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 KLIENTSKÁ ČÁST

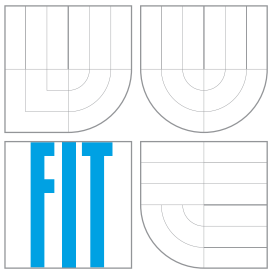
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JAN FIEDOR

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 KLIENTSKÁ ČÁST

INSTANT MESSENGING SYSTEM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — JAN FIEDOR
AUTHOR

VEDOUCÍ PRÁCE — Ing. JAROSLAV RÁB
SUPERVISOR

BRNO 2007

Zadání bakalářské práce/5795/2006/xfiedo00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2006/2007

Zadání bakalářské práce

Řešitel: **Fiedor Jan**
Obor: Informační technologie
Téma: **Systém pro zasílání textových zpráv - klientská 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 klienta.
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: **Jan Fiedor**
Id studenta: 84406
Bytem: Lutyňská 1733, 735 32 Rychvald
Narozen: 02. 04. 1985, Bohumín
(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 - klientská 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: .....

..... Jan Fiedor

Nabyvatel Autor

Abstrakt

V dnešní době existuje mnoho klientů pro komunikaci pomocí textových zpráv. Bohužel je pravdou, že velká část z nich nijak neřeší zabezpečení této komunikace, často se potýká s problémy při přenosu souborů a je spjata se specifickým protokolem. Tato práce si klade za cíl nalézt nejlepší řešení těchto problémů a aplikovat je při implementaci bezpečného a spolehlivého klienta nezávislého na konkrétním protokolu.

Klíčová slova

wxWidgets, MVC, XML, XRC, návrhový vzor composite, crypto++, AES, RSA

Abstract

Nowadays many instant messenging clients exist. However, it is true, that many of them do not solve the security issues connected with communication, often deal with file transfer problems and are linked with specific protocol. This thesis is trying to find out the best solution of those problems and use it to develop secure and reliable client independent of concrete protocol.

Keywords

wxWidgets, MVC, XML, XRC, composite pattern, crypto++, AES, RSA

Citace

Jan Fiedor: Systém pro zasílání textových zpráv
klientská část, bakalářská práce, Brno, FIT VUT v Brně, 2007

Systém pro zasílání textových zpráv
klientská část

Prohlášení

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

.....
Jan Fiedor
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 Marku Gachovi za výbornou spolupráci a tvorbu serverové části tohoto projektu.

Obsah

1 Úvod **3**

2 Teoretická část **4**
2.1 Typy klientů . . . 4
2.1.1 Tlustý klient . . . 5
2.1.2 Tenký klient . . . 5
2.1.3 Hybridní klient . . . 6
2.2 Koncepce klientských aplikací . . . 6
2.2.1 Architektura klient-server . . . 6
2.2.2 Architektura peer-to-peer . . . 6
2.3 Model-View-Controller . . . 9
2.4 XML a XML Resource . . . 10
2.4.1 XML . . . 10
2.4.2 XML Resource . . . 12
2.5 Unicode . . . 12
2.6 Zabezpečení informací . . . 14
2.6.1 Kryptografie . . . 14
2.6.2 Symetrická kryptografie . . . 14
2.6.3 Asymetrická kryptografie . . . 15

3 Návrh řešení **17**
3.1 Formulace úlohy . . . 17
3.2 Doplnění úlohy . . . 18
3.3 Návrh protokolu . . . 18
3.4 Návrh klienta . . . 19
3.4.1 Jádro . . . 19
3.4.2 Spojení . . . 21
3.4.3 Protokol . . . 21
3.4.4 Uživatel a seznam kontaktů . . . 21
3.4.5 Uživatelské rozhraní . . . 21

4 Realizace **23**
4.1 Implementační prostředí . . . 23
4.2 Protokol . . . 23
4.2.1 XML protokol . . . 23
4.2.2 Interní reprezentace . . . 24
4.2.3 Parser . . . 25
4.3 Lokalizace . . . 26

4.4 Uživatelské rozhraní . . . 27
4.4.1 Panely . . . 27
4.4.2 XRC . . . 28
4.5 Seznam kontaktů . . . 30
4.5.1 Implementace modelu . . . 30
4.6 Komunikace . . . 32
4.6.1 Protokol . . . 32
4.6.2 Spojení . . . 34
4.7 Zprávy . . . 35
4.7.1 Posílání a příjem zpráv . . . 35
4.7.2 Historie zpráv . . . 35
4.8 Přenos souborů . . . 36
4.9 Šifrování . . . 37

5 Závěr a zhodnocení **40**
5.1 Směry dalšího vývoje . . . 40
5.2 Závěr . . . 40

Kapitola 1

Úvod

Cílem této práce je navrhnout systém pro zasílání textových zpráv a implementovat k tomuto systému klientskou aplikaci. Systém by měl umožňovat komunikaci pomocí textových zpráv, zpětné zobrazení všech odeslaných a přijatých zpráv pomocí historie a přenos souborů mezi uživateli za jakékoliv situace. Také by měl zajistit bezpečnost komunikace.

Kapitola 2 obsahuje informace o výhodách a nevýhodách jednotlivých typů klientů a jejich využití. Popisuje využití architektonického vzoru MVC v aplikacích. Zabývá se jazykem XML a jeho širokým využitím. Vysvětluje problematiku kódování, která může způsobovat nemalé problémy při přenosu zpráv, a bezpečnosti, na kterou je poslední dobou kladen stále větší důraz.

Kapitola 3 analyzuje jednotlivé požadavky na systém a klienta a předkládá návrh pro realizaci těchto požadavků s vysvětlením výhod tohoto návrhu.

Kapitola 4 popisuje vlastní realizaci klienta. Podrobně rozebírá některé části implementace a jejich výhody a popisuje princip práce některých služeb.

Kapitola 2

Teoretická část

2.1 Typy klientů

Klientem je možné rozumět počítačový systém, který využívá služby nějakého jiného zařízení, například dalšího počítače, na sítí. Poprvé byl tento termín použit pro zařízení, které nemohly samostatně provozovat žádné služby, ale mohly komunikovat s dalšími zařízeními na síti. Příkladem takových klientů jsou terminály. Ty sloužily pouze k vkládání příkazů a dat a k zobrazování výstupu, samy tedy neposkytovaly žádné služby. Tyto terminály ale byly klienty salových počítačů, kterým posílaly vstupní data pro zpracování a přijímaly zpět výsledky, které zobrazily užitateli. Ovšem s nástupem moderních operačních systémů, které umožňovaly multitasking[1] již nebyl počítač omezen pouze na jeden běžící program v jednu dobu. S nárustem počtu běžících aplikací na jednom počítači se tyto aplikace začaly od sebe odlišovat svou funkčností jako samotné počítače a vznikla zde skupina aplikací, která se začala označovat jako klientské aplikace. Tyto aplikace stejně jako klientské počítače samy nemusí poskytovat žádné služby, ale mohou komunikovat s jinými aplikacemi na sítí, které již nějaké služby poskytují. Příkladem takovéto klientské aplikace je klient pro zasílání textových zpráv, který je v této práci diskutován. Tento klient komunikuje se serverovou aplikací, která přijme jeho zprávu a postará se o její doručení cílovému klientovi. Pokud ale nemá klient k dispozici server, není schopen zajistit službu doručení zprávy jinému klientovi.

Lze rozlišit několik typů klientů. Toto dělení je stejné ať již pojem klient vyjadřuje klientský počítač nebo klientskou aplikaci. V tomto textu bude pojem klient vždy reprezentovat klientskou aplikaci. Obecně se uvádějí tří typy klientů [7]:

- **Tlustý klient**
- **Tenký klient**
- **Hybridní klient**

Základní rozdíly mezi těmito třemi typy shrnuje tabulka 2.1.

[1] Termínem multitasking se označuje schopnost operačního systému provádět několik úloh současně. Úlohy se jeví jakoby byly vykonávany paralelně, ale ve skutečnosti jsou vykonávany po částech sériově, kdy každá úloha má k dispozici na určitou dobu procesor. Takovéto chování se označuje jako kvaziparalelismus.

	Tlustý klient	Hybridní klient	Tenký klient
Lokální zpracování dat	✓	✓	✗
Lokální uložení dat	✓	✗	✗

Obrázek 2.1: Základní rozdíly mezi typy klientů

2.1.1 Tlustý klient

Tlustým klientem se označuje počítač v síťové architektuře klient-server, který poskytuje velké množství služeb nezávisle na centrálním serveru. Tento klient ovšem stále potřebuje alespoň periodické spojení s centrálním serverem, ikdyž je schopen poskytovat hodně služeb i bez tohoto spojení. Příkladem může být například klient pro stahování elektronické pošty, který musí periodicky stahovat nové zprávy ze serveru, ale jakmile je stáhne, může je uživatel procházet i bez spojení k serveru.

Výhodou tohoto přístupu je, že snižuje zátěž serveru, protože klient částečně zpracovává požadavky sám aniž by potřeboval server. Z tohoto vyplývají další výhody jako možnost práce *offline*[2] a nižší zátěž samotné sítě, protože na server putuje jen část požadavků, zbylé vyřídí samotný klient. Výhodou je také lepši výkon multimédií, které by byly jinak velice náročné na výkon sítě.

2.1.2 Tenký klient

Opakem tlustého klienta je tenký klient. Tenkým klientem se označuje počítač v síťové architektuře klient-server, který je přímo závislý na centrálním serveru pro vyřízení požadavků uživatele. Většinou tento klient slouží jako jakési vstupně-výstupní rozhraní mezi uživatelem a serverem. Uživatel klientovi zadává požadavky a ten je posílá centrálnímu serveru pro zpracování. Klient pak zpětně přijme výsledky požadavku od serveru a zobrazí je uživateli. Příkladem takového klienta je internetový prohlížeč, který od uživatele přijme adresu internetové stránky, pošle serveru požadavek na zobrazení této stránky a po přijetí odpovědi, která obsahuje vyžádanou stránku, tuto stránku zobrazí uživateli.

Hlavní výhodou tohoto přístupu je centralizace. Téměř všechna nutná administrativa se totiž děje na straně serveru. Je zde lepší možnost zabezpečení dat, kdy jsou data pouze na serveru, kde k nim lze kontrolovat přístup. V případě potřeby přidání nové služby často postačuje provést úpravy pouze na straně serveru a klienta není potřeba nijak modifikovat. Náročnost klienta na zdroje na hostitelském počítači je minimální, není potřeba ani žádné trvalé úložiště dat, protože ty jsou uloženy na straně serveru, což umožňuje provozovat klienta i na bezdiskovém počítači. Klient je také podstatně kompaktnější a velikostně menší oproti tlustým klientům.

Nevýhoda tohoto přítupu je velké vytížení serveru, kde jsou požadavky klientů vyřizovány. Se zvětšujícím se počtem klientů se zvedají i nároky na výkon serveru a ten nemusí stačit v reálném čase uspokojit všechny požadavky. Zároveň je server tzv. *single point of failure*[3] , takže pokud není dostupný, nejsou schopni klienti plnit požadavky uživatele.

[2] Práci *offline* se myslí práce bez vytvořeného spojení klienta k serveru.

[3] Pojmem *single point of failure* se označuje stav, kdy porucha nějaké části systému způsobí vyřazení celého systému z provozu.

2.1.3 Hybridní klient

Jak již název napovídá hybridní klient je někdé na půl cesty mezi tenkým a tlustým klientem. Stejně jako tenký klient nemá k dispozici data lokálně, ale na rozdíl od tenkého klienta tyto data lokálně zpracovává. Je nutno podotknout, že tato definice není úplně přesná, existuje mnoho klientů, které jsou různě mezi těmito třemi typy a daly by se zařadit taky jako určité hybridní klienty.

Jako příklad je možné uvést právě diskutovaného klienta pro přenos textových zpráv, který lze označit také jako hybridní klient. Tento klient pro posílání zpráv potřebuje přítomnost serveru, serveru posílá pouze požadavek na doručení zprávy nějakému dalšímu uživateli, klient sám neví, kde se má zpráva doručit, působí jen jako rozhraní mezi uživatelem a serverem, což ho řadí k tenkým klientům. Přenos souborů pro inicializaci vyžaduje server, aby získal informace, zda je možné přenos začít a kde se má připojit. Poté ale již samotný proces přenosu probíhá pouze mezi dvěma klienty, kteří data sami zpracovávají a lokálně čtou nebo ukládají, což řadí tohoto klienta někde mezi, protože je částečně závislý na serveru pro poskytování této služby. Nakonec je zde ještě procházení lokálně uložené historie, které pracuje pouze s lokálními daty bez potřeby mít spojení na server a klient sám je zpracovává. Při této službě zase můžeme označit klienta jako tlustého.

Je vidět, že určit typ klienta může být značně obtížné a většinou lze najít u každé klientské aplikace známky více typů. Většinu klientů je tedy možné brát jako určitý typ hybridního klienta.

2.2 Koncepce klientských aplikací

2.2.1 Architektura klient-server

Tato síťová architektura patří v současnosti k jedním z nejrozšířenějších. Hlavní podíl na tom má obrovský rozmach internetu a s tím související přesun mnoha aplikací do internetových prohlížečů, které pomalu začínají plnit funkci jakéhosi univerzálního tenkého klienta. Tato architektura se skládá ze dvou základních částí - klientské a serverové aplikace. Tyto dvě komponenty mohou mezi sebou komunikovat pomocí síťového spojení. Z kapitoly 2.1 je již známo, že se tyto klientské aplikace mohou dělit podle závislosti na využití právě serveru se kterým komunikují. Schéma této architektury můžeme vidět na obrázku 2.2.

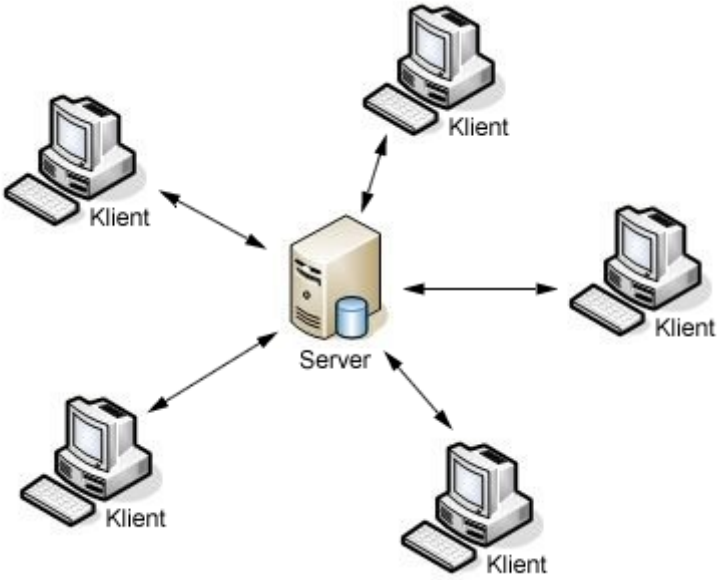
2.2.2 Architektura peer-to-peer

Síťová architektura peer-to-peer [10] (doslova přeloženo rovný-s-rovným) se skládá z uzlů. Tato architektura je přímo závislá na výkonu a propustnosti sítě jednotlivých uzlů narozdíl od architektury klient-server, která je závislá pouze na výkonu a propustnosti sítě zůčastněných serverů. Rozlišujeme dva typy peer-to-peer síťí:

- **Čisté peer-to-peer sítě**
- **Hybridní peer-to-peer sítě**

V čistých peer-to-peer síťích jsou všechny uzly rovnocenné a zastávají roli jak klientů tak serverů zároveň. Tato síť nezná pojem server, neexistuje zde žádný centrální server se kterým by uzly komunikovaly. Schéma čisté peer-to-peer sítě je možné vidět na obrázku 2.3.

V hybridních peer-to-peer síťích se vyskytují centrální servery, které uchovávají informace o uzlech a odpovídají na žádosti o zaslání těchto informací dalším uzlům. Uzly samotné



Obrázek 2.2: Schéma architektury klient-server

jsou odpovědné za uchovávání zdrojů, které mají k dispozici, za informování serverů, jaké zdroje chtěji sdílet s dalšími uzly a za umožnění, aby tyto zdroje byly k dispozici, pokud si je nějaký jiný uzel vyžádá. Schéma hybridní peer-to-peer sítě je možné vidět na obrázku 2.4.

Kromě uvedeného rozdělení je možné peer-to-peer sítě také zařadit do několika kategorií, jsou to:

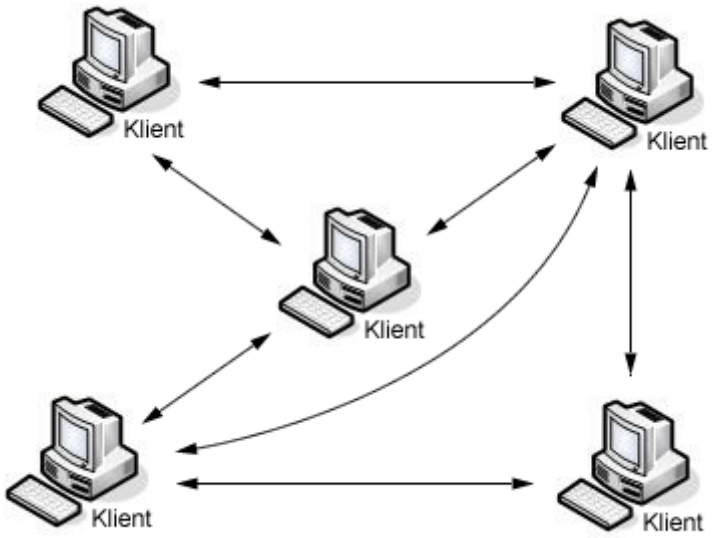
- **Centralizované peer-to-peer sítě**
- **Decentralizované peer-to-peer sítě**
- **Strukturované peer-to-peer sítě**
- **Nestrukturované peer-to-peer sítě**
- **Hybridní peer-to-peer sítě**

Centralizovaná peer-to-peer síť obsahuje centrální server, který poskytuje uzlům některé informace o jiných uzlech. Decentralizovaná peer-to-peer síť žádný centrální server nemá, obsahuje pouze jednotlivé uzly.

Další dvě kategorie souvisí s tzv. *překryvnou síťí*[4] . Většina peer-to-peer sítí jsou právě překryvné sítě, které jsou vytvořeny nad sítí internetu. Rozdíl mezi strukturovanou a nestrukturovanou peer-to-peer sítí spočívá v tom, jak jsou jednotlivé uzly této sítě navzájem propojeny.

V nestrukturované peer-to-peer sítí jsou jednotlivá spojení vytvářena libovolně. Taková síť se jednoduše vytváří. Nový uzel, který se chce připojit do sítě, jednoduše zkopíruje

[4] *překryvná síť* (anglicky *overlay network*) je počítačová síť, která je vytvořena nad jinou sítí. Spojení mezi uzly v této síťi je možné si představit jako jakési virtuální nebo logické linky, kdy každá tato linka může procházet přes několik fyzických linek překrývané síťě.



Obrázek 2.3: Schéma čisté peer-to-peer sítě

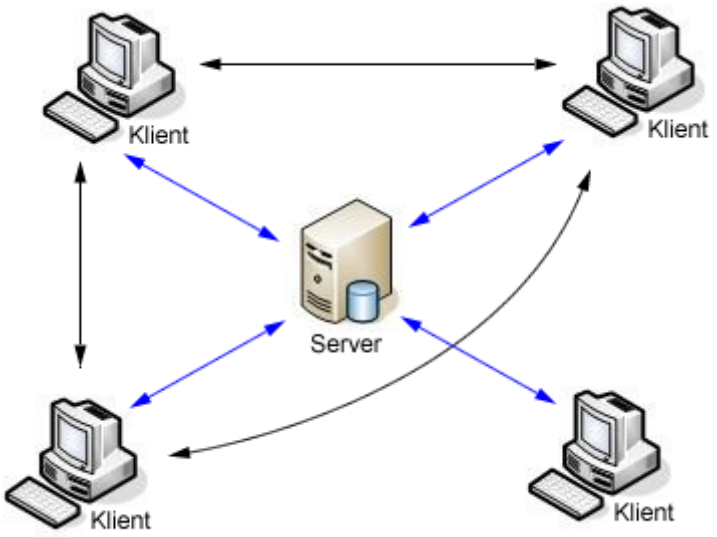
existující spojení[5] a postupem času si vytvoří vlastní. Když chce uzel v takovéto síťi najít nějaká data, musí dotaz projít zkrz síť, aby našel čím jak nejvíce uzlů, které mají požadovaná data. Nevýhodou těchto sítí je, že dotazy někdy nemusí najít žádné uzly s hledanými daty, ikdyž tyto uzly existují. Oblíbená data půjdou obvykle najít dobře, protože je bude mít velký počet uzlů, ovšem vzácné data, která má jen pár uzlů, nemusí být vůbec nalezeny.

Oproti tomu strukturované peer-to-peer sítě využívají globálně konzistentní protokol, který zaručuje, aby uzly mohly efektivně směrovat hledání k uzlům, které mají požadovaná data, i když jsou tyto data vzácná. Proto je důležité, aby byla zaručena určitá strukturovanost jednotlivých spojení. Nejvíce používaným typem strukturované peer-to-peer sítě je tzv. DHT (*Distributed Hash Table*), která využívá konzistentní hashování[6]

Poslední kategorií je hybridní peer-to-peer síť, která kombinuje prvky centralizované a decentralizované sítě. Ikdyž neobsahuje žádný centrální server, nejsou všechny její uzly rovnocenné. V této síťi existuje několik druhů uzlů, které plní trochu odlišné funkce. Například hybridní síť JXTA (více viz. [4]) definuje dvě kategorie uzlů - okrajové uzly a super-uzly (které se ještě dále dělí) a každý uzel má svou přesně definovanou roli v tomto peer-to-peer modelu.

[5] Zkopírováním existujících spojení je myšlen proces, kdy nový uzel získá od jiného uzlu určité informace (například seznam IP adres) o uzlech, ke kterým ma tento uzel vytvořené spojení a nový uzel použije tyto informace k vytvoření vlastních spojení k těmto uzlům.

[6] Konzistentní hashování (anglicky *Consistent hashing*) je schéma, které poskytuje funkcionalitu hashovací tabulky tak, že přidání nebo odebrání jednoho slotu (slotem se myslí místo v poli dvojic klíč-hodnota) nezmění nijak podstatně mapování klíčů na jednotlivé sloty. Většina tradičních hashovacích tabulek při změně počtu slotů musí přemapovat většinu klíčů, zde je nutné přemapovat pouze zhruba K/n klíčů, kde K je počet klíčů a n je počet slotů.



Obrázek 2.4: Schéma hybridní peer-to-peer sítě

2.3 Model-View-Controller

Model-View-Controller (MVC) [9] je architektonický vzor využívaný v softwarovém inženýrství. Při návrhu aplikací potřebujeme často oddělit data (model) a uživatelské rozhraní (pohled) tak, aby změny v uživatelském rozhraní neovlivňovaly práci s daty a aby data mohla být reorganizována bez změny uživatelského rozhraní. Architekrura model-view-controller řeší tento problém oddělením přístupu k datům a logiky aplikační vrstvy[7] od reprezentace dat a interakce uživatele zavedením další komponenty - řadiče.

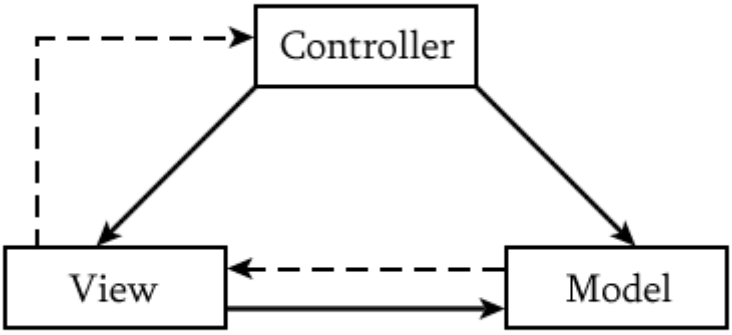
Je běžné rozdělit aplikaci do oddělených vrstev na presenční vrstvu (uživatelské rozhraní), doménovou vrstvu a vrstu přístupu k datům. U MVC je presenční vrstva ještě dále rozdělena na pohled a řadič. Celkově MVC obsahuje tři komponenty:

- **Model (Model)** - doménově specifická reprezentace informací s nimiž aplikace pracuje.
- **Pohled (View)** - převádí data reprezentovaná modelem do podoby vhodné k interaktivní prezentaci uživateli.
- **Řadič (Controller)** - reaguje na události (typicky pocházející od uživatele) a zajišťuje změny v modelu nebo v pohledu.

Jednoduchý diagram, který zobrazuje vztahy mezi těmito třemi komponentami je na obrázku 2.5 (Plné čáry vyjadřují přímou asociaci a čárkované čáry vyjařují nepřímou asociaci).

MVC se často vyskytuje u webových aplikací, kdy pohled je aktuální HTML stránka, kód, který sbírá data a generuje obsah uvnitř HTML stránek, je řadič a model je reprezentován aktuálním obsahem, který je uložen v databázi nebo souboru XML. Ačkoliv může být koncept MVC realizován různým způsobem, obecně platí tento princip činnosti:

[7] Logika aplikační vrstvy (anglicky *Business logic*) je pojem obecně používaný pro popis algoritmů, které řídí výměnu informací mezi databází a uživatelským rozhraním.



Obrázek 2.5: Vztahy mezi komponentami MVC

1. Uživatel provede nějakou akci v uživatelském rozhraní (např. stiskne tlačítko).
2. Řadič obdrží oznámení o této akci z objektu uživatelského rozhraní, často skrz zaregistrovanou obsluhu události nebo *callback*[8] .
3. Řadič přistoupí k modelu a v případě potřeby ho zaktualizuje na základě provedené uživatelské akce (např. zaktualizuje nákupní košík uživatele).
4. Komponenta pohled použije zaktualizovaný model pro zobrazení zaktualizovaných dat uživateli (např. vypíše obsah košíku). Komponenta pohled získává data přímo z modelu, zatímco model nepotřebuje žádné informace o komponentě pohled (je na ní nezávislý). Nicméně je možné použít návrhový vzor pozorovatel, umožňující modelu informovat jakoukoliv komponentu o případných změnách dat. V tom případě se komponenta pohled zaregistruje u modelu jako příjemce těchno informací. Je důležité podotknout, že řadič nepředává doménové objekty (model) komponentě pohledu, nicméně jí může poslat příkaz, aby svůj obsah podle modelu zaktualizovala.
5. Uživatelské rozhraní čeká na další akci uživatele, která celý cyklus zahájí znovu.

2.4 XML a XML Resource

2.4.1 XML

XML (*eXtensible Markup Language*, česky *rozšiřitelný značkovací jazyk*), je obecný značkovací jazyk, který byl vyvinut a standardizován konsorciem W3C. Umožňuje snadné vytváření konkrétních značkovacích jazyků pro různé účely a široké spektrum různých typů dat.

Jazyk XML, narozdíl od jazyka HTML, nemá žádné předdefinované značky (tagy, názvy jednotlivých elementů) a také jeho syntaxe je podstatně přísnější. Je určen především pro výměnu dat mezi aplikacemi a pro publikování dokumentů. Umožňuje popsat strukturu dokumentu z hlediska věcného obsahu jednotlivých částí, nezabývá se sám o sobě vzhledem

[8] *Callback* je spustitelný kód, který je předán jako parametr jinému kódu. Zde je to většinou funkce, která je předána objektu, u kterého může nastat nějaká událost, jako funkce, která se má zavolat, aby tuto událost obsloužila. Tento přistup umožňuje definovat vlastní obsluhu nějaké události.

dokumentu nebo jeho částí. Vzhled se definuje pomocí připojených stylů nebo se provádí transformace do jiného typu dokumentu nebo do jiné struktury XML.

Mezi vlastnosti jazyka XML patří (viz. [12]):

- **Standardní formát pro výměnu informací** - V dnešní době již není pratické posílat dokumenty ve formátu, který vyžaduje speciální software nějaké firmy. Je používána celá řada operačnách systémů a není zaručeno, že uživatel bude mít potřebný software k dispozici. Je tedy potřeba používat nějaký jednoduchý otevřený formát, který není úzce svázán s nějakou platformou nebo proprietární technologií[9] . Tím může být právě formát XML, který je založen na jednoduchém textu a je zpracovatelný (v případě potřeby) libovolným textovým editorem. Specifikace XML konsorcia W3C je zdarma přístupná všem. Každý tak může bez problémů do svých aplikací implementovat podporu XML.

- **Mezinárodní podpora** - XML již od počátku myslelo na potřeby i jiných jazyků, než je angličtina. Jako znaková sada se implicitně používá ISO 10646[10] (nebo Unicode). Proto je v XML možné vytvářet dokumenty, které obsahují texty v mnoha jazycích najednou a libovolně mezi nimi přepínat. Současně je přípustné i jiné libovolné kódování (např. windows-1250, iso-8859-2), musí však být v každém dokumentu přesně určeno. Odpadají tak problémy s konverzí z jednoho kódování do druhého.

- **Vysoký informační obsah** - Pomocí XML značek (tagů) je možné vyznačit v dokumentu význam jednotlivých částí textu. Dokumenty tak obsahují více informací, než kdyby se používalo značkovaní zaměřené na vzhled (definice písma, odsazení a podobně). XML dokumenty jsou proto informačně bohatší. To lze samozřejmě s výhodou využít v mnoha oblastech, například při prohledávání, kdy je možné určit i jaký význam má mít hledaný text.

- **Snadná konverze do jiných formátů** - Při používání XML dokumentu je potřeba také dokument zobrazit. XML samo o sobě žádné prostředky pro definici vzhledu nenabízí. Existuje ale několik stylových jazyků, které umožňují definovat, jak se mají jednotlivé elementy zobrazit. Souboru pravidel nebo příkazů, které definují jak se dokument převede do jiného formátu, se říká styl. Jeden vytvořený styl je možné aplikovat na mnoho dokumentů stejného typu, stejně tak je možné na jeden dokument aplikovat několik různých stylů. Výsledkem může být např. PostScriptový soubor, HTML kód nebo XML s obsahem původního dokumentu. Stylových jazyků existuje dnes několik. Mezi nejznámější patří asi kaskádové styly (CSS). Ty lze použít pouze pro jednoduché formátování, které dobře poslouží pro zobrazení dokumentu na obrazovce. Další možností je rodina jazyků XSL (*eXtensible Stylesheet Language*). Ta umožňuje dokument různě upravovat a transformovat, například vybírat části dokumentu nebo generovat obsahy a rejstříky.

- **Automatická kontrola struktury dokumentu** - XML neobsahuje předdefinované značky (tagy). Je třeba definovat vlastní značky, které bude dokument používat.

[9] Proprietární technologie je proces nebo řešení, které exkluzivně náleží určité společnosti. K takovéto technologii vetšinou nejsou k dispozici veřejnosti žádné dokumenty, které by umožňovaly její využití třetí osobou.

[10] Mezinárodní standard ISO 10646 definuje UCS (*Universal Character Set*). UCS je univerzální znaková sada, která zahrnuje všechny ostatní standardy znakových sad. Je definována jako 31bitová znaková sada, která obsahuje znaky nutné k reprezentaci prakticky všech známých jazyků.

Tyto značky je možné definovat v souboru DTD (*Document Type Definition*) nebo v souboru XSD (*XML Schema Definition*). Díky této definici lze, například pomocí parseru, automaticky kontrolovat, zda vytvářený XML dokument odpovídá této definici.

- **Hypertext a odkazy** - XML stejně jako HTML umožňuje vytváření odkazů v rámci jednoho dokumentu i mezi dokumenty, má však více možností. Je možné vytvářet i vícesměrné odkazy, které spojují více dokumentů dohromady. Tvorba odkazů je popsána ve třech standardech - XLink, XPointer a XPath.

Více informací o formátu XML je možné najít na stránkách konsorcia W3C[1].

2.4.2 XML Resource

XML Resource (XRC, přeloženo do češtiny *XML Zdroj*) je platformě nezávislý UIML[11] založený na jazyce XML, který používá toolkit wxWidgets. XRC umožňuje, aby části grafického uživatelského rozhraní, jako dialogy, menu a nástrojové lišty, byly uloženy v XML souboru, ze kterého je aplikace může za běhu načíst a zobrazit.

Mezi výhody XRC patří:

- Rekompilace a slinkování programu nemusí být vůbec potřeba, pokud se provedly změny pouze v uživatelském rozhraní.
- Odděluje uživatelské rozhraní od programové logiky.
- Možnost zvolit si mezi různými zdrojovými XRC soubory za běhu.
- XRC je standardem *wxWidgets*, takže může být vygenerován a zpracován každým programem, který tomuto formátu rozumí.
- XML je jednodušší pro zpracování než vetšina programovacích jazyků.
- Existující XML editory mohou být použity pro editaci XRC souborů.

2.5 Unicode

Ke konci osmdesátých let 20. století vznikla naléhavá potřeba sjednotit různé kódové tabulky znaků pro národní abecedy. Například český jazyk používal v informatice nejméně pět různě kódovaných tabulek. Vznikaly značné problémy při spolupráci aplikací a při přenosech dat mezi programy a různými platformami. Podobná situace byla ve všech jazycích, které nevystačily se základní 7bitovou tabulkou ASCII znaků. Proto začly vznikat snahy o vytvoření jednotné univerzální kódovací tabulky znaků. Současně vznikly dva projekty, které se touto problematikou zabývají - projekt ISO 10646, který definuje UCS (*Universal Character Set*), a projekt Unicode [11]. Ikdyž oba projekty stále publikují své standardy samostatně, jsou obě tabulky znaků kompatibilní a jejich rozšiřování je koordinováno.

Znak Unicode může být až 31 bitů dlouhý. Tento rozsah (maximálně $2^31 = 2147483648$ různých znaků) pokrývá všechny známé znakové sady jazyků na Zemi, včetně japonského nebo čínského písma. Používá se dále pro fonetické abecedy (pro zápis výslovnosti), speciální

[11] UIML (*User Interface Markup Language*) je značkovací jazyk používaný k definování uživatelských rozhraní.

vědecké a matematické symboly, kombinované znaky a podobně. Každý znak má jednoznačný číselný kód a svůj název. Standard Unicode se oproti ISO 10646 navíc zabývá implementací algoritmů pro písma psaná zprava doleva (arabština), podporou oboustranných textů (jako např. směs hebrejštiny a latinky) a algoritmy pro řazení a porovnávání textů. V současné chvíli existuje Unicode ve verzi 5.0.0, které vyšlo v roce 2006. Oproti předchozí verzi 4.1.0 bylo do nové verze zařazeno 1 369 nových znaků. Celkem se jejich počet rozšířil na 238 676 znaků a symbolů různých jazyků. Unicode consortium již v této chvíli zaručuje, že všechny nové verze budou zpětně kompatibilní s předchozími, tj. že nové standardy budou přidávat další znaky, ale žádné již nebudou odstraňovat ani měnit.

Bohaté možnosti Unicode mají i nevýhody, především vznikají problémy s nekompatibilitou se staršími aplikacemi, které jsou orientovány na jednobytové znaky. Také velmi narůstá délka textů. Textové řetězce v Unicode mohou obsahovat byty, které mají zvláštní význam pro programovací jazyky (např. binární nuly), nebo operační systémy (např. lomítka oddělující adresáře ve specifikaci souboru). Z tohoto důvodu byl navržen systém kódování znaků Unicode, nazývaný UTF (*UCS/Unicode Transformation Format*). UTF kódování odstraňuje všechny nevýhody neupraveného Unicode. V současnosti existují tři formáty kódování UTF:

- **UTF-8 (8-bit UCS/Unicode Transformation Format)** - UTF-8 je způsob kódování řetězců znaků Unicode/UCS do sekvencí bajtů. Původní specifikace je obsažena v RFC 2279 a definuje kódování znaků Unicode do sekvence 1 až 6 bytů. Tato specifikace byla nahrazena v roce 2003 novou, která je obsažena v RFC 3629 a definuje kódování znaků Unicode pouze do sekvence 1 až 4 bytů s tím, že znaky, které byly kódovány pěti nebo šesti bajty se prakticky vůbec nepoužívají. V současnosti se uvažuje o dalším zůžení tohoto rozsahu pouze na 1 až 3 bajty, je to dáno tím, že UTF-8 potřebuje k zakódování znaku z BMP[12] právě 1 až 3 bajty. Pro českou abecedu stačí pro znaky bez diakritiky jeden byte a pro znaky s diakritikou dva byty. Tento formát je nejpoužívanějším formátem kódování znaků Unicode vůbec. Například jazyk XML, který implicitně používá právě UCS nebo Unicode jako znakovou sadu, přímo ve specifikaci vyžaduje, aby systém, který zpracovává XML uměl pracovat právě s UTF-8 (a UTF-16), což činí tento způsob kódování téměř výhradním kódováním XML dokumentů.

- **UTF-16 (16-bit Unicode Transformation Format)** - UTF-16, narozdíl od UTF-8, kóduje unicode znaky do sekvence slov (dvoubajtů). Pro zakódování znaku z BMP potřebuje pouze jedno slovo (2 bajty), znaky z jiných rovin kóduje jako dvojici slov, která je označována jako tzv. zástupný pár (anglicky *surrogate pair*). UTF-16 nahrazuje kódování UCS-2 (*2-byte Universal Character Set*), které je téměř identické, ale neumožňuje použití zástupných párům, takže umožňuje kódovat pouze znaky z BMP.

- **UTF-32 (32-bit Unicode Transformation Format)** - UTF-32 a UCS-4 (*4-byte Universal Character Set*) jsou prakticky jen jiné označení pro Unicode a UCS. Tyto kódování používají pro uložení jednoho znaku vždy 4 bajty. Protože znak unicode může být maximálně 31 bitů dlouhý, ukládají tyto kódování vlastně přímo nezměněný

[12] Znaky Unicode se rozdělují to tzv. rovin (anglicky *planes*), každá rovina obsahuje 2^16 (65 536) znaků. Rovina 0 se označuje jako BMP (*Basic Multilingual Plane*) a patří zde znaky z téměř všech moderních jazyků. Znaky mimo tuto nultou rovinu maji všeobecně velmi specializované využití a ve většině případu si vždy vystačíme právě jen s nultou rovinou znaků.

unicode znak. Protože ale pro znaky z BMP nám stačí mnohem mémě bytů pro uložení, je toto kódování velice neefektivní a v praxi se pro ukládání nepoužívá, využívá se ovšem pro interní reprezentaci Unicode například v některých operačních systémech Linux.

2.6 Zabezpečení informací

Zabezpečení informací je proces ochrany dat před neoprávněným přístupem, použitím, odhalením, zničením, pozměněním nebo přerušením. Hlavním cílem je zaručit důvěrnost, neporušenost a dostupnost informací. V dnešní době, kdy je kladen stále větší důraz na bezpečnost, je nutné zajistit, aby se odesílané informace dostaly k cíli aniž by je někdo nepovolaný mohl získat a použít. Většina aplikací, které mezi sebou potřebují komunikovat na větší vzdálenosti než jen v rámci lokální sítě ale využívají ke komunikaci síť internet. Tato síť není vůbec bezpečná a prakticky kdokoliv na světě s přístupem do této sítě má možnost zasílané informace získat. Proto je nutné při přenosu přes takovéto sítě informace zabezpečit. Řadu způsobů jak informace zabezpečit předkládá kryptografie.

2.6.1 Kryptografie

Kryptografie [8] neboli šifrování je nauka o metodách utajování smyslu zpráv převodem do podoby, která je čitelná jen se speciální znalostí. Cílem kryptografie je utajit význam zpráv, ne jejich samotnou existenci. Kryptografie tedy nechrání před samotným odcizením zprávy, zaručuje ale, že bez speciálních znalostí o této zprávě je tato zpráva nečitelná a tímto i bezcenná.

Slovem šifra se označuje dvojice kryptografických algoritmů, které převádí čitelnou zprávu (prostý text) na její nečitelnou podobu (šifrovaný text) a obráceně. Šifrováním se pak označuje proces, který tyto transformace provádí. Celý proces šifrování je řízený kryptografickým algoritmem a klíčem. Klíč je tajná informace, která je potřeba pro zašifrování nebo dešifrování zprávy. Podle toho, jaký klíč je pro tyto dvě operace potřeba, rozlišujeme dva druhy kryptografie - asymetrická kryptografie a symetrická kryptografie.

2.6.2 Symetrická kryptografie

Symetrická šifra, někdy též nazývaná konvenční, je takový šifrovací algoritmus, který používá k šifrování i dešifrování jediný klíč. Podstatnou výhodou symetrických šifer je jejich nízká výpočetní náročnost. Algoritmy pro šifrování s veřejným klíčem můžou být i stotisíckrát pomalejší. Na druhou stranu velkou nevýhodou je nutnost sdílení tajného klíče, takže se odesilatel a příjemce tajné zprávy musí předem domluvit na tajném klíči. Symetrické šifry se dělí na dva druhy:

- **Proudové šifry** - zpracovávají otevřený text po jednotlivých bitech. Patří zde například FISH nebo RC4.
- **Blokové šifry** - rozdělí otevřený text na bloky stejné velikosti a doplní vhodným způsobem poslední blok na stejnou velikost. U většiny šifer se používá blok o 64 bitech, AES používá 128 bitů. Zde patří například DES, Triple-DES nebo již zmíněný AES[13].

[13] AES (*Advanced Encryption Standard*) není sám o sobě šifra. V roce 1997 oznámil NIST (*National Insti-*

Na obrázku 2.6 je možné vidět ukázku průběhu procesu symetrického šifrování a dešifrování.

Obrázek 2.6: Proces symetrického šifrování a dešifrování

2.6.3 Asymetrická kryptografie

Asymetrická šifra je takový šifrovací algoritmus, který používá k šifrování i dešifrování **odlišné** klíče. Jeden klíč se používá pro šifrování zpráv (tento klíč ani příjemce zprávy nemusí znát), druhá pro dešifrování (tento klíč zase nezná odesílatel zprávy). Je vidět, že ten, kdo šifruje, nemusí s dešifrujícím příjemcem zprávy sdílet žádné tajemství[14] , čímž eliminují potřebu výměny klíčů. Tato vlastnost je základní výhodou asymetrické kryptografie oproti symetrické.

Nejběžnější verzí asymetrické kryptografie je využívání tzv. veřejného a soukromého klíče. Šifrovací klíč je veřejný, majitel klíče ho volně uveřejní, a kdokoli jím může šifrovat jemu určené zprávy. Dešifrovací klíč je soukromý, majitel jej drží v tajnosti a pomocí něj může tyto zprávy dešifrovat. Průběh procesu šifrování a dešifrování pomocí asymetrické šifry je možné vidět na obrázku 2.7.

Obrázek 2.7: Proces asymetrického šifrování a dešifrování

Je zřejmé, že šifrovací klíč a dešifrovací klíč spolu musí být matematicky svázány, avšak nezbytnou podmínkou pro užitečnost šifry je praktická nemožnost ze znalosti šifrovacího klíče spočítat dešifrovací. Opačný proces samozřejmě možný je a využívá se právě při generování dvojice soukromého a veřejného klíče.

Kromě očividné možnosti pro utajení komunikace se asymetrická kryptografie používá také pro elektronický podpis, tzn. možnost, jak u zprávy prokázat jejího autora. Pro podepsání zpráv se využívá opačný postup jako při šifrování. Zpráva se zašifruje soukromým

tute of Standards and Technology), že hodlá vybrat nástupce zastaralého DES (*Data Encryption Standard*), který se bude označovat AES. Rozdohovalo se asi mezi 15 šiframi a v roce 2000 byla šifra zvaná Rijndael označena za vítěze a prohlášena za AES standard.

[14] Tajemstvím se myslí tajný klíč (anglicky *secret key*).

klíčem odesilatele a příjemce ji dešifruje pomocí volně dostupného veřejného klíče odesilatele a pokud se to podaří ví příjemce, že zprávu odeslal opravdu uvedený odesilatel. Průběh procesu podepisování zpráv a ověřování je možné vidět na obrázku 2.8.

Obrázek 2.8: Proces podepisování a ověřování zpráv

Kapitola 3

Návrh řešení

Cílem této kapitoly je navrhnout vhodné řešení systému z pohledu jeho klientské části. Navržený systém by měl brát zřetel na rozšiřitelnost a nezávislost. Význam nezávislosti je zde dokonce dvojnásobný, protože zde má dvojí smysl.

První smysl nezávislosti je z hlediska programačního, kde je důležité zajistit čím jak největší nezávislost mezi jednotlivými částmi programu. Tato nezávislost umožňuje provádět úpravy jedné části proramu aniž by se tyto změny dotkly jiných částí. Toto je také základní princip OOP (Objektově orientovaného programování). Bohužel úplná nezávislost mezi jednotlivými částmi není většinou možná, nebo by neúnosně zkomplikovala celý návrh, proto je většinou důležité najít optimální míru nezávislosti mezi jednotlivými částmi.

Druhý smysl nezávislosti je z hlediska aplikačního, kdy je důležité zajistit nezávislost aplikace na operačním systému, na kterém poběží. Čím více operačních systémů bude aplikace podporovat, tím větší je její využitelnost v praxi.

Rozšiřitelnost úzce souvisí s nezávislostí. Klientská aplikace je dobře rozšířitelná, pokud nejsou její jednotlivé části na sobě příliš závislé.

3.1 Formulace úlohy

Je požadováno vytvořit klientskou část systému pro výměnu textových zpráv (instant messenging system). Klient pro tuto výměnu využívá služeb serveru. Obě části systému (klientská i serverová) mezi sebou komunikují pomocí aplikačního protokolu, jehož sémantika je jim známá.

Klient musí poskytovat následující funkce:

- **Posílání a příjem zpráv** - Klient musí být schopen sestavit zprávu zadanou uživatelem a odeslat ji ve formátu specifikovaného protokolem serveru, který již obstará doručení zprávy cílovému uživateli. Klient musí také umět zprávy odeslané serverem přijmout a vizuálně interpretovat uživateli.
- **Zabezpečení zpráv** - Klient musí zajistit, aby důležitý obsah posílaných zpráv byl čitelný pouze uživateli, kterému je zpráva posílána.
- **Přenos souborů** - Klient musí umožnit přenášet soubory mezi uživateli a to i za předpokladu, že některý z klientů učastnících se přenosu sídlí na interní síťi, která využívá privátní adresy.

- **Historie zpráv** - Klient musí být schopen zobrazit předešlé zprávy odeslané a přijaté daným uživatelem.

3.2 Doplnění úlohy

Kromě základních požadavků na funkcionalitu klienta je dobré také zvážit některé další aspekty, které by zlepšily použitelnost klienta jak po uživatelské stránce, tak po programátorské stránce. Při návrhu celého systému by tedy bylo dobré brát zřetel na:

- **Mezinárodní podporu** - Jako klient systému jehož primární účel je zprostředkování písemné komunikace mezi dvěma subjekty není příliš praktické, aby zde existovala nějaká úzká vazba mezi prostředím, kde je tento klient provozován, a samotným klientem. Zpráva by měla dojít v přesném znění v jakém byla odeslána, ať již oba komunikující subjekty jsou v lokální síťi nějaké firmy, nebo na opačných stranách zeměkoule. Pokud například uživatel z Japonska odešle zprávu obsahující japonské znaky uživateli z České republiky, musí příjemci dojít zpráva, která tyto znaky zachovává, ikdyž samotný systém uživatele je nemusí umět korektně zobrazit.

- **Lokalizaci** - Ikdyž je anglický jazyk dnes již hodně rozšířený, zvláště v oblasti počítačů, uživatelé stále raději pracují s aplikací, která s nimi komunikuje v jejich mateřském jazyce. Bylo by proto vhodné umožnit u klienta výběr z různých lokalizací. Je ovšem důležité, aby rozšíření klienta o nové lokalizace bylo jednoduché (aby nové lokalizace mohli přidávat i normální uživatelé) a nevyžadovalo žádné zásahy do klienta samotného (ve smyslu jeho opětovné rekompilace nebo dokonce modifikace zdrojových kódů).

- **Multiplatformnost** - Dnes, v době rozličných operačních systémů, není vhodné svázat aplikaci jen s nějakou konkrétní platformou. Pro zvýšení využitelnosti klienta je výhodné, aby pracoval na alespoň nejvíce rozšířených operačních systémech, jako jsou Microsoft Windows, Linux a Apple Mac OS.

- **Oddělení uživatelského rozhraní od programu** - Uživatelské rozhraní bývá velice často hojně upravovaná část programu. Proto by bylo výhodné oddělit tuto část od samotného kódu programu. Kromě toho, že se tímto krokem sníží riziko vytvoření chyb v programovém kódu při úpravách kódu náležicího popisu uživatelského rozhraní (které je obsaženo typicky přimo v programovém kódu), dojde hlavně k určité centralizaci celého uživatelské rozhraní, čímž je myšleno, že věškerý popis uživatelského rozhraní bude na jednom místě. Tímto se zlepší celková orientace v popisu uživatelského rozhraní a bude možné v něm provádět změny bez zásahu do programového kódu.

3.3 Návrh protokolu

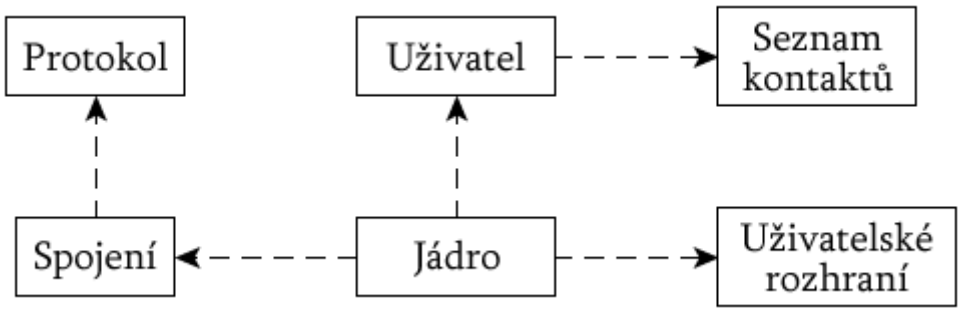
Sémantika protokolu může značně ovlivnit funkčnost klienta v mnoha ohledech, je tedy třeba postupovat při návrhu opatrně. Protokol by měl mít tyto tři vlastnosti:

- **Univerzálnost** - Protokol musí být schopen zakódovat text v jakémkoliv jazyce. Toto souvisí s mezinárodní podporou zmíněnou v kapitole 3.2, kterou může klient splňovat jedině tehdy, pokud ji bude poskytovat i samotný protokol, pomocí kterého je celá komunikace realizována.

- **Rozšiřitelnost** - Protokol musí být dobře rozšiřitelný. Při přidání nových typů zpráv nebo při úpravě stávajících by se měla minimalizovat nutnost úprav v programovém kódu.
- **Nezávislost** - Protokol by měl být nezávislý na verzích klientů. Pokud mezi sebou komunikují dvě verze klientů, kdy jedna je novější a podporuje nové funkce, nesmí tento fakt bránit komunikaci. Starší klient bude v této situaci poskytovat své funkce jako doposud a nové funkce, které jsou často spjaty s novým typem zprávy, bude ignorovat.

3.4 Návrh klienta

Pro vytvoření dobrého návrhu je potřeba celý problém rozložit na jakési ucelené části, které spolu logicky souvisí. Nejvhodnější dekompozici tohoto celku je možné vidět na obrázku 3.1. Hlavní část celého klienta je *jádro*, které veškerou komunikaci zajišťuje skrz část *spojení*. *Spojení* samo o sobě nezná semantiku protokolu, k tomu využívá *protokolovou* část. Kromě komunikace také jádro zpracovává události vzniklé v části *uživatelského rozhraní* a příslušně na ně reaguje. V poslední řadě také jádro spravuje část *uživatele*, která využívá část *seznamu kontaktů*.



Obrázek 3.1: Navrhovaná architektura klienta

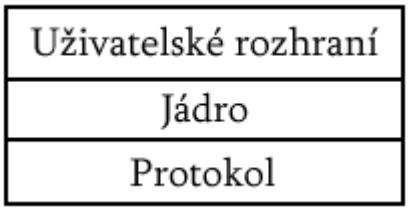
3.4.1 Jádro

Jádro je stavební kámen celého klienta, hlavní část, která prakticky zajišťuje jeho celý chod. Obstarává veškerou komunikaci, spravuje jednotlivé uživatele a zprostředkovává jejich interakci s celým systémem.

Jak již bylo řečeno na začátku této kapitoly, je potřeba klást důraz na nezávislost jednotlivých částí. Ikdyž v případě uživatelské části asi nikdy úplné nezávislosti nebude dosaženo, stejně tak i v případě uživatelského rozhraní, neplatí toto v části komunikační, kde je nezávislost vůbec nejdůležitější. Jádro vůbec nezajímá jakou sémantiku mají posílané a přijímané zprávy, potřebuje pouze zaručit jejich posílání a příjem. Ovšem tato potřeba, v kombinaci s nezávislosti, zde znamená problém.

Narozdíl od serveru, kde jádro prakticky nemusí nic vědět o zprávě, je u klienta situace poněkud odlišná. Klient sice nemusí nic vědět o sémantice zprávy, ale zároveň ji musí vlastně vytvořit, což vypadá, že se vzájemně vylučuje. Pro pochopení a analýzu celého problému

je třeba některé části klienta z obrázku 3.1 namodelovat pomocí vrstev, výsledek je vidět na obrázku 3.2.



Obrázek 3.2: Rozvrstvení části architektury klienta

Jak je vidět na obrázku, působí jádro jako jakýsi prostředník mezi protokolem a uživatelským rozhraním. U klienta je totiž, narozdíl od serveru, nutné interpretovat přichozí zprávy uživateli skrz uživatelské rozhraní a získavat od něj zprávy, které je nutné odeslat. Jádro díky této roli ovšem musí mít informace o obsahu samotných zpráv. Je tedy třeba navrhnout způsob, který zaručí jádru potřebné informace o obsahu zprávy, ale nijak ho nesváže s konkrétní implementací protokolu. Toto řešení ale musí být opravdu univerzální, aby nijak neomezovalo jakékoliv úpravy nebo rosšiřování protokolu.

Nejlepším způsobem jak tento problém vyřešit je vytvořit jakousi abstraktní reprezentaci zprávý, se kterou bude jádro pracovat a kterou předá metodám protokolu. Toto řešení sice může znamenat určité zpomalení, protože protokol bude muset transformovat abstraktní reprezentaci zprávy do svého formátu a opačně, ovšem klient tímto získá úplnou nezávislost na sémantice protokolu. Tuto abstraktní reprezentaci je možné modelovat několika způsoby:

- **Struktura s atributy** - Zpráva bude reprezentována strukturou, která obsahuje atributy. Každý z těchto atributů bude reprezentovat nějakou informaci obsaženou ve zprávě (např. odesilatele, příjemce, text zprávy). Tento způsob je ale téměř nepoužitelný ze dvou důvodů:
  - Je rozšiřitelný jen za cenu úpravý programového kódu, kdy je třeba doplnit nové atributy.
  - Je neefektivní z hlediska paměťového, protože struktura vždy obsahuje všechny atributy, ikdyž je jich pro daný typ zprávy potřeba jen malá část.
- **Seznam dvojic** - Zpráva bude reprezentována lineárním nebo jiným seznamem dvojic obsahujících pár atribut zprávy a hodnota. Tento způsob je celkem výhodný, reprezentace vždy obsahuje jen potřebné atributy k danému typu zprávy, a je lehce rozšiřitelný, v případě potřeby se jen do seznamu dodají nové dvojice. Jedinou nevýhodou je, že tento způsob nijak nebere ohled na strukturu zprávy, je to pouze seznam atributů obsažených ve zprávě.
- **Strom** - Zpráva bude reprezentována binárním nebo jiným stromem obsahujícím dvojice atribut zprávy a hodnota. Tento způsob se jeví jako nejlepší. Stejně jako seznam obsahuje pouze potřebné atributy a je lehce rozšiřitelný, ale kromě toho může strom popisovat i strukturu původní zprávy.

3.4.2 Spojení

Spojení zajišťuje navazování komunikace mezi dvěmi entitami, ať již to jsou server a klient, nebo dva klienti. Zastřešuje tak veškerou komunikaci a poskytuje jádru jednotnou obsluhu této komunikace. Pro komunikaci samotnou pak spojení využívá služeb protokolu.

Ikdyž se může zdát, že využití spojení, jako mezičlánku mezi protokolem a jádrem, je celkem zbytečné a jádro by mohlo využívat přímo služeb protokolu, má tento postup své výhody:

- **Univerzální ovládání** - Samotný protokol, který spojení využívá, může pro přenos používat různé typy protokolů nižších vrstev (jako například TCP (*Transmission Control Protocol*) nebo UDP (*User Datagram Protokol*)) a ty mohou vyžadovat rozdílné postupy navazování a udržování spojení. Je proto vhodné odstínit tyto odlišnosti a poskytnout jádru univerzální obsluhu těchto spojení.
- **Uchovávaní informací** - Spojení si může uchovávat různé informace důležité k navazování komunikace jako například informace nutné pro samotné navázání spojení (např. IP adresa a číslo portu) nebo pro následné přihlášení, pokud je potřeba. Pokud například dojde k přerušení spojení, jsou k dispozici veškeré informace k jeho obnovení.
- **Uchovávaní stavu** - Spojení si může udržovat informace o aktuálním stavu spojení, hlavně o přihlášení daného uživatele.
- **Rozšíření služeb protokolu** - Spojení poskytuje jádru funkce protokolu, který využívá. Než samotné spojení požadovanou funkci zavolá, může provést určité modifikace nebo rozšíření požadavku jádra, nebo může podle stavu spojení oznámit jádru nemožnost vykonání požadované funkce protokolem.

3.4.3 Protokol

Protokol specifikuje sémantiku přijímaných a odesílaných dat. Jeho úkolem je transformovat zprávy mezi svou reprezentací a abstaktní reprezentací, se kterou pracují jiné části klienta.

Jak již bylo řečeno v předchozích kapitolách, spojení ani jádro samotné neznají sémantiku protokolu a jsou na něm nezávislé. Pracují pouze s abstraktní reprezentací. Protokol proto musí zajistit příjem dat, jejich správnou interpretaci a transformaci do této abstaktní reprezentace, kterou předá vyšším vrstvám (viz. obrázek 3.2). Stejně tak musí zaručit správnou transformaci z abstraktní reprezentace do svého definovaného formátu a následné odeslání takto sestavené zprávy. Kromě posílání a přijmu zpráv musí také poskytovat funkce pro posílání a příjem souborů a pro přihlašování a odhlašování uživatelů.

3.4.4 Uživatel a seznam kontaktů

Tyto dvě části jsou jakýsi informační celek. Obsahují informace o uživateli klientské aplikace, jeho nastavení a kontaktech. Tyto informace ovlivňují chování celé aplikace a jádro je často využívá.

3.4.5 Uživatelské rozhraní

Uživatelské rozhraní zde působí jako presenční vrstva mezi jádrem a samotným uživatelem. Jak již bylo zmíněno v kapitole 3.2, je vhodné ho oddělit od samotného programu. Je proto

vhodné zvážit využití nějakého jazyka používaného k definování uživatelských rozhraní.

Kapitola 4

Realizace

Tato kapitola se zabývá realizací celé klientské aplikace. Jsou zde popsány postupy, které byly využity při vývoji, vybrané řešení jednotlivých problémů a samotná implementace.

4.1 Implementační prostředí

Výběr implementačního prostředí je svázán s požadavkem na multiplatformnost (viz. kapitola 3.2) a také se zkušenostmi programátora s daným prostředím.

Jako možné implementační prostřední by bylo možné využít *.NET*, *Javu* nebo *jazyk C++*. Bohužel platforma .NET zatím neposkytuje multiplatformnost a vyžaduje přítomnost virtuálního stroje pro chod programů. Jazyk Java sice multiplatformnost poskytuje, ale stejně jako .NET vyžaduje pro chod aplikací virtuální stroj. Tímto zde vzniká určitá závislost, sice ne na operačním systému samotném, ale na přítomnosti virtuálního stroje, což omezuje využití klienta. Navíc aplikace jsou kompilované pouze do tzv. mezikódu, který je poté virtuálním strojem interpretován, což zpomaluje samotný běh aplikace.

Poslední možností je jazyk C++. Ten multiplatformnost zajišťuje jen do určité míry. Například pro grafické rozhraní totiž musí využívat funkce operačního systému, které jsou platformě závislé. Řešením je využití nějakého toolkitu[1] jako je *QT* nebo *wxWidgets*. Pro implementaci klienta byl nakonec vybrán toolkit *wxWidgets*.

4.2 Protokol

Jak již bylo popsáno v kapitole 3.3 měl by navržený protokol mít tři vlastnosti - univerzálnost, rozšiřitelnost a nezávislost. Je tedy třeba navrhnout takový protokol, který tyto vlastnosti bude splňovat.

4.2.1 XML protokol

Nejlepší řešení je vytvořit protokol na bázi jazyka XML. Jak je již známo z kapitoly 2.4.1 slouží jazyk XML k popisu dokumentů, to ovšem vůbec nebrání jeho využití k popisu zpráv, vždyť zpráva je vlastně také určitá forma dokumentu. Vlastnosti jazyka XML zaručí splnění všech nároků, které jsou na protokol kladeny a to jsou:

[1] Toolkit zde označuje sadu knihoven, které odstiňují závislosti mezi jednotlivými platformami a poskytují určité jednostné rozhraní pro obsluhu grafického rozhraní i dalších služeb operačního systému.

- **Univerzálnost** - Jazyk XML implicitně využívá unicode, což zaručuje schopnost zakódovat text v jakémkoliv jazyce. Vzhledem k tomu, že norma XML přímo vyžaduje, aby parsery podporovaly kódování UTF-8, je nejlepší, aby toto kódování využíval i samotný protokol.
- **Rozšiřitelnost** - Parser je vždy schopen zpracovat text, který obsahuje validní XML, není proto problém tento text jakkoliv rozšiřovat a upravovat aniž by nastala nutnost provádět jakékoliv změny v kódu programu (kromě přidání obsluhy dané zprávy) a dokonce aniž by bylo nutné upravovat samotný parser protokolu.
- **Nezávislost** - Zpracování nových typů zpráv starým klientem nijak neovlivní jeho činnost. Jak již bylo řečeno v předchozím bodě, není při přidávání nových typů zpráv dokonce ani třeba nějak parser upravovat. Zprává, která je pro starého klienta neznámým typem, bude prostě ignorována.

Jak je vidět, je vytvoření protokolu na bázi jazyka XML ideální. Je zde ovšem jeden problém, který je třeba vyřešit a tím je zjištění délky zprávy. Existují dva přístupy, které se dnes využívají. Jsou to:

- **Koncová značka** - Zpráva je načítána dokud se nenarazí na speciální značku, která označuje konec této zprávy. Problém ovšem nastane, pokud se tato značna vyskytuje v samotném textu zprávy, která by tímto označila konec zprávy na špatném místě. Tento problém řeší tzv. *Byte stuffing*[2] , který výskyty koncové značky v textu eliminuje. Tento způsob ovšem zatěžuje jak odesilatele, tak příjemce (v tomto přídadě klienta i server), kdy jsou oba nuceni neustále testovat přišlé znaky na přítomnost koncové značky a poté ještě provádět transformaci samotné zprávy z důvodu *byte stuffingu*.
- **Specifikace délky** - Zpráva obsahuje někde na počátku informaci o své celkové délce, program tedy načte malou část zprávy a z ní zjistí kolik dat ještě musí načíst, aby byla zpráva kompletní. Tento způsob má hlavní nevýhodu v tom, že vyžaduje sestavení celé zprávy před jejím odesláním (aby bylo možní zjistit její delku a zaznačit tuto délku do této zprávy). Proto je tento způsob nepoužitelný v případech, kdy je třeba začít posílat data, aniž by bylo jasné, kolik jich bude. Tento způsob ale zatěžuje pouze odesilatele zprávy, který musí počítat délku zprávy, a umožňuje rychlé a výpočetně nenáročné načtení zprávy u příjemce.

Z pohledu implementovaného systém je výhodnější druhé řešení pomocí specifikace délky, protože rozložení zátěže je v tomto případě ideální. Server je v drtivé většině případů pouze příjemce a přeposilatel zpráv a klient odesilatel, takže dojde k přesunutí vetšiny zátěže v tomto ohledu ze serveru na klienty, kteří většinou tento nárust zátěže ani nezaznamenají.

Příklad zprávy navrženého protokolu je vidět na obrázku 4.1.

4.2.2 Interní reprezentace

V kapitole 3.4.1 byla zmíněna nutnost vytvoření abstraktní reprezentace zprávy, se kterou bude klient pracovat. Jak je patrné, tak nejlepším způsobem se jeví využití stromu.

Typů stromů ovšem existuje několik, protože ale samotný protokol je reprezentován jazykem XML, nabízí se zde možnost využití tzv. DOM modelu. DOM (*Document Object*

[2] *Byte stuffing* je proces, při němž se sekvence dat, která obsahuje nepovolené nebo vyhrazené znaky, transformuje na, většinou delší, sekvenci, která již tyto znaky neobsahuje.

```
<message length=‘‘148’’ type=‘‘plain’’>
  <head>
    <from id=‘‘jan@saber.cz’’/>
    <to id=‘‘marek@saber.cz’’/>
  </head>
  <body>
    <mess-text>Text</mess-text>
  </body>
</message>
```

Obrázek 4.1: Příklad zprávy navrženého protokolu

Model) je objektový model, který popisuje XML dokument a reprezentuje jej formou stromu. Protože hodně XML parserů při zpracování XML dokumentu vytvářejí právě jeho DOM model v paměti, je toho řešení velice výhodné. Protokol pouze zpracuje zprávu v jazyce XML a tím vlastně vytvoří interní reprezentaci zprávy, kterou předá jádru. Toolkit *wxWidgets* obsahuje objektové zastřešení XML parseru *expat*, který je sice SAX (*Simple API for XML*) parser, ale toto zastřešení reprezentuje dokument pomocí DOM stromu. Zprává z obrázku 4.1 by byla ve *wxWidgets* reprezentována stromem na obrázku 4.2.

```
wxXML_ELEMENT_NODE name=‘‘message’’, content=‘‘’’
||-- wxXML_ATTRIBUTE_NODE name=‘‘length’’, content=‘‘148’’
||-- wxXML_ATTRIBUTE_NODE name=‘‘type’’, content=‘‘plain’’
|
|-- wxXML_ELEMENT_NODE name=‘‘head’’, content=‘‘’’
|   |-- wxXML_ELEMENT_NODE name=‘‘from’’, content=‘‘’’
|   |    |-- wxXML_ATTRIBUTE_NODE name=‘‘id’’, content=‘‘jan@saber.cz’’
|   |
|   |-- wxXML_ELEMENT_NODE name=‘‘to’’, content=‘‘’’
|   |    |-- wxXML_ATTRIBUTE_NODE name=‘‘id’’, content=‘‘marek@saber.cz’’
|
|-- wxXML_ELEMENT_NODE name=‘‘body’’, content=‘‘’’
    |-- wxXML_ELEMENT_NODE name=‘‘mess-text’’, content=‘‘’’
        |-- wxXML_TEXT_NODE name=‘‘’’, content=‘‘Text’’
```

Obrázek 4.2: Strom reprezentující zprávu z obrázku 4.1

4.2.3 Parser

Pro celkové usnadnění a zrychlení práce s protokolem a XML dokumenty obecně je dobré mít parser, který poskytuje mnoho funkcí. Proto obsahuje klient vlastní dvě třídy, které zastřešují třídy *wxWidgets* pro práci s XML.

První třída je `SaberXML`, která zastřešuje třídu `wxXmlDocument`. Tato třída pracuje s interní reprezentací popsanou v kapitole 4.2.2. Kromě metod pro načítání a ukládání XML dokumentů poskytuje tato třída také metodu `SavePart` pro uložení pouze části stromu s interní reprezentací. `SaberXML` se v interní reprezentaci pohybuje pomocí tzv. skupin.

Skupinou se označuje uzel, který obsahuje nějaké synovské uzly. Třída si interně udržuje ukazatel na právě vybranou skupinu a pomocí metod `Enter/LeaveGroup` umožňuje jejich procházení. Prvotní nastavení ukazatele se provede metodou `EnterRootGroup`, která nastaví ukazatel na první uzel dokumentu (u zprávy je to uzel pojmenovaný `message`), pokud je tento uzel skupina. Ostatní metody poté operují s normálními uzly (ne s uzly, které jsou skupiny) pod danou skupinou. Třída poskytuje `Get/Set` metody pracující s velkou řadou typů jako jsou `string`, `integer`, `long`, `double`, `bool` nebo dokonce i třídami `wxColor` a `wxFont`, které reprezentují barvy a fonty. Ikdyž samotné XML reprezentuje vše jako řetězce, tyto funkce obstarávají veškeré konverze mezi textem a příslušným typem, což usnadňuje programátorovi celkovou práci s XML.

Druhá třída je `SaberMessageParser`, která obstarává obsluhu interní reprezentace protokolu. Tato třída zastřešuje třídu `SaberXML` a poskytuje speciální metody vázané k protokolu. Metoda `Load` umožňuje zpracování zprávy a její transformaci do interní reprezentace. Metoda `Create` slouží k vytvoření základu zprávy, který se dále upravuje podle typu zprávy. Asi nejdůležitější metodou je ale `Save`, která vytváří zprávu z interní reprezentace. Při vytváření zprávy rovnou dochází k výpočtu její délky a následnému zaznamenání této délky do vytvořené zprávy, zde je využita již zmíněná metoda `SavePart` třídy `SaberXML`, protože je třeba vytvářet nejprve části s tělem a hlavičkou, aby bylo možné determinovat výslednou délku zprávy, která musí být vložena na začátek, a poté teprve dodat první část zprávy (element `message`) s informací o celkové délce. Dále také poskytuje tato třída metody pro získavaní / vkládání informací z / do hlavičky a těla (metody `Get/Set Header/Body`) a speciální metody pro získávání nejčastěji potřebných informací (jako např. `Get/Set Sender/Receiver`). V případě potřeby ještě poskytuje přístup přímo k objektu třídy `SaberXML` pomocí metody `GetXml`, čímž umožňuje využívat veškeré metody této třídy pro obsluhu protokolu.

4.3 Lokalizace

Jelikož zdaleka né všichni uživatelé ovládají anglický jazyk, který byl zvolen jako základní jazyk uživatelského rozhraní, je vhodné umožnit tuto skutečnost změnit. Toolkit *wxWidgets* poskytuje třídu `wxLocale`, která využívá systému *gettext*. Tento systém pracuje se zkompilovanými soubory `.mo`[3] , ve kterých hledá překlady jednotlivých řetězců textu. Pro získání překladu je k dispozici funkce `wxGetTranslation`, která vrací překlad přijatého řetězce, místo této funkce se ovšem častěji využívá definované makro `_()`, kterým se obalují veškeré řetězce, které mají být při použití automaticky přeloženy podle použité lokalizace.

Klient obsahuje pro účely lokalizace třídu `AppLanguage`, která rozšiřuje funkcionalitu třídy `wxLocale`. Třída je modelována jako singleton, k němuž se přistupuje voláním statické metoty `Get` a umožňuje dynamické a jednoduché přidávání nových lokalizací bez nutnosti úprav v kódu. Kromě souborů s překladem potřebuje třída ještě soubor `languages.xml`, který obsahuje seznam dostupných lokalizací. Ukázku obsahu tohoto souboru je možné vidět na obrázku 4.3.

Parser předpokládá UTF-8 jako použité kódování tohoto souboru, parametr `id` obsahuje textovou identifikaci lokalizace, podle které se vyhledá příslušný `.mo` soubor s překladem a `text` obsahuje název jazyka, který bude zobrazen uživateli. Třída poskytuje metodu `GetLanguageList`, kterou mohou jiné části programu získat seznam jazyků, které jsou

[3] Tyto soubory se vytvářejí zkompilováním `.po` textových souborů, které obsahují dvojice řetězec (`msgid`) - překlad (`msgstr`), pomocí utility `msgfmt`.

```
<languages>
  <language id=‘‘English’’ text=‘‘English’’ />
  <language id=‘‘Czech’’ text=‘‘Čeština’’ />
</languages>
```

Obrázek 4.3: Příklad obsahu souboru `languages.xml`

v tomto souboru obsaženy a jsou tedy pravděpodobně k dispozici. K nastavení požadované lokalizace slouží metoda `SetLanguage`, která přijímá textovou identifikaci jazyka (obsah parametru `id`). Pokud se nepodaří nalézt `.mo` soubor s překladem k vybranému jazyku, bude použit implicitní jazyk, což je v tomto případě angličtina.

Kromě popsaných funkcí ale dokáže tato třída ještě automaticky generovat nabídku jazyků do zadaného menu. Nabídka má formát submenu, které obsahuje seznam jazyků k dispozici a možnost obnovení tohoto seznamu. Třída požaduje zadání menu, do kterého se vloží prvek, skrz který se k tomuto submenu bude přistupovat, a umožňuje specifikaci pozice tohoto prvku v rámci zadaného menu. Obsluhu událostí této části menu zajišťuje sama třída `AppLanguage`, která se u třídy okna, obsahujícího menu, automaticky zaregistruje jako odběratel událostí pro tyto položky.

Postup pro přidání nové lokalizace je tedy následující:

- Získat soubor `saber.po`, který je umístěn v adresáři `languages`, dopsat do něj příslušné překlady a zkompilovat ho do `.mo` formátu (k editaci i kompilaci `.po` souboru je možné využít například program poEdit (http://www.poedit.net/)).
- Vytvořený `saber.mo` soubor s překladem uložit do adresáře `languages/<lang>/`, kde `<lang>` je textová identifikace jazyka, (např. `languages/cs_CZ/`).
- Zaznamenat nový jazyk do souboru `resources/languages.xml`.

4.4 Uživatelské rozhraní

4.4.1 Panely

Pro možnost vlastního uspořádání jednotlivých částí hlavního okna je možné tyto části rozdělit do tzv. plovoucích panelů. Tyto panely je možné přeskupovat, měnit jejich velikost, maximalizovat přes ostatní panely nebo je úplně vyjmout z hlavního okna a ponechat jako samostatné plovoucí okna. Toolkit *wxWidgets* obsahuje novou třídu `wxAUI` (*Advanced User Interface*), která tuto funkcionalitu poskytuje.

Hlavní třída `wxAUI` je `wxAuiManager`, která zajišťuje veškerou obsluhu jednotlivých panelů. Klient ovšem nepoužívá tuto třídu, nýbrž třídu z ní poděděnou `SaberFrameManager`, která přepisuje metodu `ProcessDockResult` rozhodující o platnosti ukotvení přesouvaného panelu. Toto je nutné pro omezení některých rozvržení, kdy by například mohlo docházet k odsunutí panelu za okraj klientské části okna, kde by panel nebyl vůbec vidět.

Bohužel zde existuje ještě jeden problém, který nastává při změně velikosti hlavního okna. Protože `wxAUI` je relativně nová třída a na jejím vývoji se neustále pracuje, není ještě úplně funkční. Ikdyž tato chybějící funkcionalita nijak nebrání jejímu využití, zde nastává problém. `wxAUI` totiž momentálně ignoruje nastavení maximální velikostí panelů. Díky tomu není možné při zmenšení velikosti okna zařídit, aby se panely přizpůsobily

nové velikosti. Tento problém klient momentálně obchází pomocí nastavení minimální velikosti a přenastavení velikosti panelů na jedna. Při změně velikosti se nejprve načte z třídy `SaberFrameManager` tzv. pohled, což je textový řetězec, který popisuje rozmístění a nastavení panelů, v tomto řetězci se upraví části specifikující velikosti panelů na jedna a nový pohled se zpět uloží do této třídy. Tímto se dosáhne jakéhosi znovunastavení všech panelů do minimální velikosti a následným nastavením minimální velikosti jednotlivých panelů, podle velikosti klientské části okna, budou panely nuceny změnit svou velikost, aby toto nastavení dodržely. Tímto se získá možnost specifikovat velikost panelů pouze nastavováním minimální velikosti, čímž přestane být ignorování nastavení maximální velikosti problém. Toto řešení je bohužel momentálně jediné možné, ikdyž je trochu náročnejší.

4.4.2 XRC

Jak je již známo z kapitoly 2.4.2 je XRC neboli XML Resource jazykem UIML založeným na jazyce XML. Jelikož je XRC standard *wxWidgets* nic nebrání k jeho plnému využití pro popis uživatelského rozhraní.

Pro práci s XRC slouží třída `wxXmlResource`. Tato třída je implementována jako singleton a přistupuje se k ní pomocí statické metody `Get`. Umožňuje zpracování `.xrc` souborů s popisem rozhraní a poskytuje metody pro automatické vytvoření uživatelského rozhraní z těchto informací. XRC soubor může obsahovat popis většiny grafických komponent implementovaných ve *wxWidgets* i popis některých jiných objektů tříd jako například `wxBitmap` nebo `wxIcon` a zde možnosti zdaleka nekončí. Samotná třída `wxXmlResource` totiž neumí objekty popsané v `.xrc` souboru zpracovat, k tomu využívá tzv. handlerů, což jsou instance tříd poděděných z třídy `wxXmlResourceHandler`, které jsou schopny vytvořit daný objekt podle jeho popisu v `.xrc` souboru. Kdykoliv je tedy možné vytvořit vlastní handler, který zpracuje určitý objekt definovaný XRC popisem a tím rozšířit možnosti tohoto systému.

Na obrázku 4.4 je vidět příklad `.xrc` souboru, který obsahuje popis popup menu[4] , které se objeví při stisku pravého tlačítka nad položkou seznamu kontaktů.

Z příkladu je vidět, že dokument se skládá z položek `object`, které reprezentují jednotlivé objekty a jsou hierarchicky uspořádané podle jejich rozložení v dané grafické komponentě. Každá položka `object` vetšinou obsahuje dva atributy:

- atribut `name`, který identifikuje danou instanci objektu. Třída `wxXmlResource` z XRC popisu sice vytvoří uživatelské rozhraní, ale to je samo o sobě k ničemu, je třeba aby program reagoval na události, které v něm nastanou. Při definici obsluhy události je ovšem potřeba specifikovat ID (identifikátor) komponenty, jejíž událost se má obsloužit. Toto ID lze získat pomocí této textové identifikace metodou `GetXRCID`, která vrací ID asociované s danou instancí objektu. Pro pohodlnější použití v tabulkách událostí, které *wxWidgets* používá ke specifikaci obsluh událostí, je definováno makro `XRCID`. Zpracování událostí ale zaručí jen omezenou interakci uživatele s programem, často je třeba také zpracovat vložený text nebo naopak vypsat odpověď, k čemuž je nutné mít přístup k jednotlivým komponentám. K tomu slouží makro `XRCCTRL`, kterým lze pomocí textové identifikace objektu získat ukazatel na tento objekt. Je nutno poznamenat, že systém nijak nekontroluje duplicitu této identifikace, takže sám programátor musí zajistit unikátnost textové identifikace.

[4] Popup menu by se dalo přeložit jako vyskakovací nabídka. Zde označuje nabídku, která se objeví pro pravém stisknutí tlačítka myši. Protože český překlad je celkem matoucí a v praxi se stejně vetšinou používá anglické označení, bude upřednostněno i zde.

```
<?xml version=‘‘1.0’’ encoding=‘‘UTF-8’’ standalone=‘‘yes’’ ?>
<resource xmlns=‘‘http://www.wxwindows.org/wxxrc’’ version=‘‘2.3.0.1’’>
  <object class=‘‘wxMenu’’ name=‘‘cListPopupMenu’’>
    <label>Contact List Menu</label>
    <object class=‘‘wxMenuItem’’ name=‘‘cListPopupMenu_SendMessage’’>
      <label>Send Message</label>
      <help></help>
    </object>
    <object class=‘‘wxMenuItem’’ name=‘‘cListPopupMenu_SendFile’’>
      <label>Send File</label>
      <help></help>
    </object>
    <object class=‘‘separator’’ />
    <object class=‘‘wxMenuItem’’ name=‘‘cListPopupMenu_ContactInfo’’>
      <label>View Contact Information</label>
      <help></help>
    </object>
    <object class=‘‘wxMenuItem’’ name=‘‘cListPopupMenu_Rename’’>
      <label>Rename</label>
      <help></help>
    </object>
    <object class=‘‘separator’’ />
    <object class=‘‘wxMenuItem’’ name=‘‘cListPopupMenu_ContactHistory’’>
      <label>View Contact History</label>
      <help></help>
    </object>
  </object>
</resource>
```

Obrázek 4.4: Příklad XRC popisu popup menu

- atribut `class`, který určuje třídu, které je objekt instancí. Podle tohoto atributu zjišťuje `wxXmlResource` který ze svých registrovaných handlerů má použít ke zpracování tohoto objektu.

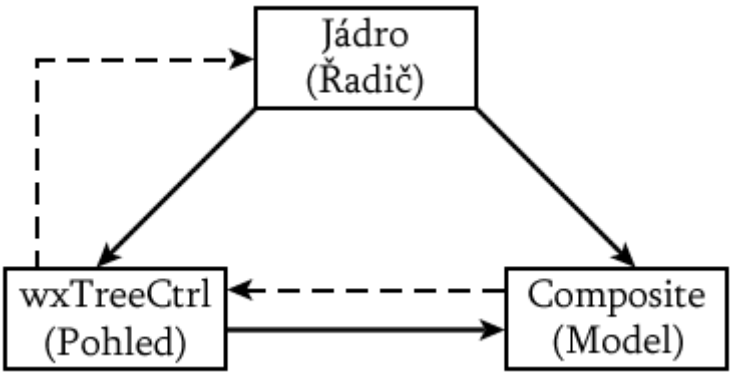
Kromě těchto dvou atributů může položka `object` ještě obsahovat synovské položky s informacemi o nastavení a vlastnostech daného objektu, tyto položky jsou specifické pro každou třídu objektu. Pro více informací k XRC viz. [2].

Klient kromě třídy `wxXmlResource` využívá ještě vlastní malou třídu `SaberXRC`, kde centralizuje načítání všech .xrc souborů využívaných klientem. Veškeré tyto soubory jsou umístěny v komprimovaném souboru `resources.zip` v adresáři `resources`. Protože třída `wxAUI` je nová, není momentálně k dispozici handler, který by umožňoval zpracovat nastavení a vlastnosti panelů z XRC popisu. Proto má klient implementován vlastní handler `auiPaneInfoXmlHandler`, který umožňuje zpracování objektu třídy `wxAuiPaneInfo`, obsahující všechny parametry `wxAUI` panelu, z XRC popisu. Třída `SaberXRC` obsahuje metodu `LoadAuiPaneInfo`, která vytvoří objekt této třídy z XRC popisu.

4.5 Seznam kontaktů

Seznam kontaktů udržuje informace o uživatelích, se kterými může uživatel komunikovat. Jednotlivé kontakty lze roztřizovat pomocí skupin. Skupiny mohou obsahovat jak uživatele, tak jiné skupiny. Celkově tedy vzniká stromová struktura několika různých entit.

Pro implementaci by bylo samozřejmě možné využít komponenty `wxTreeCtrl`, která slouží k zobrazení stromových struktur, ale nabízí se zde taky jiné řešení a tím je využití architektonického vzoru Model-View-Controller popsaného v kapitole 2.3. Tento vzor pracuje se třemi komponentami - model, pohled a řadič. Jako pohled, který bude zobrazovat seznam kontaktů uživateli, lze použít již zmíněmou komponentu `wxTreeCtrl`. Jako řadič, který vše obsluhuje, je možné využít přímo jádro. Poslední otázkou zůstává model. V seznamu kontaktů existují dvě entity - uživatel a skupina. Ikdyž se zdá, že tyto dvě entity jsou dosti odlišné a je třeba s nimi pracovat odděleně, není tomu tak. Klienta nezajímá jestli pracuje s uživatelem nebo skupinou, pokud chce uživatel odeslat zprávu vybranému uživateli, pošle se pouze tomuto uživateli, pokud chce zprávu poslat skupině, pošle se všem uživatelům patřícím do této skupiny. Je tedy vidět, že entity musí být v modelu hierarchicky uspořádané, aby bylo možné zjistit například kteří uživatelé patří do které skupiny, a musí k nim být umožněn jednotný přístup. K řešení tohoto problému se přímo nabízí návrhový vzor *composite* [5], který se zabývá řešením, jak uspořádat jednoduché objekty a z nich složené (kompozitní) objekty a zajistit, aby se k oběma typům těchto objektů přistupovalo jednotným způsobem. Výsledný návrh jednotlivých komponent je vidět na obrázku 4.5.

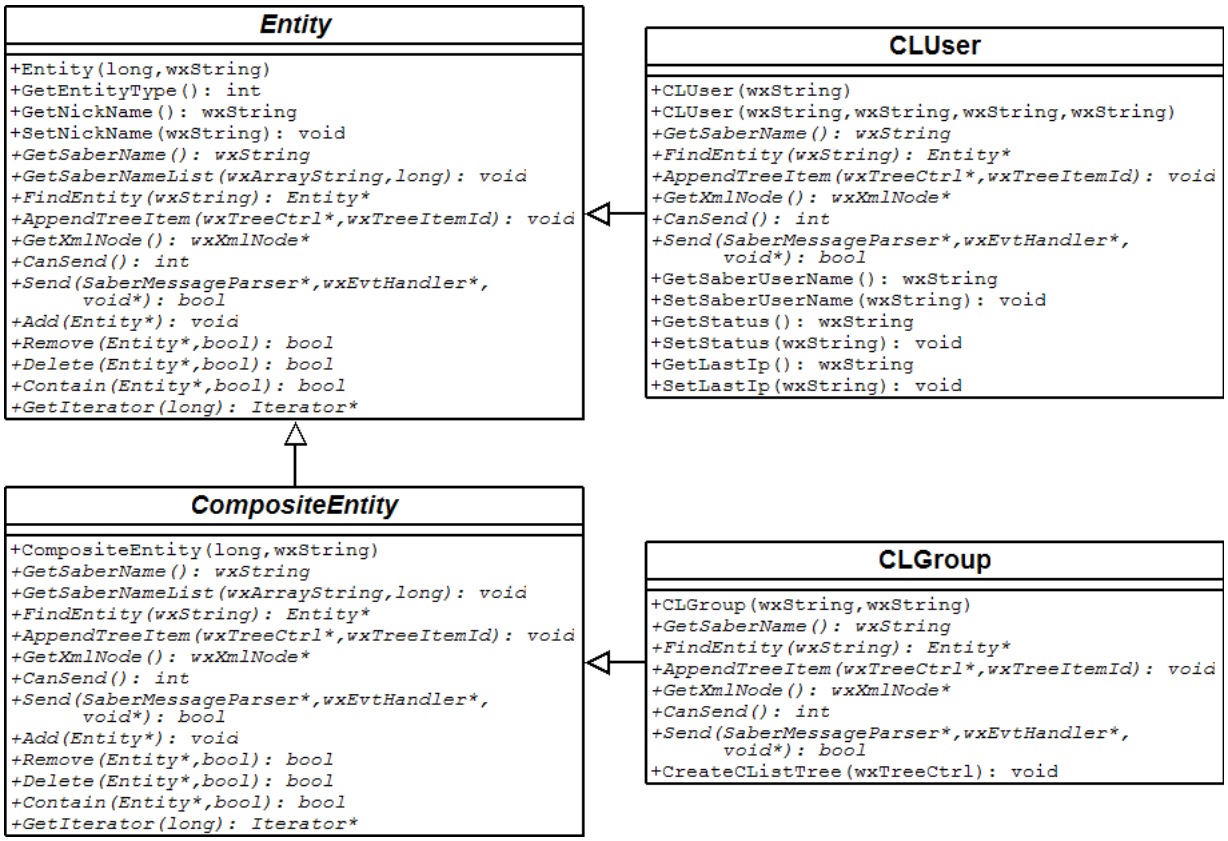


Obrázek 4.5: Komponenty MVC implementace seznamu kontaktů

4.5.1 Implementace modelu

Implementace vychází ze specifikace návrhového vzoru *composite*. Struktura tříd, které tento model implementují je na obrázku 4.6.

Z obrázku je patrné, že celý model implementují čtyři třídy. Jako rozhraní, které umožňuje přistupovat ke všem entitám seznamu kontaktů jednotným způsobem, je zde třída `Entity`, ze které dědí všechny ostatní třídy, ať již přímo nebo nepřímo. Tato třída slouží jako bázová třída všech jednoduchých objektů. Druhou stěžejní třídou modelu je `CompositeEntity`, která slouží jako bázová třída všech složených objektů, které mohou obsahovat další jednoduché nebo jiné složené objekty. Tyto dvě třídy umožňují konstruovat stromovou hierarchickou strukturu, kde `CompositeEntity` je možné si představit jako uzel stromu, který může obsa-



Obrázek 4.6: Struktura tříd implemetujících model seznamu kontaktů

hovat synovské uzly a listy, a `Entity` jako list stromu, který již nic obsahovat nemůže. Obě tyto třídy zde ovšem působí jako abstraktní třídy, které mají `protected` konstruktor a tímto je nelze přímo vytvářet. Samotný model obsahuje pouze objekty dalších dvou tříd `CLUser`, která zastupuje uživatele jako jednoduchou entitu a dědí proto z třídy `Entity`, a `CLGroup`, která zastupuje skupinu jako složenou entitu a dědí proto z třídy `CompositeEntity`.

Implementace modelu využívá principů polymorfismu[5] , který je v jazyce C++ zajištěn pomocí virtuálních metod (více viz. [6]). Třída `Entity` definuje veškeré virtuální metody, které používají jak jednoduché, tak složené entity. Metody vázané k složeným entitám jsou v této třídě prázdné, nebo vracejí `false` pro vyjádření nemožnosti provedení požadované operace. Pro možnost zjištění typu entity za běhu programu obsahuje třída metodu `GetEntityType`, která vrací číselnou identifikaci typu. Tento typ se definuje v konstruktoru a nelze po vytvoření nijak změnit. Třída `CompositeEntity` již definuje funkce vázané k obsluze složených entity. Stejně jako ve specifikaci návrhového vzoru *Composite* poskytuje metody `Add`, `Remove` a `GetIterator`, již zde jsou ale mírné úpravy. Kromě metody `Remove` definuje třída i metodu `Delete`, která kromě vyjmutí entity jí i automaticky vymaže, obě tyto metody také umožňují rekurzi těchto metod skrz další složené entity. Dále obsahuje metodu `Contain`, která zjistí, zda složená entita obsahuje předanou entitu. Nejzajímavější je ale metoda `GetIterator`, která umožňuje vytvořit filtrovatelný iterátor. Tento iterátor se může

[5] Polymorfismus v OOP je schopnost objektu, který může být více typů, reagovat na volání metody spuštěním příslušné metody třídy, které je objekt instancí.

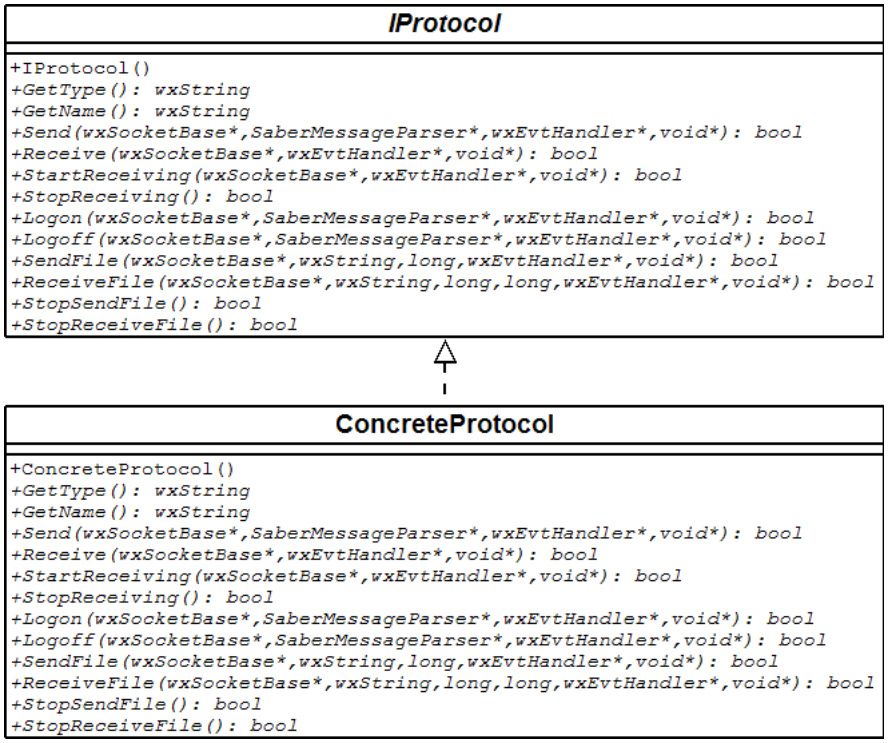
nastavit, aby iteroval pouze přes specifikované entity, čímž umožňuje aplikovat jednotlivé metody pouze na specifickou část modelu. Ikdyž tato funkce zatím nemá přílišné využití, v připravovaných rozšířeních bude nepostradatelná.

Kromě těchto základních metod poskytují třídy ještě metody vázané již k samotnému klientovi. Tyto metody plně využívají výhod použitého návrhového vzoru *composite*. Metoda `AppendTreeItem` slouží k vložení reprezentace entity do `wxTreeCtrl`, zavoláním této metody na kořenovou skupinu modelu dojde k sestavení celého obsahu `wxTreeCtrl` a tímto k zobrazení seznamu kontaktů uživateli. Metoda `GetXmlNode` vrací reprezentaci entity jako uzlu XML dokumentu, zavoláním této metody na kořenovou skupinu modelu dojde k vytvoření reprezentace seznamu kontaktů v jazyce XML. Tato funkce je stěžejní funkcí pro uložení seznamu kontaktů do XML souboru. Metoda `Send` odešle předanou zprávu uživateli (pokud je entita uživatel) nebo uživatelům (pokud je entita skupina). Jedinou metodou, která zatím využívá výhod filtrovacího iterátoru, je `GetSaberNameList`, která vytváří seznam identifikátorů entity a iterátor umožňuje omezit tento seznam pouze na určité typy entit.

4.6 Komunikace

4.6.1 Protokol

Protože hlavní cíl při implementaci klienta byl kladen na nezávislost na sémantice protokolu, samotný klient nepracuje s žádnou konkrétní třídou protokolu, ale pouze s třídou, která slouží jako univerzální rozhraní mezi klientem a konkrétní třídou s implementací. Tímto rozhraním je třída `IProtocol` a její definici ukazuje obrázek 4.7

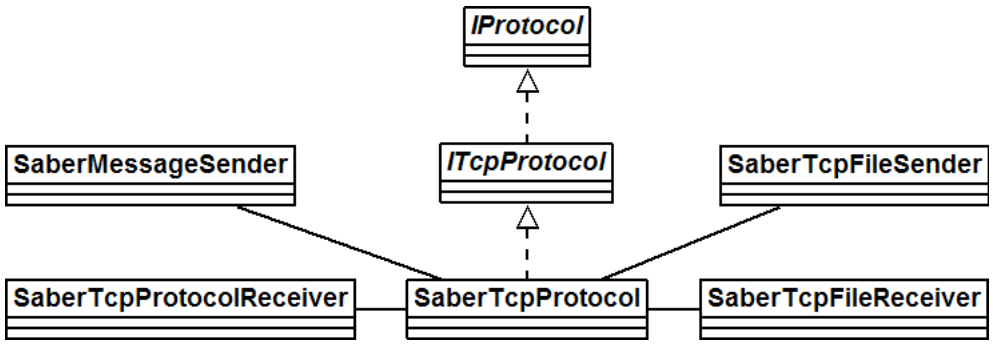


Obrázek 4.7: Definice rozhraní protokolu

První dvě metody třídy (`GetType` a `GetName`) jsou definovány jako standartní virtuální funkce. Tyto metody slouží hlavně k identifikaci třídy za běhu programu. Důležité jsou zbylé metody, které jsou čistě virtuální a tudíž je musí poděděná třída všechny implementovat. Metody `Send` a `Receive` slouží k odeslání nebo přijmu jediné zprávy. Metoda `StartReceiving` spouští proces neustálého příjmu zpráv a metoda `StopReceiving` tento proces zastavuje. Metody `Logon` a `Logoff` slouží k přihlášení a odhlášení uživatele. A poslední čtyři metody (`SendFile`, `ReceiveFile`, `StopSendFile` a `StopReceiveFile`) slouží k odeslání a příjmu souboru a k stornování těchto procesů. Postup vytvoření vlastního protokolu je tedy následující:

- Vytvořit třídu poděděnou z třídy rozhraní `IProtocol`.
- Implementovat všech deset čistě virtuálních metod.

Klient samotný využívá protokol implementovaný třídou `SaberTcpProtocol`. Tato třída nedědí ale přímo z třídy `IProtocol`, ale z třídy `ITcpProtocol`. Na obrázku 4.8 jsou zobrazeny závislosti mezi třídou `SaberTcpProtocol` a jinými třídami.



Obrázek 4.8: Závislosti třídy `ITcpProtocol` na jiných třídách

Třída `ITcpProtocol` prakticky pouze redeklaruje rozhraní zděděné z třídy `IProtocol`, jedinou změnou v této třídě je konečná implementace metody `GetType`, která říká, že daný protokol využívá TCP protokol jako transportní protokol. Snahou této třídy je pouze přehledně rozlišit protokoly podle použitých protokolů nižších vrstev (např. TCP a UDP).

Samotná třída `SaberTcpProtocol`, jak je vidět, využívá pro svou práci hned několik tříd pro zajištění veškeré funkcionality:

- `SaberMessageSender` obstarává posílání zpráv. Využívá k tomu thread pool s jedním vláknem. Tento přístup je výhodný, protože o veškerou obsluhu vlákna se stará thread pool a vlákno pracuje pouze pokud má odesílat nějakou zprávu, po zbytek času je pozastavené. Pro více informací o implementaci thread poolu viz. [3].
- `SaberTcpProtocolReceiver` obstarává příjem zpráv. Pro příjem vytvaří tzv. *detached thread*[6] , který umožňuje jak příjem jedné zprávy, tak nepřetržité naslouchání. Po příjmu celé zprávy dojde k její transformaci do interní reprezentace a vyvolání události `saberEVT_MESSAGE_RECEIVED`, jejíž obsluhu obstará jádro. Jako parametry

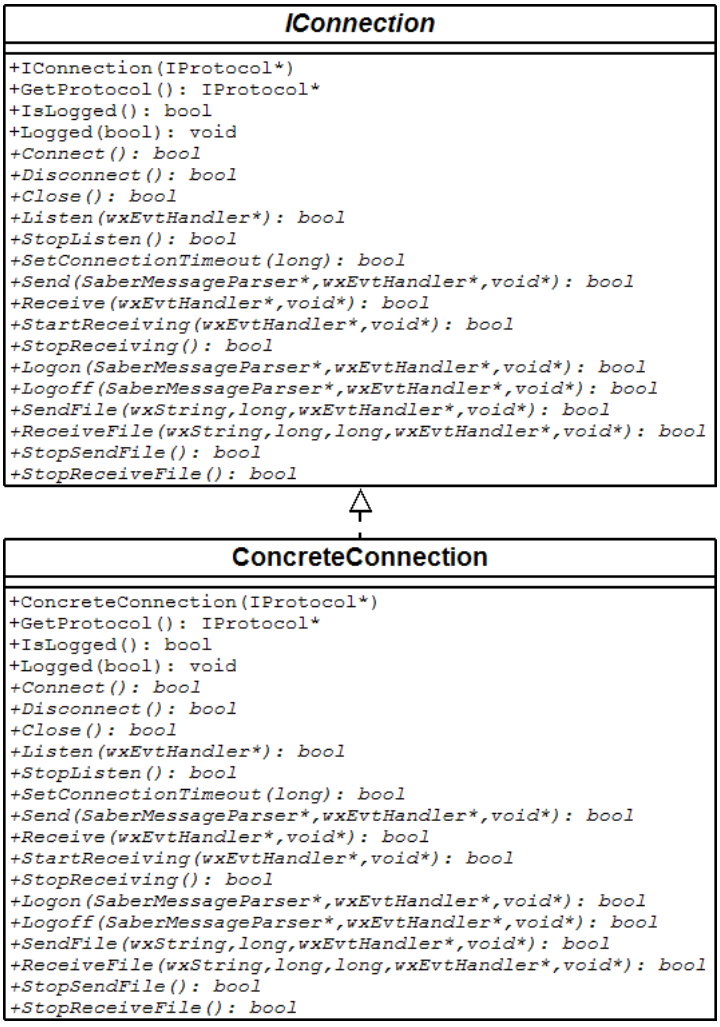
[6] *Detached thread* narozdíl od *joinable thread* se po skončení své vstupní funkce sám uvolní z paměti. *Joinable thread* je nutné uvolnit explicitně z jiného vlákna.

této události předané obslužné funkci jsou interní reprezentace zprávy a ukazatel na objekt reprezentující spojení, na kterém byla tato zpráva přijata.

- `SaberTcpFileSender` obstarává posílání souborů. Pro každý odesílaný soubor se vytváří separátní *detached thread*, který v průběhu posílání dat informuje jádro skrz události `saberEVT_FT_DATA_SENDED` a `saberEVT_FT_FILE_SENDED` o postupu přenosu.
- `SaberTcpFileReceiver` obstarává příjem souborů. Pro každý přijímaný soubor se vytváří separátní *detached thread*, který v průběhu přijímání dat informuje jádro skrz události `saberEVT_FT_DATA_RECEIVED` a `saberEVT_FT_FILE_RECEIVED` o postupu přenosu.

4.6.2 Spojení

Jako v případě protokolu i zde klient nepracuje přímo s konkrétní třídou spojení, ale s univerzálním rozhraním, které je implementováno třídou `IConnection` jejíž definice je na obrázku 4.9.



Obrázek 4.9: Definice rozhraní spojení

Třída `IConnection` vyžaduje v konstruktoru instanci nějaké třídy protokolu, který bude toto spojení využívat, přístup k tomuto protokolu lze poté získat pomocí metody `GetProtocol`. Pomocí metod `IsLogged` (`Logged`) lze zjistit (nastavit), zda je uživatel na daném spojení přihlášen. Zbylé metody jsou již virtuální a definují rozhraní. Narozdíl od třídy `IProtocol` nejsou ale tyto metody čistě virtuální, takže lze v poděděných třídach implementovat jen část těchto metod. Tento přístup je zvolen z důvodu rozličnosti účelu jednotlivých spojení, které většinou nepotřebují implementovat všechny metody, které poskytuje rozhraní, v tomto případě se použijou implementace z třídy `IConnection`, které všechny vracejí `false`, aby signalizovaly, že požadovaná operace nelze provést.

Klient obsahuje tři třídy implementující spojení pro různé účely:

- `SaberClientConnection` poskytuje metody pro připojení k serveru nebo jinému klientovi jako `Connect` a `Disconnect` a veškeré metody, které poskytuje protokol. Tento typ spojení se využívá pro připojení klienta k serveru a veškerou komunikaci s ním nebo třeba při spojení ke klientovi, kterému se zasílá soubor.
- `SaberServerConnection` poskytuje pouze metody pro naslouchání na zadaném portu jako `Listen` a `StopListen`. Tento typ spojení se používá k naslouchání na příchozí spojení pro přenos souborů.
- `SaberGenericConnection` poskytuje pouze metody protokolu. Tento typ spojení se využívá pro příjem souborů.

4.7 Zprávy

4.7.1 Posílání a příjem zpráv

Posílání a příjem zpráv je primární službou celého systému. Interakce této služby s uživatelem probíhá pomocí komunikačního okna. Toto okno obsahuje komunikační panel pro každého uživatele se kterým je vedena textová konverzace.

Jakmile uživatel zadá zprávu do textového pole komunikačního panelu a odešle ji, zpracuje jádro tento text a vytvoří interní reprezentaci zprávy, která ovšem zatím neobsahuje specifikaci ani příjemce, ani odesilatele. Tato interní reprezentace se předá entitě seznamu kontaktů, které se zpráva posílá, a ta do ní automaticky doplní příjemce a předá ji objektu serverového spojení, který do ní přídá odesilatele a předá ji instanci protokolu. Tato instance ji transformuje do svého formátu a odešle. Jak je vidět i jednoduché odeslání zprávy vyžaduje spolupraci skoro všech částí klienta.

Zprávy lze posílat i uživatelům, kteří nejsou momentálně přihlášeni. Tyto zprávy jim budou doručeny, jakmile se přihlásí.

Příjem zpráv zajišťuje protokol, který je transformuje do interní reprezentace. Ta je pak zpracována jádrem, který přijatou zprávu zobrazí uživateli a pošle serveru odpověď, že daná zpráva byla úspěšně doručena.

4.7.2 Historie zpráv

Při odeslání nebo příjmutí zprávy je tato zpráva uložena do lokálního souboru s historií. Tyto soubory jsou ve formátu jazyka XML a obsahují informace o čase odeslání zprávy (tento čas je vyjádřen v UTC (*Coordinated Universal Time*)), čase přijetí zprávy (tento čas je vyjádřen lokálním časem), typem zprávy (zda je zpráva příchozí nebo odchozí)

a samotným textem zprávy. Informace o odesilateli a příjemci zde nejsou nutné, protože je lze odvodit z typu zprávy.

Pro hledání v historii je klient vybaven prohlížečem historie, ten umožňuje hledání jak v lokální historii, tak v historii uložené na serveru. Hledání pouze v lokální historii má výhodu, že nepotřebuje k práci server a lze tedy provádět i offline (tzn. bez připojení k serveru), ale zase není zaručeno, že lokální historie obsahuje veškeré odeslané a přijaté zprávy. Hledání pouze na serveru sice zatěžuje síť a musí se čekat na odpověď serveru, ale zaručuje hledání mezi veškerými zprávami. Určitým kompromisem je využití hledání v lokální historii i na serveru zároveň. Nejprve se prohledá lokální historie a zobrazí se výsledky, poté se pošle požadavek na server, ve kterém se specifikuje počet nalezených zpráv pro daný dotaz v lokální historii, pokud server zjistí, že našel více zpráv pro daný dotaz, pošle tyto zprávy klientovi, který je zobrazí. Tento způsob tímto zatíží síť pouze pokud server obsahuje nějaké zprávy, které v lokální historii chybí.

Prohlížeč historie poskytuje dvě možnosti prohledávání historie. První je standardní hledání v textu zprávy a druhá je zobrazení historie pro daný den, k čemuž je v prohlížečí k dispozici kalendář pro snadný výběr požadovaného dne.

4.8 Přenos souborů

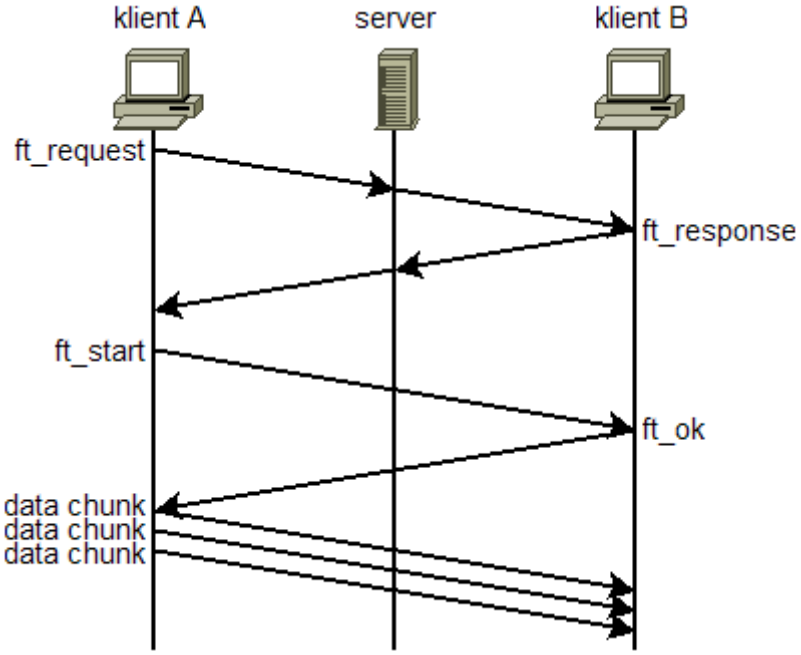
Přenos souborů je další důležitá služba, kterou musí systém zajistit. Tuto službu musí poskytovat za všech okolností.

Stejně jako u přenosu zpráv i samotný přenos souborů je realizován protokolem. Vizuální interpretaci tohoto přenosu zprostředkováná uživateli stavové okno, které obsahuje veškeré informace o přenosu a jeho průběhu. Jak již bylo řečeno v kapitole 4.6.1, protokol pomocí událostí informuje jádro o průběhu přenosu a to podle těchto informací aktualizuje stavové okno.

Samotný přenos může být realizován dvěma způsoby. První je přenos v režimu peer-to-peer, kdy jsou data souboru přenášena přímo mezi dvěma klienty. Tento způsob je také nejčastější. Druhý je přenos dat souboru přes server. Tento způsob je použit v případě, když není možné vytvořit spojení mezi dvěma klienty. Tato situace může nastat například pokud jsou klienti v privátní síťi a k internetu přistupují pomocí systému NAT[7] . Oba způsoby posílají data souboru po částech (tzv. chunks) předem specifikované velikosti. Protože se jednotlivé části posílají ihned za sebou, může se jevit, že toto rozdělení na části je zbytečné. Ovšem některé protokoly nižších vrstev (jako např. UDP) umožňují najednou odeslat pouze určitý objem dat a tímto je nutné data souboru rozdělit na menší části.

Na obrázku 4.10 je znázorněn průběh celého procesu přenosu souborů pokud se oba klienti mohou spojit. V první části, přípravné fázi přenosu, se vyžaduje přítomnost serveru, protože klienti nemají ještě informace o možnostech jejich spojení navzájem. Odesilatel pošle příjemci zprávu `ft_request`, kterou žádá o přenos souboru, příjemnce na tuto zprávu reaguje zprávou `ft_response` kde specifikuje, že přijal nebo odmítnul žádost o přenos souboru. Pokud příjemce přijal žádost o přenos, dodá do odpovědi informace o IP adrese a portu, které umožňují odesilateli se s ním spojit. Odesilatel následně vytvoří spojení s příjemcem a pošle mu zprávu `ft_start`, která obsahuje informace o přenosu jako velikost a počet částí, po kterých budou data souboru posílána. Pokud je daný přenos očekáván,

[7] NAT (*Network Address Translation*) je systém pro překlad adres z privátní sítě do sítě veřejné. Adresy používané v privátní síťi totiž nelze použít v síťi internet. NAT přepíše při komunikaci IP adresu klienta svou veřejnou IP adresou a tím mu umožní komunikovat na internetu.



Obrázek 4.10: Průběh přenosu souborů

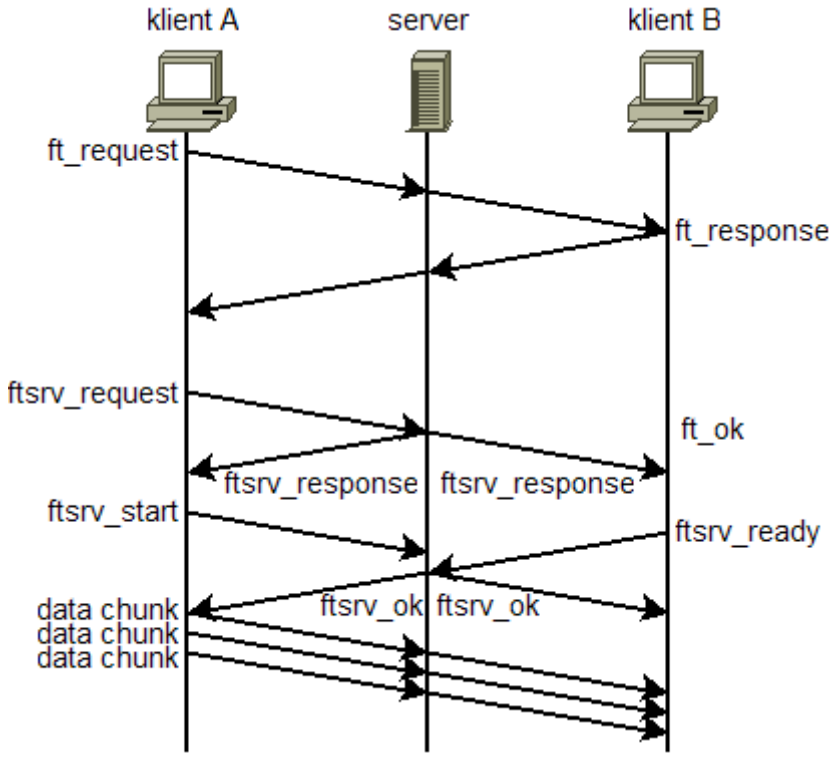
odpoví příjemce zprávou `ft_ok` a začne přijímat data na vytvořeném spojení. Jakmile obdrží odesilatel potvrzení, že přenos je očekáván, začne odesílat data souboru.

Pokud není možné, aby se klienti mezi sebou spojili, musí se k přenosu souboru využít server. Průběh přenosu souboru přes server je na obrázku 4.11. Přípravná fáze je zde stejná jako v předchozím případě, jakmile se ale odesilatel nedokáže spojit s příjemncem, odešle serveru zprávu `ftsrv_request`, kterou žádá o přenos přes server. Pokud server souhlasí s přenosem, odešle odesilateli i příjemci zprávu `ftsrv_response`, kde informuje klienty o IP adrese a portu, kde ho mají kontaktovat. Jakmile se odesilatel spojí se serverem, odešle mu zprávu `ftsrv_start`, která obsahuje informace o velikosti a počtu částí odesílaných dat. Příjemce po spojení se serverem zprávou `ftsrv_ready` signalizuje připravenost přijímat data. Teprve po kontaktování serveru oběma účastníky přenosu odešle server oběma zprávu `ftsrv_ok`, která odesilateli signalizuje, že může začít posílat data, a příjemce informuje o velikosti a počtu dat, které má začít přijímat.

4.9 Šifrování

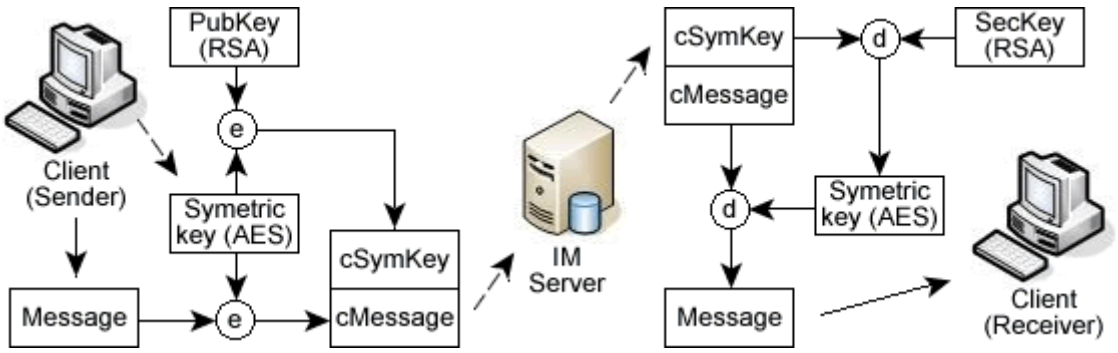
Pro zaručení bezpečného přenosu obsahu zasílaných zpráv obsahuje klient možnosti jejich šifrování, k čemuž využíva služeb knihovny *Crypto++*, která poskytuje širokou škálu kryptografických algoritmů.

V kapitole 2.6 již byly zmíněny postupy zabezpečení informací pomocí symetrické a asymetrické kryptografie a jejich výhody a nevýhody. Implementovaný systém využívá obou těchto přístupů zároveň. Text zasílané zprávy je zašifrován symetrickou šifrou, která pracuje velice rychle, s použitím náhodně vygenerovaného textového řetězce jako klíče. Následně dojde k zašifrování tohoto klíče asymetrickou šifrou s použitím veřejného klíče příjemce zprávy. Tyto informace jsou pak vloženy do zprávy a odeslány. Příjemce poté pomocí svého



Obrázek 4.11: Průběh přenosu souborů s využitím serveru

soukromého klíče dešifruje klíč použitý k zašifrování textu zprávy a tímto klíčem následně dešiftuje text zprávy. Tento postup poskytuje výhodu rychlého šifrování dat, které poskytují symetrické šifry, a zároveň zaručuje bezpečný přenos klíče symetrické šifry k příjemci, což je hlavní slabina symetrického šifrování. Průběh celého procesu je vidět na obrázku 4.12.



Obrázek 4.12: Průběh přenosu šifrované zprávy

Klient pro šifrování využívá třídu `SaberCrypt`. Tato třída poskytuje pro symetrické šifrování momentálně dvě šifry označené `aesphm` a `3des-mac`. `3des-mac` využívá k šifrování algoritmus Triple-DES a zajišťuje kontrolu integrity zprávy pomocí kontrolního součtu. `aesphm` (*AES Padded and Hashed Message*) využívá k šifrování algoritmus AES, který mo-

mentálně poskytuje největší zabezpečení ze symetrických algoritmů. Poskytuje také kontrolu integrity zprávy pomocí kontrolního součtu a tzv. vycpává zprávy, což znamená, že doplňuje k původní zprávě náhodný počet různých znaků, čímž zabraňuje možnosti zjištění délky původní zprávy z délky její šifry. Pro asymetrické šifrování poskytuje třída implementaci šifry RSA, která lze použít jak pro šifrování, tak pro podepisování, narozdíl od šifry DSA, kterou prakticky vystřídala. Kromě šifrování poskytuje také třída metodu pro vytváření kontrolních součtů řetězců s využitím algoritmu SHA1 a metodu pro generování páru veřejného a soukromého klíče pro asymetrickou kryptografii.

Kapitola 5

Závěr a zhodnocení

5.1 Směry dalšího vývoje

Tato realizace systému obsahuje pouze zlomek možností, než mohla mít. Návrhů na vylepšení nebo nové funkce by šla vymyslet celá řada. Stručně lze zmínit alespoň některé z nich:

- Doplnit možnost spojení mezi jednotlivými klienty. Tímto by klienti získali nezávislost na přítomnosti serveru a umožnilo by to komunikaci i v případě, že server není k dispozici. Pro toto rozšíření je potřeba implementovat v klientovi částečnou funkcionalitu serveru.
- Implementovat podporu emotikon a jejich snadné přidávání uživatelem.
- Implementovat stavové zprávy vyjadřující stav uživatele (např. uživatel není přítomen, uživatel je zaneprázdněn) a umožnit uživateli definici vlastních zpráv.
- Přidat další možnosti interakce mezi uživateli jako např. hlasová komunikace, video komunikace, tabule umožňující kreslení a dalších.
- Navrhnout a implementovat systém, který by umožnil klientovi využití externích zásuvných modulů, které by rozšiřovaly jeho funkcionalitu.

5.2 Závěr

Cílem této práce bylo vytvořit klientskou aplikaci, která bude sloužit systému pro zasílání textových zpráv k interakci s uživatelem.

Vytvořená aplikace poskytuje uživateli možnost bezpečné komunikace s dalšími uživateli systému formou textových zpráv a zpětné dohledání odeslaných a přijatých zpráv v historii. Dále uživateli umožňuje realizovat přenos souborů a to i za situace, že ke klientovi nelze vytvořit spojení. Klient také umožňuje přidávat nové lokalizace a částečne upravovat uživatelské rozhraní bez nutnosti rekompilace.

Bohužel z důvodu nedostatku času a rozsáhlosti kódu není aplikace momentálně úplně stabilní a někdy se může vyskytnout nečekaná chyba. Aplikace také ještě nebyla otestována na jiným operačních systémech než Microsoft Windows. Všechny povinné body zadání však byly úspěšně splněny.

Literatura

[1] World Wide Web Consortium. *W3C XML homepage*. Dostupné na URL: <http://www.w3.org/XML/>.

[2] Julian Smart et al. *wxWidgets manual* [online]. Poslední modifikace: 24. března 2007. [cit. 2007-05-11]. Dostupné na URL: <http://www.wxwidgets.org/manuals/stable/wx_contents.html>.

[3] Marek Gach. *Systém pro zasílání textových zpráv - serverová část*, 2007.

[4] Sun Microsystems. *JXTA official website*. Dostupné na URL: <http://www.jxta.org/>.

[5] The Gang of Four. *Design Patterns*. Addison-Wesley Publishing Company, 1994. ISBN 0-201-63361-2.

[6] Peringer P. *Seminář C++* [online]. Poslední modifikace: 12. května 2004. [cit. 2007-05-11]. Dostupné na URL: <https://www.fit.vutbr.cz/study/courses/ICP/public/Prednasky/ICP.pdf>.

[7] Wikipedia. *Client (computing)* [online]. Poslední modifikace: 7. května 2007. [cit. 2007-05-11]. Dostupné na URL: <http://en.wikipedia.org/wiki/Client_%28computing%29>.

[8] Wikipedia. *Cryptography* [online]. Poslední modifikace: 10. května 2007. [cit. 2007-05-11]. Dostupné na URL: <http://en.wikipedia.org/wiki/Cryptography>.

[9] Wikipedia. *Model-view-controller* [online]. Poslední modifikace: 9. května 2007. [cit. 2007-05-11]. Dostupné na URL: <http://en.wikipedia.org/wiki/Model-view-controller>.

[10] Wikipedia. *Peer-to-peer* [online]. Poslední modifikace: 10. května 2007. [cit. 2007-05-11]. Dostupné na URL: <http://en.wikipedia.org/wiki/P2p>.

[11] Wikipedia. *Unicode* [online]. Poslední modifikace: 15. dubna 2007. [cit. 2007-05-11]. Dostupné na URL: <http://cs.wikipedia.org/wiki/Unicode>.

[12] Wikipedia. *XML* [online]. Poslední modifikace: 15. dubna 2007. [cit. 2007-05-11]. Dostupné na URL: <http://cs.wikipedia.org/wiki/XML>.