VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

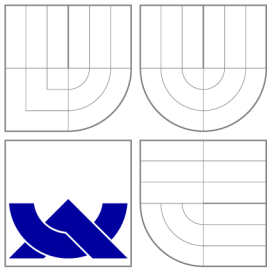
# SERVEROVÁ ČÁST WEBOVÉ APLIKACE PRO INSTANT MESSAGING S VYUŽITÍM AJAX

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

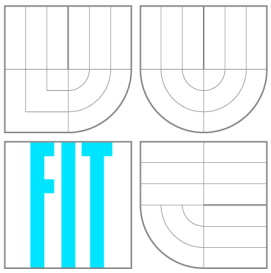
AUTOR PRÁCE
AUTHOR

VÁCLAV ŠVIRGA

BRNO 2014



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

# SERVEROVÁ ČÁST WEBOVÉ APLIKACE PRO INSTANT MESSAGING S VYUŽITÍM AJAX

SERVER COMPONENT OF A WEB APPLICATION FOR INSTANT MESSAGING WITH AJAX

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

VÁCLAV ŠVIRGA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. VLADIMÍR BARTÍK, Ph.D.

BRNO 2014

# Abstrakt

Tato práce se zabývá vytvořením serverové části webové aplikace pro instant messaging. V teoretické části jsou popsány běžné služby pro instant messaging a obvyklé webové technologie tvorby webových aplikací. V praktické části je navržena architektura aplikace, popsána její implementace a propojení jednotlivých částí skrze API. Závěrem práce je zhodnocení výsledné aplikace.



# Abstract

The subject of this thesis is to create a server component to an instant messaging web application. The theoretical part of the thesis describes common instant messaging services and technologies used for building web applications. The practical part of the thesis focuses on the design of the application architecture, the implementation and the connections between the components of the application via the API. The conclusion of the thesis covers evaluation of the application.



# Klíčová slova

instant messaging, IM, webová aplikace, AJAX, HTML, CSS, JavaScript, libpurple, C, PHP, MySQL, Nette, dibi, BSD sockety, Jabber, XMPP, ICQ, Google Talk, Facebook Chat

# Keywords

instant messaging, IM, web application, AJAX, HTML, CSS, JavaScript, libpurple, C, PHP, MySQL, Nette, dibi, BSD sockets, Jabber, XMPP, ICQ, Google Talk, Facebook Chat


# Citace

Václav Švirga: Serverová část webové aplikace pro instant messaging s využitím AJAX, bakalářská práce, Brno, FIT VUT v Brně, 2014

# Serverová část webové aplikace pro instant messaging s využitím AJAX

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Vladimíra Bartíka Ph.D.

.....
Václav Švirga
19. května 2014

## Poděkování

Je mou povinností poděkovat panu Ing. Vladimíru Bartíkovi za vedení při tvorbě práce. Dále Františkovi Sabovčikovi za obětavý vývoj klientské části aplikace a rovněž všem ostatním, co mě během práce podporovali.

© Václav Švirga, 2014.
*Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.*

# Obsah

# Kapitola 1

# Úvod

V poslední době vznikají webové aplikace zastávající činnosti, které byly kdysi ryze doménou desktopových aplikací. Webové rozhraní e-mailu dnes už nikoho nepřekvapí, ale jako webové aplikace vznikají i sofistikovanější nástroje, jakožto kancelářský balík, grafický editor či video střižna. I když webové implementace aplikací a služeb nemusí vždy poskytovat stejný komfort, jako desktopová verze, mají jednu velkou výhodu – dají se používat z jakéhokoliv počítače vybaveného internetem, bez nutnosti instalace, která není vždy možná. Uživatel takovéto služby, díky uložení dat u vzdáleného poskytovatele, nemusí synchronizovat data mezi počítači. Na druhou stranu je zde riziko ukončení služby poskytovatelem a ztráty funkcionality, hůře dat.

Instant messaging (dále IM) není v tomto trendu výjimkou. Všechny rozšířené IM služby/protokoly (Jabber, Facebook Chat, ICQ, ...) mají webovou aplikaci, přes kterou je lze z prohlížeče používat. Problém nastává ve chvíli, kdy uživatel používá více služeb. Už v dávných časech IM služeb vznikali desktop klienti, kteří dokázali obsluhovat více IM služeb, než jednu. Zmíněn může být například populární opensource klient Pidgin [1]. Tyto klienty využívali s velkou oblibou uživatelé, kteří využívají více IM služeb a nechtěli mít nainstalováno více různých klientů a přepínat mezi nimi. Další výhodou je jednotná správa dat (kontaktů a historie) a jednotné uživatelské rozhraní.

Na webu není situace jiná, mít v jednom panelu internetového prohlížeče otevřený např. Facebook a v druhém ICQ není příjemné a už v minulosti vznikaly služby, co měly tento problém vyřešit. Je nutné zmínit dvě nejvýznamnější služby, a to Meebo a Imo.im. Meebo byla velmi průkopnická služba, která si troufám tvrdit (autor práce byl dlouholetým uživatelem) poskytovala uživateli IM vše potřebné. Je to také velký zdroj inspirace pro tuto práci. Bohužel služba Meebo byla ukončena (2.1). Meebo později nahradila služba Imo.im, nyní ji ale potkal podobný osud (2.2).

Domnívám se, že po podobných aplikacích stále poptávka existuje a má práce je snahou na ni odpovědět.

# Kapitola 2

# Popis dřívějších řešení

Jak bylo zmíněno v úvodu, hlavní klíčové služby (a průkopníci) v oboru byly dvě: Meebo a Imo.im. Bohužel první služba již byla ukončena a druhá přestala plnit účel, který si tato práce bere za klíčový. Přesto je důležité obě služby zanalyzovat jako zdroj inspirace a zkušeností.

## 2.1 Meebo

Služba byla založena v roce 2005 [6]. Podporovala mnoho IM protokolů jako Yahoo! Messenger, Windows Live Messenger, AIM, ICQ, MySpaceIM, Facebook Chat, Google Talk, CafeMom, Zorpia a další.

Služba rovněž nabízela webovou aplikaci pro Android a iPhone.

Služba byla v roce 2012 odkoupena společností Google a ukončena [2].

Z vlastních zkušeností musím na Meebo docenit jednoduché a přehledné uživatelské rozhraní, které je vzorem pro tuto práci.

Jako knihovnu pro komunikaci s IM protokoly/službami používala libpurple.

## 2.2 Imo.im

Služba byla podobného zaměření jako Meebo, ovšem od 3. 3. 2014 přestala podporovat cizí protokoly/služby [3], čímž přestala být alternativou k předmětu práce.

Imo.im bych vytknul uživatelské rozhraní, které mi oproti Meebo přišlo méně přehledné.

## 2.3 Další služby

Meebo a Imo.im nejsou naštěstí jedinými službami. V současné době existují ještě např. služby Plus.im [4] a Iwantim [5]. Otázkou je, jak dlouhé budou mít trvání, když předchozí dvě významné služby tuto oblast opustily a kolik si získají uživatelů. V každém případě další konkurence v podobě této práce ničemu nemůže uškodit, možná bude nakonec nejlepší alternativou. Osobně mi, stejně jako na bývalé službě Imo.im, nevyhovuje uživatelské rozhraní obou služeb a zde vidím prostor proto, aby nová služba mohla zaujmout.

# Kapitola 3

# Instant messaging

Instant messaging (dále IM) je internetová služba umožnující uživatelům vyměňovat si zprávy (chatovat) v reálném čase. Dále umožňuje zobrazit stav uživatele, což je informace, zda je uživatel přítomen u počítače či ne a zda si přeje či nepřeje být rušen. V porovnání s e-mailem nabízí rychlejší formu komunikace, na druhou stranu není vhodná pro vyměňování delších textů, kvůli horším možnostem archivace, třídění a vyhledávání zpráv.

Rovněž můžeme IM rozdělit na komunikaci mezi dvěma osobami a skupinový chat, při němž spolu komunikuje více osob.

## 3.1 Protokoly IM

Popíšeme si nejvýznamnější IM služby/protokoly:

### 3.1.1 IRC

Internet Relay Chat (IRC) je nejstarší službou umožňující chatování po internetu. Protokol IRC je otevřený (např. RFC 1459 [7]) a postavený nad protokolem TCP. Na světě existuje mnoho IRC sítí, které obsluhují desítky IRC serverů. Nejznámější IRC sítě současnosti jsou freenode a IRCnet. Na IRC se komunikuje převážně v kanálech (channel), které představují chatovací místnosti pro skupinový chat. Komunikace dvou uživatelů je rovněž možná a nazývá se query. Integrace IRC do webové aplikace by měla smysl v případě, že by aplikace zvládala chat více uživatelů, jelikož pro komunikaci dvou lidí se IRC příliš nevyužívá. IRC je zmíněno hlavně z historických důvodů.

### 3.1.2 Jabber – XMPP

Extensible Messaging and Presence Protocol (XMPP – také známý jako Jabber) je protokol, který kromě IM může být použít pro vzdálené ovládání programů a služeb. Protokol je otevřený a postavený nad protokoly TCP a XML. O vývoj protokolu se stará XMPP Standards Foundation [8]. XMPP síť je decentralizovaná a existuje spousta poskytovatelů. Důležité je, že mezi jednotlivými sítěmi lze navzájem komunikovat. Každý uživatel má své Jabber ID (zkráceně JID) ve tvaru `uzivatel@server`. JID je tedy velmi podobný e-mailové adrese.

Protokol XMPP je důležitý i proto, že je přes něj možné komunikovat se dvěma významnými službami Google Talk (3.1.6) a Facebook (3.1.5). Proto by ho webová aplikace měla podporovat.

### 3.1.3 ICQ

ICQ [9] je program pro IM využívající protokolu OSCAR. Protokol OSCAR je proprietární a byl dlouhou dobu uzavřený (společnost AOL v roce 2008 uvolnila specifikaci), ovšem podařil se re-implementovat pomocí reverzního inženýrství. Síť je centralizovaná a vázaná na oficiálního klienta. ICQ je dlouhodobě kritizované [10] za restriktivní licenční podmínky (omezení alternativních klientů, zákaz používání služby pro komerční účely). Na druhou stranu je v České republice dosud rozšířen, proto by byla jeho podpora vhodná.

### 3.1.4 Skype

Skype [11] je program a služba umožňující provozovat internetovou (video)telefonii (VOIP) a také IM chat. Síť je centralizovaná a protokol je uzavřený a zatím se ho nepodařilo dešifrovat. Skype klient poskytoval oficiální API pro IM chat, bohužel v novějších verzích klienta bylo API zrušeno. Z toho důvodu ho není možné podporovat.

### 3.1.5 Facebook Chat

Chat ve webové službě Facebook se v poslední době stává (spolu s rozšířeností Facebooku) velmi rozšířenou formou IM komunikace. Naštěstí služba Facebook poskytuje k webovému chatu otevřené XMPP rozhraní [12].

### 3.1.6 Google Talk

Google Talk je VOIP a IM služba společnosti Google založená na protokolu XMPP. V současné době je ale plánované její ukončení a nahrazení službou Google+ Hangout [13]. Zatím ale Google Talk funguje a díky XMPP ho není obtížné podporovat.

## 3.2 Otevřenost protokolů

Pro maximální užitečnost webové aplikace s podporou více služeb/protokolů je potřeba, aby co nejvíce protokolů bylo otevřených nebo aspoň mělo přístupné API, aby je aplikace mohla podporovat. Bohužel současný trend je spíše opačný, což lze vidět na příkladu Skype (3.1.4) a Google Talk (3.1.6).

# Kapitola 4

# Webové aplikace

Webová aplikace je aplikací poskytovanou uživatelům z webového serveru přes počítačovou síť Internet (či její vnitropodnikovou obdobu intranet) do internetového prohlížeče, který se pak chová jako tenký klient. Oblíbenost webových aplikací pramení z možnosti přistupovat k nim skrze jakýkoliv počítač vybavený Internetem, bez nutnosti instalovat software či řešit synchronizaci dat – data jsou uložena u poskytovatele [14, s. 12].

Dále si popíšeme technologie nutné pro realizaci webové aplikace s tím, že se zaměříme na technologie využité v této práci. Budou zmíněny i technologie klientské části aplikace, přestože klientská část není přímo předmětem této práce. Při tvorbě webových aplikací se od klientských technologií nikdy nemůžeme 100% oddělit a je nutné jim porozumět.

## 4.1 Klient – server

Webové aplikace využívají síťové architektury klient – server, která odděluje klienta (konkrétně webový prohlížeč) a server (konkrétně webový server). Obě části spolu komunikují přes počítačovou síť, konkrétně Internet. Webový prohlížeč posílá požadavky na webový server skrze protokol HTTP, či jeho zabezpečenou verzi HTTPS a webový server naslouchá požadavkům. Při přijetí požadavku odešle odpověď [16, s. 30]. Webový prohlížeč nezná logiku aplikace, pouze ji interpretuje skrze HTML, CSS a JavaScriptový kód, proto ho můžeme nazývat tenkým klientem.

Pro realizaci plnohodnotné webové aplikace musíme vyřešit několik úskalí. HTTP protokol je bezstavový, HTTP dotazy nemají spolu souvislost. U webové aplikace je ale potřeba uchovávat například stav, jaký uživatel je k aplikaci přihlášen. Toto se realizuje skrze sessions, které umožňují webovému serveru uchovat si údaje o uživatelích, kteří k němu přistupují. Na straně klienta identifikace probíhá skrze HTTP cookies, což je malé množství dat, které může webový server skrz HTTP protokol poslat prohlížeči a prohlížeč tato data uchová. Do cookie se v případě session uloží session ID, které identifikuje vlastníka session.

Dalším problémem je odesílání požadavků z webového prohlížeče a přijímaní odpovědí od serveru. U webových stránek může postačovat klasický způsob interakce uživatele stránek se serverem skrze hypertextové odkazy a odesílání webového formuláře. Pro webové aplikace je toto řešení nedostačující, a to minimálně ze dvou hledisek:

- Po kliknutí na odkaz či po odeslání formuláře se musí znova načíst celá stránka. To jednak způsobí viditelnou prodlevu způsobenou stahováním a překreslováním celé stránky a také to zvyšuje síťový provoz, což je například u mobilního internetu problém.

- Klient se takřka nemůže opakovaně dotazovat serveru na nové stavy. Existují řešení jako meta tag pro automatické znovu-načtení stránky po určitém časovém intervalu nebo odkaz, na který uživatel musí manuálně klikat, a který způsobí aktualizaci stránky (případně stejný účel plní klávesa F5). Ovšem všechny tyto řešení jsou vesměs nepraktické.

Řešení tohoto problému přináší JavaScript (4.2.2) skrze API XMLHttpRequest, které umožňuje na pozadí odeslat HTTP(S) požadavek a přijmout a zpracovat odpověď. Tato technologie se nazývá AJAX (4.2.3).

Při výběru technologií na klientské straně jsme omezeni tím, co dnešní prohlížeče dokážou. V současné době se objevují nové zajímavé technologie pro tvorbu klient – server aplikací, jako například Web Sockets poskytující implementaci schránek pro přímou komunikaci mezi webovým serverem a klientem. Při jejich použití je třeba zvážit otázku zpětné kompatibility. Na druhou stranu nejsme omezení výběrem technologií na straně serveru. Zde je škála velmi pestrá a je pouze na nás, jaké komponenty budeme při tvorbě aplikace využívat.

Mezi stěžejní serverové části webové aplikace patří webový server a aplikační server či skriptovací jazyk, ve kterém je aplikace vytvořena. Dále úložiště dat, což je obvyklé nějaká forma databáze.

## 4.2 Technologie

### 4.2.1 Webový prohlížeč

Webový prohlížeč je program, který slouží k prohlížení World Wide Webu (WWW či web), což jsou aplikace internetového protokolu HTTP(S) propojené skrze hypertextové odkazy. Prohlížeč musí implementovat protokol HTTP(S) a technologie, ze kterých je vytvořená HTML stránka, jako (X)HTML, CSS, JavaScript a multimédia (obrázky, . . . ). Pro webovou aplikací plní prohlížeč roli tenkého klienta.

Historie webových prohlížečů je velmi pestrá a plná různých zvratů. Dokonce se hovoří o válkách webových prohlížečů [16, s. 55]. V současné době se ale situace ustálila na tři hlavní rodiny prohlížečů:

- Různé verze prohlížeče Internet Explorer společnosti Microsoft

- Rodina prohlížečů založena na renderovacím jádře Gecko, jehož vlajkovou lodí je prohlížeč Firefox společnosti Mozilla

- Rodina prohlížečů založena na renderovacím jádře WebKit, jehož vlajkovou lodí je v současnosti prohlížeč Google Chrome společnosti Google

Webové technologie v minulosti dosáhly určité míry stagnace, kdy vývoj nových technologií nepostupoval příliš kupředu. Změna přišla v roce 2004 spolu s komunitou WHATWG, která položila počátky standardu HTML5, jakožto nástupce HTML 4.01 [17, s. 25]. Od té doby se prohlížeče předhánějí v implementaci nové funkcionality z HTML5 (tag video,

tag audio, ...), nové funkcionality v JavaScriptu (WebSockets, Geolocation, ...) či nové funkcionality stále nedokončené specifikace CSS3. Hodně pokroku pomohla i společnost Microsoft, která začala inovovat prohlížeč Internet Explorer, který dlouho zůstával v, časem silně zastarávající, verzi 6.

Úskalím nových webovým technologií je ovšem to, že je zvládají pouze nové prohlížeče. Určité procento uživatelů bohužel zůstává u starých verzí a je na tvůrci webové aplikace, jakou míru kompatibility zvolí. Obvykle se ohlíží hlavně na verzi prohlížeče Internetu Exploreru. Je to z toho důvodu, že konkurenční prohlížeče na bázi WebKitu a Gecka jsou i ve starých verzích podstatně vyspělejší než staré verze Interetu Exploreru a dále i díky automatickým aktualizacím je mají uživatele často aktuální. Aktualizace Internetu Exploreru je oproti tomu často spojena s aktualizací operačního systému, protože například verze Internetu Exploreru 9 nefunguje na dosud rozšířeném operačním systému Windows XP a proto jeho uživatelé musí zůstat u verze 8 nebo přejít na konkurenční prohlížeč.

Dle statistik nadace Wikimedia, která spravuje mimo jiné Wikipedii, ze začátku tohoto roku, Internet Explorer 7 používá 0.9 % uživatelů, zatímco Internet Explorer 8 již 3.3 %. Celkově Internet Explorer (všechny verze) 9.3 % uživatelů [15]. Dle mého názoru je vhodné, pokud je to možné, Internet Explorer 8 ještě podporovat, starší verze je už možné zanedbat.

### 4.2.2 HTML + CSS + JavaScript

Kombinace technologií HTML + CSS + JavaScript + multimédia (obrázky, zvuky, ...) vytváří webovou stránku, což je dokument, který je možné zobrazit pomocí webového prohlížeče. Z webových stránek je utvořena klientská část webové aplikace.

#### HTML

HTML (HyperText Markup Language) [18] je značkovací jazyk pro popis obsahu webové stránky. HTML dokument je tvořen HTML elementy a jejich vlastnostmi (atributy). Elementy mohou být nepárové či párové. Párové se skládají z otevíracího a uzavíracího tagu (značky) a mezi nimi mohou být další elementy či text. Zanořené elementy do sebe tvoří stromovou hierarchii dokumentu. Elementy mohou informaci, kterou obalují, měnit obsahový (sémantický) význam či její formátování. Současný trend je ale takový, že by v HTML měl být definován převážně obsah a formátování by mělo být přesunuto do stylopisu definovaného jazykem CSS [20, s. 17]. Aktuální verze HTML je HTML5.

Existuje ještě možnost psát stránky v XHTML, což je jazyk pro tvorbu webových stránek postavený nad XML (HTML bylo postaveno nad SGML až do verze HTML 4.01), který měl ambici HTML nahradit, ovšem souboj s HTML5 prohrál. Je ale možné tvořit stránky v HTML5 se XHTML syntaxí [19].

#### CSS

CSS (Cascading Style Sheets) – Kaskádové styly [16, s. 127] je jazyk pro popis způsobu zobrazení webových stránek. Jeho hlavní snahou je oddělit vzhled dokumentu od obsahu. Obsah je definován jazykem HTML. Definice kaskádových stylů se skládá z pravidel. Každé pravidlo obsahuje selektor a blok deklarací. Selektor identifikuje objekt (či objekty), na který bude pravidlo aplikováno. Selektorem může být například množina HTML elementů, třídy (atribut `class`) či identifikátor (atribut `id`). Deklarace obsahuje vlastnosti, které pravidlo nastaví na určité hodnoty. Vlastnostmi může být formátování textu (barva, velikost, font, ...), vlastnosti blokových elementů (velikost, okraj, pozice, ...), atd.

#### JavaScript

JavaScript [20] je skriptovací, prototypově založený jazyk s dynamickým typováním. Syntaxí je inspirovaný jazykem C. Nemá nic společného s programovacím jazykem Java vyjma názvu. Ve webových stránkách se používá pro skriptování na straně klienta (ve webovém prohlížeči). Důležité je, že JavaScript má přístup k objektovému modelu dokumentu DOM (Document Object Model), což je API umožňující přístup či modifikaci obsahu, struktury nebo stylu webové stránky či její části. I když prohlížeče umožňují vypnutí JavaScriptu například z bezpečnostních důvodů, realita je taková, že stále více webových stránek bez JavaScriptu nefunguje ani v základní funkcionalitě, která je bez JavaScriptu realizovatelná. To ale neplatí pro moderní webové aplikace, které se bez klientského skriptovacího jazyka neobejdou.

### 4.2.3 AJAX

Jak jsem již lehce zmínil v části Klient – Server (4.1), AJAX (Asynchronous JavaScript and XML) [21] je technologií pro asynchronní (na pozadí) komunikaci mezi klientskou a serverovou částí webové aplikace. AJAX není technologií sám o sobě, ale jde o jiné využití existujících technologií jako HTML, CSS, JavaScript, DOM, XMLHttpRequest a případně dalších [21, s. 21].

Hlavní myšlenkou je, že klientská část aplikace může na pozadí požádat server o data, a to na základě události. Událostí může být interakce uživatele (například kliknutí na nějaký objekt ve stránce) nebo třeba vypršení časovače. Konkrétně JavaScriptový kód webové stránky skrze API XMLHttpRequest pošle HTTP(S) požadavek na webový server a zpracuje odpověď.

Při dotazování je možné využít HTTP(S) metod jako GET a POST a dotaz parametrizovat. Parametry dotazu můžeme předat pomocí URL adresy a parametrů. U metody GET jsou parametry součástí URL, při použití metody POST mohou být součástí HTTP dotazu, tudíž je POST vhodnější pro obsáhlejší data. Jednou z možností, jak posílat parametry požadavku, či požadavky celé je serializace (zabalení) do některého vhodného formátu (XML, JSON, YAML, ...) a předání skrze POST. Tato varianta byla využita v práci.

Nejjednodušší (na implementaci) odpovědí serveru může být HTML kód, který JavaScriptem nahradí kód určitého elementu. Výhodou je, že na klientské straně nemusí být žádná pokročilá logika, co by musela zpracovávat odpověď serveru, nevýhodou je, že se zbytečně přenáší (a na serveru sestavuje) HTML kód, který by se mohl vytvořit u klienta a dále to, že tímto způsobem nejde utvořit pokročilejší funkcionalita. Praktičtější je posílat strukturovaná data opět serializovaná ve vhodném formátu a nechat je zpracovat JavaScriptem.

Jako vhodný formát by se už z názvu technologie AJAX nabízelo XML (Extensible Markup Language). Není to ale jediná možnost. Velmi rozšířený je také JSON (JavaScript Object Notation) [22], jehož používaní je v JavaScriptu z podstaty přirozenější. Takováto kombinace JSONu a AJAXu se někdy přezdívá AJAJ a je využita v této práci.

### 4.2.4 Webový server

Pro serverovou část webové aplikace potřebujeme prostředí, ve kterém ji realizujeme. Možností je opravdu mnoho, téměř jakýkoliv programovací jazyk se dá pro tento účel využít. Z důvodu svých dřívějších zkušeností jsem se rozhodl pro osvědčenou variantu v podobě skriptovacího jazyka PHP v kombinaci s Nette Frameworkem a jako úložiště databázi MySQL.

Webová aplikace používající standardní technologie, což PHP a MySQL splňuje, by mohla být provozována (hostována) na některé hostingové službě. Při takové volbě není třeba příliš řešit výběr operačního systému a webového serveru a je to možné nechat na provozovateli hostingu. Pro účely práce ale budeme potřebovat komunikovat s IM službami, což už do standardní webové aplikace nespadá. Proto bylo zvoleno řešení v podobě zakoupení VPS (Virtual Private Server) a instalace webového serveru Apache do operačního systému Debian GNU/Linux, opět z důvodu osvědčenosti této kombinace.

#### PHP

PHP (PHP: Hypertext Preprocessor) je skriptovací jazyk vyvinutý pro účely vývoje webu. I když bývá někdy kritizován za špatný návrh [23], troufám si tvrdit, že to platilo hlavně pro historické verze jazyka. Ve verzi 5 podstatně vylepšuje možnosti objektového programování (podpora abstraktních tříd, rozhraní, ...) a od verze 5.3 přidává podporu jmenných prostorů, anonymních funkcí a uzávěrů (closures) [24, s. 30]. Tím se ale vývoj nezastavil a do jazyka v nových verzích přibývají další novinky. Na druhou stranu se snaží udržet zpětnou kompatibilitu. To, že může být vhodnou volbou i pro větší aplikace, dokazuje např. sociální síť Facebook, která ho pro svůj web používá [25].

U PHP je nutné zmínit, že při při zpracování požadavku neudržuje kontext aplikace (neposkytuje aplikační server), ale vytváří jej vždy znova. To je při při vývoji aplikace komunikující s IM službami problém a bude ho nutné vyřešit.

#### Nette Framework

Nette Framework [26] je český framework pro tvorbu webových aplikací v PHP. Výhodou použití frameworku je ušetření si práce při využívání funkcionality, kterou obsahuje, a která by musela být jinak znova naprogramována. Framework staví na MVC (Model View Controler) [21, s. 574] architektuře. Jeho silnou stránkou jsou mimo jiné šablony Latte, třídy pro vytváření a zpracování formulářů a ladící nástroj Laděnka.

#### MySQL

MySQL je multiplatformní relační databázový systém komunikující pomocí jazyka SQL. Podporuje několik úložišť, nejznámější jsou MyISAM a InnoDB. MyISAM je oproti InnoDB výkonnější, ale nepodporuje transakce a cizí klíče, InnoDB ano [24, s. 473]. V práci je využito databázové úložiště InnoDB.

# Kapitola 5

# Specifikace aplikace

Cílem práce je vytvořit serverovou část webové aplikaci pro komunikaci s IM službami. Popíšeme si, jak by měla vypadat celá aplikace, včetně její klientské části.

Aplikace musí poskytnout uživateli běžný komfort IM, na který je zvyklý. Vyjmenujme si stěžejní komponenty a vlastnosti takové IM aplikace:

- Správa IM účtů s možností přidat, upravit, či odebrat účet
- Seznam kontaktů s možností přidat či odebrat kontakt. U kontaktů je také nutné vyřešit autorizaci cizího kontaktu.
- Změna stavu uživatele, možnost měnit režimy jako online, offline, nedostupný
- Okna s konverzacemi, ve kterých uživatel přijímá a odesílá zprávy
- Možnost zobrazit si historii předešlých konverzací
- Aplikace by měla při výpadku internetu na tento stav vhodně zareagovat a po obnovení připojení se obnovit

Jelikož práce tvoří serverovou část takovéto aplikace, musí vyřešit tyto problémy:

- Způsob komunikace s IM službami
- API, prostřednictvím kterého bude klientská část komunikovat se serverovou částí aplikace

V práci se omezíme na IM komunikaci pouze mezi dvěma uživateli, nebudeme tedy implementovat skupinový chat.

# Kapitola 6

# Návrh a implementace aplikace

## 6.1 Komunikace s IM službami

U IM aplikace je pochopitelně vyřešení komunikace s cizími IM službami klíčová záležitost, která ovlivňuje skutečnou užitečnost takovéto aplikace. Pokud bude aplikace umět služeb málo, nebude ji mít příliš velký smysl používat. Při řešení tohoto problému máme v zásadě dvě možnosti:

- Napsat si řešení sami
- Využít externí knihovnu

Napsání vlastního řešení jsem zavrhl proto, že není v mých silách udržovat implementaci nepříliš dobře zdokumentovaných proprietárních služeb, jejichž protokol se navíc čas od času změní, jako je např. ICQ (3.1.3).

Při využití externí knihovny jsem zase závislý na jejich vývojářích, pokud její vývoj začne upadat, nebo bude ukončen, při změnách protokolů aplikace přestane fungovat a já budu muset znova vynaložit čas pro přepsání aplikace na jinou knihovnu.

Velmi pečlivě jsem vážil výběr vhodné knihovny a rozhodl jsem se pro knihovnu libpurple od vývojářů opensource desktopové multiplatformní IM aplikace Pidgin [1]. Její dlouhodobý vývoj a dobrá podpora různých protokolů je pro mě dostatečnou zárukou, že vývoj bude pokračovat, navíc tuto knihovnu zvolilo i předchozí řešení Meebo (2.1), kterým se tento projekt inspiruje.

Knihovna libpurple je napsána v jazyce C a je primárně určena pro vývoj desktopových klientů. Práce se ale zabývá webovým klientem, u kterého jsem se rozhodl pro vývoj v jazyce PHP na serverové straně. Proto bylo nutné vyřešit, jak tyto části propojit. Možnosti byly následující:

- Najít existují binding (propojení) knihovny libpurple s jazykem PHP
- Vytvořit binding knihovny libpurple pro jazyk PHP
- Vzdát se jazyka PHP a celou serverovou část aplikace napsat v jazyce C
- Napsat část pro komunikaci s IM službami jako externí část v jazyce C a propojit ji s PHP částí některou technologií meziprocesové komunikace – IPC (Inter-Process Communication)

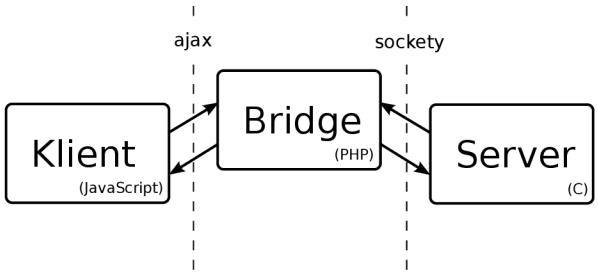
Binding libpurple pro PHP existuje, ale jeho možnosti zdaleka nedostačují požadavkům aplikace. Celkově je myšlenka použití bindingu spíše nevhodná. Jak je uvedeno v sekci webový server (4.2.4), PHP vždy při novém požadavku vytváří znova kontext aplikace. To v případě IM služeb znamená připojit se na IM službu, přijmout či odeslat zprávy a odpojit se. Nejenom, že by takovéto řešení trpělo vážnými výkonnostními problémy, ale způsobilo by i neustálou změnu stavu uživatele z režimu online na offline, což by bylo pro druhou stranu jednak nepříjemné a dále by to mohlo vést k zablokování účtu provozovatelem IM služby. Takovéto řešení je tedy vhodné pouze pro občasné poslání IM zprávy z aplikace např. kvůli monitoringu.

Vzdát se jazyka PHP a napsat celou serverovou část aplikace v C možné je a uvažoval jsem o tomto řešení. Nevýhodou je, že práce s technologiemi jako JSON, MySQL či tvorba HTML šablon je v jazyce C mnohonásobně pracnější, než v tomu uzpůsobeném jazyce PHP. Alternativou by mohlo být využití jazyka, který poskytuje aplikační server, co uchovává kontext a má vhodný binding pro knihovnu libpurple, ale nakonec zvítězilo poslední řešení:

Rozhodl jsem se napsat samostatnou část v jazyce C, využívající knihovnu libpurple, která prostřednictvím BSD schránek (sockety) bude komunikovat s PHP částí a uchovávat kontext IM služeb. Výhody takového řešení jsou rozepsány v sekci Server (6.3).

## 6.2 Architektura aplikace

Aplikace se skládá ze tří částí. Část pro komunikaci s IM službami, napsaná v jazyce C a využívající knihovnu libpurple, bude dále v textu označována jako Server (6.3), klientská část aplikace ve webovém prohlížeči jako Klient (6.5) a PHP část aplikace komunikující s databází MySQL jako Bridge (6.4), jelikož převážně přeposílá dotazy a odpovědi mezi Serverem a Klientem. Znázorněno je to na schématu 6.1.




Obrázek 6.1: Schéma architektury

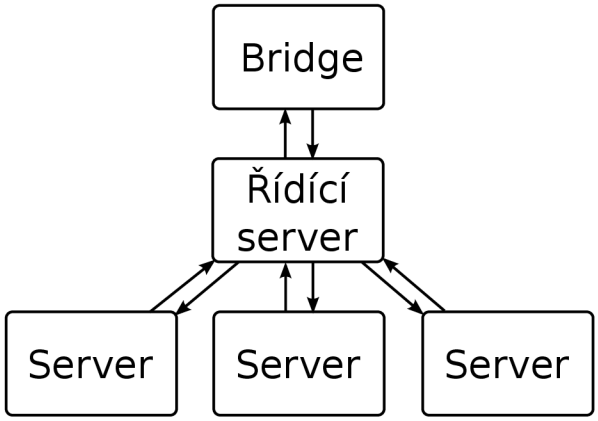
## 6.3 Server

Pro server je využit jazyk C kvůli knihovně libpurple, důvody pro takovéto řešení jsou uvedeny v sekci Komunikace s IM službami (6.1). Proto bylo potřeba nalézt vhodný způsob komunikace mezi Bridge (6.4) a Serverem.

### 6.3.1 Komunikace s Bridge

Zvolil jsem BSD schránky a protokol TCP z důvodu jednoduché možnosti provozu Serveru na jiném počítači, než bude provozován Bridge. Je rovněž možné zapojit více Serverů. I když

současná implementace počítá pouze s jedním Serverem, pokud bude projekt úspěšný, proběhne přechod na architekturu více Serverů. Architektura by fungovala tak, že by se Servery nekomunikovala přímo Bridge, ale přibyl by mezičlánek v podobě Řídícího serveru, který by se pro Bridge choval jako Server, ale ve skutečnosti by přeposílal požadavky jednotlivým Serverům a sám by měl za úkol pouze Servery vytvářet a ukončovat dle aktuální zátěže. Architektura je znázorněnou na schématu 6.2.




Obrázek 6.2: Schéma architektury s více Servery

Takováto architektura přinese několik výhod:

- Škálování výkonu – Servery mohou běžet na různých počítačích
- Větší stabilita – Pokud dojde k výpadku jednoho Serveru, restart postihne jen část uživatelů
- Distribuce IP adres – některé služby mohou pojmout podezření, pokud je z jedné IP adresy připojeno příliš mnoho klientů, a IP adresu zablokovat. S konceptem výše zmíněné architektury půjde zátěž distribuovat na více IP adres.

Přes BSD schránky je vhodné přenášet co nejpodobnější formát dat, jakým bude komunikovat Klient s Bridge, aby nebylo nutné udržovat více API. Více je formát komunikace rozepsán v sekci API (6.6), podstatné je, že data jsou serializovaná do formátu JSON. Naštěstí pro C existuje knihovna Jansson [28], díky které se dá v jazyce C s formátem JSON pohodlně pracovat.

Jelikož webový server, na kterém běží Bridge pracuje konkurentně, může přicházet na Server více požadavků od Bridge v jeden okamžik. Proto i Server musí pracovat konkuretně, aby je zvládl obsloužit.

Naivní životní cyklus serveru by mohl vypadat takto:

- Hlavní proces Serveru je spuštěn
- Hlavní proces poslouchá na portu, při připojení klienta vytvoří nové vlákno:
  - Vlákno příjme kolekci požadavků od Bridge
  - Vlákno zpracuje požadavky, pro IM požadavky zavolá patřičné funkce knihovny libpurple a odešle kolekci odpovědí
  - Vlákno se ukončí

### 6.3.2 Integrace libpurple

Bohužel i zde nastaly komplikace. Většina IM požadavků je asynchronních. Například připojení na server může trvat až několik desítek sekund. Stejně tak odeslání zprávy. Navíc se ne vždy akce musí podařit. Běžný webový požadavek trvá desítky milisekund, není proto možné, aby Bridge čekal desítky sekund na odpověď Serveru a Klient desítky sekund na odpověď Bridge. Je proto nutné u asynchronních požadavků požadavek přijmout, vrátit informaci, že je požadavek zpracováván a ve chvíli, kdy se ho podaří dokončit, informovat o výsledku.

Dalším problémem je, že knihovna libpurple není uzpůsobena pro práci ve vícevláknové aplikaci (není thread safe) a přebírá řízení programu skrz hlavní událostní smyčku. Autor aplikace pak do chodu programu může zasahovat skrze zpětná volání (callbacky) na signály vyvolané událostmi a navíc může do hlavní smyčky pověsit vlastní vyvolávače událostí, převážně pro GUI, které mohou vyvolat událost například ve chvíli, kdy uživatel klikl na určité tlačítko. Daný událostní systém je vymyšlen převážně pro tvorbu desktopových IM aplikací, což není případ práce.

Problém byl vyřešen tak, že aplikace byla rozdělena na dvě části, kde každá běží v samostatném vlákně. Hlavní vlákno obsluhuje knihovnu libpurple s její hlavní řídící smyčkou, do které byl přidán časovač, který jednou za krátký časový okamžik vyvolá událost, jehož obsluha zamkne zámek (mutex), ze společných struktur přečte požadavky, které předá knihovně libpurple. Ve chvíli, kdy libpurple požadavek vykoná, vyvolá událost, jejíž obsluha opět zamkne mutex a do společných struktur uloží odpověď. Při takovém použití není nutné, aby byla knihovna libpurple thread safe, protože s jejími funkcemi pracuje pouze vlákno, ve kterém knihovna běží. Toto vlákno budeme dále nazývat **IM část**.

Vedlejší vlákno poslouchá na portu, vytváří další vlákna pro každého připojeného klienta – Bridge (konkurentnost), čeká na požadavky, validuje je a vrací na požadavky odpovědi. Pokud požadavek potřebuje IM část, je požadavek uložen do společných struktur a jako odpověď je vráceno, že byl požadavek zařazen do fronty (`accepted`). Pro vrácení stavu již zpracovaných požadavků musí Bridge odeslat speciální požadavek `get_responses`, na základě čehož Server vrátí odpovědi zpracovaných požadavků a vyprázdní frontu odpovědí ve společných strukturách. Toto vlákno bude dál nazýváno **Síťovou části**.

Dále je nutné vyřešit možnost, že se Klient odmlčí (například mu spadne internetové připojení). Bridge kontext neuchovává, takže časovače (timeout) pro odhlášení klienta a uklizení struktur po určité době nečinnosti rovněž řeší Server.

Je nutné zdůraznit, že Server nemá žádné permanentní úložiště a data ve strukturách (které jsou v operační paměti) včetně uživatelů a jejich IM účtu dynamicky vznikají a zanikají na základě požadavků Bridge, který jediný má přístup k databázi.

### 6.3.3 Datové struktury

Pro lepší pochopení Serveru si objasníme klíčové datové struktury, které jsou definovány v souboru `myim_server/src/myim.h`.

### MYIM

`MYIM`[1] je společná struktura pro sdílení dat mezi IM a Síťovou částí (6.3.2). Obsahuje:

- `requests`: seznam požadavků (instance `MYIM_Request`) čekajících na zpracování IM částí
- `requests_in_process`: seznam požadavků (instance `MYIM_Request`) zpracovávaných libpurple – čeká se na reakci libpurple, tedy spuštění patřičného callbacku
- `users`: hash tabulka uživatelů (instance `MYIM_User`), vyhledávání podle ID uživatele
- `users_by_purple`: hash tabulka uživatelů (instance `MYIM_User`), vyhledávání podle ukazatele na libpurple IM účet (nutné kvůli komunikaci s libpurple)
- `lock`: zámek (mutex) instance struktury

### MYIM_User

MYIM_User (uživatel) reprezentuje uživatele tohoto projektu. Obsahuje:

- `user_id`: ID identifikuje uživatele, je součástí každého požadavku na Server
- `last_action_at`: čas poslední akce umožňuje Serveru odhlašovat a rušit neaktivní uživatele
- `accounts`: seznam IM účtů (instance `MYIM_Account`) uživatele
- `responses`: Fronta zatím neodeslaných odpovědí (instance `MYIM_Response`) na požadavky, které přišly s ID daného uživatele

### MYIM_Request

`MYIM_Request` reprezentuje požadavek. Obsahuje:

- `action`: název akce (typ požadavku)
- `request_id`: ID identifikující požadavek
- `user_id`: ID identifikující uživatele (`MYIM_User`)
- `purple_account`: libpurple IM účet nalezený skrze `MYIM_User:accounts` z předaného `account_id` v požadavku
- `data`: další parametry požadavku
- `started_at`: časová značka, kdy se požadavek začal zpracovávat (kvůli odhalení ztraceného požadavku – timeout)

[1] myim je pracovní název projektu, pravděpodobně se stane i názvem oficiálním, až bude projekt spuštěn.

#### MYIM_Response

`MYIM_Response` reprezentuje odpověď na požadavek nebo událost (třeba příchod zprávy). Obsahuje:

- `type`: jestli je odpověď událost, typ reprezentuje druh události, pokud je odpověď odpovědí na požadavek, je typ `NULL`
- `user_id`: ID identifikující uživatele (`MYIM_User`)
- `request_id`: ID identifikující požadavek, pokud odpověď není událostí (`MYIM_Request`)
- `state`: pokud je odpověď odpovědí na požadavek, stav reprezentuje výsledek požadavku (uspěl – `done`, neuspěl – `failed`)
- `data`: další data odpovědi (například text přijaté zprávy a odesílatel u události přijaté zprávy)

#### MYIM_Account

`MYIM_Account` reprezentuje jeden IM účet uživatele. Obsahuje:

- `account_id`: ID IM účtu
- `protocol`: Protokol IM účtu, používá se pro detekci, jestli se klient nesnaží přihlásit více stejných účtů zároveň
- `name`: Jméno IM účtu, uchovává se pro detekci, jestli se klient nesnaží přihlásit více stejných účtů zároveň
- `purple_account`: libpurple reprezentace IM účtu
- `authorizations`: hash tabulka autorizací, tedy požadavků na přidání cizího kontaktů, čekajících na odpověď uživatele (přijetí/odmítnutí)

### 6.3.4 Životní cyklus

Skutečný životní cyklus Serveru nakonec vypadá takto:

- Hlavní proces je spuštěn
- Hlavní proces inicializuje knihovnu libpurple, vytvoří časovač generující událost, jejíž obsluhu nazveme Kontrolér
- Hlavní proces vytvoří síťové vlákno, tedy výše uvedenou Síťovou část
- Hlavní proces spustí hlavní smyčku knihovny libpurple, nyní plní roli výše zmíněné IM části (6.3.2)
- Síťová část poslouchá na portu, při připojení klienta vytvoří nové vlákno:
  - Vlákno příjme kolekci požadavků od Bridge
  - Vlákno zpracuje požadavky, pro IM požadavky uzamkne mutex a uloží je do společných struktur

  - Pokud klient chce odpovědi (požadavek `get_responses`), uzamkne mutex a vrátí odpovědi ze společných struktur
  - Vlákno se ukončí po odpojení klienta
- Pokud v IM části nastane událost časovače, spustí se Kontrolér:
  - Kontrolér uzamkne mutex
  - Vybere požadavky ze společných struktur a spustí patřičné funkce libpurple
  - Kontroler odemkne mutex
- Pokud IM části nastane nějaká IM událost, spustí se patřičný callback:
  - Callback uzamkne mutex
  - Na základě předaných parametrů (např. text příchozí zprávy) vytvoří odpověď a uloží ji do společných struktur
  - Callback odemkne mutex

## 6.4 Bridge

Bridge je část aplikace běžící na webovém serveru. Z pohledu webu se jedná o klasickou serverovou část webové aplikace, ale jelikož je primárním účelem aplikace komunikovat s IM službami, kde většinu práce plní Server (6.3) a Bridge je prostředníkem, zvolil jsem takovéto pojmenování.

Bridge je napsaný v programovacím jazyce PHP (verze 5.4), jako úložiště využívá relační databázi MySQL (verze 5.5). Pro ulehčení práce bylo využito Nette Frameworku (verze 2.1), knihovny dibi [27] pro jednodušší práci se SQL dotazy a ORM (Objektově relační mapování) vrstvy Golem[2]. Tyto technologie jsou rozepsané v sekci Webový server (4.2.4).

### 6.4.1 Úkoly

Popišme si úkoly, které musí Bridge zajišťovat:

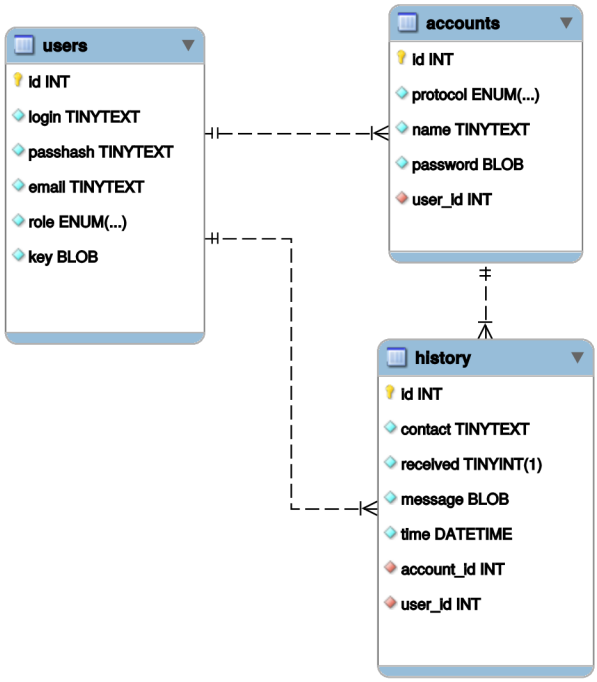
- Registrace, přihlášení a změna údajů uživatele
- Po přihlášení odeslání klientské části aplikace do webového prohlížeče
- Přijímá požadavky od Klienta (6.5) a validuje je
- U požadavků týkajících se údajů v databázi rovnou odpovídá
- Odesílá požadavky na Server a vrací odpovědi
- Některé požadavky modifikuje/rozšiřuje, například požadavek `login`, viz dále

Bridge je jediná část aplikace, která má přístup k databázi a pro ostatní části ji zpřístupňuje, proto se na ní nyní podíváme.

[2] Golem je knihovna autora práce, zatím není dokončena a ani nikde publikována.

### 6.4.2 Model databáze

Bridge v databázi potřebuje uchovávat uživatelské účty (dále uživatele), IM účty uživatelů a historii. Uživatele reprezentuje tabulka `users`, IM účty `accounts` a historii `history`. Integritní omezení hlídá Bridge na modelové vrstvě architektury MVC (Model View Controler). Databázové tabulky včetně vztahů jsou zakreslené v diagramu 6.3.




Obrázek 6.3: Diagram modelu databáze

Popíšeme si jednotlivé tabulky podrobněji:

#### Tabulka users

Zastupuje uživatele, obsahuje tyto atributy:

- `login`: přihlašovací jméno
- `passhash`: otisk hesla vytvořený jednosměrnou hashovací funkcí
- `email`: emailová adresa
- `role`: role uživatele, možnosti jsou `registered` a `admin`, normální uživatel je `registered`
- `key`: zašifrovaný klíč určený k šifrování hesel IM účtů a historie, viz 6.4.4

#### Tabulka accounts

Zastupuje IM účty, obsahuje tyto atributy:

- `protocol`: protokol/služba, možnosti jsou `jabber` a `icq`, ovšem díky XMPP rozhraní (`jabber`) pokrývá i službu Google Talk a Facebook
- `name`: jméno přihlašovacího jména k účtu
- `password`: zašifrované heslo, viz 6.4.4

#### Tabulka history

Zastupuje historii, obsahuje tyto atributy:

- `contact`: jméno kontaktu, od kterého přišla, nebo kterému se odeslala zpráva
- `received`: boolean hodnota, zda byla zpráva obdržena (`true`), či odeslána (`false`)
- `message`: zašifrovaný text zprávy, viz 6.4.4
- `time`: datum a čas odeslání/přijmutí zprávy

### 6.4.3 Implementace

Implementace Bridge není nikterak rozsáhlá, Bridge je skutečně z velké části pouze prostředníkem mezi Serverem (6.3) a Klientem (6.5). Navíc velkou část práce odvedl samotný Nette Framework.

Uživatel, který poprvé zavítá na naši aplikaci, se musí zaregistrovat a poté přihlásit. Registrace je realizována klasickým webovým formulářem, po jehož odeslání se v databázi vytvoří záznam v tabulce `users`. Po té je uživatel přesměrován na stránku s přihlašovacím formulářem.

Po zadání správných přihlašovacích údajů Bridge vytvoří relaci, která je uložena v session, a přesměruje na stránku obsahující Klienta, který se načte uživateli do webového prohlížeče.

Teprve nyní se dostávají na řadu AJAX (4.2.3) požadavky, případně samotný Server. Požadavky posílá klient na URL `/api/request` metodou POST ve formátu JSON. Kvůli snížení zátěže na server je jich možné poslat více zároveň. Bridge je projde, zvaliduje a zpracuje. U každého požadavku Bridge ověří jeho akci, podle ní pozná, zda je požadavek třeba odeslat na Server, nebo ho má Bridge vyřídit přímo.

Mezi požadavky, jejichž zpracování je v režii Bridge, patří požadavky pracující s databází. Konkrétně je to získání, přidání, úprava či odebrání IM účtů (pracuje se s tabulkou `accounts`) a zaznamenání, odstranění či vrácení zpráv z historie (pracuje se s tabulkou `history`).

Většina požadavků se ale pomocí BSD socketů odesílá Serveru a Bridge pouze vrátí odpovědi. Speciální je požadavek `login`, který zajišťuje přihlášení k IM účtu. Klient nezná přihlašovací údaje a bylo by zbytečné, aby je Bridge klientovi posílal, jelikož je přenos hesel po síti kvůli bezpečností vhodné minimalizovat. Proto, pokud se chce klient přihlásit na IM účet, pošle ve svém požadavku `login` pouze ID IM účtu a přihlašovací údaje před odesláním na Server doplní Bridge z databáze. Více informací k požadavkům je uvedeno v sekci API (6.6).

Z popisu Bridge vyplývá, že hlavní logiku aplikace obstarává samotný Klient (6.5).

### 6.4.4 Bezpečnost

Webová aplikace pro IM pracuje s privátními daty uživatelů. Nejde pouze hesla, ale také o historii zpráv, která se uchovává v databázi. Určitě by uživatele aplikace nepotěšilo, kdyby se někomu podařilo tyto informace získat a zneužít. Podívejme se, jaké jsou možné nejběžnější útoky na aplikaci:

- Man in the middle (člověk uprostřed) útok: útočník se bude nacházet na síťové cestě mezi Klientem a Bridge a bude ji moci odposlouchávat. Může to být třeba provozovatel veřejné Wi-Fi sítě. Kromě odposlechu přihlašovacích údajů k aplikaci, případně IM službám, může odposlouchávat posílané zprávy. Obranou proti tomuto útoku je používat zabezpečenou komunikaci HTTPS šifrovanou pomocí SSL, což činím.
- Nabourání serveru a získání obsahu databáze: bohužel, i když je aplikace psaná s důrazem na bezpečnost, nikdy nelze zaručit, že nemůže obsahovat nějakou bezpečností díru, kterou potenciální útočník může zneužít. Pokud se díky chybě útočníkovi podaří získat obsah databáze a nebudou ošetřené privátní data, získá hesla a historii všech uživatelů aplikace. Aplikace se tomuto problému snaží předejít šifrováním, které je popsáno níže.

Zabezpečení hesla k aplikaci se provádí standardně tak, že se uchovává pouze jeho otisk vytvořený nějakou jednosměrnou hashovací funkcí. Při přihlášení se z hesla, které uživatel zadal do přihlašovacího formuláře, vytvoří otisk a porovná se s otiskem uloženým v databázi. Pokud se otisky shodují, je heslo zadáno správně a uživatel může být přihlášen. Pro zvýšení bezpečnosti se k heslu přidává sůl (salt), která má zabránit tomu, aby přece jen útočník z otisku hesla nemohl původní heslo určit například pomocí porovnání otisku hesla s množinou otisků vygenerovaných ke známým slovům. Poté předpokládáme, že množinu otisků ke známým slovům si útočník nevygeneruje, protože sůl nezná. Také je vhodné před vytvořením otisku k heslu přidat login uživatele, aby dva páry heslo – login se stejným heslem měly pokaždé jiný otisk. Aplikace tento postup pro hesla uživatelů využívá. Při studiu, jak vytvářet správně otisky hesel, jsem vycházel, mimo jiné, z toho zdroje [29].

Touto technikou ovšem není možné zabezpečit hesla k IM účtům. Aplikace se totiž potřebuje pomocí hesel autentizovat na IM servery, což bez možnosti rozšifrovat heslo nepůjde. Jednosměrná hashovací funkce tedy nepřichází v úvahu. Stejný problém se týká zpráv v historii. Nechceme, aby si je přečetl útočník, ale chceme je zobrazit uživateli aplikace.

Řešením je šifrování privátních dat pomocí klíče, který je v databázi uložen zašifrovaný heslem uživatele a rozšifrovaný se uchovává pouze po dobu uživatelské relace v session. K rozšifrování klíče dojde při přihlášení uživatele, což je jediný okamžik, kdy aplikace zná uživatelovo heslo. Proto později bez znalosti hesla uživatele (a bez prolomení šifrovací funkce) není možné získat privátní data uživatele. Inspirací pro zmíněný postup mi byl tento zdroj [30].

## 6.5 Klient

Z pohledu uživatele je Klient nejdůležitější částí aplikace. Dává aplikaci tvář, určuje její logiku. Úkol Klienta je přetavit funkcionalitu serverové části do líbivého a funkčního uživatelského rozhraní, jelikož funkcionalita by mohla být sebedokonalejší, ale s nevyhovujícím Klientem naprosto nepoužitelná a tím pádem zmařená. Klientská část sice není předmětem práce, ale je nutné vytvořit takové podmínky, aby vhodný Klient mohl vzniknout. Což se podařilo, jelikož Klient existuje (7).

### 6.5.1 Úkoly

Nyní se podíváme na úkoly, které musí Klient zajišťovat:

- Vytvořit přehledné uživatelské rozhraní (UI) umožňující přístup ke všem částem aplikace, jako je seznam kontaktů, okna s konverzacemi, nastavení IM účtů, historie, apod.
- Periodicky se dotazovat Bridge (6.4) na nové události, jako jsou přijaté zprávy, změny stavů kontaktů, ...
- Na základě interakcí uživatele s UI odesílat požadavky na Bridge a zpracovat odpovědi. Příkladem může být odeslání zprávy.
- Vhodně reagovat na chybové stavy, jako nedostupnost Bridge, výpadek internetového připojení, atd.

### 6.5.2 Rozhraní mezi klientskou a serverovou částí

Původní představa byla taková, že v rámci práce budu implementovat pouze Server (6.3) a Bridge a autorovi Klienta poskytnu popis API. Autor Klienta už samostatně implementuje pomocí JavaScriptu, CSS a HTML (4.2.2) celého Klienta včetně obsluhy AJAX (4.2.3) požadavků. Později se ukázalo, že tato představa není nejvhodnější. Jakožto autor serverové části znám nejlépe různé eventuality, které je nutné ošetřit na klientské straně. Pokud bych se je snažil vysvětlit autorovi Klienta, aby je sám implementoval, oba bychom zbytečně ztráceli čas. Já vysvětlováním, on jejich implementací, místo aby se soustředil na podstatu Klienta. Proto jsem naprogramoval JavaScriptovou knihovnu, která obstarává AJAX požadavky a tvoří rozhraní mezi serverovou a klientskou části. Tato knihovna je ještě předmětem této práce. Autor Klienta pak tuto knihovnu využívá. Pro pochopení problematiky se ale nyní musíme podívat na API (6.6). Samotná knihovna je popsána zde 6.6.6.

## 6.6 API

API (Application Programming Interface), česky programové rozhraní, se v aplikaci objevuje na více místech. Jednak je třeba komunikovat mezi Serverem a Bridge, dále mezi Bridge a Klientem (6.5) a v samotném Klientovi je API obaleno funkcemi JavaScriptové knihovny implementující rozhraní mezi klientskou a serverovou částí (6.6.6), které využívá autor Klienta. Podstata API ale zůstává napříč aplikací stejná.

### 6.6.1 Objekty API

V API se vyskytují tři základní objekty: požadavky, odpovědi a události. Podíváme se na ně podrobněji:

- **Požadavek (request):** reprezentuje jakýkoliv požadavek směrem od Klienta na server. Každý požadavek obsahuje typ požadavku – akci (`action`), číslo požadavku (`request_id`) a další parametry (`data`). Příkladem požadavku může být `send_message`, co odešle zprávu.

- **Odpověď (response):** reprezentuje odpověď na požadavek nebo událost, přicházející směrem od serveru ke Klientovi.
  - **odpoveď na požadavek:** v tomto případě odpověď obsahuje číslo požadavku (`requst_id`) a stav vykonání požadavku (`state`). Stav může nabývat hodnot, že se požadavek nepodařilo vykonat (`failed`), podařilo vykonat (`done`), nebo že byl zařazen do fronty k pozdějšímu zpracování (`accepted`). Dále odpověď může obsahovat navrácená data požadavku (`data`), například odpověď na požadavek pro získání kontaktů IM účtu `get_contacts`, obsahuje jako data kontakty.
  - **událost:** je speciální druh odpovědi, která neodpovídá na požadavek, ale oznamuje událost ze strany IM serveru, např. došlou zprávu. Místo čísla požadavku obsahuje typ události (`type`) a data události (`data`), např. text došlé zprávy. Pro získání události je potřeba odpovědi vyžádat skrze požadavek `get_responses`.

### 6.6.2 Požadavky

Požadavky můžeme rozdělit na rychlé a pomalé. Rychlé požadavky jsou ty, které může Bridge, případně Server, vyřešit okamžitě, tedy odpověď na ně dosahuje stavů úspěch/neúspěch. Všechny požadavky, které dokáže vyřešit přímo Bridge jsou rychlé. Naopak pomalé požadavky jsou ty, jenž se na Serveru zařadí do fronty a zpracují se později. Všechny požadavky, které obsluhují IM záležitosti, jsou pomalé. Navrácená odpověď má stav `accepted`, což znamená, že je požadavek přijatý a čeká na zpracování. Později na tyto požadavky dostaneme odpověď skrze požadavek `get_responses`, který vrací odpovědi na dokončené požadavky a události.

Nyní si popíšeme jednotlivé požadavky. Požadavky jsem shrnul do dvou tabulek. V tabulce 6.1 jsou zobrazeny požadavky, které se posílají od Klienta až na Server. Mezi Serverem a Bridge se ještě ke každému požadavku přidává `user_id` identifikující, jakého uživatele se požadavek týká. Mezi Klientem a Bridge se neposílá, jelikož se uživatel identifikuje pomocí session. Po spuštění Klient musí uživatele na serveru inicializovat, aby pro něj Server vytvořil patřičné datové struktury, což provede požadavkem `register`. Po ukončení Klienta (například odhlášením či zavřením okna prohlížeče) by měl Klient struktury uklidit požadavkem `unregister`, který rovněž odhlásí IM účty.

V tabulce 6.2 jsou zobrazeny požadavky, které zpracovává přímo Bridge. [ ] v obou tabulkách značí pole a sloupec výstupní data popisuje data odpovědi na požadavek v případě úspěchu požadavku.

Je nutné více vysvětlit požadavky pro práci s historií `history_find`, `history_add` a `history_remove`. Historie je navrhnuta tak, že při příchodu zprávy musí Klient, pokud chce uchovávat historii, zprávu uložit pomocí `history_add`. Nevýhodou je, že to mírně zvyšuje síťový provoz, výhodou je snazší implementace a možnost nechat Klienta řídit, zda se má uchovávání provádět, či ne. Pokud chce Klient historii zobrazit, pošle požadavek `history_find`, který kromě prostého vrácení záznamů, umí záznamy i spočítat, vrátit omezenou podmnožinu, nebo vyhledat záznamy dle řetězce. Díky tomu lze skrze požadavek `history_find` implementovat například stránkování nebo vyhledávání záznamů historie.

Odpovědi na požadavky v případě úspěchu máme shrnuté v tabulkách. V případě neúspěchu přijde odpověď se `state failed`. V parametru `data` se nachází `code` s kódem chyby a `message` s popisem chyby. Přehled kódů je ve zdrojových kódech Serveru.

| akce | popis | pomalý | vstupní data | výstupní data |
|---|---|---|---|---|
| `register` | inicializace uživatele na Serveru, viz dále | ne | | |
| `unregister` | deinicializace a úklid uživatele | ne | | |
| `login` | přihlášení účtů na Serveru | ano | `account_id:` ID účtu<br>`protocol:` protokol<br>`name:` jméno<br>`password:` heslo<br>U Klienta údaje doplní Bridge | |
| `logout` | odhlášení účtu | ne | `account_id:` ID účtu | |
| `get_responses` | vrací odpovědi a události | ne | | [ `request_id:` ID požadavku<br>`state:` stav požadavku\|null<br>`type:`typ události<br>`data:`data ] |
| `get_contacts` | vrací kontakty | ano | `account_id:` ID účtu nebo all pro všechny účty<br>`offline`: true\|false jestli vracet offline kontakty | [ `account_id:` ID účtu<br>`contacts:` kontakty<br>[ `name:` jméno<br>`alias:` alias<br>`state:` stav]] |
| `get_account_states` | vrací stavy účtů na serveru | ano | | [ `account_id:` ID účtu<br>`state:` stav] |
| `send_message` | odeslání zprávy | ano | `account_id:` ID účtu<br>`to:` kontakt<br>`message:` zpráva | `time:` čas odeslání |
| `account_change_state` | změna stavu účtu na serveru | ano | `account_id:` ID účtu<br>`state:` stav | |
| `account_add_contact` | přidání kontaktu k účtu | ano | `account_id:` ID účtu<br>`name:` jméno kontaktu | |
| `account_remove_contact` | odebrání kontaktu z účtu | ano | `account_id:` ID účtu<br>`name:` jméno kontaktu | |
| `authorization` | potvrzení autorizace | ano | `account_id:` ID účtu<br>`name:` jméno kontaktu<br>`accept:` true\|false potvrdit či nepotvrdit | |

Tabulka 6.1: Tabulka požadavků Klient – Bridge – Server

| akce | popis | vstupní data | výstupní data |
|---|---|---|---|
| `get_accounts` | vrátí účty | | [`account_id:` ID účtu<br>`protocol`: protokol<br>`name`: jméno] |
| `account_add` | přidá účet | `protocol`: protokol<br>`name`: jméno<br>`password`: heslo | `account_id:` ID účtu |
| `account_change` | upraví účet | `account_id:` ID účtu<br>`protocol`: protokol\|null<br>`name`: jméno\|null<br>`password`: heslo\|null | |
| `account_remove` | odstraní účet | `account_id:` ID účtu | |
| `history_find` | vrátí záznamy z historie | `account_id:` ID účtu<br>`contact:` jméno kontaktu<br>`count:` true\|false<br>vrátit počet záznamů<br>`fromTime:` datum od\|null<br>`toTime:` datum do\|null<br>`search:` hledat text\|null<br>`offset:`<br>přeskočit N zaznamů\|null<br>`limit:`<br>omezit na N zaznamů\|null<br>`order:` time asc\|time desc<br>směr řazení | [ `time:` čas odeslání<br>`message:` zpráva<br>`id:` id zprávy<br>`received:` true\|false<br>přijatá či odeslaná ] |
| `history_remove` | odebere záznamy z historie | `account_id:` ID účtu<br>`contact:` jméno kontaktu<br>`fromTime:` datum od\|null<br>`toTime:` datum do\|null<br>`search:` hledat text\|null<br>`offset:`<br>přeskočit N zaznamů\|null<br>`limit:`<br>omezit na N zaznamů\|null<br>`order:` time asc\|time desc<br>směr řazení | [ `id:` id ostraněné zprávy ] |
| `history_add` | přidá záznam do historie | `account_id:` ID účtu<br>`contact:` jméno kontaktu<br>`time:` datum<br>`message:` zpráva<br>`received:` true\|false<br>přijatá či odeslaná | `id:` id přidané zprávy |

Tabulka 6.2: Tabulka požadavků Klient – Bridge

### 6.6.3 Události

Události posílají IM servery. Událostí může být změna stavu kontaktu, příchod nové zprávy, požadavek na přijetí kontaktu – autorizace nebo odhlášení účtu z IM serveru z důvodu nějakého problému. Události jsou shrnuty v tabulce 6.3.

| typ | popis | data |
|---|---|---|
| `message` | přijatá zpráva | `time`: čas příchodu<br>`account_id`: ID účtu<br>`from`: jméno kontaktu<br>`message`: text |
| `contact_status_changed` | změna stavu kontaktu | `time`: čas změny<br>`account_id`: ID účtu<br>`name`: jméno kontaktu<br>`state`: stav kontaktu |
| `authorization` | přijatá autorizace | `account_id`: ID účtu<br>`name`: jméno kontaktu<br>`alias`: alias kontaktu<br>`message`: zpráva autorizace |
| `account_disconnect` | odpojení účtu z důvodu<br>chyby či výpadku na serveru | `account_id`: ID účtu |

Tabulka 6.3: Tabulka událostí

### 6.6.4 Formát

Vysvětlili jsme si požadavky a odpovědi. Nyní se podíváme, jak vypadá samotný formát dat. API mezi Serverem a Bridge, i mezi Bridge a Klientem využívá formát JSON. Vždy komunikace probíhá tak, že jedna strana (Buď Klient směrem na Bridge, či Bridge směrem na Server) odešle kolekci požadavků a zpátky obdrží kolekci odpovědí.

#### Požadavky

Ve formátu JSON vypadá dotaz s požadavky takto:

```
[
  {
    "request_id":id_požadavku,
    "action":typ_požadavku,
    "data":data_požadavku_či_null
  },
  {"request_id":...}, ...
]
```

#### Odpovědi

A odpovědi takto:

```
[
  {
    "request_id":id_požadavku,
    "state":stav_odpovědi_na_požadavek,
```

```
    "data":data_odpovědi_nebo_null
  },
  {"request_id":...}, ...
]
```

Co přesně znamenají jednotlivé atributy požadavků a odpovědí je vysvětleno zde 6.6.1.

### 6.6.5 Příklad komunikace

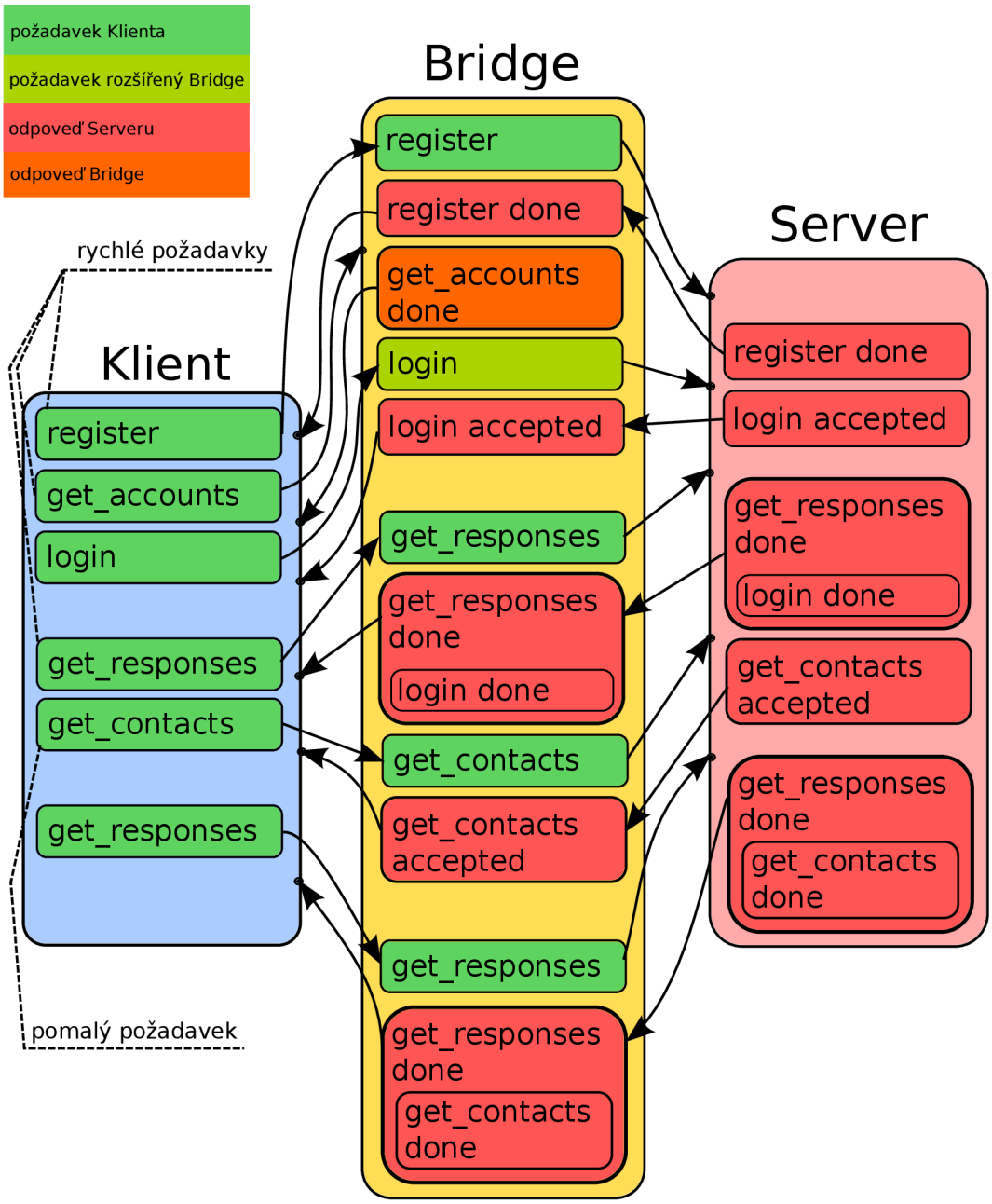
Pro lepší objasnění, jak API funguje a jak spolu jednotlivé části aplikace komunikují, budu API demonstrovat na příkladu. Jako příklad jsem zvolil zjednodušenou inicializaci Klienta. Při inicializaci se musí Klient registrovat na Serveru požadavkem `register`. Dále se potřebuje dotázat Bridge na IM účty požadavkem `get_accounts`. V příkladu budeme předpokládat, že uživatel má pouze jeden IM účet, na který se Klient pokusí přihlásit požadavkem `login`. Jelikož je požadavek `login` pomalým požadavkem, musí nyní Klient periodicky získávat odpovědi příkazem `get_responses`. Ve chvíli, kdy získá odpověď na požadavek `login` (budeme předpokládat, že úspěšnou), požádá Server o kontakty IM účtu požadavkem `get_contacts`. Opět se periodicky bude ptát na odpovědi příkazem `get_responses`, dokud kontakty neobdrží. Po té může Klient například vykreslit seznam kontaktů a inicializace je dokončena. Diagram komunikace v tomto příkladu je zakreslen na schématu 6.4.

U skutečného Klienta je inicializace složitější. Musí se např. ošetřit stav, kdy uživatel je už na serveru registrován a ke všem IM účtům přihlášen, ale Klient to neví, protože uživatel například znovu-načetl stránku. Podobných zádrhelů při tvorbě Klienta nastává spousta a jejich opomenutí a neošetření způsobí neočekávané chování. S některými zádrhely se počítalo už při implementaci, protože šly důkladnou analýzou odhalit, jiné se objevují až v reálném nasazení a daří se je opravit díky testovaní ve skutečném provozu.

### 6.6.6 Knihovna pro práci s API na klientské straně

Jak je zmíněno v sekci Klient 6.5.2, pouhé poskytnutí API pro vývoj Klienta nestačí. Součástí práce je i knihovna, která má práci s API zjednodušit. Podíváme se na to, co knihovna musí zajišťovat:

- Navázat registraci se serverem (příkaz `register`), pokud už registrace existuje (uživatel například znovu-načetl stránku), Klienta o tom informovat
- Poskytnout rozhraní pro práci s požadavky:
  - Možnost přidat požadavek do fronty
  - Jednou za časový interval frontu odeslat na server
  - Přijmout odpovědi a předat je Klientovi
  - Umožnit okamžité vyřízení fronty
- Poskytnout rozhraní pro práci s událostmi – možnost navázat na události zpětná volání (callbacky)
- Odhlášení uživatele od serveru
- Řešení detekce on-line a off-line režimu (výpadky internetu)
- Kontrola, zda nějaký požadavek nezůstal dlouho bez odpovědi – neztratil se




Obrázek 6.4: Diagram komunikace mezi všemi částmi aplikace při zjednodušené inicializaci Klienta

V popisu funkcí knihovny je zmíněna fronta požadavků. Bylo by nehospodárné při každém požadavku Klienta posílat AJAXový (4.2.3) dotaz. Vhodnější je požadavky seskupit a vyřídit naráz jednou za časový interval. Občas se ale může hodit okamžité vyřízení fronty.

Samotný požadavek předaný pro vyřízení knihovně je JavaScriptovým objektem. Může mít nastavené zpětná volání (callbacky) na stavy, kdy uspěl, kdy neuspěl a nebo byl ztracen (dlouho nepřišla odpověď). Díky podpoře uzávěrů (closures) v jazyce JavaScript, což je možnost jednoduše do proměnné uložit funkci, která má přístup k vnějšímu kontextu, v době vytvoření funkce, a později ji zavolat [31, s. 75], se s callbacky velmi dobře pracuje.

V části 6.6.5 jsme si ukázali příklad komunikace při inicializaci Klienta. Konkrétně si na ni můžeme demonstrovat užití knihovny. Zdrojový kód je v příloze A.

Na začátku kódu deklarujeme globální proměnné `accounts`, což je objekt seskupující IM účty a `roster`, což je objekt seskupující kontakty daných účtů. Po té inicializujme knihovnu přes proceduru `init`, jíž předáme inicializační parametry. V ukázce jsou zkráceny, podstatný je parametr `onInit`, jenž očekává callback, který bude zavolán po inicializaci knihovny. `onInit` je předána reference na funkci `clientInit`. Knihovna sama pošle serveru příkaz `register`, který inicializuje datové struktury na Serveru.

Funkce `clientInit` má argument `reconnected`, který informuje, zda Server Klienta nezná (hodnota je `false`), či zná (hodnota je `true`) a tedy Klient nebyl minule korektně odhlášen a Server má stále jeho datové struktury v paměti. Podle stavu argumentu `reconnected` musí probíhat inicializace Klienta, která se v obou variantách liší. Dále si popíšeme obvyklejší variantu, kdy je `reconnected` rovno `false`.

Prvně je potřeba poslat požadavek `get_accounts`, jenž požádá Bridge o seznam účtů z databáze. Vytvoříme instanci třídy `GetAccounts`, která je poděděná od třídy `Request`, reprezentující požadavek. Předpokládáme zde úspěch, proto ji nastavíme callback `onSuccess`, ve kterém obdržíme odpověď (`response`) se všemi IM účty.

Dále inicializujeme globální objekt `accounts`, který obsahuje seznam všech účtů a jejich aktuální stav. V tuto fázi je stav všech účtů `offline`, jelikož k nim nejsme přihlášeni. Poté deklarujeme počítadlo účtů `accountsN`, které nastavíme na počet všech účtů. Začneme iterovat účty a každý se pokusíme přihlásit přes požadavek `login`, v knihovně reprezentovaný třídou `Login`.

Zde už je kromě úspěchu (callback `onSuccess`) potřeba počítat i s neúspěchem (callback `onFailed`, případně ztrátou požadavku (callback `onLost`), jelikož v každém callbacku dekrementujeme počítadlo účtů a když poklesne na nulu, poznáme, že je inicializace dokončena. Pokud bychom dekrementovali počítadlo jen v případě úspěchu, neúspěšný pokus přihlášení k IM účtu by způsobil zamrznutí aplikace.

V případě úspěchu rovněž změníme stav v objektu `accounts` u daného účtu na `online` a pokusíme se získat kontakty účtu, jenž jsou uložené na IM serveru. To se provede požadavkem `get_contacts`, zde reprezentovaný třídou `GetContacts`. V jejím callbacku `onSuccess` kontakty uložíme do objektu `roster` a dekrementujeme počítadlo účtů.

Ve chvíli, kdy je počítadlo `accountsN` rovno nule, voláme proceduru `afterInit`, ve které pokračuje běh Klienta.

Na příkladu je vidět, že programování Klienta je často událostně řízené a tím klade důraz na programátora Klienta, jenž musí ošetřit různé chybové situace, jinak hrozí neočekávané chování. Na druhou stranu se knihovna snaží autorovi Klienta pomoci využitím technik objektově orientovaného a událostně řízeného programování.

Přesnější popis knihovny je v komentářích ve zdrojových kódech v souborech `myim_bridge/www/js/myimLib.js`, `myim_bridge/www/js/myimRequest2.js` a `myim_bridge/www/js/client2.js`.

### 6.6.7 Zhodnocení možností API

API umožňuje implementaci všech stěžejních částí IM aplikace, které jsme si stanovili ve specifikaci (5). Skrze požadavky `account_add`, `account_edit` a `account_remove` umožňuje přidat, upravit či odebrat IM účet. API podporuje služby/protokoly ICQ a XMPP, díky kterému je možné podporovat i Google Talk a Facebook. Požadavek `login`, `account_change_state` a `logout` umožňuje přihlášení, změnu stavu a odhlášení IM účtu.

Přes požadavek `send_message` je možné odeslat zprávu a skrze událost `message` zprávu obdržet. Zprávy je možné zaznamenávat do historie přes požadavek `history_add` a historii vybrat či vymazat přes požadavky `history_find` a `history_remove`.

Také je možné získat, přidat a odebrat kontakty IM účtu skrze `get_contacts`, `account_add_contact`, `account_remove_contact`. Skrze požadavek `authorization` a událost `authorization` je možné vyřizovat autorizace kontaktů. Změnu stavu kontaktu ohlašuje událost `contact_status_changed`.

API také počítá s výpadkem IM serveru skrze událost `account_disconnect`. Pro pohodlnější využívání API autorem Klienta vznikla knihovna, která obaluje API a umožňuje s ním pracovat skrze JavaScriptové funkce (6.6.6).

# Kapitola 7

# Závěr

Cílem práce bylo navrhnout architekturu aplikace, implementovat serverovou část včetně API a tím připravit půdu pro vznik Klienta. Zkompletováním všech částí měla vzniknout použitelná webová aplikace pro instant messaging. S potěšením musím oznámit, že se cíl splnit podařilo.

Klienta vytvořil student VUTBR FIT, František Sabovčik. Klient splňuje cíle stanovené specifikací (5), což dokazuje, že návrh serverové části, včetně API a jeho implementace, funguje správně. Klient je ukázán na snímku obrazovky v příloze B.

Nyní se projekt nachází v testovací fázi. Aplikace byla otestována na úzkém okruhu uživatelů a reakce byly vesměs pozitivní. Nyní je potřeba aplikaci testovat při větším počtu uživatelů a vyladit její stabilitu a výkon. U klientské části je potřeba sledovat preference uživatelů, jejich reakce na UI, a to s ohledem na ně přizpůsobit. Práce na klientské straně je ale v režii autora Klienta.

Jakmile bude aplikace dostatečně stabilní a uživatelsky přizpůsobená, je možné přejít z testovací fáze k oficiálnímu spuštění aplikace. Podle zájmu o aplikaci je pak možné hledat vhodný obchodní model. Jednou z možností je aplikaci prodávat do firemní sféry, případně inkasovat za zobrazování reklamy.

Budoucnost aplikace, kromě zájmů uživatelů, také ovlivní trendy v IM službách/protokolech. Jestli se budou nadále uzavírat, a nebude je možné podporovat, využitelnost aplikace bude omezena.

Aplikaci je možno dále rozšiřovat. Aplikaci by bylo vhodné lokalizovat do více jazyků, minimálně angličtiny. Je možné vytvořit mobilní verzi (například jako mobilní aplikaci využívající API serverové části), přidat další služby (interaktivní tabule), či videotelefonii (např. přes technologii WebRTC [32]).

Pokud by o aplikaci projevila zájem firemní sféra, je možné implementovat administraci pro pohodlnou správu uživatelů, např. s možností kontroly, jestli se věnují pouze firemní komunikaci.

U webové služby spuštěné pro běžné uživatele, je možné dodávat při registraci jabber účet a tím vlastně vytvořit novou IM síť.

Skutečný potenciál aplikace ale ukáže až budoucnost.

# Literatura


[1] *Pidgin* [online]. [cit. 2014-04-10]. Dostupné z: https://pidgin.im/

[2] POLESNÝ, David. Univerzální Meebo Messenger bude za měsíc zrušen. *Živě.cz* [online]. [cit. 2014-04-10]. Dostupné z: http://www.zive.cz/bleskovky/univerzalni-meebo-messenger-bude-za-mesic-zrusen/sc-4-a-164095/default.aspx

[3] POLESNÝ, David. Služba Imo.im dnes zabije svou největší výhodu. *Živě.cz* [online]. [cit. 2014-04-10]. Dostupné z: http://www.zive.cz/bleskovky/sluzba-imoim-dnes-zabije-svou-nejvetsi-vyhodu/sc-4-a-172673/default.aspx

[4] *plus.im* [online]. [cit. 2014-04-13]. Dostupné z: https://plus.im/

[5] *Iwantim* [online]. [cit. 2014-05-10]. Dostupné z: http://www.iwantim.com/

[6] Meebo. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation [cit. 2014-04-13]. Dostupné z: http://en.wikipedia.org/w/index.php?title=Meebo&oldid=604714195

[7] RFC 1459. *Internet Relay Chat Protocol*. 1993. Dostupné z: http://tools.ietf.org/html/rfc1459.html

[8] *XMPP Standards Foundation* [online]. [cit. 2014-04-14]. Dostupné z: http://xmpp.org

[9] *ICQ* [online]. [cit. 2014-04-13]. Dostupné z: http://www.icq.com/

[10] KRČMÁŘ, Petr. ICQ mění licenci: chce kontrolovat alternativní klienty. *Root.cz* [online]. [cit. 2014-04-13]. Dostupné z: http://www.root.cz/clanky/icq-meni-licenci-chce-kontrolovat-alternativni-klienty/

[11] *Skype* [online]. [cit. 2014-04-14]. Dostupné z: http://www.skype.com/

[12] Chat API. *Facebook* [online]. [cit. 2014-04-17]. Dostupné z: https://developers.facebook.com/docs/chat/

[13] Google's chat client drops Jabber compatibility. *The H* [online]. [cit. 2014-04-17]. Dostupné z: http://www.h-online.com/open/news/item/Google-s-chat-client-drops-Jabber-compatibility-1866129.html

[14] GUTMANS, Andi, Stig Sæther BAKKEN a Derick RETHANS. *AJAX a PHP: tvoříme interaktivní webové aplikace profesionálně*. Vyd. 1. Překlad Bogdan Kiszka. Brno: CP Books, 2006, 320 s. ISBN 80-868-1547-1.

[15] Wikimedia Traffic Analysis Report – Browsers e.a. *Wikimedia* [online]. [cit. 2014-04-23]. Dostupné z: http://stats.wikimedia.org/wikimedia/squids/SquidReportClients.htm

[16] HAUSER, Marianne, Stig Sæther BAKKEN a Derick RETHANS. *HTML a CSS: tvoříme interaktivní webové aplikace profesionálně.* Vyd. 1. Překlad Bogdan Kiszka. Brno: Computer Press, 2006, 912 s. ISBN 80-251-1117-2.

[17] GOLDSTEIN, Alexis, Louis LAZARIS a Estelle WEYL. *HTML5 a CSS3 pro webové designéry: velká kniha řešení.* Vyd. 1. Překlad Bogdan Kiszka. Brno: Zoner Press, 2011, 286 s. ISBN 978-80-7413-166-0.

[18] HTML5. *W3C* [online]. [cit. 2014-04-24]. Dostupné z: http://www.w3.org/TR/html/

[19] KOSEK, Jiří. XHTML je mrtvé! Ať žije HTML5! Nebo ne?. *zdroják.cz* [online]. [cit. 2014-04-24]. Dostupné z: http://www.zdrojak.cz/clanky/xhtml-je-mrtve-at-zije-html5-nebo-ne/

[20] YANK, Kevin a Cameron ADAMS. *Začínáme s JavaScriptem.* Vyd. 1. Překlad Jakub Zemánek. Brno: Zoner Press, 2008, 591 s. ISBN 978-80-86815-94-7.

[21] HOLZNER, Steven. *Mistrovství v Ajaxu.* Vyd. 1. Překlad Jakub Zemánek. Brno, 2007, 591 s. ISBN 978-80-251-1850-4.

[22] *JSON* [online]. [cit. 2014-04-25]. Dostupné z: http://www.json.org/

[23] EMBRY, Darren. Why PHP Sucks. In: *webonastick.com* [online]. [cit. 2014-04-25]. Dostupné z: http://webonastick.com/php.html

[24] GILMORE, W. *Velká kniha PHP 5 a MySQL: kompendium znalostí pro začátečníky i profesionály.* Nové, 3. vyd. Překlad Jan Pokorný. Brno: Zoner Press, 2011, 736 s. Encyklopedie Zoner Press. ISBN 978-80-7413-163-9.

[25] The Spirit of Openness. *Facebook* [online]. [cit. 2014-04-25]. Dostupné z: https://www.facebook.com/notes/facebook/the-spirit-of-openness/2223862130

[26] *Nette Framework* [online]. [cit. 2014-04-25]. Dostupné z: http://nette.org/

[27] *dibi* [online]. [cit. 2014-04-26]. Dostupné z: http://dibiphp.com/

[28] *Jansson* [online]. [cit. 2014-04-28]. Dostupné z: http://www.digip.org/jansson/

[29] TICHÝ, Jan. Solení hesel aneb Sůl nad zlato. In: *PHP Guru.cz* [online]. [cit. 2014-05-02]. Dostupné z: http://www.phpguru.cz/clanky/soleni-hesel

[30] VRÁNA, Jakub. Ukládání citlivých informací. In: *PHP triky* [online]. [cit. 2014-05-02]. Dostupné z: http://php.vrana.cz/ukladani-citlivych-informaci.php

[31] CRANE, Dave. *Ajax in practice.* Greenwich: Manning, c2007, xxiii, 508 s. ISBN 19-323-9499-0.

[32] *WebRTC* [online]. [cit. 2014-05-04]. Dostupné z: http://www.webrtc.cz

# Příloha A

# Ukázka inicializace Klienta s využitím knihovny pro práci s API

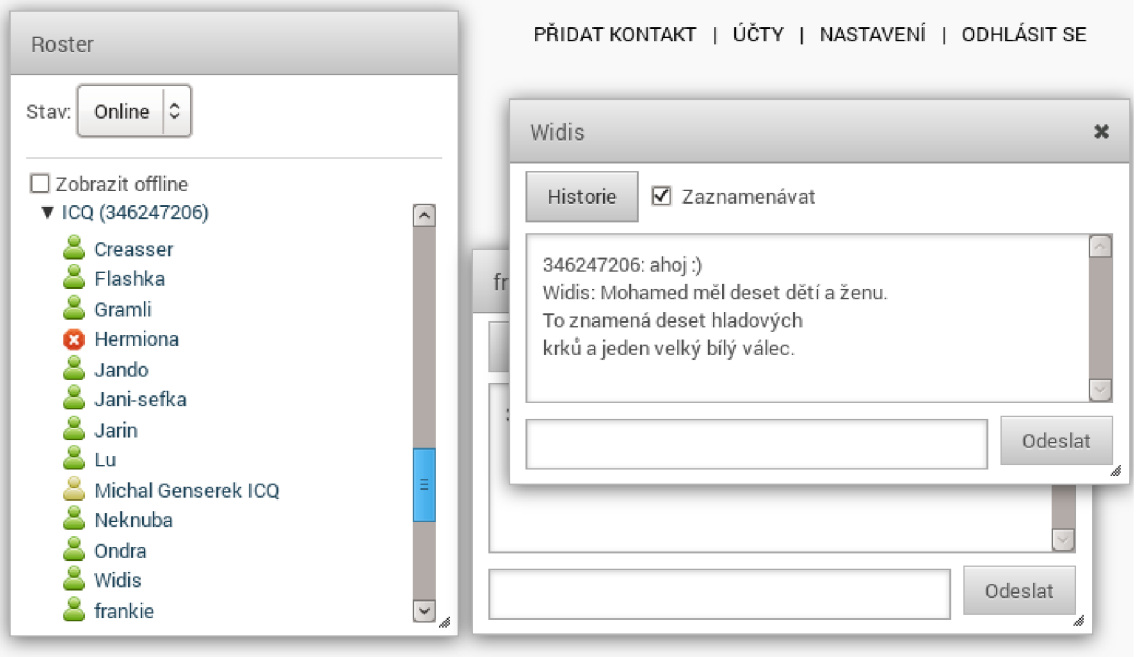
```
//Uživatelské účty
var accounts;

//Kontakty účtů
var roster;

//Inicializace myim
myimLib.init({
  'onInit':clientInit,
  ...
});

/**
 * Inicializuje klienta
 * Konkrétně zjistí ze serveru uzivatelske účty (do accounts)
 * A seznam kontaktů v nich (do roster)
 * @param bool existovala na serveru registrace (uživatel znova-načetl
 * stránku)?
 */
function clientInit(reconnected)
{
  accounts = {};
  roster = {};

  //Požádáme Bridge o seznam účtů
  myimLib.request(new myimRequests.GetAccounts()
    .onSuccess(function(request, response)
    {
      //Inicializujeme strukturu účtů
      response.data.forEach(function(account)
```

```
{
  accounts[account.account_id] = account;
  //zatím všechny považujeme za odpojené
  accounts[account.account_id].state = "offline";
});

//Jsme nově připojeni
if(!reconnected)
{
  //Počítadlo, díky kterému zjistíme, že jsme připojili už všechny
  //účty
  var accountsN = response.data.length;

  //Existují vůbec nějaké účty
  if(accountsN > 0)
  {
    //Projdeme je
    $.each(accounts, function(index, account)
    {
      //Pokusíme se účet přihlásit
      myimLib.request(new myimRequests.Login(account.account_id,
        "online")
        .onSuccess(function(request, response)
        {
          //Můžeme mu nastavit správný stav
          account.state = "online";
          //Načteme kontakty účtu
          myimLib.request(new myimRequests.GetContacts(
            account.account_id)
            .onSuccess(function(request, response)
            {
              //Přidáme je do seznamu kontaktů
              roster[request.getData().account_id] =
                response.data[0].contacts;
              if(--accountsN === 0)
                afterInit();
            })
          );
        })
        //Né vždy se podaří přihlásit
        .onLost(function(request, response)
        {
          if(--accountsN === 0)
            afterInit();
        })
        .onFailed(function(request, response)
        {
          if(--accountsN === 0)
```

```
                afterInit();
              })
            );
          });
        }
        else
        {
          afterInit();
        }
      }
      //Už jsme na serveru registrovaní
      else
      {
        //...
      }
    });
  );
}

/**
 * Dokončená inicializace
 */
afterInit()
{
  //...
}
```

# Příloha B

# Snímek obrazovky Klienta




Obrázek B.1: Snímek obrazovky Klienta vytvořeného Františkem Sabovčikem

# Příloha C

# Obsah přiloženého CD

Přiložené CD obsahuje:

- **Tuto práci ve formátu PDF** – nachází se v adresáři `prace_pdf`
- **Tuto práci ve formátu LaTeX** – nachází se v adresáři `prace_tex`
- **Zdrojové kódy části aplikace Server** – nachází se v adresáři `myim_server`
- **Zdrojové kódy části aplikace Bridge** – nachází se v adresáři `myim_bridge`
- **Zdrojové kódy části aplikace Klient** – nachází se v adresáři `myim_klient`[1]

Pokyny pro instalaci aplikace ze zdrojových kódů všech částí aplikace jsou uvedeny v souborech `install.txt`, případně `INSTALL`, v adresářích jednotlivých částí.

[1] Klient je vytvořen, dle zadání, studentem VUTBR FIT Františkem Sabovčikem a je dodáván za účelem zprovoznitelnosti aplikace.