názorným příkladem může být aplikační server, jenž poskytuje uživateli databázová data. Tento typ se nazýva *třívrstvá architektura* a je nejrozšířenější. Název a celá koncepce byla specifikována konsorciem **Rational Software**.

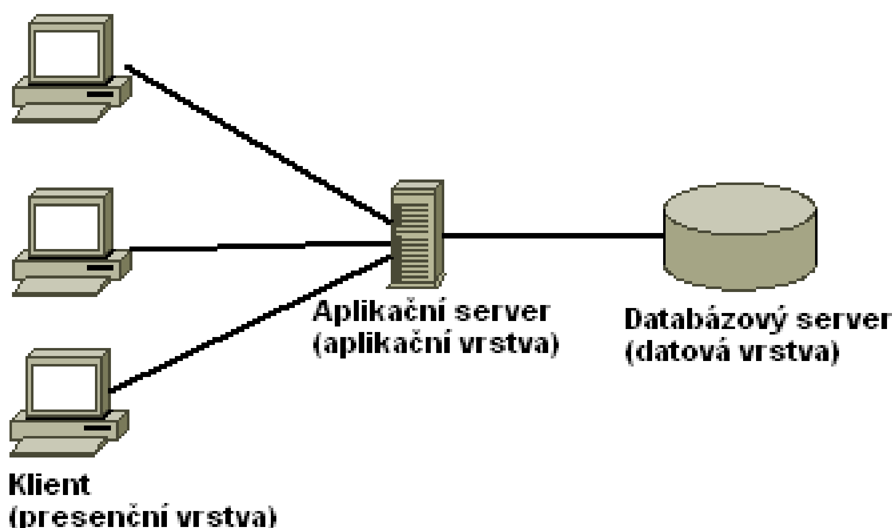
Jak je patrné na obrázku 2.2, tato architektura se skládá z těchto tří částí:

- **Presenční vrstva** – Representuje aplikaci, která využívá služeb logické vrstvy. Posíláním dotazů na server získává požadovaná data. Ty jsou pak interpretována, zpracována a srozumitelným způsobem (většinou uživatelským rozhraním) poskytnuta uživateli. Jak je patrné z modelu, oproti jiným typům architektur (např. MVC), nemá aplikace přístup k dalším vrstvám. Proto není pro aplikaci nutné znát strukturu dalších vrstev nacházejících se za vrstvou logickou.

- **Logická (aplikační) vrstva** – Představuje aplikační server samotný. Server je centralizovanou aplikací zpracovávající data z ostatních vrstev. Uživatel posílá dotazy, které jsou následně interpretovány. V případě nutnosti server využívá služeb datové vrstvy. Jak je patrné, centralizovaná architektura zvyšuje síťový provoz a nároky na výkon serverového systému.

- **Datová vrstva** – Poskytuje serveru možnost ukládání a následného získávání dat. Datovou vrstvou nemusíme chápat jen databázový server. Existují i jiné technologie umožňující spravování dat. Mezi ně můžeme zařadit OLAP nebo souborové systémy (sdílené textové nebo XML soubory). Nezávisle na technologii, server využívá podobné, ne-li stejné principy pro práci s daty v této vrstvě.

S ohledem na svou strukturu je tento princip velice flexibilní a hojně používaný.



Obrázek 2.2: Třívrstvá architektura klient–server

Použití vícevrstvé architektury má značné výhody:

- Vícevrstvá architektura se skládá z více částí, které komunikují s aplikační vrstvou. Aplikační vrstva má pro tuto komunikaci ve většině případů standardizované rozhraní. Proto jakákoliv změna jedné komponenty neovlivní chod dalších. Jako příklad se muže uvažovat aplikační vrstva komunikující s databázovým serverem MySQL pomocí ODBC rozhraní. Při změně serveru na Oracle nedojde k narušení dalších komponent, a jelikož k databázovému serveru Oracle existuje ODBC ovladač, nebude potřeba ani měnit strukturu komunikačního rozhraní
- Na rozdíl od MVC architektury, ve které je komunikace triangulární a všechny prvky mohou vzájemně bez zábran komunikovat, vícevrstvá architektura toto umožnuje jen částečně. Dochází jen ke komunikaci aplikační a datové vrstvy. Klient nemá přístup k datové vrstvě a přistupuje k ní prostřednictvím aplikační vrstvy. Proto je toto řešení velice bezpečné a autentizace uživatele je centralizována. To má za následek menší zatížení datové vrstvy a větší šanci na odhalení případného útoku.

V následujících dvou bodech jsou shrnuty zásadní rozdíly mezi vícevrstvým a dvouvrstvým přístupem síťové komunikace. Každy z těchto přístupů má své klady i zápory a záleži na konkrétní situaci, která rozhodne, které řešení je více vyhovující.

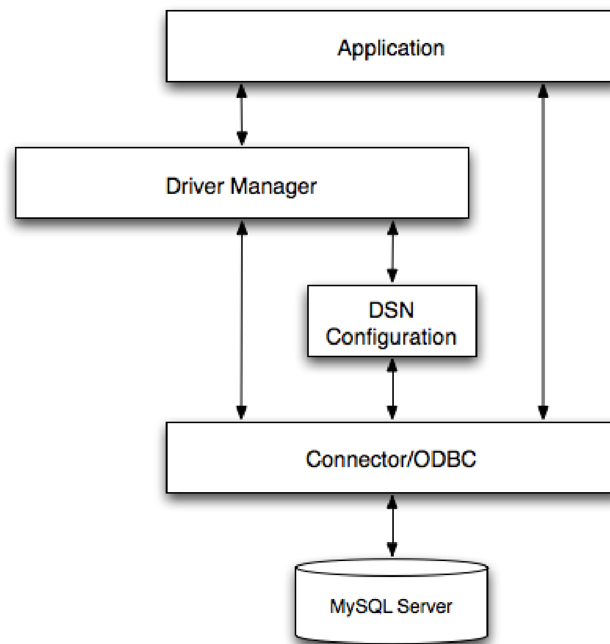
- V porovnání s dvouvrstvou klient–server architekturou je tato architektura značně náročnější na síťový provoz, jelikož všechna data z dalších zdrojů jsou zpracovávána a posílána jen výlučně aplikační vrstvou. Zpracovávání informací z ostatních zdrojů je výkonově náročné a způsobuje dodatečnou zátež.
- Oproti dvouvrstvé klient–server architektuře je zde větší počet mezi sebou komunikujících zdrojů. Proto je velký problem s naprogramováním a následným testováním výsledné aplikace. Tato skutečnost zvyšuje cenu vývoje a údržby aplikací tohoto typu.

2.1.3 Přístup k datové vrstvě pomocí ODBC

Pro připojení k datové vrstvě ve vicevstvých architekturách klient–server se ve většině případů využívá rozhraní ODBC. Toto rozhraní je absolutně nezávislé na použitém programovacím jazyku, databázovém a operačním systému. Proto je dobrým řešením pro implementaci rozhraní ve vícevrstvé architektuře klient–server. Použití ODBC zvyšuje modularitu a nabízí možnost rychlé změny komponent systému. Existuje ale i řada jiných technologií stéjného zaměření. Mezi ně můžeme zařadit tyto: OLE DB, ADO.NET nebo JDBC. ODBC má ale výhodu v tom, že tato technologie již existuje řadu let (od roku 1992) a je stabilnější a více odladěná nežli ostatní systémy podobného typu. ODBC dosahuje podobných výsledků v rychlosti jako nativní databázová rozhraní (rozdíl v jednotkách procent [6]). Nativní rozhraní navíc oproti ODBC nabízí některé specifické operace, které z důvodu přenositelnosti nemohou být součástí ODBC. Od aktualní situace a požadavků na systém se poté může odvíjet zvolení ODBC nebo nativního řešení.

Architektura ODBC na obrázku 2.3 se skládá z těchto pěti částí (podle [5]):

- **Aplikační** – Jedná se o samotnou aplikaci, která využívá standardizované rozhraní (*ODBC API*) k přístupu k datům databázového serveru. Toto rozhraní je implementováno ODBC ovladačem. Přítomnost unifikovaného rozhraní umožňuje odstínit aplikaci od databáze samotné. Proto pro aplikaci není podstatné znát způsob uložení dat ani konfiguraci systému k přístupu k datům. Dokonce pro aplikaci nemusí být podstatný ani fakt, že se nejedná o databázový server. (Například pro XML a EXCEL souborory existují také ODBC ovladače.) Samotná aplikace musí také komunikovat se správcem ovladačů, což umožňuje správu a konfiguraci aktuálně používaného ODBC ovladače. Jedinou úlohou aplikace je připojení se pomocí



Obrázek 2.3: Struktura ODBC

správce ovladačů k vybranému zdroji dat (DSN), posílání SQL dotazů a následná interpretace příchozích odpovědí.

- **Správce ovladačů** – Je knihovnou, která umožňuje komunikaci mezi ovladačem (ovladači) a aplikací samotnou. V systému Windows se jedná o kolekci DLL knihoven, která nese označení (Microsoft ODBC). Ta je v současné době součástí všech distribucí systému Windows. Lze nalézt i jiné konkurenční řešení jako například iODBC, UnixODBC nebo UDBC. Správce ovladačů umožňuje získávání DSN spojení (unikátní alfanumerický identifikátor připojení), načítání ovladače pro daný typ ODBC spojení a obsluhu volání ODBC funkcí.

- **DSN konfigurace** – DSN (Data source Name) je unikátní řetězec, který reprezentuje konkrétní databázový zdroj. K tomuto jsou přidruženy informace pro navázaní spojení s databází (ovladač, databáze, hostitel a případně autentizační údaje). Aplikace potřebuje k přístupu k databázi znát jen název zdroje (DSN), který poskytne správci ovladačů a ten se postará o navázání spojení.

- **ODBC ovladače** – Soubor knihoven implementujících podporu funkcí rozhraní ODBC (ODBC API). Mezi základní činnosti patří: obsluha volání funkcí rozhraní ODBC, zasílání dotazů na konkrétní typ databázového serveru a navrácení výsledků dotazů zpět aplikaci. Dotazy jsou komponovány s použitím jazyka SQL. V některých případech je nutno tyto dotazy upravit do formy podporované daným databázovým serverem.

- **Databázový server** – Konkrétní databázový server, na který se pokoušíme připojit pomocí ODBC rozhraní. V současnosti je ODBC ovladač dostupný k většině databází. Jelikož, jak již bylo zmíněno, ODBC je univerzálním rozhraním, nemusí být všechny funkce daného databázového serveru dostupné z použitím ODBC rozhraní. Pro tento případ je nutno zvolit

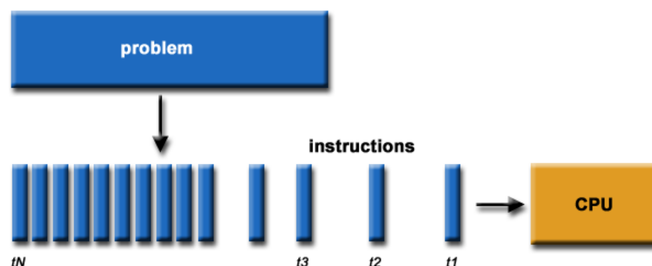
přímo proprietární (firemní) řešení. Pro databázi MySQL je dostupné MySQL API, ke kterému lze přistupovat z jazyka C++ použitím MySQL++.

2.2 Principy paralelních systémů

2.2.1 Paralelní programování

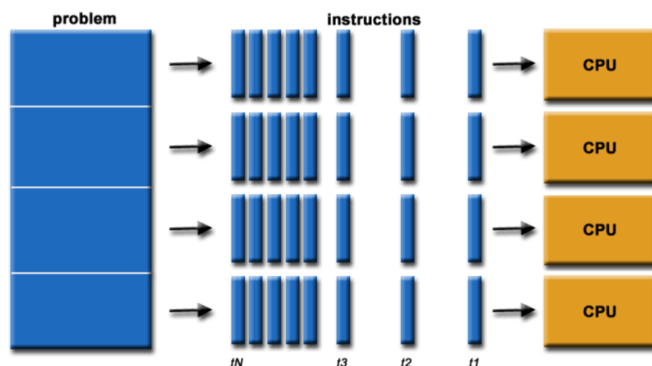
Je-li dán algebraický problém, který je potřeba vyřešit, lze aplikovat dva typy přístupu. Je možné přistoupit ke klasickému sériovému přístupu nebo použít řešení paralelní exekuce.

Sériové řešení je postaveno na klasické struktuře definované ve čtyřicátých letech minulého století Maďarským matematikem *Johnem von Neumanem*. Klasický von Neumanův stroj se skládá ze 4 částí: vstupně/výstupního rozhraní, řídící jednotky, paměti a jedné centrální aritmetické jednotky (CPU). Pro takovýto počítač je napsán program. Jak je zřejmé na obrázku 2.4, program je přeložen do instrukcí CPU, které tvoří diskrétní sérii. Procesor poté postupně (sériově) načítá a vykonává dané instrukce. Je nutno podotknout, že v tomto řešení může být v danou chvíli vykonávána jen jedna instrukce. Druhým přístupem, který je v dnešní době vicejádrových procesorů dosti aktu-



Obrázek 2.4: Sériový přístup k řešení problému (zdroj [1])

alním trendem, je paralelní zpracování 2.5. Systém v tomto případě je podobný tomu popsanému výše až na fakt, že místo jedné aritmetické jednotky jich má hned několik. Problém je rozdělen do více částí, které lze vykonávat konkurentně. Každá z těchto částí je poté rozdělena na sérii instrukcí. Instrukce z každé části jsou pak postupně vykonávány tak, že pro každou část je alokována jedna aritmetická jednotka.



Obrázek 2.5: Paralelní přístup k řešení problému (zdroj [1])

2.2.2 Rychlost paralelních řešení

Paralelní řešení za jistých okolností urychluje řešení daného problému. Otázkou je do jaké míry se dané řešení urychlí. Na toto je hodně obtížné nalézt odpověď. Zrychlení daného paralelního řešení se odvíjí od mnoha faktorů.

Existuje i matematický aparát, který nám umožňuje určit faktor zrychlení daného algoritmického problému při použití paralelního vykonávání. Gene Amdahl formuloval zákon s jehož pomocí můžeme určit faktor zrychlení daného systému při změně rychlosti běhu jeho určité části. Tento vztah lze aplikovat i na specifikaci zrychlení při změně dané části systému ze synchronního běhu na běh paralelní. Pro tento případ je definován tento vztah (podle [7]):

$$S = \frac{1}{F + \frac{1-F}{N}}$$

Kde:

- **F** – určuje podíl výpočtu, který zůstává sekvenční (nemůže být paralelizována).
- **N** – počet aritmetických jednotek (CPU), které daný systém poskytuje.
- **S** – celkový faktor zrychlení (kolikanásobně je toto řešení rychlejší).

Tímto vztahem lze určit maximální možné zrychlení, na jehož hodnotu mají vliv faktory jako použitý programovací jazyk, operační systém, rychlost paměti atd. Modelovým příkladem je čistě teoretické lineární zrychlení (F=1), kde s počtem procesorů lineárně stoupá výkon. V reálných systémech lze ale stanovit toto:

- Paralelní programování je efektivní jen pro malý počet procesorů.
- **nebo** pro řešení problémů, jejichž sekvenční část je co nejmenší. Proto se v paralelním programování vyvíjí snahy o co nejrozsáhlejší paralelizaci.

2.2.3 Modely paralelního programování

Modely paralelního programují existují jako abstrakce nad hardwarem a architekturou paměti. Jedná se o jakýsi doporučený přístup k řešení daného problému. Nelze ale obecně určit, který model je v dané situaci vhodnější. Využití modelů paralelního programování je otázkou kompromisů. Rozlišujeme tuto pětici modelů (podle [1]):

- **Sdílená paměť** – V tomto modelu úlohy sdílejí společný adresový prostor. V něm mohou provádět operace čtení i zápisu. Pro zaručení konzistence dat lze použít zámků nebo semaforů. Výhodou tohoto modelu je, že data nemají specifikovaného vlastníka. To přispívá ke zjednodušení výsledného kódu aplikace. Nevýhodou je skutečnost, že nastává problém s umístěním sdílenych dat při použití tohoto principu v distribuovaných systémech.
- **Vlákna** – V současnosti nejčastěji využívaný přístup. Je to způsob, jakým lze program rozdělit do více souběžně bežících úloh. Všechny vytvořená vlákna, ale i program samotný, sdílejí stéjný paměťový prostor. Proto je nutné zavést synchronizační metody, které zaručí výlučný přístup ke sdíleným zdrojům. O běh vláken se stará plánovač jádra operačního systému. Ačkoliv se zdá, že vlákna jsou vykonávána souběžně, je to jen klamný dojem. Všechny operace, které vykonává jádro operačního systému jsou spouštěny s využitím techniky dávkování času (time slicing). Každému procesu běžícímu na daném počítačí je podle určitých kritérií

přidělen časový okamžik, ve kterém může plně využívat procesorové jádro. Na dnešních vícejádrových procesorech je situace podobná. Do jisté míry je zde aplikováno souběžné zpracování. Ale i přesto je zde stále aplikováno dávkování času. Jen s rozdílem, že zde se přiděluje více jader (procesorů) a teoreticky v daném okamžiku muže běžet na každém jádře jiný proces.

- **Zprávy** – Jedná se o sadu úloh, které mají svůj vlastní adresový prostor. Tyto úlohy mohou být vykonávány na jednom počítači nebo současně na několika strojích propojených počítačovou sítí. Jak vyplývá z názvu, úlohy vzájemně komunikují posiláním zpráv obsahujících data. Přenos samotných dat vyžaduje kooperující operace, které musí být vykonány každým procesem. Například operace posílání dat musí mít odpovídající operaci, která data přijme. Tento způsob je často využíván pro distribuované výpočty a pro toto využití bylo dokonce definováno i přenositelné rozhraní[1].
- **Datový paralelismus** – Tento přístup se zaměřuje na práci se sadou dat. Tyto data jsou organizována do struktury, jakou je může být pole. Všechny úlohy pracují se stéjnou strukturou a provádějí nad ní totožné operace, ale každá pracuje s jinou částí dat.
- **Hybridní** – Je kombinací dvou či více modelů uvedených výše. Nejčastějším použitím je kombinace posílání zpráv a vláknového modelu nebo sdílené paměti.

2.2.4 Synchronizační techniky

V káždé paralelní aplikaci vznikají problémy při přístupu vláken ke sdíleným zdrojům. Z důvodu udržení konzistence sdílených zdrojů je nutno definovat speciální synchronizační objekty. Níže jsou vyjmenovány základní druhy těchto objektů.

Semafor – Jeden z nejčastěji využívaných synchronizačních mechanismů nevyžadujicích aktivní čekání. Jedná se o celočíselnou sdílenou proměnnou s těmito dvěma atomickými operacemi: *lock* a *unlock*. Při obsazování semaforu se jeho hodnota snižuje a při uvolňování zvyšuje. Jeli tato hodnota nulová znamená to, že je kritická sekce obsazená a nikdo na ní nečeká. Klesá-li hodnota proměnné do záporných hodnot, znamená to, že kritická sekce je plně obsazena a absolutní hodnota sdílené proměnné udává počet čekajících procesů ve frontě dané kritické sekce.

Mutex – Stéjně jako semafor, zajišťuje mutex výlučný[2] přístup ke sdílenému zdroji. Mutex je z teoretického pohledu binárním semaforem. Definuje stavy zamknutý (locked) a odemknutý (unlocked).

Monitor – Pro komfortnější implementaci synchronizace byl navržen tento vysokoúrovňový synchronizační mechanismus, který je často vystavěn nad semafory. Proto umožňuje použití vzájemného vyloučení. Monitor je výhodný kvůli možnosti čekání na určitou událost s použitím podmínek (conditions). Je definována operace `wait`, která pozastaví daný proces v rámci monitoru, automaticky uvolní monitor a čeká na určitou událost. Operace signal nebo broadcast[3] upozorní procesy čekající na dané podmínce na vznik události, na kterou čekají. Monitor je definovám POSIXem metodami začinajicími pthread_cond_. Do Javy pronikl monitor (s implicitně jednou odmínkou) do tříd či metod chráněných konstrukcí synchronized. Ve Win32 API mají stejnou sémantiku metody

[1] **MPI (Message Passing Interface)** je příkladem platformově nezávislého rozhraní pro paralelní programování aplikací s použitím posílání zpráv. Vice na: `www.mcs.anl.gov/mpi/`

[2] V teorii pravděpodobnosti události $E_1, E_2, \ldots, E_n$ jsou vzájemně výlučné jestliže platí: Výskyt jakékoliv události automaticky implikuje nemožnost výskytu dalších zbylých $n-1$ událostí.

[3] Základní rozdíl mezi těmito funkcemi je počet objektů, které informují o události. *broadcast* informuje všechny čekající na podmínce. *signal* informuje jen jednoho a příjemce není možno s jistotou určit při více čekajících.

`WaitForSingleObject` a `WaitForMultiObjects` spolu s použitím `Win32 event` objektu jako podmínky[4].

Barrier – Vysokoúrovňový synchronizační objekt umožňující synchronizaci běhu vláken. Objekt barrier pro daný počet procesu (vláken) určuje, že musí dojít k zastavení běhu na daném místě a čekání do doby než všichni dosáhnou daného místa. Tento objekt slouží k zastavení všech ostatních vláken a vykonání nějakého typu sériové operace. Lze jej také použít pro sesynchronizování startu všech vláken.

K problému synchronizačních metod se váže problém *uváznutí*. Jestli existují procesy A a B a sdílené zdroje X a Y. Proces A požádá o zdroj X a proces B požádá o zdroj Y. Poté může nastat situace, že proces A bude požadovat ještě zdroj Y a proces B zdroj X. Tato situace vede k zásadnímu problému zvanému *uváznutí*.

2.3 Paralelní exekuce

Architektura mnoha serverový aplikací, jako webových serverů, databázových serverů nebo souborových serverů, je postavena na vykonávání velkého množství relativně krátkých úloh, které přicházejí ze vzdáleného zdroje. Výkon většiny serverů poté záleží na efektivitě a rychlosti způsobu, jimž se tyto úlohy exekuují. Záměrem této kapitoly je popsat většinu používaných přístupů. V kapitole Realizace (4) bude poté na základě této kapitoly vybrán nejvhodnější způsob přístupu k problému paralelní exekuce s ohledem na implementaci výkonného a rychlého jádra aplikačního serveru, které bude nuceno vykonávat velké množství asynchronně příchozích dotazů.

2.3.1 Možné přístupy

Jedním ze základních principů je vytvoření vlákna na pozadí (backgound thread). Jak je patrvé z obrázku 2.6, je vytvořena fronta příchozích požadavků. Tyto požadavky jsou postupně vykonávány jedním vláknem. Tento přístup se hojně používá v dnešních aplikacích. AWT nebo Swing využívají tohoto principu pro zpracování událostí grafického rozhraní. Pro vykonávání úloh, které nemají dlouhého trvání je tento přístup idální.

Uvažujeme-li úlohu, která pracuje s databází, můžeme očekávat relativně dlouhé čekání na dokončení databázové operace. Další úlohy budou zbytečně čekat ve frontě a bude docházet k plýtvání výpočetních zdrojů počítače, jelikož procesor v tuto chvíli nebude vůbec zatížen. V grafickém rozhraní Swing (z důvodu urychlení obsluhy zpráv) lze toto řešit vytvořením separátního vlákna pro vykonávání dlouho trvajících operací uživatelského rozhraní.

Jak lze vidět na obrázku 2.7, dalším přístupem je vykonávání úloh v sepátním vlákně. Pro každý příchozí požadavek je vytvořeno vlákno, které ho vykoná. Takovýto přístup je výhodný pro vykonávání malého množství dlouhotrvajících úloh.

Pro případ velkého množství krátce trvajících úloh je toto řešení nevhodné, jelikož největším negativem je náročnost operace vytvoření a zrušení vlákna. Další nevýhodou je fakt, že nelze kontrolovat počet simultánně bežících vláken. Při nepřiměřeně vysokém počtu vláken může dojít k situaci, v níž bude procesor plně vytížen a většina času bude místo vykonávání operací použita k přepínáni kontextu[5].

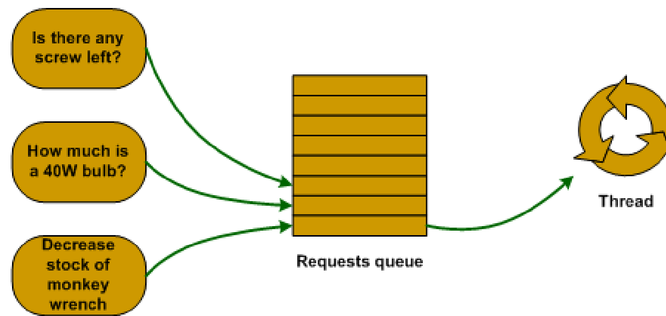
Jiným přístupem, jenž kombinuje některé aspekty předešlých případů je thread pool[6]. Na obrázku

---

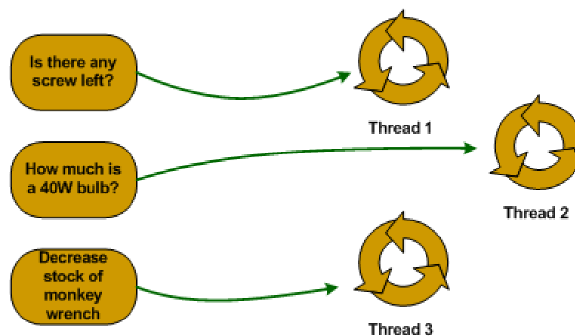
[4] Ve Win32Api lze za podmínku pokládat hned 9 různých typů. Detaily o této možnosti lze nalézt na: `http://msdn2.microsoft.com/en-us/library/ms687032.aspx`

[5] Přepínání kontextu umožňuje kvaziparalelní vykonávání jednotlivých vláken. Dochází k přidělování časových kvant jednotlivým úloham a je potřeba pamatovat předchozí stav registrů všech úloh.

[6] Pro tento termín neexistuje vhodný český ekvivalent.



Obrázek 2.6: Exekuce pomocí vlákna na pozadí (zdroj [2])



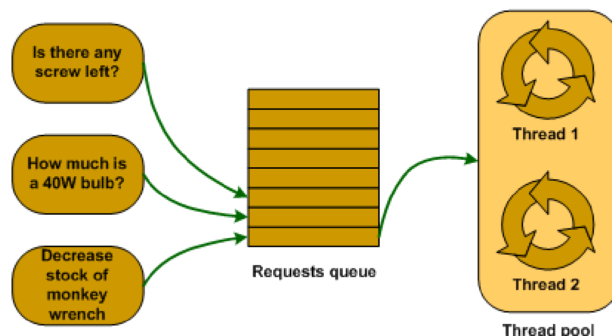
Obrázek 2.7: Exekuce úlohy v separátním vlákně (zdroj [2])

2.8 je patrné, že příchozí požadavek je zařazen do fronty. Poté existuje přiměřené množství vláken, která vybírají daný požadavek ze sdílené vstupní fronty a vykonávají jej. Jak je patrné, tento přístup eliminuje většinu nevýhod ostatních postupů. Je možné volit optimální počet vláken, která vykonávají dané operace. Díky tomuto návrhu nedochází ke zbytečnému plýtvání systémových zdrojů. Další výhodou je znovupoužití exekučních vláken. Ty jsou vytvořeny jen jednou a po celou dobu existence thread poolu již nedochází k jejich vzniku nebo zániku, což pozitivně ovlivňuje rychlost výsledného řešení. Nevýhodou je značná náročnost tohoto řešení co se týká použití synchronizačních metod.

2.3.2 Thread pool

Jedná se bezesporu o velice efektivní a rychlé řešení. Jako každá struktura, která je vystavěna na principu paralelní exekuce, je vystavena několika klasickým synchronizačním problémům (podle [4]):

- **Uváznutí** – Jako v každé jiné vícevláknové aplikaci, i zde vzniká možnost uváznutí. Klasický problém uváznutí je popsán v kapitole 2.2.4. V daném případě tento problém získává další rozměr. Uvažujeme-li dva exekuční vlákna, která vykonávají nějakou operaci. Tato operace čeká na výsledek jiné operace, kterou vložila do vstupní fronty thread poolu. Tímto vzniká uváznutí, jelikož obě vlákna vykonávají operací, která čeká na dokončení jiné operace, která se nachází ve vstupní frontě thread poolu. Jak je patrné, výsledku operace se nedočká, jelikož operace ze vstupní fronty nemužou být vykonány, protože není a ani nebude dostupné ani jedno volné vlákno.



Obrázek 2.8: Exekuce s použitím thread pool (zdroj [2])

- **Plýtvání zdrojů** – Je nutno volit vhodný počet exekučních vláken. Při jejich nepřiměřeně vysokém počtu může docházet k plýtvání systémových zdrojů z důvodu nutnosti přepínání kontextu. Dále je nutno pamatovat, že objekty jako například soubory, ODBC spojení nebo schránky jsou limitovaným zdrojem. Při nepřiměřeném konkurentním běhu muže docházet k nepřidělení zdrojů z důvodu jejich vyčerpání.
- **Race conditions** – Jedná se o klasický problém, který v dané situaci může způsobit značné problémy. Uvažujeme-li dva totožné požadavky: požadavek A a požadavek B. Oba tyto požadavky zapisují do stejného sdíleného zdroje. Je spuštěn požadavek A a poté požadavek B. Z čistě teoretického hlediska by mělo dojít k situaci, že k zápisu dojde v pořadí, ve kterém tyto požadavky byly vloženy do sdílené vstupní fronty. Ale jelikož, jak již z názvu vyplývá, dochází k závodění těchto dvou požadavku o dosažení cíle (zápisu do sdíleného zdroje), může nastat situace, ve které požadavek B předběhne v zápisu požadavek A.

Skutečností, která do značné míry ovlivňuje rychlost tohoto řešení, je počet exekučních vláken. Obecným problémem je najít kompromis mezi malým a příliš vysokým počtem vytvořených vláken.

Výhodou používání vláken je možnost vykonávání dalších operací i přesto, že některá vlákna čekají na výsledek I/O dotazů. Tento přístup také plně využíva výhod víceprocesorových systémů.

Optimální počet exekučních vláken se odvíjí od počtu procesorů (případně jader) a povahy vykonávaných operací. Obecně lze konstatovat, že pro maximální možné využití zdrojů N-procesorového stroje je optimální počet N nebo $N+1$ exekučních vláken.

Pro vstupně/výstupní operace (databázový dotaz, zápis do souboru) je potřeba volit počet vláken vyšší nežli počet procesorů. Důvodem je skutečnost, že některá vlákna, která čekají na výsledek například vstupně/výstupní operace nemusí v daném okamžiku pracovat. Pro N-procesorový (jadrový) systém je definován tento vztah (podle [4]):

$$C = N * \left(1 + \frac{WT}{ST}\right)$$

Kde:

- **C** – udává optimální počet exekučních vláken.
- **N** – počet procesorů daného systému.
- **WT** (waiting time) – čas čekání na dokončení vstupně/výstupní operace.
- **ST** (service time) – čas vykonávání celé úlohy.

Výsledná veličina **C** udává optimální počet vláken pro dosažení plného využití systémových zdrojů. Podmínkou nejlepší rychlosti je shlukování do daného threadpoolu operací stéjných charakteristik (času exekuce, náročnosti na systémové zdroje). Jestli nelze dosáhnou exekuce podobných operací v daném thread poolu, je nutno zvážit z důvodu lepší rychlosti vytvoření více thread poolů pro různe typy operací. V reálných systémech pro dosažení dobrých výsledků je nutno vytvořit jednu instanci pro posílání dat, další pro práci s databází a další například pro práci se soubory.

Kapitola 3

Návrh řešení

Cílem této kapitoly je nalézt nejvhodnější koncepci výsledného systému. Hlavní aspekty, které je nutno zvážit jsou rychlost, rozšiřitelnost a nezávislost.

Výsledné řešení musí využít nejvhodnějších technik pro tvorbu paralelní aplikace. Jak již bylo zmíněno v kapitole 2.2.3, existuje celá řada možných přístupů vedoucích k paralelizaci daného řešení. Úkolem je zhodnotit možnost využití některého modelu (modelů) s ohledem na povahu vytvářené aplikace.

Jednotlivé části systému by měly být vzájemně co nejméně závislé. Změna jakékoliv části systému by měla minimalně ovlivnit chování jiných. Takovýto princip je základem postupu OOP návrhu. Je snaha tvořit programové moduly (třídy) poskytující co nejlepší operace ostatním. Ostatní části programu nesmí být závislé na konkrétních akcích vykonávaných danými metodami. Pro úplnost příkladem by mohlo být využití třídy, která poskytuje metodu write pro zápis do souboru. Jestliže se prováděná operace změní ze zápisu do souboru na zápis do schránky, neměla by tato skutečnost ovlivnit ostatní, jelikož pro ně jsou implementační detaily dané metody nepodstatné.

Rozšiřitelnost je úzce spjata s pojmem nezávislost. Daná aplikace je dobře rozšiřitelná, jestliže závislost ostatních částí na měněném modulu je co nejmenší. Při dobrém návrhu třívrstvého serveru typu klient-server by měla být možná rychlá změna využívaného databázového serveru, jelikož již z podstaty této architektury vyplývá jakási modularita aplikační i datové vrstvy.

3.1 Formulace úlohy

Primárním požadavkem je navrhnout serverovou aplikaci podporující protokol pro výměnu textových zprav (instant messenging). Všechna pernamentní data budou uloženy v databázi, k níž bude mit přístup výhradně server. Jak je patrné, jedná se o klasický příklad třívrstvé klient-server architektury.

Obě aplikace (klient i server) využívají pro komunikaci mezi sebou určitý druh protokolu, jehož sémantice rozumí. Presenční vrstva (zde klientská aplikace) posílá požadavek na server. Server přijíma data, která následně analyzuje a determinuje, jaký je požadavek uživatele. Poté dochází k následnému kroku, kterým je formulace odpovědi. Data obsažená v odpovědí jsou buď získáná z interních datových struktur, nebo v některých případech je nutný dotaz na databázi. Takto získaná data jsou poté zpracována, vložena do odpovědi a převedena do formy odpovídající specifikaci protokolu.

Server musí byt schopen zajistit tyto základní operace:

- **Spravování klientských spojení** – Zahrnuje operace související s připojením a odpojením klienta. Musí rovněž existovat mechanismy schopné evidovat všechny připojené klienty.

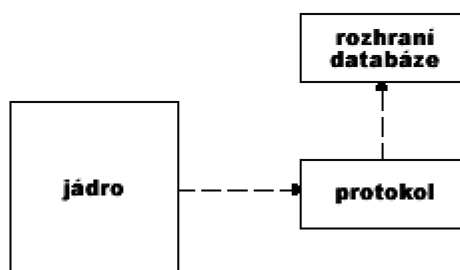
- **Příjem/posílání dat** – Musí existovat operace schopné přijímat data, která jsou representována zpravou v daném komunikačním protokolu. Aplikace musí být také schopna sestavit odpověď v daném formátu specifikovaném protokolem a odeslat ji požadovanému uživateli.
- **Interpretace sémantiky protokolu** – Je potřeba implementovat sémantiku komunikačního protokolu. Tento musí umět načítat zprávy v daném formátu, interpretovat je a vykonávat požadované operace. Podmínkou je také schopnost sestavení odpovědi ze získaných dat.
- **Komunikace s datovou vrstvou** – Jak již bylo zmíněno, serverová aplikace musí být schopná komunikace s databází. Pro tento účel je potřeba nalézt vhodné rozhraní pro praci s daty v této vrstvě.

Jako doplňující požadavky jsou definovány:

- posílání souboru
- šifrovaná komunikace
- mnohonásobné přihlášení na stéjný účet

3.2 Určení částí systému

V této části je potřeba dekomponovat celý problém. Danou doménu je potřeba rozdělit do logicky soudržných celků. Asi nejvhodnější rozdělení je viditelné na obrázku 3.1. Je definováno jádro systému, nad nimž je vystavěna sémantika pomoci části implementující protokol. Protokolová část komunikuje jak s jádrem, tak i datovou částí. Jedná se o centralizované misto, ve kterém dochází k vykonávání většíny funkcí.



Obrázek 3.1: Navrhovaná architektura systému

3.2.1 Jádro

Jedná se o klíčovou část celého programu. Hlavní úlohou je zajištění síťové komunikace. Obstarává také příjem příchozího uživatelského spojení. Informaci o všech připojených uživatelích je potřeba uchovávat pro pozdější použití. Jádro rovněž obstarává posílání dat konkrétnímu připojenému uživateli.

Zásadní otázkou pro zamyšlení je způsob načítání příchozích dat. Příchozí data jsou ve speciálním formátu specifikovaném protokolem. Jelikož, jak již bylo zmíněno, při návrhu aplikací je potřeba minimalizovat závislosti mezi moduly, proto je potřeba odstínit jádro od sémantiky protokolu. Základní funkcí jádra musí stále býti přijímání spojení, posílání a příjem dat a nesmí vzniknout

nežadoucí závislost na sémantice dat, která je pro jádro samotné nepodstatná. Proto nejlepším řešením je situace, ve které samotné načítání příchozích dat provádí protokol, který zná strukturu dat a umí s použitím dané sémantiky určit parametry příchozí zprávy.

Stéjnou koncepcí se musí řídit i posílání dat. Zpráva, která se má odeslat je vytvořena protokolem. Poté je předána jádru k odeslání. Tento model umožňuje odeslání všech dat danému příjemci bez ohledu na jejich strukturu nebo délku.

Tímto přincipem je možné dosáhnout nezávislosti jádra na struktuře a sémantice dat, se kterými pracuje. Z pohledu návrhu programu je dosaženo absolutní nezávislosti jádra na protokolu, ve kterém probíhá veškerá síťová komunikace. Dokonce je dosažen stav, ve kterém je jádro samotné nejenom nezávislé na sémantice protokolu, ale i na jeho typu. Proto je v dané situaci vedlejší, jestli protokol je charakteru XML, čistého textu nebo jakéhokoliv jiného.

Pro shrnutí jsou toto hlavní požadované operace, které musí jádro vykonávat:

- obsluha příchozích spojení
- příjem dat (samotnou operaci získáváni dat ze schránky provádí protokol)
- odesílání dat

3.2.2 Protokol

Veškerá sémantika příchozích dat je charakterizována specifikací protokolu. Jak již bylo řečeno, v daném návrhu jádro neví nic o významu dat (jsou to pro něj prakticky řetězce bezvýznamných znaků). Data dostanou svůj význam až při interpretaci protokolem. Zde se poté analyzuje celá zpráva. Podle získaných informací se rozhodne, jaký typ operace bude vykonán. Typ operace se přímo odvíjí od specifikace konkrétního protokolu. Celý postup třídy protokolu je možno shrnout do těchto bodů:

- analýza sémantiky příchozí zprávy
- získání požadovaných dat (je-li potřeba)
- sestavení odpovědi
- poslání odpovědi danému uživateli (nemusí být shodný s odesilatelem)

Zásadní otázkou pro tuto část textu je, jestli je nutno vytvořit určité mechanismy, díky nimž bude možno spouštět konkrétní operaci vykonávanou v reakci na sémantiku příchozí zprávy[1]. Existují dva možné přístupy, kterými lze tohoto dosáhnout:

- **Třída protokolu** – Pro každý konkrétní případ je ve třídě protokolu definována metoda. Budou existovat metody například pro přihlášení uživatele, přeposlání zprávy příjemci nebo získání informací o daném kontaktu. Nastává otázka jakým způsobem budou dané operace spuštěny. Operace je možno vykonávat v separátním vlakně, které se vytvoří při zavolání dané metody. Toto řešení je z hlediska paralelismu nepřijatelné. Události příchozí zprávy vznikají asynchronně, proto v dané situaci nemůžeme jednoduše kontrolovat počet paralelně[2] vykonávaných operací. Další nevýhodou je nutnost aplikace synchronizačních

[1] Označuje operaci, které získá potřebná data nutná pro sestavení odpovědi, vytvoří samotnou odpověd a obstará její poslání danému uživateli.

[2] Z důvodu multitaskingu dnešních systémů, se operace nevykonávají čistě paralelně. Jedná se o typ exekuce, která se nazývá kvaziparalelismus.

metod mezi vláknem volajícím a exekučním, což by bylo velice pracné a ještě více by znepřehlednilo třídu protokolu. Nevýhodou by byl také vznik závislosti na těchto synchronizačních metodách. Dalším možným řešením je vykonávání všech operací v aktuálním vlákně v němž je pracováno s třídou protokolu. Toto řešení by mohlo být za jistých okolností akceptovatelné (při krátkém čase exekuce daných metod). Uvažujeme-li situaci, ve které se čeká na výsledek databázové operace, která může trvat relativně dlouho, je toto řešení přinejmenším neefektivní. Čekání totiž zamezuje vykonání jakýmkoliv jiným operacím v daném vlákně. Nevýhodou, která se týka obou přístupů je obrovský nárust počtu metod třídy protokolu a vznik závislosti třídy samotné výlučně na svých vlastních metodách. V neposlední řadě se takováto třída stává při větším počtu metod těžko modifikovatelná a udržovatelná.

- **Uživatelem definované funkce** – Další možnou alternativou je přesunutí veškeré logiky vykonávání uživatelem definovaných funkcí do jádra serveru. V tomto případě, oproti příkladu výše, by třída protokolu nemusela obsahovat žádnou exekuční logiku. Tento přístup do značné míry respektuje fakt, že třída protokolu by se měla zabývat čistě analýzou sémantiky a jádro zastřešovat provádění všech potřebných operací. Jádro nabízí uživateli (pod tímto pojmem rozumíme protokol) možnost zaregistrovat si vlastní funkce. Každá takto registrovaná funkce může být například identifikována jedinečným identifikátorem, jakým je například textový řetězec. Jádro bude poskytovat také možnost exekuce požadované funkce. Pro vykonání dané uživatelské funkce je potřeba znát její název, pod kterým je registrovaná. Volitelně lze poskytnout uživatelem definovaná data. Úkolem jádra je zajistit co nejrychlejší a nejefektivnější exekuci. Je nutno pečlivě zvážit způsob řešení exekuční části, jelikož na ní ve značné míře závisí celá rychlost výsledné aplikace. Je potřeba zvážit použití přístupu popsaného v kapitole 2.8.

3.2.3 Elementy datové vrstvy

Datovou vrstvou v kontextu této aplikace rozumíme databázový server. Na tomto serveru jsou uložena všechna pernamentní data. Je nutno navrhnou vhodný typ přístupu k datům tohoto zdroje. V tomto případě je nutno pamatovat na fakt, že většina přístupů k databázi bude prováděna paralelně. Proto je nutné specifikovat postupy umožňující spravování více databázových spojení. Je nutno zvážit použití rozhraní ODBC, které bylo již popsáno v kapitole 2.1.3. Toto rozhraní by bylo vhodné pro tuto aplikaci z těchto důvodů:

- Podpora ODBC rozhraní většinou dnešních operačních systémů.
- Nezávislost na zvoleném databázovem serveru.
- Existence množství knihoven pro práci s ODBC.

Je nutno podotknout, že technologie přístupu k datové vrstvě jsou součástí logiky třídy protokolu. V žádném případě nesmí dojít k závislosti jádra systému na některých součástech využívaných zaregistrovanými uživatelskými funkcemi.

Co se týká samotné databáze, je potřeba navrhnout vhodný model uložení dat. Zohlednit je potřeba fakt, že k databázi se bude ve vetšině případů přistupovat pomocí více souběžných spojení. Dále je nutno, pro urychleni běhu aplikace, vhodně definovat indexy tabulek. Dobře zvolené indexy tabulek mohou v konečném důsledku urychlit vykonávané operace. Struktura SQL dotazů samotných je také velmi důležitá. Je nutno volit rychlejší verze některých dotazů a vyvarovat se některým náročnějším dotazům, jakými jsou například agregační funkce nebo vnořené dotazy.

Kapitola 4

Realizace

Tato kapitola se zabývá realizací jádra serverové aplikace. Je popsáno několik implementačně zajimavých částí.

4.1 Implementační prostředí

Výběr implementačního prostředí je úzce spjat s požadavkem multiplatfomnosti. Také zkušenosti programátora hrají značnou roli.

Uvažovat lze tří prostředích. A to tyto: *jazyk Java*, *platforma .NET* společnosti Microsoft a *jazyk C++*. Platforma .NET v dnešní době není přenositelná a stejně jako jazyk Java vyžaduje překlad do mezikódu a běh s použitím virtuálního stroje. Jazyk Java by mohl být pro daný případ dobrým řešením, ale v porovnání s jazykem C++ je velice pomalý.

Jazyk C++ je platformově nezávislý jen z části. Pro dosažení plné nezávislosti bylo potřeba zvolit kombinaci *jazyka* C++ a *wxWidgets*, která je využita pro tvorbu serverové aplikace.

4.2 Uživatelem definované funkce

4.2.1 Thread pool

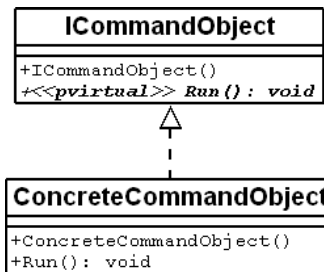
Jak již bylo diskutováno v kapitole 2.3.2, thread pool je jedním z možných přístupů k problému paralelní exekuce. Tako kapitola se zaměří na konkrétní implementaci tříd pro paralelní vykonávání dotazů.

Hlavní otazkou je, co lze chápat pod pojmem dotaz. Z pohledu thread poolu je dotazem objekt, jenž může nějakým způsobem spustit určitou programátorem definovanou posloupnost akcí. Pro takovýto účel bylo vytvořeno rozhraní `ICommandObject`. Na obrázku 4.1 lze vidět, že toto rozhraní má kromě povinného konstruktoru definováno čistě virtuální metodu `Run()`.

Vytváření instancí třídy `ICommandObject` nedává hlubší smysl. Pro formulaci uživatelského dotazu je nutno vytvořit konkrétní třídu poděděnou z rozhraní `ICommandObject`. Tato třída poté implementuje metodu `Run`, která je vstupní bodem exekuce.

Jako příklad lze uvažovat požadavek na vytvoření dotazu, který pošle klientu určitá data z databáze. Programátor bude postupovat následovně:

- Vytvoří konkrétní třídu poděděnou z `ICommandObject` (lze specifikovat i parametrizovaný konstruktor pro možnost vložení vstupních dat).
- Implementuje metodu `Run`, která je vstupním bodem exekuce a obsahuje vykonávané operace, kterými jsou v tomto konkrétním případě dotaz na databázi a poslaní odpovědi.

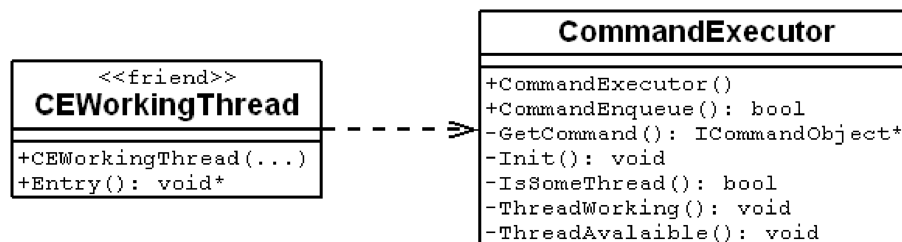


Obrázek 4.1: Exekuční objekt

- Vytvoří instanci dané třídy.
- Instanci vloží do vstupní fronty exekutoru, který poté tento požadavek vykoná zavoláním metody `Run`.

Jak je patrné, vytváření dotazů je jednoduché a přímočaré. Konkrétní třida může obsahovat jakýkoliv druh operace, která může trvat jakkoliv dlouho. Je třeba ale pamatovat na skutečnost, že příliš velký počet dotazů s dlouhou dobou exekuce může způsobit problémy, jakými může být přeplnění vstupní fronty nebo extrémní nárůst doby odpovědi.

Samotná implementace thread poolu na obrázku 4.2 se skládá z hlavní třídy `CommandExecutor` a spřátelené třídy `CEWorkingThread`. Výhodou deklarace třídy `CEWorkingThread` jako spřátelené k hlavní třídе je možnost přístupu k privátním metodám a proměnným. Tento přístup je výhodný, jelikož není potřeba zbytečně měnit viditelnost metod.



Obrázek 4.2: Struktura objektu implementujícího thread pool

Třída `CommandExecutor` definuje jen dvě veřejné metody. První z nich je metoda `CommandEnqueue`, která vloží objekt implementující dotaz do sdílené vstupní fronty. Tato fronta je ve výchozím nastavení implementována pomocí klasické C++ fronty(`std::queue`). Je ale také možnost zvolit implementaci postavenou na využití vázaného seznamu, který je přímo nativním typem *wxWidgets*. Další veřejnou metodou je konstruktor, který obstarává vytvoření instance třídy s počtem exekučních vláken, který je specifikován vstupním parametrem.

Vytvoření daného počtu vláken zajišťuje privátní metoda `Init`. Hlavním požadavkem je vytvoření daného počtu exekučních vláken. Všechny tyto vlákna mezi sebou sdíleji mutex, nad kterým je definován monitor. Všechny vlákna je nutno vytvořit tak, aby po vytvořeni přešly voláním `Wait` nad sdíleným monitorem do stavu čekání na signalizaci změny podmínky, kterou je v daném případě příchod požadavku do sdílené fronty a nutnost jeho obsluhy. Jediným problémem, který bylo potřeba v dané situaci řešit, byla nutnost existence všech exekučním vláken, čekajících na dané podmínce, v okamžiku dokončení metody `Init`. Tento zásadní problém bylo nutno řešit pomocí synchronizačního principu barrier, který byl již popsán v kapitole 2.2.4.

V tomto případě je problém implementace mechanismu barrier řešen pomocí synchronizačního objektu monitor, který je pro tento případ nejvhodnější alternativou, která je nativně podporována prostředím *wxWidgets*. Obrázek 4.3 ilustruje vytvoření dvou exekučních vláken metodou `Init`. Pro účely synchronizace je v dané ukázce využito dvou synchronizačních objektů monitor `CreationMonitor` a `ExecutionMonitor`. Postup vytváření je pro každé vlákno stejný. Hlavní vlákno, ve kterém je spouštěna metoda `Init`, vytváří novou instanci třídy `CEWorkingThread`. Toto nově vytořené vlákno je následně spuštěno voláním metody `Run`. V tomto okamžiku je nutno aplikovat synchronizační mechanismy, díky nimž hlavní vlákno bude čekat na dokončení vytvoření exekučního vlákna. Pro tento účel je použita proměnná `CreationMonitor`, která je typu monitor. Po spuštění vytvořené instance exekučního vlákna, aplikuje hlavní vlákno pomocí metody `Wait` monitoru `CreationMonitor` čekání na signalizaci podmínky, kterou je v daném případě dokončení výtváření exekučního vlákna. Vytvářená instance exekučního vlákna provede všechny potřebné inicializační kroky a signalizuje hlavnímu vláknu splnění podmínky, kterou je dokončení inicializace. Poté exekuční vlákno voláním metody `Wait` monitoru `ExecutionMonitor`, započne čekání na signalizaci podmínky, kterou je požadavek na exekuci dotazu, který byl vložen do sdílené vstupní fronty.

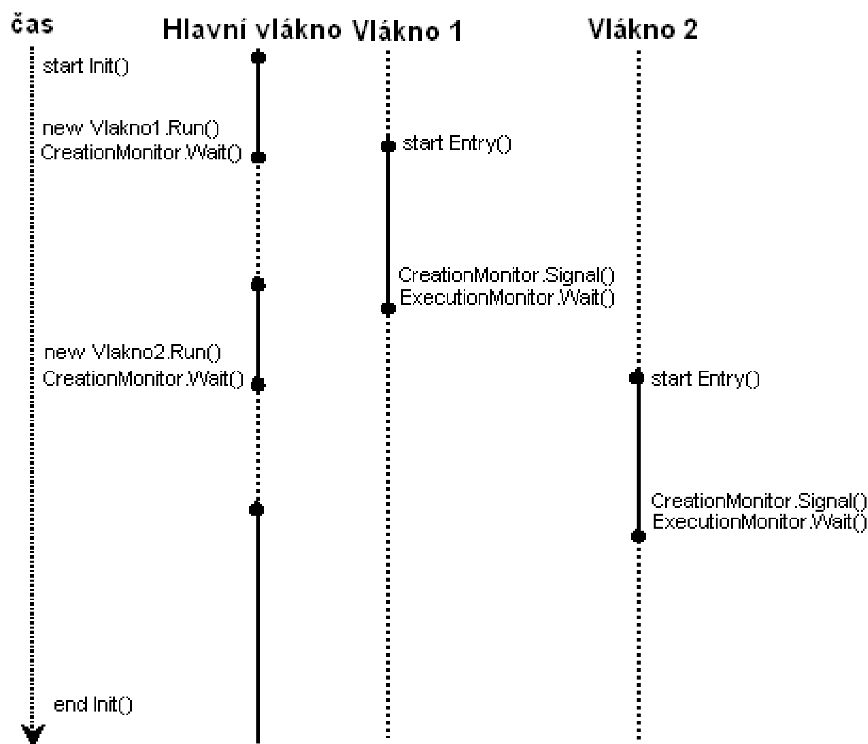
Zásadním synchronizačním problémem, který bylo nutno v dané situaci řešit, je zabezpečení doručení signálu. Při špatném přístupu může totiž dojít k situaci, že signál změny podmínky dorazí v okamžiku, kdy dané vlákno nezačalo ještě čekat na tuto skutečnost. V aktuálním případě by mohla nastat situace, kdy hlavní vlákno dostává signál splnění podmínky o dokončení vytváření exekučního vlákna ještě předtím, než na splnění této podmínky začne čekat. V takovémto případě může dojít k nekonečnému čekání hlavního vlákna na příchod signálu, který již byl poslán. Jak je patrné, jedná se o značně nebezpečnou situaci, která může způsobit pád celé aplikace.

Způsobem popsaným výše byl po dokončení metody `Init` vytvořen daný počet exekučních vláken, která čekají na signalizaci příchodu požadavku do vstupní sdílené fronty thread poolu.

Chování daných exekučních vláken po zjednodušení některých synchronizačních aspektů lze jednoduše popsat pomocí několika vět. Exekuční vlákno se může nacházet ve dvou stavech: pracující (aktivní) nebo čekající na přidělení požadavku (pasivní). Ve stavu pracující vlákno bere požadavek ze sdílené fronty a vykonává jej. V tomto stavu se vlákno nacházi tak dlouho, dokud jsou k dispozici požadavky k vykonávání, čili do doby, než není vstupní fronta prázdná. Jestli dojde k vyprázdnění vstupní fronty, začne exekuční vlákno čekat na sdíleném monitoru na signalizaci podmínky, kterou je požadavek na exekuci dotazu. V tomto okamžiku se vlákno nachází v druhém stavu. Aktuální stav, ve kterém se vlákno nachází, sděluje třídě `CommandExecutor` pomocí metod `ThreadAvalaible` a `ThreadWorking`. Po volání těchto metod je upravena hodnota počítadla dostupných vláken.

Je nutno poznamenat, že čekání na příchod požadavku na exekuci s použitím objektu monitor je nejrychlejším a nejlepším řešením, které lze za daných podmínek aplikovat. Existují i jiné, na synchronizaci méně náročné techniky, jako uspání vlákna metodou `Sleep` nebo kombinace metod vlákna `Pause` a `Resume`. Tyto zmíněné přístupy nejsou ale vhodné z důvodů vysokých časových nároku na svůj běh.

Samotné vykonávání požadavku exekučním vláknem je zobrazena na obrázku 4.4. Jak je patrné, situace je podobná jako v případě akcí prováděných metodou `Init`. Pro synchronizaci běhu je vytvořen objekt monitor nazvaný `ExecutionMonitor`. Pro uskutečnění exekuce, je nutno nejprve daný požadavek získat ze sdílené vstupní fronty. Poté je vytvořeno a spuštěno pomocné vlákno a primární vlákno začíná čekat na příchozí signalizaci podmínky, kterou je dokončení běhu sekundárního vlákna. Pomocné vlákno spouští konkrétní požadavek závoláním jeho vstupní metody `Run`. Po dokončení všech akcí daného požadavku je pomocné vlákno ukončeno a signalizuje své dokončení vláknu primárnímu. Výhodou vykonávání požadavků v pomocném vlákně je možnost



Obrázek 4.3: Postup vytváření exekučních vláken

kontroly délky exekuce daného požadavku[1]. Další výhodou je, že ikdyž požadavek vyvolá za svého běhu nějakou závažnou chybou, která způsobí pád daného pomocného vlákna, neohrozí to existenci exekučního vlákna. Proto toto žešení je velice rafinované a odolné vůči chybám, jelikož exekuční vlákno nemůže být ohroženo například zacyklením nebo fatální chybou způsobenou vykonávaným požadavkem (nebo jakoukoliv jinou chybovou situací).

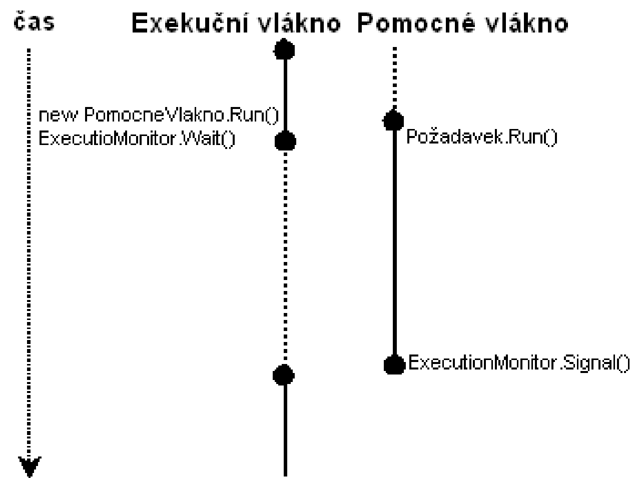
Samotné vkládání do vstupní fronty je implementováno metodou `CommandEnqueue` třídy `CommandExecutor`. Tato metoda zjistí (metodou `IsSomeThread`), jestli je dostupné nějaké volné vlákno, které může vykonat požadavek. Pokud existují nejaká nevyužítá vlákna, je signalizována změna podmínky, kterou je příchod požadavku do sdílené fronty. Pro signalizaci je použita metoda `Signal` synchronizačního objektu monitor. Při více vláknech čekajících na dané podmínce je příjemce náhodný. V této aplikaci to ale nevadí, protože je jedno, které vlákno z množiny neaktivních vláken je vybráno.

4.2.2 Vykonávání uživatelských funkcí

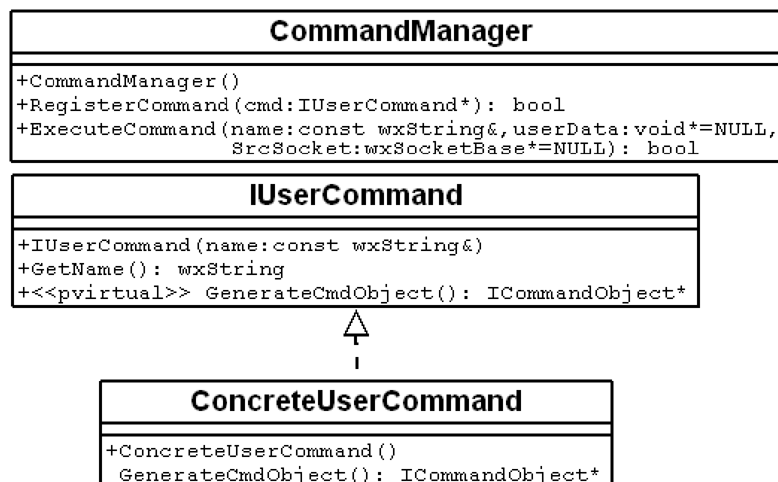
Je nutno implementovat mechanismy, které umožní spravovat a vykonávat uživatelem definované funkce přímo v jádru systému. Toho poté využije i konkrétní implementace IM protokolu pro registraci a spouštění sebou definovaných funkcí. Je nutno implementovat průhledné a jednoduché řešení, které plně využije možností paralelní exekuce s pomocí konceptu thread pool, který byl popsán v minulé kapitole. Funkcionalita mechanismů uživatelem definovaných funkcí je poskytována přimo jádrem serveru.

Pro správu uživatelem definovaných funkcí, jak je patrné na obrázku 4.5, byla definována třída

[1]Misto metody `Wait` lze aplikovat metodu `WaitTimeout`, která po uplynutí daného maximálního času vynutí ukončení sekundárního vlákna a automaticky signalizuje podmínku, na kterou se čeká.



Obrázek 4.4: Spouštění dotazu exekučním vláknem



Obrázek 4.5: Struktura tříd implementujících vykonávání uživatelských funkcí

`CommandManager`. Instance této třídy umožňuje registrovat uživatelem definované funkce, implementované konkrétní instancí třídy IUserCommand. Konkrétní instance třídy IUserCommand, která specifikuje své jméno parametrem konstruktoru, je pomocí volání metody `RegisterCommand` třídy `CommandManager` registrována pod specifikovaným jménem mezi uživatelem definované funkce. Všechny registrované funkce lze volat pomocí metody `ExecuteCommand`, které je jako parametr poskytnut název volané funkce. Jako volitelné parametry lze poskytnout ukazatel na uživatelem definovaná data a ukazatel na schránku, který může specifikovat zdroj příchozí zprávy.

Všechny funkce jsou interně registrovány do nativní struktury `wxWidgets` `wxStringHashMap`, která implementuje obdobu klasické C++ struktury `std::hashmap` s klíčem v podobě řetězce a záznamem typu `IUserCommand*`. Díky využití tohoto přístupu je hledání dané funkce podle jména mnohem rychlejší, nežli například s použitím klasického pole. Pro interní exekuci je použita instance již dříve popsané třídy `CommandExecutor`, která umožňuje paralelní exekuci s využitím principu thread pool.

Jediným problémem, který byl popsán již v kapitole 2.3.2 je problém nazvaný race conditions. Z teoretického hlediska je možná situace, kdy bezprostředně za sebou vykonáme dvě funkce, jejichž výsledek bude zapsán do stejné schránky. Ikdyž tyto požadavky byly vloženy do vstupní fronty třídy `CommandExecutor` bezprostředně za sebou a v daném pořadí, díky paralelnímu vykonávání více než jedním exekučním vláknem mužou být požadavky vykonávány zároveň a to každý jiným vláknem. Poté může nastat situace, ve které dojde k dokončení požadavku a následnému zápisu do schránky v jiném pořadí než by se očekávalo. Toto může způsobit promíchání příchozích zpráv. Tento problém je velmi těžko řešitelný, jelikož ve většině případů je až v době exekuce známa cílová schránka. Z důvodů časových, rozsahu projektu a univerzálnosti řešení, problém nebyl řešen. Je nutno podotknout, že i jiné implementace principu thread pool v .NET nebo jazyce Java tento problém neřeší. V konkrétním aplikačním serveru, který je popisován touto prací, výskyt tohoto problému nebyl zatím zaznamenán, což ale nevylučuje možnost jeho výskytu. Důvodem, proč k tomuto problému nedocházi je fakt, že pravděpodobnost exekuce dvou požadavků se stéjnou cílovou schránkou zároveň je hodně nízká, již z důvodu frekvence posílání dotazů na server přímo klientem. Toto ale nevylučuje možnost výskytu této, v dané implementaci hodně málo pravděpodobné situace. Pro naprosté vyloučení tohoto problému je možno provádět veškerou exekuci pomocí jen jednoho exekučního vlákna, což do značné míry neovlivní rychlost na jednoprocesorovém počítači (viz. kapitola 2.3.2).

Uživatelskou funkci lze definovat vytvořením konkrétní třídy poděděné z třídy IUserCommand. U takto vytvořené třídy je nutno specifikovat v konstruktoru unikátní název dané třidy, jimž bude identifikována při registraci. Dále je nutno implementovat metodu `GenerateCmdObject`, která svým učelem napodobuje návrhový vzor Factory. Zmíněná metoda může generuje různé instance konkrétních tříd na základě rozhodnutí podle uživatelem specifikovaných dat.

Pro ilustraci uvažujme konkrétní třídu `SendMessage`, která je poděděná ze třídy `IUserCommand`. Je vytvořena instance této třídy a ta je registrována jako uživatelská funkce. Jestli budeme chtít vykonat tuto funkci, je spuštěna zmíněná metoda `GenerateCmdObject`, jejiž výsledek bude vložen do vstupní fronty exekutoru. Tímto způsobem může například podle uživatelem specifikovaných dat metoda `GenerateCmdObject` v uvažované třídě vracet různé objekty, které implementují různé funkce. Například se v jedné situaci může jednat o instanci třídy `SendNormalMessage` a v jiné například o instanci `SendEncryptedMessage` (obě jsou poděděny ze třídy `ICommandObject`). Toto umožňuje definovat uživatelské funkce, které po zvážení určitých podmínek mohou generovat různé požadavky.

4.3 Operace se schránkami

Hlavní úlohou serverového jádra je správa uživatelských spojení a poskytování operací umožňujících manipulaci se schránkami. Jádro poskytuje také mechanismy umožňující uchovávat informace o právě připojených uživatelích. Pro pochopení některých částí, jako například příjmu dat, který je v konečném důsledku prováděn až metodou konkrétní instance třídy protokolu, je nutno se seznámit s kapitolou 5.

4.3.1 Správa spojení

Server má v aktuální implementaci otevřeny dva porty. Oba tyto porty lze využít pro připojení klientů. Jeden z nich je ale nastaven jako výchozi pro provádění veškeré síťové komunikace. Druhý je vyhrazen pro posílání souborů, které se realizuje přímo přes server[2].

[2]Mimo tuto možnost existuje způsob posílání souborů přímo v režimu peer to peer. Negociace tohoto typu přenosu ale probíhá na standartním portu a do následného přenosu již server není zapojen.

Jakéhokoliv klienta, který se připojí k serveru je potřeba obsloužit. Célá koncepce *WxWidgets* je postavena na principu posílání zpráv (events). Kdykoliv se připojí klient, nastane zpráva o této události. Tuto zprávu je nutno obsloužit (v této implementaci metodami jádra) a provést potřebné akce, které zahrnují vytvoření nové schránky pro daného uživatele. Poté již může docházet k posílání a příjmu dat způsobem, který je popsán v kapitole 4.3.3.

Nutné je také ošetřit situaci, ve které dojde k náhlému uzavření spojení klientskou stranou. Toto může nastat při pádu klientské aplikace nebo chybě síťového spojení. V takové situaci je, stejně jako v případě připojení nového klienta, generována zpráva. Tato zpráva je obsloužena metodou jádra, která korektně ošetří zrušení dané schránky a informuje o této skutečnosti registrovanou třídu protokolu. Protokol v reakci na toto sdělení může například smazat danou schránku z registru schránek připojených uživatelů.

4.3.2 Registr připojených uživatelů

Bylo nutno poskytnout mechanismy, které umožňují spravovat a manipulovat s informacemi o aktuálně přihlášených uživatelích. Pro tento účel byla vytvořena třída `ConnectionInfo`, která je hojně využívána konkrétní implementací protokolu.

Uživatel je v rámci této třídy identifikován pomocí unikátního alfanumerického řetězce. Tento řetězec obsahuje v aktuální implementaci uživatelovu e-mailovou adresu. Uživatelský záznam je representován v rámci třídy `ConnectionInfo` instancí třídy `ConnectionInfoEntry`. Tato třída může obsahovat předem neurčený počet schránek. Poté lze s těmito schránkami jakkoliv manipulovat (mazat nebo vkládat). Tímto lze zaručit možnost vlastnictví jedním uživatelem více než jedné schránky (tohoto je využito při vícenásobném přihlášení jednoho uživatele).

Na obrázku 4.6 je zjednodušený model třídy `ConnectionInfo`. Jak je patrné, jedná se o třídu, která je navržena na základě návrhového vzoru Singletone. Toto umožňuje vytvoření jen jedné unikátní instance v rámci celého programu. Pro vkládání schránky lze využít metody `InsertSocket`. Tato metoda vloží novou instanci třídy `UserInfoEntry`, která representuje uživatelovy sockety do interního registru třídy `ConnectionInfo` a vloží do ní poskytnutou schránku. Interní registr všech uživatelských záznamů je realizovám pomocí nativní struktury `wxStringHashMap`, kde klíčem je řetězec (zde e-mailová adresa uživatele) a záznamem je konkrétní instance třídy, která obsahuje všechny uživatelovy schránky. Při existenci záznamu s daným jménem, je schránka přidána jako další uživatelova schránka.

ConnectionInfo

```
-ConnectionInfo()
+GetInstance(): ConnectionInfo*
+InsertSocket(key:const wxString&,socket:wxSocketBase*): bool
+GetSocket(key:const wxString&): wxSocketBase*
+RemoveSocket(socket:wxSocketBase*): bool
+RemoveSocket(key:const wxString&,socket:wxSocketBase*): bool
+GetIdBySocket(key:wxSocketBase*): wxString
```

Obrázek 4.6: Struktura třídy registru uživatelských schránek

Dále pro získání všech schránek daného uživatele je využívána metoda `GetSocket`. Tato metoda volá stéjnojmennou metodu třídy `ConnectionInfoEntry`. Tato metoda při svém opakovaném volání vrací postupně všechny uživatelovy schránky. Navrácená hodnota *NULL* signalizuje, že již nejsou žádné další záznamy k dispozici. Takovéto řešení je velice flexibilní co se týká paralelního

přístupu. V rámci struktury obsahující dané schránky je vytvořena sdílená struktura mutex. Ta je zamknuta v době prvního volání funkce `GetSocket` a při posledním volání je odemknuta. Toto řešení je v dané situaci nejlepší. Vracení ukazatelů nebo iterátorů není v tomto případě z důvodu zachování integrity vůbec bezpečné.

Poté je definována metoda `RemoveSocket`, která, jak již název napovídá, smaže danou uživatelskou schránku. Pro zjištění jména vlastníka dané schránky lze použít metodu `GetIdBySocket`, která podle specifikované schránky vrátí přidružený unikátní klíč.

Pro zachování integrity je nad všemi zmíněnými metodami třídy `ConnectionInfo` definována sdílená synchronizační struktura mutex.

4.3.3 Posílání a příjem

Tato část pojednává o základních principech posílání a příjmu dat. Při implementaci bylo nutno zajistit nezávislost přijímací funkce od použitého protokolu. Každý protokol totiž může specifikovat jiný formát zpráv.

Příchod dat do schránky je signalizován, tak jako většina skutečností v dáne implementaci, pomocí vzniku události. Pro obsluhu této události je definována v jádru serveru obslužná funkce. Jedinou úlohou této funkce je sdělení konkrétní třídě protokolu, že daná schránka obsahuje data, která mohou být načtena. Samotné načítání je prováděno až pomocí funkce konkrétního protokolu. Jakým způsobem přesně jsou načítána data, lze zjistit přímo v kapitole 5.1, která je věnována protokolu.

Posílání dat je značně složitější. Metoda pro posílání dat byla implementována přímo jádrem serveru. Tento problém lze řešit v tomto místě, jelikož zde nemusí být známa sémantika. Celé posílání je řešeno již dříve definovanou třídou `CommandExecutor`, která implementuje koncepci paralelního vykonávání s použitím přístupu zvaného thread pool. Pro samotné posílání byl definován požadavek, který je reprezentován třídou `MessageSendCommand` (poděděnou ze třídy `ICommandObject`). Tento požadavek zajistí poslání daných dat do specifikované schránky.

Celý systém posíláni spočívá v zavolání metody jádra, které se jako parametr předá určitá struktura obsahující data a cílovou schránku. Tato metoda interně z daných dat vytvoří instanci třídy `MessageSendCommand`. Tento objekt se poté vloží do exekuční fronty instance třídy `CommandExecutor`, která je odpovědná za posílání dat. Zde se daný požadavek spustí a pošle data.

Využití již dříve definované třídy `CommandExecutor` značně urychluje výsledné řešení. Problémem bylo zvolit počet exekučních vláken. Na základě kapitoly 2.3.2 bylo rozhodnuto o použití jednoho exekučního vlákna. Jedno vlákno se použije i na víceprocesorových strojích, protože vykonávaná operace je velice rychlá a použití více než jednoho vlákna by výsledné řešení udělalo jen zbytečně pomaléc (hlavně z důvodu složitejší mezivláknové synchronizace).

4.3.4 Struktura jádra

Jádro tvoří základ, nad kterým je vystavěna konkrétní třída protokolu. Všechny funkcionality jádra byly již popsány v dřívějších kapitolách. Cílem této části je shrnout vše, co jádro nabízí.

Jádro zajišťuje všechny tyto funkce:

- **Obsluhu stavu spojení** – Definuje akce prováděné při připojení uživatele a také při ztrátě spojení.
- **Manipulaci se schránkou** – Obsluhuje situace příchodu dat a posílá data.
- **Práci s uživatelem definovanými funkcemi** – Zajišťuje registraci a vykonávání těchto funkcí.

- **Registrovat protokol** – Umožňuje registrovat konkrétní instanci třídy protokolu.
- **Správu připojených uživatelů** – Definuje mechanismy umožňující spravovat informace o právě připojených uživatelích.

Další funkcionality lze definovat pomocí protokolu a tříd, kterých využívá. Výstavbou dalších rozšíření nad stávajícím jádrem se zabývá následující kapitola.

Kapitola 5

Výstavba komunikačního protokolu nad jádrem

Tato kapitola buduje na serverovém jádře protokol a další rozšířené funkcionality. Je definován konkrétní protokol a k němu přidružené uživatelem definované funkce. Pro práci s pernamentními daty je vytvořeno databázové spojení a způsob representace databázovych dat přímo v lokální paměti serveru.

5.1 Protokol

Protokol je částí, ve které je implementována veškerá sémantika síťové komunikace. Jak již bylo řečeno dříve, jádro tvoří samostatnou část, čili je nezávislé na jakýchkoliv sémantických akcích. Pro samotné použití protokolu je nutno vytvořit konkrétní třídu poděděnou ze třídy `ServerProtocol`. Tato se poté zaregistruje jako konkrétní implementace protokolu přímo do jádra serveru.

Bázová třída protokolu definuje 3 základní metody (čistě virtuální), které musí být implementovány všemi konkrétními třídami dědicími toto rozhraní. Metoda `RegisterAllCommand` je volána jádrem hned po registraci daného protokolu. Její úlohou má být registrace všech uživatelem definovaných funkcí. Samozřejmě, jelikož se tato metoda spoušti jen jednou a to při zavedení jádra serveru, může obsahovat i kroky inicializace protokolu. Další z definovaných metod je `Receive`. Tato metoda je notifikována jádrem vždy při příchodu dat do jakékoliv schránky. Zde je nutno načíst data s ohledem na sémantiku daného konkrétního protokolu. Poslední z vyjmenovávaných metod je `Lost`. Touto cestou je třída protokolu informována o pádu spojení se specifikovaným klientem. Zde jsou provedeny všechny kroky ošetřující zneplatnění dané schránky.

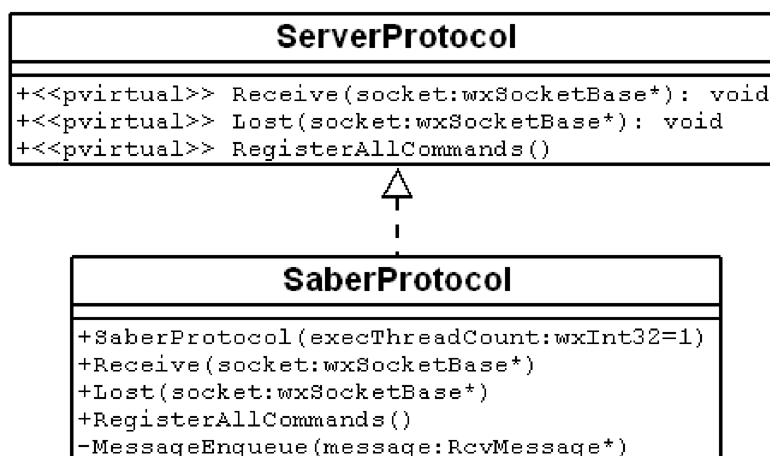
5.1.1 Konkrétní protokol

Obsahem této části je popis třídy konkretního `Saber protokolu`. Jelikož návrh protokolu nebyl náplní tohoto zadání práce, veškeré dodatečné informace lze získat v materiálu jejího tvůrce v pramenu [3].

Protokol používaný tímto projektem je formátu XML zprávy. Jak je patrné níže, samotná zpráva se skládá z hlavičky a těla. Hlavička obsahuje základní informace o odesilateli a příjemci a také rozšířený typ zprávy, na základě kterého je tato zpráva identifikována přímo parserem protokolu. Tělo zprávy může obsahovat jakékoliv informace předem specifikovaného formátu. Výsledná zpráva může vypadat takto:

<message length="218" type="plain">
<head>
<extype val="message"/>
<from id="noobiQ@saber.cz"/>
<to id="noob99@saber.cz"/>
</head>
<body>
<mess-text>Hi, this is the test message. Enjoy. SABER TEAM.</mess-text>
</body>
</message>

Pro začlenění `Saber protokolu` do tohoto projektu, bylo nutno vytvořit konkrétní třídu `SaberProtocol` poděděnou ze třídy `ServerProtocol` (obr. 5.1). V této třídě bylo nutno implementovat všechny tři metody, které byly popsány výše. Dále bylo potřeba definovat mechanismy pro analýzu sémantiky protokolu. Výsledný protokol je poté zaregistrován v jádru.



Obrázek 5.1: Schéma definice konkrétního protokolu

Pro potřeby sémantické analýzy byla vyvinuta speciální třída XML parseru. Tato třída umožňuje získávat informace z příchozí zprávy a také je schopna vytvářet zprávy nové. Je využita většinou funkcí konkrétní třídy protokolu.

Jak již bylo zmíněno, je potřeba implementovat všechny 3 metody definované bázovou třídou. První z nich je `Receive`. Zde jsou definovány mechanismy, které jsou schopny načíst zprávu v daném formátu specifikovaném konkrétním protokolem. Při příchodu jakýchkoliv dat, je tato metoda notifikována jádrem serveru. Zde se spustí separátní vlákno, které se pokusí načíst zprávu. Z důvodu ušetření serverových zdrojů jsou tímto vláknem načteny úplně všechny zprávy, které se aktuálně nachází v dané schránce. Celé načítání funguje na principu zjištění délky příchozí zprávy z její hlavičky a načtení dat dané velikosti.

Pro obsluhu sémantiky jsou protokolem zaregistrovány do jádra serveru uživatelem definované funkce. Registrace se děje na požádání jádra metodou `RegisterAllCommands`. Protokolem jsou registrovány funkce, které se starají o:

- posílání zpráv

- přenos souborů
- práci s historií
- manipulaci seznamem kontaktů
- práci s uživatelským účtem
- stav připojení uživatele

Pro samotnou sémantickou interpretaci zprávy byla vytvořena speciální třída `SaberParsingCmd` poděděná ze třídy `ICommandObject`. Všechny sémantické operace jsou prováděny instancí třídy `CommandExecutor`, která je součástí třídy protokolu a implementuje mechanismus thread pool s jedním exekučním vláknem.

Po načtení zprávy třídou protokolu je vytvořena instance třídy `SaberParsingCmd`, která implementuje kroky, které v závislosti na rozšířeném typu zprávy volají danou uživatelem definovanou funkci, která již provede potřebné akce. Následně je tento objekt vložen do vstupní fronty instance třídy `CommandExecutor` (popsána výše) a je vykonán.

Protokol a jím registrované funkce musí podporovat správu připojených uživatelů. Také je nutností existence autentizačních mechanismů, umožňujících kontrolu vlastníka dané schránky. Toto zamezuje případnému útočníkovi procházet historii jiného uživatele nebo posílat zprávu s podvrženým uživatelským jménem.

Další nutně implementovanou metodou je `Lost`. Ta je volána jádrem a musí ošetřit situaci, ve které dojde k pádu spojení s uživatelem. Proto zde je nutno korektně zneplatnit schránku a odebrat ji z registru právě přihlášených uživatelů.

5.2 Databáze

K této aplikaci je databáze připojená pomocí rozhraní ODBC. Nad tímto rozhraním byly definovány struktury pro práci s databázovým spojením a tabulkami. Bylo navrženo optimální schéma uložení dat s ohledem na rychlost prováděných databázových operací.

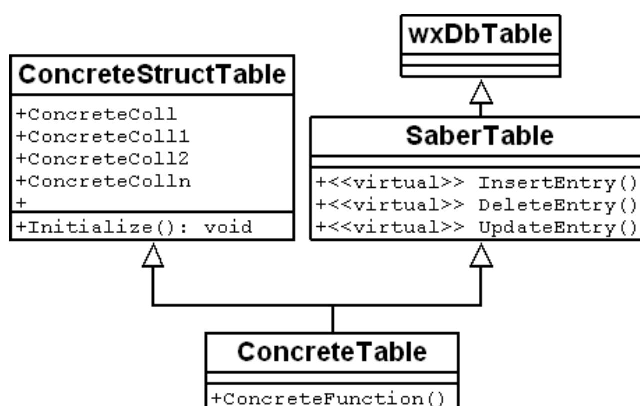
5.2.1 Implementace databázových operací

Pro účel správy databázových spojení byla implementována třída `ConnectionManager`. V této třídě je využito principu znovupoužitelnosti databázového spojení. Jak je vidět na obrázku 5.2, zavolání metody `Start` jsou poskytnuty parametry databázového spojení, které jsou interně uloženy. V tomto okamžiku už je možno generovat databázová spojení. Celkový princip této třídy spočíva v zamezení zbytečného vytváření a rušení databázových spojení, protože tyto operace jsou zbytečně náročné. Třída si udržuje nějaký, předem určený počet otevřených spojení. Každá funkce, která potřebuje databázové spojení, může o ně požádat voláním metody `GetConnection`. Tato funkce vrátí databázové spojení z množiny otevřenych spojení nebo v situaci, kdy není žádné k dispozici, je vytvořeno nové. V okamžiku, kdy dané databázové spojení už není potřebné, je navráceno zpět do množiny dostupných spojení voláním metody `ReleaseConnection`. Zásadní problém může nastat v situaci, kdy funkce před svým ukončením neoznačí vypůjčené spojení za dostupné. V takovéto situaci dochází k nenávratné ztrátě daného spojení. A jelikož je databázové spojení omezeným zdrojem, může tato situace při svém častém opakování způsobit veliké problémy.

Konkrétní tabulky databáze jsou implementovány způsobem naznačeným na obrázku 5.3. Principem všech databázových operací v dané aplikaci je mapování sloupců databáze do proměnných uložených

Obrázek 5.2: Schéma správce databázových spojení

v paměti. Poté je možné vykonat dotaz a použít funkcionalit ODBC ovladače a databázových kursorů. Posunem kursoru lze procházet navracenou odpověď a nahrávat ji do proměnných uložených v paměti. V této situaci je vždy vrácena hodnota jen jednoho řádku, což je výhodné, jelikož v některých případech není potřeba pracovat s celou odpovědí. Tímto přístupem je snížen počet přenesených dat.



Obrázek 5.3: Struktura implementace tabulek databáze

Pro implementaci konkrétní tabulky je nutno vytvořit třídu poděděnou ze třídy definující paměťové proměnné a třídy `SaberTable`. První třída, ze které se dědí (na obrázku třída `ConcreteStructTable`) definuje paměťové proměnné, do kterých se nahrávají data. Třída `SaberTable` rozšiřuje funkcionality nativní třídy `wxDbTable` o možnost vložení dat obsažených v paměťových proměnných do databáze, smazání z databáze dat, která jsou aktuálně načtena a možnost aktualizace hodnot v databázi. Mimo těchto operací, může vytvořená třída definovat metody, které uskutečňují dotazy nad databází pomocí jazyka SQL.

5.2.2 Struktura tabulek

Struktura databázových tabulek byla navržena s ohledem na rychlost prováděných operací. Všechny tabulky, které jsou v databázi, zachycuje obrázek 5.4. Databáze obsahuje tyto tabulky:

- **cl_content** – obsahuje seznamu kontaktů všech uživatelů. Podle této tabulky se rozesílají zprávy o změně stavu uživatele. Její obsah je měněn při každé změně uživatelova seznamu kontaktů.

- **history** – tabulka obsahující historii všech zpráv. Z důvodu častého hledání byl vytvořen index.

- **status** – obsahuje stavy všech připojených uživatelů. Je zde uložen typ a text aktuálního stavu.
- **transfer** – representuje seance přenosu souborů. Tabulka obsahuje informace o všech seancích serverového přenosu souborů, které ještě nezačaly.
- **users** – obsahuje informace o všech registrovaných uživatelských účtech. Zde jsou uloženy oba asynchronní klíče i seznam kontaktů daného uživatele.

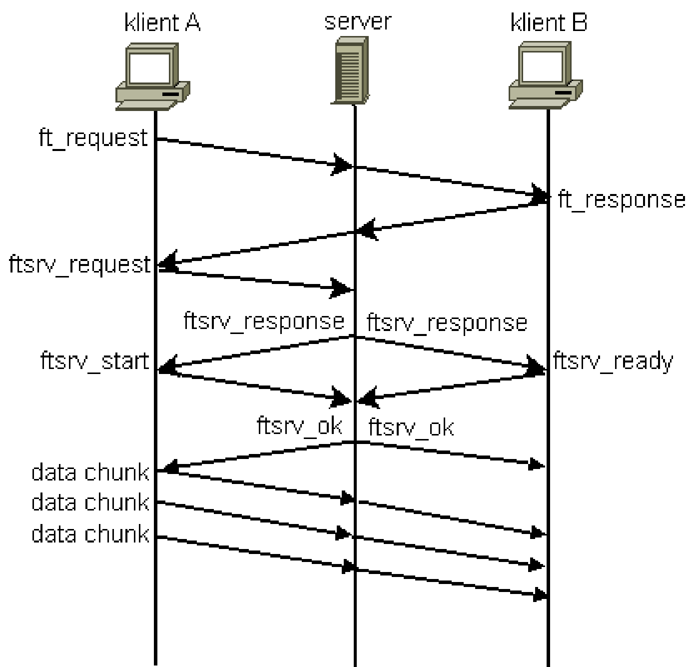
Obrázek 5.4: Struktura tabulek databáze

Databázové tabulky jsou optimalizovány s ohledem na rychlost prováděných operací. Další nutností je výběr vhodných a rychlých dotazů. Je minimalizován počet agregačních a vnořených dotazu.

5.3 Posílání souborů

Jedním z požadavků zadání byla nutnost implementace mechanismů umožňujících posílání souborů. V aktuální specifikaci protokolu je možno posílat soubory dvojím způsobem. První způsob realizuje přenos bez účasti serveru v režimu peer to peer. Tento přenos je nejběžnější, jelikož většina klientů ma možnost se mezi sebou bez zábran spojit. Záměrem této kapitoly neni popisovat tento typ přenosu. Více o tomto případě lze zjistit v [3]. Další možnou alternativou přenosu je posílání souboru s účastí serveru. Toto řešení je aplikováno v situaci, kdy dva klienti nemohou navázat přímé spojení. Diskutovaná situace může nastat ve většině případů při použití překladu adres (NAT). Samotný přenos souborů je realizován pomocí posílání shluků dat (chunks). Data jsou posílána po

segmentech předem dané velikosti. Obě zůčastněné strany musí znát parametry daného přenosu. Těmito parametry je velikost shluku, jejich počet a velikost poslední části. Soubor lze v dané situaci posílat i uživateli, který je přihlášen vícenásobně. Všem instancím příjde požadavek na přenos a jen klient, který tento požadavek potvrdí jako první se stane účastníkem přenosu.



Obrázek 5.5: Schéma komunikace při posílání souboru s účastí serveru

Obrázek 5.5 popisuje situaci posílání souboru s účastí serveru. V dané situaci se pokouší klient A poslat soubor klientu B. Prvním krokem celé komunikace je negociace přenosu souboru. Klient A posílá na server požadavek na poslání souboru (ft_request). Požadavek je přeposlán připojené instancí klienta B. Příjemce odpovídá na požadavek posláním zprávy rozšířeného typu ft_response. Daná zpráva potvrzuje nebo zamítá požadavek na přenos. Server tuto zprávu přijme a přepošle ji instancí klienta A. Klient A akceptuje odpověď. V případě, že odpověď je pozitivní, byla zdárně dokončena negociace přenosu.

Po zdárném dokončení negociace dochází k pokusu přímého spojení obou klientů a následného poslání souboru. Samotný přenos je poté uskutečněn bez komunikace se serverem. V zde popisovaném případě tento pokus selhal a je nutno uskutečnit přenos s účastí serveru.

Posílající strana informuje server o potřebě uskutečnění přenosu posláním zprávy ftsrv_request. Server přenost buď odmítne nebo potvrdí posláním zprávy rozšířeného typu ftsrv_response. V případe potvrzení přenosu obsahuje zpráva také adresu a port, kam se mají oba účastníci připojit. Jak již bylo diskutováno v kapitole 4.3.1, server má pro přenos souborů vyhrazen speciální port.

Zde bylo nutno vytvořit speciální způsob umožňující kontrolu přenosů. Byla implementována seance, která umožňuje čekat na připojení obou zůčastněných stran. Seance je vytvořena při poslání zprávy o připojení na port pro přenos souborů oběma účastníkům. Všechny informace o ní jsou uloženy v databázi a identifikuje se časové razítko (V dané situaci jde o časové razítko rozšířené o čas v milisekundách.). Délka trvání seance, čili maximální čas čekání na připojení obou zůčastněných stran, je nastavitelná. Po uplynutí daného času je záznam z databáze automaticky smazán a seance se stane neplatná. Všechny pokusy o přístup k neplatné seanci jsou zamítnuty.

První z klientu se připojí k serveru na daný port. Jestli se jedná o posílajícího, je uložena jako proměnná seance informace o přenášených shlucich dat (počet a velikost) a také se ukládá do databáze v podobě binárních dat ukazatel na klientovu schránku. V případě, že se připojí jako první příjemce, je uložen do databáze jen ukazatel na jeho schránku. Informace o posílaných datech se neukládají, jelikož příjemci nejsou zatím známy. U připojení druhého klienta je situace podobná a v obou případech je nutno získat proměnnou seance, která obsahuje ukazatel na schránku klienta, který se připojil jako první. Jestli je druhým připojeným příjemce, je nutno jestě z databáze získat informace o parametrech přenášených dat. Poté je vytvořena a poslána oboum klientům zpráva obsahující informace o parametrech přenosu a přenos může začít. Posílající odesílá shluky dat, které jsou potom serverem přeposlány příjemci. Celý přenos se uskutečňuje v separátním vláknu, které bylo vytvořeno pro tento účel.

Jak je patrné, samotný přenos souboru nezatěžuje server, jelikož většina požadavků je uskutečněna bez jeho účasti. Přenos je uskutečňován pomocí posílání shluků dat, proto i v případě přenosu s účastí serveru není jeho celkové zatížení vysoké. Problémem může ale v této situaci být zvýšené využití přenosového pásma.

5.4 Kódování

Tato část krátce popisuje kódování používané serverovou aplikací. Pro hlubší pochopení této problematiky je možno sáhnou po publikaci [3].

XML protokol, který je použit pro komunikační kanál, používá kódování *UTF-8*. Toto kódování je pro tento účel vhodné z důvodu variabilní délky, která minimalizuje velikost zprávy. V aplikaci jsou data representovány datovým typem *wxString*, který je interně ukládá data do kódování *UCS-2*. Z důvodu zrychlení práce s databází bylo vybráno kódování databáze *UCS-2*. V daném případě není potřeba žádných konverzních funkcí. Implementace této aplikace ale nabízí možnost přepsání dvou metod pro konverze z a do databázového kódování. V konečném důsledku je možné vytvořit konverzní funkce mezi *UCS-2* kódováním a jakýmkoliv jiným kódováním databáze.

5.5 Zprávy

Základním principem serveru je posílání textových zpráv. Princip posílání zpráv je složitější, než se na první pohled zdá. Uvažujme posílání textové zprávy uživatelem A uživateli B. Uživatel A pošle zprávu a ta je přijata serverem, který provede potřebné operace. Mezi toto patří uložení zprávy do historie poslaných zpráv. Tato zpráva je také označena jako nedoručená. Poté je potřeba zjistit, jestli je příjemce připojen k serveru, jestli ano, je mu poslána daná zpráva a čeká se na potvrzení jejího přijetí. Při obdržení potvrzení je v historii tato zpráva nastavena jako odeslaná.

Jestli daný uživatel není v daném okamžiku připojen, při jeho připojení mu jsou odeslány všechny zprávy, které mu ještě nebyly doručeny. I v tomto případě je vyžadováno potvrzení o doručení, což zamezuje případnému nedoručení zprávy.

Server umožňuje připojení více instancí stejného účtu. V tomto případě jsou zprávy doručovány všem instancím daného účtu. Záznam o zprávě v historii je samozřejmě vytvořen jen jeden.

Z hlediska implementace je vytvořena a zaregistrována jedna instance konkrétní třídy poděděné z třídy `IUserCommand`. Tato uživatelem definovaná funkce je volána protokolem pro zajištění poslání zprávy. Požadavky generované touto třídou vykonávají všechny výše popsané operace.

5.5.1 Historie zpráv

Jak již bylo zmíněno, všechny posílané zprávy se ukládají v historii. V tabulce databáze `history` se nacházejí všechny zprávy, které daný uživatel poslal nebo příjmul. Jsou zde i zprávy, které zatím nebyly doručeny.

Funkcionalitou serveru, která je podporována v klientském grafickém rozhraní, je možnost vyhledávání v historii zpráv. Jelikož všechny příchozí zprávy jsou ukládány do lokální historie klienta, existují prakticky paralelně dva typy historie. Klientská historie nemusí ale obsahovat všechny položky, proto směrodatnou je historie na serveru, která obsahuje vše. V historii lze vyhledávat podle data nebo podle textu obsaženého ve zprávě. Historie je vždy omezena na uživatele u kterého ji zobrazujeme. Při výběru zobrazení historie zpráv s uživatelem `marek@saber.cz` uvidíme jen zprávy vyměněné s tímto uživatelem.

Pro samotné hledání lze specifikovat 3 typy. Prvním je hledání podle daných kritérií přímo v lokální historii klienta. Toto řešení hledá bez účasti serveru a nikdy nezaručí, že v lokální historii budou archivovány všechny zprávy. Proto výsledky tohoto hledání nemusí být kompletní. Dalším je vyhledávání přímo na serveru. Toto řešení je náročnější na sít'ový přenos a serverové zdroje. Výsledkem je ale vždy kompletní odpověd', protože na serveru je zálohována veškerá komunikace. Existuje i méně náročné řešíní, ve kterém se vyhledává na serveru i klientu zárověň. Klient zjistí počet záznamů, které splňují daná kritéria a sdělí to serveru, který udělá tuto stéjnou operaci. V případě, že na serveru je stejný počet záznamů jako na klientu, je mu tato skutečnost sdělena a vyhledává se lokálně. V opačném případě se vyhledá na serveru a výsledek je odeslán klientu.

5.6 Šifrování

Šifrování z pohledu klienta je zachyceno v publikaci [3]. Tato část se zabývá touto problematikou z pohledu serveru.

Hlavní úlohou serveru v této situaci je správa dvojice RSA asynchronních klíčů. V aktuální implementaci server neprovádí žádné operace šifrování. Šifrované textové zprávy jsou jen přeposílány příjemci. Uživatelovo heslo je jednocestně šifrováno algoritmem SHA1. Heslo v této podobě je i uloženo v databázi.

Kapitola 6

Závěr a zhodnocení

6.1 Směr dalšího vývoje

Tato realizace systému obsahuje jen zlomek zamýšlených součástí. Z časového hlediska a s ohledem na rozsah projektu, nebyly všechny části implementovány. Aplikaci je možno rozšířit například těmito funkcionalitami:

- Vytvořením protokolu pro vzdálenou administraci serveru. Nad tímto protokolem je poté možné vystavět například webové rozhraní napsané v jazyce PHP.
- Poskytnutím podpory zásuvných modulů, které mohou rozšiřovat funkcionalitu serveru.
- Zavedením podpory videohovorů.
- Implementováním hlasového přenosu.
- Umožněním posílání souboru skupině uživatelů.
- Poskytnutím možnosti šifrování veškerých zpráv (aktuálně jsou šifrovány jen textové zprávy).

6.2 Závěr

Cílem této práce bylo vytvoření serveru třívrstvé klient–server architektury.

Výsledná aplikace podporuje bezpečné posílání zpráv a souborů. Soubory lze posílat za jakékoliv situace, jelikož je podporován přenos souborů s účastí serveru, který je vždy úspěšný. Dále je uživateli umožněno hledání v historii zpráv. Je možno připojit se k datové vrstvě pomocí rozhraní ODBC.

Všechny povinné body zadání byly úspěšně splněny. Bohužel z důvodu nedostatku času není kód zcela odladěn. Aplikace také ještě nebyla testována na jiném operačním systému než Microsoft Windows. Z povahy přenositelnosti *WxWidgets* ale vyplývá, že by neměl nastat problém s portabilitou na jiné platformy. Připojení pomocí ODBC bylo úspěšně otestováno pro databázový server MySQL. Jelikož je ODBC univerzálním rozhraním, měla by fungovat i jiná databázová řešení.

Literatura

[1] Blaise Barney. Introduction to parallel computing [online]. last modified: 07/10/2006 21:35:51. [cit. 2007-04-30]. Dostupné na URL: <http://www.llnl.gov/computing/tutorials/parallel_comp/>.

[2] David Carmona. Programming the thread pool in the .net framework [online]. june 2002. [cit. 2007-04-30]. Dostupné na URL: <http://msdn2.microsoft.com/en-us/library/ms973903.aspx>.

[3] Jan Fiedor. Systém pro zasílání textových zpráv – klientská část. FIT VUT v Brně, 2007.

[4] Brian Goetz. Java theory and practice: Thread pools and work queues [online]. 01 jul 2002. [cit. 2007-04-30]. Dostupné na URL: <http://www-128.ibm.com/developerworks/java/library/j-jtp0730.html>.

[5] MySql.com. General information about odbc and connector/odbc [online]. [cit. 2007-04-30]. Dostupné na URL: <http://dev.mysql.com/doc/refman/5.0/en/myodbc-general-information.html>.

[6] Ken North. Comparative performance tests of odbc drivers and proprietary database application programming interfaces [online]. august 3, 1995. [cit. 2007-04-30]. Dostupné na URL: <http://ourworld.compuserve.com/homepages/Ken_North/API_test.htm>.

[7] Wikipedia. Amdahl's law [online]. last modified 17:42, 28 march 2007. [cit. 2007-04-30]. Dostupné na URL: <http://en.wikipedia.org/wiki/Amdahls_Law>.

[8] Wikipedia. Client-server [online]. last modified 14:06, 28 april 2007. [cit. 2007-04-30]. Dostupné na URL: <http://en.wikipedia.org/wiki/Client/server>.