

HTTP proxy server

An HTTP Proxy Server

Bc. Ondřej Doležal

Diplomová práce
2012



Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení: **Bc. Ondřej DOLEŽAL**
Osobní číslo: **A08756**
Studijní program: **N 3902 Inženýrská informatika**
Studijní obor: **Informační technologie**

Téma práce: **HTTP proxy server**

Zásady pro vypracování:

1. Vypracujte literární rešerši na téma proxy server v internetu.
2. Vytvořte program, který bude zastávat funkci jednoduchého http proxy serveru.
3. Cílová platforma bude GNU/Linux. Programování C/C++ (sokety, vláknové programování).
4. Vytvořte jednoduché webové rozhraní pro sledování komunikace a stavu proxy pro výukové účely.
5. Vypracujte manuál k vytvořenému programu.

Rozsah diplomové práce:

Rozsah příloh:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

1. RFC 791. Internet Protocol. [s.l.] : IETF, 1981. 45 s. Dostupné z WWW: <http://tools.ietf.org/html/rfc791>.
2. RFC 793. Transmission Control Protocol. [s.l.] : IETF, 1981. 85 s. Dostupné z WWW: <http://tools.ietf.org/html/rfc793>.
3. RFC 1945. Hypertext Transfer Protocol – HTTP/1.0. [s.l.] : IETF, 1996. 60 s. Dostupné z WWW: <http://tools.ietf.org/html/rfc1945>.
4. RFC 2616. Hypertext Transfer Protocol – HTTP/1.1. [s.l.] : IETF, 1999. 176 s. Dostupné z WWW: <http://tools.ietf.org/html/rfc2616>.
5. RFC 2617. HTTP Authentication: Basic and Digest Access Authentication. [s.l.] : IETF, 1999. 34 s. Dostupné z WWW: <http://tools.ietf.org/html/rfc2617>.
6. RFC 3143. Known HTTP Proxy/Caching Problems. [s.l.] : IETF, 2001. 32 s. Dostupné z WWW: <http://tools.ietf.org/html/rfc3143>.
7. GAY, Warren. Linux Socket Programming by Example. [s.l.] : Que, 2000. 557 s. ISBN 0789722410.

Vedoucí diplomové práce:

doc. Ing. Martin Sysel, Ph.D.

Ústav počítačových a komunikačních systémů

Datum zadání diplomové práce:

24. února 2012

Termín odevzdání diplomové práce:

21. května 2012

Ve Zlíně dne 24. února 2012



prof. Ing. Vladimír Vašek, CSc.
děkan



doc. Mgr. Roman Jašek, Ph.D.
ředitel ústavu

ABSTRAKT

Internet a zejména World Wide Web jsou všudepřítomné. V řadě případů lze zvýšit efektivitu nebo bezpečnost přístupu k WWW nasazením HTTP proxy serveru. Práce uvádí do problematiky HTTP proxy. Současný HTTP proxy server nutně pracuje se sockety, a obvykle s výhodou využívá vícevláknovou architekturu. Práce podává úvod do těchto technologií a je nabízí implementací takového vícevláknového HTTP proxy serveru s funkcionalitou WWW serveru pro uživatelské rozhraní.

Klíčová slova: HTTP, proxy server, sockety, vlákna

ABSTRACT

The Internet and World Wide Web in particular are ubiquitous. The efficiency and safety of access to the Web can be enhanced by the deployment of an HTTP proxy server in many cases. This work provides an introduction to the issue of an HTTP proxy. Current HTTP proxy server requires the use of sockets a preferably multithreaded architecture. The paper provides an introduction to these technologies and an implementation of a multithreaded HTTP proxy server with an embedded WWW server used for the user interface.

Keywords: HTTP, proxy server, sockets, threads

Poděkování

Děkuji vedoucímu diplomové práce panu doc. Ing. Martinu Syslovi, Ph.D. za pomoc a rady, které mi poskytl při zpracování zadaného tématu.

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové/bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová/bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou/bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou/bakalářskou práci nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- beru na vědomí, že pokud bylo k vypracování diplomové/bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové/bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové/bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve

.....

diplomanta

Zlíně

podpis

OBSAH

ÚVOD	9
I TEORETICKÁ ČÁST	10
1 PROXY SERVER	11
1.1 DŮVODY UŽITÍ APLIKAČNÍ PROXY	11
1.2 ZÁKLADNÍ ROZDĚLENÍ APLIKAČNÍCH PROXY	13
1.2.1 ROZDĚLENÍ PODLE FUNKCIONALITY.....	13
1.2.2 ROZDĚLENÍ PODLE PŘÍSTUPU.....	14
2 PROTOKOL HTTP	16
2.1 DETAILY HTTP PROTOKOLU.....	16
2.2 ČINNOST HTTP PROXY.....	18
3 NĚKTERÉ EXISTUJÍCÍ IMPLEMENTACE HTTP PROXY	20
4 SOCKETY	22
4.1 OTEVŘENÍ SOCKETU.....	22
4.2 POUŽITÍ SOCKETŮ.....	24
5 VLÁKNA	26
5.1 POUŽITÍ VLÁKEN.....	26
II PRAKTICKÁ ČÁST	28
6 PROGRAMÁTORSKÁ DOKUMENTACE	29
6.1 IMPLEMENTACE HTTP PROXY SERVERU.....	29
6.1.1 PAMĚŤOVÉ NÁROKY.....	29
6.1.2 PROCESOROVÉ NÁROKY.....	30
6.1.3 VÝKON PROXY SERVERU.....	32
6.2 ARCHITEKTURA SYSTÉMU.....	33
6.2.1 HLAVNÍ VLÁKNO.....	34
6.2.2 VYŘIZOVÁNÍ POŽADAVKŮ.....	36
6.2.3 VYŘIZOVÁNÍ PROXY POŽADAVKŮ	36
6.2.4 VYŘIZOVÁNÍ HTTP POŽADAVKŮ	37
6.2.5 LOGOVÁNÍ A STATISTIKA.....	38
6.2.6 ŠABLONOVACÍ SUBSYSTÉM.....	39
6.3 PROGRAMOVÉ CELKY.....	40
6.4 GLOBÁLNÍ PROMĚNNÉ.....	41
6.4.1 SOUBOR MAIN.C.....	41
6.4.2 SOUBOR SERVER.C.....	41
6.4.3 SOUBOR CONSTANTS.C.....	42
7 UŽIVATELSKÁ DOKUMENTACE	43

7.1	INSTALACE.....	43
7.2	SPUŠTĚNÍ APLIKACE.....	43
7.3	POUŽITÍ PROXYSERVERU.....	45
7.4	UŽIVATELSKÉ ROZHRANNÍ.....	45
7.4.1	UŽIVATELSKÉ ROZHRANNÍ PRO SLEDOVÁNÍ PROVOZU.....	46
	ZÁVĚR.....	48
	ZÁVĚR V ANGLIČTINĚ.....	49
	SEZNAM POUŽITÉ LITERATURY.....	50
	SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK.....	51
	SEZNAM PŘÍLOH.....	54

ÚVOD

Celosvětová počítačová síť Internet se od svých počátků v 70. a 80. let 20. století neustále prudce rozrůstá. K červnu 2009 je k síti připojeno více než 1,67 miliardy uživatelů. Nejčastějším způsobem využití sítě Internet těmito koncovými uživateli je přístup k WWW - World Wide Webu. Tento přístup je natolik dominantní, že i v běžné komunikaci lze vysledovat splývání pojmů Internet a WWW. Důvodů takové obliby je více - architektura umožňující bezproblémovou spolupráci zcela heterogenních systémů, intuitivní vizuální a i interaktivní uživatelské rozhraní, kolaborativní prostředí umožňující jednoduše se zapojit do publikování informací a zároveň dostatečná komplexita pro vytváření složitých aplikací.

Samotný World Wide Web lze charakterizovat jako distribuovaný informační systém typu klient-server s tenkým klientem, přenášející informace ve formě HTML stránek a dalších objektů aplikačním protokolem HTTP. Jde o typický protokol typu požadavek-odpověď, který řídí přenos dat mezi serverem a klientem jako je WWW prohlížeč. HTTP provoz tak je dvoustrannou komunikací; existuje však řada případů užití této komunikace které mohou profitovat či jsou dokonce přímo umožněny zařazením dalšího aktivního prvku - proxy serveru - do komunikačního kanálu.

Proxy server je prvek zabezpečující funkci prostředníka v komunikaci mezi serverem a klientem. Tento prostředník může komunikaci jen předávat, nebo do ní provádět zásahy na úrovni aplikačního protokolu. HTTP proxy server může např. zrychlit přístup ke zdrojům, provádět kontrolu nebo sledování probíhajícího provozu nebo zajistit anonymitu uživatelů.

Cílem této diplomové práce je provést analýzu požadavků na internetový HTTP proxy server, specifikovat způsoby jejich řešení a vytvořit návrh aplikačního programu který bude funkce HTTP proxy realizovat a provést implementaci takového programu v prostředí GNU Linux včetně potřebné dokumentace. Vytvořený program bude vytvořen jen za pomoci produktů svobodného softwaru a sám bude publikován pod otevřenou licencí GNU GPL.

Práce je rozdělena na teoretickou a praktickou část. Teoretická se zabývá popisem požadavků, technologií a přístupů k řešení. Praktická část zahrnuje dokumentaci vytvořeného programu.

I. TEORETICKÁ ČÁST

1 PROXY SERVER

Proxy server je počítačový systém - kombinace hardwarové platformy a aplikačního software - který slouží jako prostředník v síťové komunikaci mezi jednotlivými stranami. U systémů klient-server tedy slouží jako prostředník v komunikaci mezi klientem, většinou odesílajícím požadavky, a serverem vysílajícím odpovědi. Obecně ovšem není princip proxy omezen na klient-server systémy, vhodná proxy může zprostředkovávat výměnu zpráv například v sítích typu peer-to-peer.

Základní princip činnosti proxy serveru je přijímat požadavky klientů (vůči kterým se tváří jako server), tyto požadavky analyzovat a následně odeslat cílovým serverům (vůči nimž se tváří jako klient) a poté příslušné odpovědi předávat původním klientům - v původní či upravené podobě, podle druhu proxy. Z nutnosti analyzovat přichozí požadavky vyplývá nutnost proxy serveru operovat na 7., tj. aplikační vrstvě modelu ISO/OSI. Proto se také tyto proxy nazývají proxy aplikačními. Dalším logickým důsledkem je nutnost proxy serveru pracovat se stejným aplikačním protokolem jako obsluhovaní klienti. Obsluhu různých protokolů tak lze dosáhnout různými proxy servery, nebo servery víceprotokolovými [delegate].

Vedle těchto aplikačních proxy serverů existují také proxy aplikačně nezávislé, zajišťující pouze transport paketů bez znalosti protokolů aplikační vrstvy. Jejich využití však proto vyžaduje použití specifického komunikačního protokolu pro komunikaci s proxy serverem. Typickým zástupcem protokolu univerzální proxy je SOCKS protokol, pracující na 5. tj session vrstvě ISO/OSI modelu. Nasazení SOCKS proxy vyžaduje přizpůsobení klientské aplikace - modifikace síťového kódu. Existují ovšem i klientské implementace které po spuštění přesměrují síťovou komunikaci klienta na SOCKS proxy v příslušném protokolu, odstraňují tak nutnost upravovat klientský kód. HTTP proxy je proxy aplikační, proto se tato práce nebude univerzálními proxy zabývat.

1.1 Důvody užití aplikační proxy

Primárním důvodem pro nasazení prvních proxy serverů bylo umožnění přístupu k vnějším zdrojům počítačům umístěným uvnitř sítě chráněné firewallem nebo jinak přímo nedostupných. Proxy server je v takovém případě nainstalován na počítač s firewallem, a slouží jako brána zprostředkovávající síťový provoz příslušného aplikačního protokolu. Podobného efektu - zprostředkování zdrojů - lze dosáhnout při

přítomnosti firewallu např. vhodným stavovým pravidlem a otevřením provozu pro odchozí požadavky; komunikace mezi klientem a serverem je pak běžným způsobem routována.

Nasazení aplikační proxy nicméně nabízí další rozšíření funkčnosti:

1. Ve většině případů přistupuje k jedné proxy více klientů. Skrze proxy tak projdou všechny odpovědi na požadavky všech klientů, a v případě že proxy server ukládá obsahy jednotlivých odpovědí, může zrychlit odezvu a snížit nároky na přenosové pásmo na opakované požadavky tím že místo kontaktování cílového serveru použije uloženou odpověď z předchozího identického požadavku. Takový proxy server se nazývá caching proxy. Většina HTTP proxy serverů implementuje caching v nějaké formě. Významnou otázkou pro caching proxy je platnost (validita) uložených odpovědí; a to do té míry že tyto otázky řeší samotná specifikace protokolu HTTP.

2. Proxy server umožňuje, díky zpracování požadavků na úrovni aplikačního protokolu, přesněji a s jemnější granularitou než firewall definovat možnosti přístupu k jednotlivým zdrojům. Obecně tak v rámci organizace lze omezit přístup pro jednotlivé klientské počítače podle cílových adres, protokolu, názvu nebo typu zdrojů. Specializované HTTP proxy servery podporují i časové omezení v rámci dne nebo řízení rychlosti přenosu, čímž lze například omezit neefektivní využití přenosového pásma v pracovní době nebo k nevhodným účelům.

3. V dnešní době neustále důležitější bezpečnost lze posílit použitím filtrační proxy pro detekci a blokování škodlivého obsahu. Opět díky zpracování na aplikační úrovni může proxy server provádět antivirovou kontrolu příchozího obsahu a blokovat přístup k nřikovaným zdrojům. Stejně tak lze provádět i kontrolu odchozích dat, jak z hlediska virů a obecně tzv. malwaru tak vzhledem např. k datovým únikům apod.

4. Další možnost pro využití filtrování (modifikace) průchozího obsahu je u proxy serverů zprostředkovávající obsah pro jiné než původně určené platformy, jako je třeba přístup k WWW zdrojům z mobilních zařízení. Takové aplikační proxy mohou provádět dynamickou recompresi pro snížení datového objemu, transkripci obsahu pro vynechání nepodporovaných komponent atp. S výhodou se taková funkcionalita spojí s cachingem, který umožní uložit výsledky transformací a vyhnout se tak jejich opakování.

5. Zvýšení bezpečnosti lze dosáhnout i použitím aplikačního proxy serveru k logování a auditu klientských požadavků; na rozdíl od logování na nižších vrstvách umožňuje logování na aplikační vrstvě snadný přístup ke všem vlastnostem klient/server transakce a jejich zaznamenání a následné vyhodnocení.

6. V neposlední řadě pak lze proxy server využít k anonymizaci přístupu k cílovému serveru. Tohoto efektu může být použito pro obejít omezení platných pro příslušnou zdrojovou adresu klienta, případně u vhodně nakonfigurovaných proxy i k zamaskování atributů klientského systému jako je druh a verze softwaru apod.

7. Aplikační proxy server může ve vhodných případech provádět převod mezi různými protokoly na stranách klienta a cílového serveru. V praxi pak funguje jako překladač na úrovni aplikačních protokolů, což může klientům zpřístupnit zdroje k nimž jinak vůbec nemá možnost přistupovat, či programátorovi klientského softwaru ušetřit mnoho řádek kódu jinak nutného pro to aby klient daný protokol podporoval. Vhodnými příklady je např. přístup k FTP nebo systému Gopher prostřednictvím WWW prohlížeče.

1.2 Základní rozdělení aplikačních proxy

1.2.1 Rozdělení podle funkcionality

Hlavní funkcí proxy serveru je zprostředkování spojení klient/server, mimo tuto hlavní činnost ale mohou navíc poskytovat další služby a možnosti, jak bylo nastíněno. Nejdůležitější dodatečné funkce proxy serveru jsou:

1. Caching proxy - proxy s vyrovnávací pamětí. Proxy server ukládá obsahy jednotlivých odpovědí, tím zrychlí odezvu na požadavky a sníží nároky na přenosové pásmo tím že místo spojení s cílovým serverem použije odpověď uloženou z předchozího identického požadavku.

2. Filtering proxy - filtrační proxy. Provádí inspekci a případně i modifikaci obsahu odpovědi vzdáleného serveru (a případně obsahu požadavku klienta) podle zadaných pravidel. Může tak provádět mj. antivirovou kontrolu, rekompresi, transformaci do jiného formátu nebo jen povolit či zamítnout přístup podle daných kritérií.

3. Anonymous proxy - anonymizační proxy. Slouží k anonymizaci přístupu k cílovému serveru. Tohoto lze využito pro obejít omezení pro zdrojovou adresu klienta, případně

k zamaskování atributů klientského systému jako jsou operační systém, klientský software aj.

Tyto možnosti jsou vzájemně komplementární a proxy server tedy může podporovat jejich libovolnou kombinaci.

1.2.2 Rozdělení podle přístupu

Na rozdíl od doplňkové funkcionality, je rozdělení aplikačních proxy podle přístupu klientů zásadní charakteristikou která určuje způsob i důvod nasazení konkrétního proxy serveru. Každá proxy je proto jednoho z následujících typů:

1. Forward proxy - běžná nebo též dopředná proxy. Proxy funguje běžným výše popsaným způsobem, tj. zajišťuje spojení mezi omezenou skupinou klientů v rámci lokální sítě LAN nebo WAN a zdroji v rámci větší sítě, např. Internetu. Obecně všechny proxy s výjimkou obrácené mají charakteristiku dopředné proxy.

2. Open proxy - otevřená nebo též veřejná proxy. Obecně proxy která je k dispozici bez omezení všem uživatelům sítě, obvykle Internetu. Obvykle je otevřená proxy užívána k anonymizaci klientů a mívá proto vlastnosti anonymizační proxy.

3. Transparent proxy - transparentní proxy. Jedná se o nasazení proxy serveru takovým způsobem, že je pro klienta nemožné vyhnout se jeho využití, a často klient ani netuší že jeho požadavek na server byl zpracován prostřednictvím proxy serveru. Transparentní proxování musí podporovat jak proxy software, neboť daný aplikační protokol pro komunikaci s cílovým serverem může být odlišný od protokolu používaného pro komunikaci s běžnou proxy. Transparentní proxy musí být schopná všechny potřebné informace získat z běžného (na proxy necíleného) požadavku klienta. Další podmínkou pro transparentní proxování je podpora na hraničním routeru dané sítě (bráně), která musí umožňovat přesměrování síťové komunikace. Obvykle je veškerá odchozí komunikace (daného aplikačního protokolu) přesměrována do proxy serveru který ji následně zpracuje. Transparentní proxování má velké využití ve veřejně přístupných sítích, kde umožňuje využít cachingu bez nutnosti konfigurovat jednotlivé klienty. Ještě důležitější je ale bezpečnostní využití, kdy lze transparentně filtrovat veškerý provoz. Toto využití však může být i kontroverzní - viz tzv. Velký čínský firewall, nástroj pro cenzuru Internetu prostřednictvím systému filtrujících transparentních proxy v Číně.

4. Reverse proxy - obrácená nebo reverzní proxy. Proxy server této konfigurace je instalován v systému serveru a nikoli klienta. Vůči klientům se taková proxy tváří jako běžný server dané aplikace, přijímá požadavky a ty poté předává jednomu nebo několika koncovým serverům. Klientský program tak vůbec nerozliší že komunikuje s proxy serverem, stejně jako v případě transparentní proxy. Důvody pro použití reverzní proxy jsou však odlišné; takový proxy server je nasazen pro zlepšení funkce koncových serverů pro všechny klienty. Je to umožněno zejména: možností jednoduchého cachingu odpovědí serverů pro všechny klienty, bez nutnosti takový caching zajišťovat na aplikační úrovni. Další výhodou může být předřazení uživatelské autorizace, opět bez nutnosti řešení na aplikační úrovni. Jednou z nejdůležitějších možností pak je relativně snadná konsolidace distribuovaných systémů, kdy jeden rychlý proxy server stojí před několika pomalejšími koncovými servery, na něž distribuuje požadavky - buď formou pevně rozdělených úloh, nebo nějakým cyklickým mechanismem (tzv. round-robin).

2 PROTOKOL HTTP

Hypertext transfer protokol - HTTP - je internetový aplikační protokol určený pro přenos hypertextových dokumentů, původně primárně HTML formátu. V dnešní době je však díky rozšířením možné přenášet HTTP protokolem soubory libovolného formátu. Protokol se dočkal od svého vzniku řady aktualizací, poslední verze 1.1 je definována v RFC2616. Pro identifikaci zdrojů je v HTTP komunikaci používán tzv. jednotný lokátor zdrojů (Uniform Resource Locator, URL) běžně nazývaný URL-adresou. Pomocí URL lze odkázat na libovolný přístupný dokument nebo i jeho část. Pro HTTP provoz je běžně využíván TCP port 80.

Mimo základního HTTP protokolu existuje i jeho zabezpečená varianta HTTPS, která umožňuje ověřovat identitu účastníků komunikace a šifrovat přenášená data. Pro tento účel se využívá asymetrické kryptografie s privátním klíčem (SSL/TLS). Pro HTTPS provoz je běžně využíván TCP port 443.

2.1 Detaily HTTP protokolu

HTTP je klient/server protokol typu požadavek/odpověď (request/response). Komunikaci zahajuje klient, který se připojí k serveru a pošle požadavek, nato server vrátí klientovi odpověď a tím je interakce (vyřízení požadavku) vyřešeno. Z toho důvodu může, a ve starších verzích protokolu i musí, server uzavřít spojení. V aktuální verzi protokolu je možné udržovat TCP spojení otevřené a použít jej pro vyřízení více požadavků z důvodů zrychlení odezvy - odstranění zátěže více navazovaných TCP spojení. Takto udržované spojení využívá tzv. keep-alive připojení

Nicméně i u keep-alive připojení je každá HTTP transakce požadavek/odpověď izolovaná bez toho aby udržovala kontext - z hlediska protokolu je pro klient i server každý požadavek nerozlišitelný a bez souvislosti s předchozími nebo následujícími požadavky. To zjednodušuje programování jednoduchých HTTP serverů určených pro pouhý přenos dokumentů. Postupem času kdy se ale HTTP začalo mimo pouhého přenosu používat jako základní protokol pro provoz hypertextových aplikací, komplikovala izolovanost každého požadavku jakékoli udržování stavu mezi jednotlivými požadavky. Odstranění tohoto nedostatku zajistilo rozšíření specifikace o mechanismus tzv. cookies, malé množství dat které může server odpovědí na požadavek nastavit v nějakém perzistentním úložišti

klienta, který pak tato data připojí do každého následujícího požadavku k danému serveru.

Jak požadavek, tak odpověď klienta se skládá z hlavičky ve formě čistého textu a volitelně těla (požadavku nebo odpovědi). Hlavička požadavku sestává z příkazu a dále volitelných položek. Příkaz obsahuje metodu, URL které má být danou metodou vyvoláno a verzi protokolu HTTP kterou klient používá. Mezi používané metody patří:

- GET: užívá se pro získání dokumentu na dané URL.
- HEAD: stejně jako GET, ale server vrátí jen hlavičku jakou by vrátil pro požadavek GET, bez toho aby odesílal samotný dokument.
- POST: užívá se pro odeslání dat na server, například pro zpracování HTML formuláře. Data jsou odeslána v těle požadavku. Metodou GET lze sice také předávat data, metoda POST je ale vhodná pro předání větší množství dat, případně když není vhodné aby data byla součástí URL adresy.
- PUT, DELETE: použití zejména u WebDAV klientů pro modifikaci dokumentů.
- CONNECT: umožňuje navázat spojení s uvedeným serverem na uvedeném portu. Je potřeba pro HTTPS připojení skrze HTTP proxy server.

Volitelné položky hlavičky jsou uvedeny každá na samostatném řádku jako páry jméno, hodnota oddělené dvojtečkou. Pomocí nich v požadavku klient oznamuje serveru např. jaké formáty je schopen zpracovat apod. Konec hlavičky je signalizován dvěma prázdnými řádky, pokud požadavek má obsahovat tělo toto ihned následuje. Podstatné pro server a také případnou proxy jsou zejména:

- Host: umožňuje podporu tzv. virtuálních hostů, tj. provozu více serverů na 1 IP adrese.
- If-Modified-Since: podmíněný požadavek - nebyl-li dokument modifikován od uvedeného data, server o tom informuje a klient použije vlastní kopii.
- Keep-Alive / Connection: podpora perzistentního spojení.
- Cookie: předávání cookies dat.
- Authenticate: předává autentizační údaje pro povolení přístupu k omezeným zdrojům.

Odpověď serveru také začíná hlavičkou, jejíž první řádek je stavový. Obsahuje verzi protokolu kterou používá server, stavový kód indikující zda proběhlo zpracování

v pořádku, případně jaké akce se očekávají od klienta, a textovou reprezentaci toho stavu. Následují opět jednotlivé volitelné položky na samostatných řádcích. Podstatné pro HTTP přenos, zpracování obsahu a také podporu proxování jsou zejména tyto položky:

- Content-Type: indikuje MIME-typ přenášeného obsahu.
- Content-Length: informuje o délce přenášeného obsahu.
- Expires: označuje datum do kterého klient může dokument považovat za aktuální.
- Last-Modified: informuje o poslední modifikaci dokumentu.
- Set-Cookie: slouží k nastavování cookies dat.
- WWW-Authenticate: vysílá požadavek na autentizaci uživatele, klient musí opakovat požadavek s autentizačními údaji.
- Pragma / Cache-Control: používá se pro omezení možnosti ukládat dokument do vyrovnávacích pamětí proxy serveru a/nebo i klienta.

Ukázka minimální HTTP komunikace mezi klientem a serverem může vypadat například takto:

```
HEAD / HTTP/1.1

Host: www.example.com

HTTP/1.0 200 OK

Content-Type: text/html

Content-Length: 2005
```

Je ovšem potřeba rozlišovat komunikaci HTTP klient/server a HTTP klient/proxy (vyjma transparentní). Jelikož u proxovaného spojení se klient připojuje k proxy, a Host hlavička není povinná, neměl by proxy server jak určit cílový server od něž má vyžádat odpověď. Příkaz je proto při spojení na proxy modifikován, místo pouhé cesty (URL-path) je po metodě vyžadována celá URL-adresa.

2.2 Činnost HTTP proxy

Aplikační proxy protokolu HTTP pracuje v principu tak, jak bylo uvedeno v bodu týkajícím se obecného fungování aplikační proxy. V tomto konkrétním případě tedy musí server (démon) pracující jako HTTP proxy po spuštění:

1. čekat na příchozí požadavky klientů
2. při příchodu klientského požadavku tento analyzovat, prověřit je-li možno mu vyhovět
3. pokud ano, otevřít spojení na vzdálený server dle požadavku klienta a přetlumočit jej
4. vyčkat na odpověď, případně ji také analyzovat a prověřit její platnost
5. je-li v pořádku, odeslat ji zpět klientovi
6. je-li to v souladu s metadaty odpovědi a konfigurací caching proxy, také odpověď uložit

Pochopitelně proxy servery o specifické funkčnosti (caching, filtering apod.) přidávají další nutné kroky. Např. caching proxy před odesláním požadavku na vzdálený server kontroluje zda-li jej nemůže vyřídít z vyrovnávací paměti, filtrující proxy před odesláním obsahu klientovi kontroluje zda-li vyhovuje pravidlům a případně provádí transformaci.

Z uvedeného schematu činnosti je zjevné, že HTTP proxy musí zahrnovat funkcionalitu HTTP serveru (na straně naslouchající požadavkům klientů) i HTTP klienta (na straně předávající požadavky cílovým serverům). Zároveň je zjevné že pro jakékoli smysluplné využití musí být proxy schopna souběžně plnit požadavky více klientů. Jelikož časová prodleva komunikace s cílovým serverem není zanedbatelná, není reálně možné vyřizovat požadavky v sérii po sobě ale nějakým způsobem tuto úlohu paralelizovat. Pro tento účel se nabízí dvě řešení - multiplexování, tj. střídavá obsluha více otevřených spojení, nebo vyhrazení samostatného toku řízení (vlákna nebo procesu) pro každé klientské spojení. Konkrétní řešení bude popsáno v následujících kapitolách.

3 NĚKTERÉ EXISTUJÍCÍ IMPLEMENTACE HTTP PROXY

Squid

Projekt Squid vznikl v roce 1996. Jde o robustní, univerzální řešení HTTP proxy která podporuje i FTP, HTTPS a Gopher. Umí pracovat v režimech dopředné i reverzní proxy, také jako proxy transparentní. Podporuje caching, rozšířené možnosti řízení přístupu i anonymizaci. Jde o vysoce škálovatelné řešení, pracující ve víceprocesním modelu. Jde o multiplatformní software, podporuje jak Linux a širokou škálu Unixových systémů (AIX, Solaris, BSD, MacOSX a další) tak i Windows. Jedná se o svobodný software vydaný pod licencí GNU GPL a je masově užíván jak ve firemních infrastrukturách, tak i v rámci sítí ISP a ICP.

Více informací lze získat na domácí stránce projektu: <http://www.squid-cache.org/>

Apache

Jeden z prvních open-source HTTP serverů, Apache, vznikl v roce 1995. Od verze 1.2 obsahuje ale modul `mod_proxy` jenž umí změnit funkcionalitu HTTP serveru v HTTP proxy. Od verze 2.0 obsahuje navíc i modul `mod_cache`, který poskytuje funkčnost proxy s vyrovnávací pamětí.

Proxy v Apache serveru podporuje mimo HTTP i FTP a HTTPS spojení. Je schopen provozu jak v dopředném, tak v reverzním režimu. V reverzním režimu může být výhodou možnost nasazení zároveň reverzní proxy i zdrojového HTTP serveru v jednom.

Apache je nejpopulárnějším HTTP serverem současnosti. I Apache je svobodný software, vydaný pod Apache licencí, víceplatformní podporující vedle Linuxu a Unixů obecně i MS Windows, MacOSX nebo Novell Netware.

Více informací lze získat na domácí stránce projektu: <http://httpd.apache.org/>

Tinyproxy

Tinyproxy je navržen jako HTTP proxy server s minimálními systémovými nároky. Malá proxy je užitečná pro nasazení v omezených prostředích, jako jsou routery (např. projekt OpenWRT) nebo pro sdílení konektivity (tethering) tzv. chytrých telefonů. Jedná se o software pod licencí GNU GPL a je určen k běhu v unixových operačních systémech.

Více informací lze získat na domácí stránce projektu: <http://www.banu.com/tinyproxy/>

Privoxy

Privoxy je HTTP proxy server navržený primárně pro zvýšení soukromí, bezpečnosti a pohodlí uživatelů. Jedná se o proxy bez vyrovnávací paměti, pracující v dopředném režimu, ovšem s anonymizačními funkcemi a zejména pokročilou filtrací průchozího obsahu. Privoxy umožňuje definovat pravidla, pomocí nichž odstraňuje z průchozího obsahu nežádoucí data, například reklamy, bezpečnostně rizikové skripty a podobně.

Privoxy je uvolněn pod GNU GPL licencí a jedná se o multiplatformní software, funguje mimo unixových systémů i pod MS Windows.

Více informací lze získat na domácí stránce projektu: <http://www.privoxy.org/>

Ziproxy

Ziproxy je specializovaná dopředná HTTP proxy, vytvořená za účelem zrychlení přenosu obsahu. Na rozdíl od běžných proxy však toho nedosahuje vyrovnávací paměti, ale pokročilou filtrací. Je schopna rekomprimovat obrazová data s vyšší mírou ztrátové komprese, optimalizovat (minimalizovat) HTML, JavaScript a CSS soubory.

Ziproxy je vydáván pod licencí GNU GPL a určen pro unixové systémy.

Více informací lze získat na domácí stránce projektu: <http://ziproxy.sourceforge.org/>

Varnish

První verze projektu Varnish vznikla v roce 2006. Jedná se o jednoúčelovou reverzní cachingovou proxy, již od začátku navrženou pouze za účelem fungovat jako akcelerátor pro intenzivně zatěžované, dynamicky generované WWW stránky. Jedná se o škálovatelný systém intenzivně používající vícevláknovou architekturu.

Zvláštností vyrovnávací paměti Varnishe je že veškeré stránky ukládá jen do virtuální paměti, a nechává tak na operačním systému které části budou uloženy na disk (do odkládacího souboru). Účelem tohoto řešení je odstranit kolize mezi využitím paměti pro samotný proces a diskovou vyrovnávací paměť.

Varnish je svobodný software vydávaná pod BSD licencí, určený k provozu na operačních systémech unixového typu.

Více informací lze získat na domácí stránce projektu: <http://www.varnish-cache.org/>

4 SOCKETY

Socket je univerzální komunikační mechanismus. Poprvé se objevil v operačním systému BSD. Socket je velice obecný nástroj, stejné funkce lze používat pro komunikaci pomocí různých protokolů. Socketové rozhraní (API) je shodné pro všechny systémy unixového typu - tj. na všech takových systémech se s sockety pracuje stejně.

Základní myšlenka socketů je rozvedena z unixové filozofie podle které jsou všechny objekty v systému souborem. Aby bylo možné použít tento přístup i pro síťová spojení, je třeba nějak takové soubory vytvářet. Sockety jsou mechanismus který právě to dělá. Vytvořený socket je z hlediska programátora stejným popisovačem otevřeného souboru (filedescriptor) jako kterýkoli jiný otevřený soubor. Z toho důvodu lze i na socketech používat standartní systémová volání pro práci se soubory, jako `read()` nebo `write()`. Specializovaná volání, určená primárně pro práci nad sockety, však nabízejí větší možnosti.

Pro praktické použití socketové komunikace je potřeba tuto adresovat. Vzhledem k univerzálnosti socketů je možné využít je jak pro komunikaci síťové, v praxi zejména nad rodinou protokolů TCP/IP, tak pro komunikaci meziprocesní pomocí tzv. lokálních socketů. Lokální sockety (též unixové nebo unix domain sockets) jsou adresovány jako soubor v souborovém systému, z toho důvodu jsou lokální pro jeden systém a k síťové komunikaci nevhodné. Výhodou je možnost využití identického kódu pro zpracování síťové a meziprocesní komunikace, která je při použití unixových socketů efektivnější (odpadá režie TCP/IP).

Adresace síťových socketů je u TCP/IP sítí dána kombinací IP adresy síťového rozhraní, použitého protokolu transportní vrstvy (TCP, UDP) a čísla portu které identifikuje konkrétního příjemce (proces) na daném rozhraní. Pro adresaci se v C kódu používají struktury `addrinfo`, `sockaddr` a `sockaddr_un`, `sockaddr_in`, `sockaddr6`, `sockaddr_in6` podle toho zda se jedná o unixovou, IPv4 nebo IPv6 adresu. Základní knihovní funkcí pro práci s IP adresami je volání `getaddrinfo()` kterým lze provádět zjišťování i překlad adres. Využití této funkce, zároveň s vhodným kódem, zároveň zajistí kompatibilitu kódu s oběma verzemi IP protokolu.

4.1 Otevření socketu

Před zahájením socketové komunikace, jak na straně serveru (pasivního, čekajícího na připojení), tak na straně klienta (aktivního), je potřeba nejprve vytvořit nový - nepřípojený - socket, jako (nepřípojený) komunikační kanál. Toho se dosáhne voláním `socket()`, které vytvoří a vrátí filedescriptor socketu pro uvedený typ spojení a protokolu.

Dalším krokem pro navázání komunikace je - vždy na straně serveru a v určitých případech i u klienta - připojení socketu k určité místní adrese a portu, na němž má poslouchat (nebo z nějž má odchozí spojení iniciovat u klienta). Pro tento účel slouží volání `bind()`. V unixových systémech může použít porty s číslem menším než 1024 pouze proces s oprávněním superuživatele (root).

Server který chce čekat na spojení po připojení socketu musí ještě inicializovat tzv. backlog čili frontu požadavků, do níž operační systém řadí příchozí spojení před tím než je serverový proces odbaví. Pro tento účel slouží jednoduché volání `listen()` které pro daný socket nastaví velikost fronty a informuje systém aby začal požadavky přijímat. Po naplnění fronty, nejsou-li procesem průběžně odebírány, bude systém další příchozí požadavky odmítat.

Pro vyjmutí příchozího spojení z fronty pro jeho zpracování slouží volání `accept()`. Jedná se o blokující systémové volání, které je-li ve frontě příchozí spojení, vytvoří a vrátí filedeskriptor nového socketu, určeného pouze pro komunikaci s daným klientem. Původní socket na němž server naslouchá příchozím spojením tak zůstává stále otevřený pro příjem a zpracování dalších spojení.

Minimální příklad inicializace TCP/IP serveru:

```
struct sockaddr_storage their_addr;
socklen_t  addr_size;
struct addrinfo hints, *res;
int sockfd, new_fd;
memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;
```

```
getaddrinfo(NULL, „3333“, &hints, &res);  
sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);  
bind(sockfd, res->ai_addr, res->ai_addrlen);  
listen(sockfd, 10);  
addr_size = sizeof their_addr;  
new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &addr_size);
```

Klient vytvoří nové připojení k serveru nad nepřipojeným socketem voláním `connect()` kterému dodá jednak nový socket, a pak adresu ve formě některé ze struktur `sockaddr`.

4.2 Použití socketů

Čtení a zápis (sockety jsou obousměrné komunikační nástroje) je sice možné provádět běžnými funkcemi `read()` a `write()`, pro tento účel jsou ale vhodnější funkce `recv()` a `send()`. Nespojivé (UDP) sockety mohou využít ekvivalentní `recvfrom()` a `sendto()`.

Po ukončení práce se socketem jej uzavře volání `close()`. Je možné i uzavřít socket jen v jednom směru komunikace voláním `shutdown()`.

Z uvedeného popisu práce se sockety na straně serveru je zřejmé, že každý server který má souběžně obsluhovat více než jednoho klienta musí řešit paralelizaci této úlohy, neboť každé připojený klient vytvoří nový socket s nímž je třeba pracovat, stejně tak je třeba zpracovávat hlavní (naslouchající) socket pro další příchozí konexe.

Jak již bylo nastíněno, existují v zásadě dva přístupy k tomuto problému. Prvním je multiplexování - střídavá obsluha více socketů v rámci jednoho programového toku řízení (vlákna nebo procesu). Tento způsob obsluhy je možné realizovat pomocí systémových volání `select()` nebo `poll()`, které pracují nad různým způsobem definovanými množinami filedeskriptorů, umožňují programu vyčkat do výskytu události (data připravena ke čtení, možnost odeslat, chyba apod.) na některém z deskriptorů a poté získat informaci o tom na kterém a jaké události nastaly. Programový kód poté událost zpracuje a vrací se zpět do čekání `select()/poll()`. Výhodou multiplexování je ve většině případů efektivita (alespoň v případech kdy je úzkým hrdlem síťová propustnost), nevýhodou nevyužití případných víceprocesorových systémů a komplikovanější kód. Existují také rozšiřující knihovny (`libevent`) pracující na principu registrace zpětného volání (`callback`) umožňující programátorovi pohodlnější práci s multiplexováním spojení.

Druhým přístupem k problému je zpracování každého jednotlivého socketu v samostatném programovém toku řízení (vlákně, procesu). Tradičně server pracující tímto způsobem má jedno hlavní řídicí vlákno (proces) které čeká na příchozí požadavky na naslouchajícím socketu, a s příchodem každého požadavku vytvoří nové vlákno (proces) které daný požadavek zpracuje. Tato pracovní vlákna nebo procesy mohou být předem vytvořena pro zrychlení reakce za cenu obsažené paměti, jejich počet se může dynamicky přizpůsobovat počtu aktuálních spojení (např. aby stále bylo k dispozici nejméně X a nejvýše Y nevyužitých vláken), mohou zpracovávat postupně více požadavků nebo naopak být vytvářena vždy jen na jeden konkrétní požadavek. Výhodou tohoto přístupu je využití víceprocesorových systémů, relativně přehledný programový kód. Nevýhodou zůstává vyšší paměťová a procesorová náročnost, související s nutností přepínání kontextu mezi vlákny nebo procesy.

V neposlední řadě je možné oba zmíněné přístupy i kombinovat.

5 VLÁKNA

Vlákno je v nejobecnějším smyslu samostatný tok řízení programového kódu. Jde tedy o samostatně vykonávatelnou jednotku, obsahující minimálně ukazatel na následující instrukci. V praxi mají vlákna i vlastní ukazatel na počátek zásobníkové paměti (stack) a další pomocné vlastnosti (například maska obsluhy signálů aj.). Všechny ostatní zdroje všechna vlákna patřící do jednoho procesu sdílí. Tento model zrychluje přepínání řízení mezi vlákny, ale nutí programátora řešit synchronizaci přístupu ke sdíleným prostředkům.

Každý proces je vždy tvořen alespoň jedním vláknem, může jich ale vytvořit více.

Implementace vláken v Linuxu a dalších unixových systémech odpovídá standartu POSIX. Tento standart je implementován v knihovně pthread - POSIX threads.

5.1 Použití vláken

Nové vlákno je spuštěno voláním pthread_create(). Pro spuštění je třeba definovat tzv. startovací funkci vlákna, což je funkce jež bude v novém toku řízení vyvolána.

K ukončení vlákna dojde po ukončení startovací funkce vlákna nebo voláním pthread_exit().

Příklad vícevláknového programu:

```
#include <pthread.h>
#include <stdio.h>
struct char_print_params {
    char character;
    int count;
};
void *char_print(void *parameters) {
    struct char_print_params *p = (struct char_print_params *)
parameters;
    int i;
    for(i = 0; i < p->count; ++i) fputc(p->character, stderr);
    return NULL;
```

```
}  
int main() {  
    pthread_t thread1_id,thread2_id;  
    struct char_print_params thread1_args,thread2_args;  
    thread1_args.character = 'x';  
    thread1_args.count = 30000;  
    pthread_create(&thread1_id, NULL, &char_print, &thread1_args);  
    thread2_args.character = 'o';  
    thread2_args.count = 20000;  
    pthread_create(&thread2_id, NULL, &char_print, &thread2_args);  
}
```

II. PRAKTICKÁ ČÁST

6 PROGRAMÁTORSKÁ DOKUMENTACE

6.1 Implementace HTTP proxy serveru

Praktickou částí práce je vícevláknová implementace HTTP proxy serveru zpracovávajícího příchozí požadavky v samostatných vláknech. Implementace je realizovaná v programovacím jazyce C. Program je určen pro a odladěn v prostředí operačního systému GNU/Linux. Program využívá řadu systémových a knihovnických funkcí, mimo standartní knihovny jazyka C především knihovny pthread pro práci s vlákny a knihovnu rt pro některé funkce související s reálným časem. Obě tyto knihovny jsou součástí běžné instalace systému GNU/Linux a program nevyžaduje žádné další nebo neobvyklé knihovny.

Server realizuje plnou podporu standartu HTTP 1.0, tj. metody GET, POST a HEAD, a dále podporuje podmnožinu standartu HTTP 1.1 tak, aby umožňoval bezproblémovou komunikaci existujícím implementacím klientů a serverů (tj. internetových prohlížečů a WWW serverů). Tato podpora zahrnuje jak standartizované mechanismy jako je např. podpora *ranges* tj. rozsahů umožňující klientům vyžádat si pouze učitou část dokumentu která je hojně využívána při streamování videa, tak i mechanismy nestandardizované jako je např. podpora *cookies* využívaná pro udržování stavové informace v jinak nestavové HTTP komunikaci, bez níž by nebyla většina současných WWW aplikací funkční.

Implementace proxy serveru obsahuje také vlastní HTTP server, kterým se realizuje uživatelské rozhraní proxy serveru. Toto rozhraní dovoluje dohled nad stavem a komunikací proxy prostřednictvím WWW prohlížeče, a obsahuje přehledy celkového stavu serveru, stavu jednotlivých vláken a přehled posledních vyřizovaných HTTP požadavků včetně výsledku jejich vyřízení. Mimo tento přehled proxy server také zaznamenává úplný záznam všech požadavků do logovacích souborů, zvláště pro proxy subsystém a pro HTTP server subsystém.

6.1.1 Paměťové nároky

Server je paměťově nenáročný, pro svůj běh využívá minimum globálně alokované paměti na haldě - z ní jsou největší jednorozměrná pole stavových a statistických informací pro jednotlivá vlákna, a zejména kruhový buffer (zásobník o pevné maximální

velikosti) záznamu posledních požadavků. I ten ale ve výchozím nastavení nepřesahuje velikost 1MB.

Jednotlivá vlákna využívají jednak vlastní zásobníky (v současných verzích GLIBC o pevné velikosti 2MB na vlákno), a dále si dynamicky alokují paměť z haldy dle potřeby. Pro snížení paměťové náročnosti je zprostředkovaná HTTP komunikace mezi klientem a cílovým serverem řešena prokládaně, tj. nedochází k načtení celého požadavku klienta (u požadavků obsahujících HTTP entity jako je POST) ani celé odpovědi serveru do paměti proxy, nýbrž je tato komunikace okamžitě po přijetí přeposlána protistraně. Proxy server se tímto způsobem vyhýbá nutnosti alokovat paměť pro celou, potenciálně objemnou přenášenou entitu, a vystačí si s malou vyrovnávací pamětí, v reálném provozu (v závislosti na specifikách použitého TCP/IP subsystému a síťových podmínkách) okolo 2KB.

Tento mechanismus prokládaného vyřizování požadavků má také pozitivní vliv na rychlost odezvy proxy, neboť klient začne přijímat odpověď dříve než proxy server obdrží celou zprávu.

Celková paměťová náročnost procesu http proxy se tak pohybuje okolo 23MB po startu ve výchozím nastavení (tj. při 10 inicálních vláknech).

6.1.2 Procesorové nároky

Server je efektivní i z hlediska procesorové náročnosti. Při benchmarkingu prováděném nástrojem Apache Benchmark (AB) byla na testovacím systému s procesorem Intel Pentium Dual-Core na frekvenci 3GHz sledováno využití procesoru procesem proxy serveru maximálně do 10%, a to při maximální zátěži dovolené výchozí konfigurací (tj. 100 souběžně běžících vláken). Těto efektivity je v aplikaci dosaženo zejména eliminací tzv. aktivního čekání. Jako u většiny síťové aplikace, i u proxy serveru je hlavním úzkým hrdlem propustnost síťového připojení - běžné procesory jsou schopny tak jednoduchého zpracování dat, jako je v tomto případě jejich identifikace a přeposlání, řádově rychleji než je síť schopna data poskytovat. Většinu času tak aplikace tráví čekáním na příchod dalších dat, případně dalšího spojení atp. Triviálním řešením tohoto čekání je cyklicky kontrolovat, zda-li jsou již nějaká data na socketu k dispozici. Toto řešení má však řadu nedostatků:

- především zbytečné plýtvání procesorovým časem, kdy místo trvalé kontroly může systém zpracovávat jinou úlohu (u víceúlohových systémů) nebo jiné vlákno v rámci úlohy (u vícevláknových aplikací)

- dále obtížné řešení obsluhy více síťových spojení souběžně, byť tento problém je zejména u vícevláknových aplikací odstraněn - nicméně právě v tomto případě by aktivní čekání všech vláken ještě zhoršilo problém zatížení systému.

Proto se místo aktivního čekání užívá technologie blokování, kdy aplikační program zavolá systémové volání např. pro příjem dat ze socketu, a pokud na socketu nejsou data k dispozici, je vykonávání aktuálního toku zablokováno až do příchodu dat, kdy je operačním systémem probuzen a pokračuje ve zpracovávání instrukcí.

V realizovaném proxy serveru jsou všechny operace síťového vstupu a výstupu řešeny jako blokující, a to jak příjem nových spojení v hlavním vlákně procesu, tak čtení požadavků a odesílání odpovědí u připojených socketů v jednotlivých pracovních vláknech.

Při profilování aplikace bylo zjištěno následující reálné využití procesoru: při testu sestávajícím z vyřízení 10.000 požadavků, který trval celkem 48,8 sekund, profilovací nástroj gprof indikoval celkovou kumulativní zátěž aplikace 0,12 sekund, tj. jen asi 2,5% celkového času testu proces vytěžoval procesor. Následující tabulka obsahuje procedury v nichž proces strávil největší množství času.

% času	procedura	popis
16,67	addHeader	přidání nové HTTP hlavičky do požadavku
16,67	parseHeaders	identifikace hlaviček příchozí HTTP zprávy
8,33	recvtimeout	příjem TCP zprávy do bufferu
8,33	destroyEntity	dealokace paměti načtené HTTP entity
8,33	makeHeaders	vytvoření záhlaví nové HTTP zprávy
8,33	parseRequest	kontrola a zpracování obsahu záhlaví HTTP požadavku
8,33	getResponse	přeposlání požadavku na vzdálený HTTP server
8,33	cbInsert	vložení záznamu do kruhového bufferu
8,33	makeLogEntry	vytvoření nového záznamu příchozích požadavků
8,33	fillIn	vyplnění textové šablony

Tabulka 1: Rozdělení procedur podle využití strojového času

Výstup profilovacího nástroje gprof je uveden v příloze 1 této práce.

6.1.3 Výkon proxy serveru

Jelikož primárním úkolem proxy serveru je zprostředkovávat síťovou komunikaci, je jeho maximální rychlost externě omezena šířkou přenosového pásma. Většina koncových připojení stávající síťové infrastruktury je stále realizována prostřednictvím FastEthernetu o maximální propustnosti kolem 10MBps, zatímco současná generace PC má propustnost paměťové sběrnice až 20x vyšší (Intel Pentium 4, 2GHz - 1,6GBps).

Při zmíněném benchmarkingu nástrojem AB byla testována rychlost odezvy a přenosu dat referenčního HTTP serveru, v tomto případě Apache 2.0.52, při opakovaných souběžných požadavcích na odeslání statického obsahu - HTML souboru z lokálního souborového systému počítače na kterém HTTP server běžel. Požadavky byly spouštěny ve paralelně po 20 a nepřetržitě, bez čekání na dokončení zpracování předchozích skupin, až bylo dosaženo maximální povolené saturace tj. 100 souběžně vyřizovaných spojení, od kdy test probíhal ustálenou rychlostí. Pro zjištění maximálního dopadu vložení proxy serveru do HTTP komunikace a eliminaci rušivých faktorů bylo měření prováděno v rámci 1 počítače provozujícího jak HTTP server, tak proxyserver i benchmarkovací software, tj. bez vlivu síťové latence. Použitý byl systém disponoval procesorem Intel Core2Quad 2,4GHz a 8GB operační paměti a používal operační systém GNU/Linux (CentOS 4.9).

Celý výstup měření je součástí přílohy 2 této práce, hlavní výstupy shrnuje tabulka.

	Bez proxy	S proxy serverem	% rozdíl
požadavků celkem	1 000,00	1 000,00	0,00
čas celkem (ms)	198,89	354,96	78,47
požadavků za sekundu	5 027,98	2 817,23	-43,97
dosažený přenos (kB/s)	9 789,48	5 527,40	-43,54
průměrný čas vyřízení (ms)	3,98	7,10	78,39

Tabulka 2: Srovnání rychlosti přístupu s a bez použití proxy serveru

Jak test ukazuje, došlo k přibližně $\frac{3}{4}$ prodloužení času vyřizování požadavku, a ekvivalentnímu snížení přenosové rychlosti. Toto navýšení je do značné míry způsobeno přidanou režii navazování dalšího TCP/IP spojení mezi proxyserverem a HTTP serverem,

a je v řádu jednotek milisekund na požadavek. V reálném nasazení, kdy podmínky komunikace nejsou tak optimalizované a roli hraje i síťová latence, případně vytížení webserveru nebo doba generování obsahu, kdy časy vyřízení požadavku se pohybují o řád výše, je tato přidaná režie proxyserveru zanedbatelná.

Tuto situaci by změnil masový nástup rychlostí gigabytového ethernetu na všech úrovních síťové infrastruktury, nicméně z hlediska proxy serveru je neméně důležitá jako propustnost sítě i propustnost (schopnost generovat a odesílat obsah) samotných zdrojů - HTTP serverů, a podle provedených měření je představovaná implementace proxy serveru řádově stejně rychlá jako běžné HTTP servery (v měřeném případě Apache 2.0). Z tohoto hlediska ani pro gigabytový ethernet nepředstavuje tento proxy server významné zpomalení provozu, zejména při provozu na dostatečně výkonném hardware.

6.2 Architektura systému

Aplikace proxy serveru je postavena s architekturou vícevláknového soketového serveru, a dělení práce mezi vlákny je realizováno podle modelu producer-consumer. Vícevláknový soketový server funguje tak, že jedno z vláken poslouchá na hlavním soketu pro nová příchozí spojení, a po jejich navázání, kdy dojde k vytvoření nového soketu pro komunikaci s právě nově připojenou protistranou, je tento soket předán jinému vláknu pro řešení další komunikace, zatímco původní vlákno dál naslouchá příchozím spojení.

Protože aplikace poskytuje mimo proxy služby i službu HTTP serveru jako uživatelské rozhraní, je potřeba tyto služby oddělit příjmem požadavků na různých TCP portech (8000 pro HTTP a 8080 pro proxy ve výchozím nastavení které lze změnit). Samotné naslouchání na 2 portech však není řešeno samostatnými vlákny, nýbrž je realizováno prostřednictvím monitorování více soketů v rámci jedné blokující operace poll, která čeká na změnu (příchozí spojení) na libovolném ze sledovaných soketů. Toto řešení neubírá na efektivitě aplikace, přijímající vlákno je schopno akceptovat spojení a předat jej k řešení v mnohem kratším čase než potřebuje nové vlákno pro zpracování požadavku - hlavní smyčka naslouchajícího vlákna je krátká a maximálně jednoduchá.

Z hlediska dělení práce mezi vlákny aplikace používá model producer-consumer, tj. takový kdy v tomto případě 1 vlákno vytváří (producer) úlohy (příchozí požadavky) ke zpracování a vkládá je do fronty úloh, a pracovní vlákna která čekají na příchod úlohy do

fronty, poté ji odebírají její zpracování (consumer). Toto zpracování probíhá paralelně pro tolik požadavků, kolik je aktuálně běžících consumer vláken.

Pro zajištění efektivity vyřizování požadavků aplikace obsahuje dynamickou variantu tzv. fondu vláken (thread pool). Thread pool, používaný pro řešení úloh typu producer-consumer, je množina vláken označených jako consumer sloužící ke zpracování úloh, která po spuštění přejdou do své hlavní smyčky a pokud nemohou převzít úlohu z fronty (fronta je prázdná), zablokují se. V momentě vzniku nové úlohy producer - v tomto případě vlákno přijímající nová spojení - signalizuje prostřednictvím synchronizačního mechanismu její vložení do fronty blokováným consumerům, operační systém některého z nich odblokuje a ten úlohu z fronty převezme a zpracuje. Sekvence čekání na novou úlohu, její převzetí a zpracování je pak obsahem hlavní smyčky pracovních vláken - consumerů. Výhodou tohoto řešení proti jednoduššímu vytváření pracovních vláken pouze pro zpracování každého jednotlivého požadavku (a následného ukončení vlákna) je nižší latence - vytvoření úlohy odpadá režie spojená se spuštěním vlákna, neboť pracovní vlákno je již k dispozici.

Dynamická varianta fondu vláken je rozšířena o to, že producer vlákno funguje zároveň jako hlavní vlákno které reguluje velikost thread poolu, a při vzniku množství úloh které přesahuje definovaným způsobem množství vláken v thread poolu jej zvětší a spustí nová pracovní vlákna tak aby byly úlohy vyřizovány co nejrychleji. Toto řešení ovšem není vhodné pro všechny situace, zejména u výpočetně náročných úloh množství vláken vyšší než počet procesorových jader na nichž se mohou vykonávat vede pouze k nárůstu režie přepínání vláken bez zvýšení efektivity výpočtu. Pro síťovou aplikaci jako je proxy server je však tento model ideální, neboť drtivou většinu času všechna vlákna stráví čekáním na síťovou komunikaci.

Aby nedocházelo k hromadění nepotřebného množství vláken po špičkových zátěžích, jsou všechna dodatečně spouštěná vlákna spouštěna s příznakem který způsobí že nedojde-li do definovaného času k probuzení vlákna ke zpracování nové úlohy, je jeho čekání přerušeno časovačem a vlákno ukončí svoji činnost. Tímto způsobem thread pool dynamicky reaguje na změny ve vytiženosti serveru aniž by zbytečně zvyšoval zátěž častým spouštěním vláken nebo systémové zdroje jejich nadbytečným množstvím. Zároveň toto řešení odebírá povinnost kontrolovat nadbytečnost hlavnímu vláknu. Pouze iniciální množství vláken v poolu zůstává bez ohledu na nedostatek úloh.

6.2.1 Hlavní vlákno

Po spuštění aplikace je nejprve provedeno zpracování parametřů příkazové řádky, a není-li vyžadována jiná funkčnost jako zobrazení nápovědy je následně spuštěna funkce `serverStart()`. Ta provede inicializaci serveru funkcí `initServer()` s příslušnými parametry, buď z příkazové řádky nebo s výchozími hodnotami.

Funkce `initServer()` provede alokaci globálních dat, otevře logové soubory, inicializuje kruhový buffer záznamu požadavků a všechny vláknové synchronizační struktury. Pokud dojde k chybě, je tato indikována uživateli a program končí.

Po úspěšné inicializaci následuje spuštění iniciálního množství vláken funkcí `startThreads()`, a to s příznakem bránícím jejich automatickému ukončení při nečinnosti.

Dalším krokem je otevření hlavních soketů pro příjem spojení, a to na definovaných portech pro proxy a HTTP funkcí `openListener()`. Není-li některý z těchto kroků úspěšný, je chyba opět indikována program končí. Následně hlavní vlákno vchází do své hlavní smyčky, ve které stráví celý zbytek (normálního) běhu aplikace.

Hlavní smyčka začíná čekáním na nastavení semaforu `connection_sem`, který slouží k omezení maximálního množství nezpracovávaných otevřených TCP spojení. Při inicializaci aplikace je tento semafor nastaven na hodnotu desetkrát vyšší, než je maximální množství pracovních vláken. Při otevření nového spojení hlavní vlákno semafor sníží operací `sem_wait()`, a po ukončení zpracování každého požadavku naopak pracovní vlákno hodnotu zvýší operací `sem_post()`. Smyslem tohoto semaforu je tedy omezit maximální počet čekajících otevřených soketů, nicméně k jeho úplnému "položení" tedy vynulování které by zablokovalo příjem nových spojení by bylo třeba aby setrvale delší dobu docházelo k příchodu požadavků rychlostí vyšší než jsou pracovní vlákna schopna zpracovávat. V takovém případě server bude nová spojení odmítat, takovou situaci lze řešit zvýšením maximálního povoleného množství vláken.

Je-li snížení semaforu úspěšné, hlavní vlákno se zablokuje na operaci `poll()` kde čeká dokud na některém z hlavních soketů není nový požadavek na spojení. Pak vytvoří datovou strukturu `task` pro novou úlohu, voláním `accept()` požadavek přijme a soket nového připojení vloží do struktury. Následně vlákno zjistí aktuální hodnotu semaforu `jobqueue_sem` (operací `sem_getvalue()`) který indikuje počet úloh čekajících ve frontě, uzamkne mutexu `thread_mutex` který hlídá přístup k informacím o spuštěných vláknech,

a pokud zjistí že fronta stále obsahuje nevyřízené úlohy, spustí tolik dočasných vláken kolik úloh čeká. Operace `sem_getvalue()` sice nemusí dávat zcela přesné výsledky neboť není synchronizována, to ale není na závadu - bude-li náhodou spuštěno zbytečné vlákno, může zpracovat aktuálně přidávanou úlohu a následně se ukončit.

Následně hlavní vlákno přidá novou úlohu do fronty - fronta a operace s ní jsou synchronizovány mutexem `jobqueue_mutex`. Poslední operací je zvednutí semaforu `jobqueue_sem` o jedničku, a smyčka se opakuje.

6.2.2 Vyřizování požadavků

Startovním bodem pracovních vláken je funkce `handleClient()` který obsahuje jejich hlavní smyčku. Po spuštění vlákna je z argumentu - struktury `task` - získáno číslo vlákna a příznak zda-li jde o vlákno trvalé nebo dočasné. Následně je spuštěna hlavní smyčka, která ihned provede operaci čekání na semafor `jobqueue_sem`. Jeho iniciální hodnotou je nula, proto trvalá vlákna spuštěná při startu aplikace okamžitě blokují a čekají na příchod úlohy, kdy hlavní vlákno hodnotu semaforu zvýší. Rozdíl u dočasných vláken je v tom že provedou čekání operací `sem_timedwait()`, která není-li úspěšná do nastaveného času, je ukončena což vlákno identifikuje, zamkne mutex `thread_mutex`, zruší informaci o své existenci a ukončí se.

Po úspěšném snížení semaforu pracovní vlákno zamkne přístup k frontě úloh mutexem `jobqueue_mutex` a odebere první čekající úlohu. Následně provede inicializaci zpracování - vyplní strukturu `request` časem převzetí požadavku a adresou protistrany socketu, a podle příznaku uloženého ve struktuře `task` hlavním vláknem vyvolá zpracování požadavku příslušnou funkcí - `handleProxyClient()` nebo `handleWebClient()`.

Je-li požadavek úspěšně zpracován, vrátí se ze zpracovatelské funkce datová struktura `logentry` s informacemi o vyřízení požadavku. Tato data jsou funkcí `writeLog()` zanesena do příslušného logovacího souboru, kruhového bufferu posledních požadavků a připočítána ke statistikám aktuálního vlákna a celého serveru.

Poslední činností je uzavření klientského socketu a zvednutí semaforu `connection_sem` značící vyřízení dalšího požadavku.

6.2.3 Vyřizování proxy požadavků

Funkce `handleProxyClient()` načte hlavičku příchozího požadavku - až po sekvenci `<CR><LF><CR><LF>` která ji odděluje od těla HTTP entity - ze soketu funkcí `recvHeader()`. Pokud je načtení úspěšné, je zavolána funkce `parseRequest()` které jsou předána načtená data a struktura `request`. V ní je dekodován požadavek na volanou metodu, požadované URL apod. a dále jsou identifikovány a do struktury `request` uloženy HTTP hlavičky požadavku. Není-li indikována chyba, předá se dekodovaný požadavek spolu s prázdnou strukturou `response` pro uložení odpovědi funkci `getResponse()` která provede první polovinu prokládaného předávání požadavků. Nejprve vytvoří nové záhlaví HTTP požadavku podle dat ze struktury `request` voláním `makeRequest()`, otevře soketové spojení na cílový server voláním `makeConnection()` a záhlaví do něj odešle. Následuje prokládané předání obsahu těla původního klientského požadavku na cílový server funkcí `passThrough()`, která cyklicky odebírá přijímané pakety z klientského soketu a odesílá je na cílový server. Pokud odeslání proběhne v pořádku, funkce `getResponse()` načte záhlaví odpovědi z cílového serveru dalším voláním `recvHeader()` a její zpracování funkcí `parseResponse()`. Je-li odpověď v pořádku, vrací se do `handleProxyClient()` soket na cílový server a jsou vyplněny údaje o odpovědi ve struktuře `response`.

Následně je vytvořeno záhlaví HTTP odpovědi podle dat z přijaté odpovědi serveru voláním `makeResponse()` a odesláno do klientského soketu. Pokud proběhne v pořádku a nedošlo např. k přerušení spojení stiskem tlačítka stop prohlížeče, je provedena druhá polovina prokládaného předání a to předání těla odpovědi z cílového serveru klientovi dalším voláním `passThrough()`.

Došlo-li v průběhu zpracování před finálním odesláním k chybě, je zrušen obsah struktury odpovědi `response` a klientovi je namísto toho odeslána chybová stránka voláním `sendWebResponse()` z HTTP subsystému aplikace. Nedojde-li k chybě alespoň nyní, je vytvořena struktura `logentry` pro záznam do logu a statistik a vrácena do hlavní smyčky vlákna.

6.2.4 Vyřizování HTTP požadavků

I funkce `handleProxyClient()` načte hlavičku příchozího požadavku ze soketu funkcí `recvHeader()`. Pokud je načtení úspěšné, je také zavolána funkce `parseRequest()` které

jsou předána načtená data a struktura request pro dekódování požadavku. Je-li požadavek v pořádku, musí se u HTTP požadavku který není určený k předání nýbrž k vyřízení v rámci aplikace před zpracováním celý načíst a to voláním `recvBody()`. V této verzi aplikace negeneruje žádné odpovědi, které by s tělem požadavku pracovali, nicméně je na takovou možnost připravena. Po úspěšném načtení celého požadavku je tento předán spolu s klientským soketem, prázdnou strukturou response a URL cestou požadavku namapovanou voláním `matchPath()` na některý z dostupných zdrojů interního HTTP serveru funkci `sendWebResponse()`. Ta použije funkci `getWebResponse()` k vygenerování těla odpovědi do struktury response. Vygenerování odpovědi probíhá nejprve identifikací namapovaného zdroje, který odpovídá některé z funkcí pro generování obsahu

- `createHtmlPage()` pro vytvoření některé ze zabudovaných HTML stránek
- `createFixedPage()` pro vytvoření některého ne-HTML zabudovaného souboru
- `createStatsPage()` pro vytvoření dynamicky generované stránky s aktuální statistikou
- `createErrorPage()` pro vytvoření některé chybové stránky

Není-li identifikován existující zdroj, je vytvořena chybová stránka odpovídající chyby HTTP protokolu tj. 404.

Po identifikaci obsahu je tento vygenerování a naplněn do struktury response a funkce `sendWebResponse()` pro něj vytvoří odpovídající HTTP záhlaví voláním `makeResponse()`, které spolu s obsahem samotným odešle klientovi.

Informace o úspěšnosti odeslání se vrací do `handleWebClient()`, kde pokud došlo k chybě ještě před odesláním odpovědi - tj. při zpracování požadavku nebo generování obsahu - je opětovně voláno `sendWebResponse()` tentokrát pro odeslání příslušné chybové stránky, a je-li alespoň tato operace úspěšná, je vytvořena a do hlavní smyčky vláken vrácena struktura logentry.

6.2.5 Logování a statistika

Proxy server shromažďuje statistické informace svém provozu a ukládá záznam o realizovaných požadavcích do logovacích souboru a paměťového bufferu který slouží pro generování WWW stránky s přehledem provozu.

Statistické informace obsahují součty celkového počtu požadavků, objemu přenesených dat a času který provoz trval, a dále detailní informace o posledním zpracovávaném požadavku. Informace pro jednotlivá vlákna jsou ukládány v datové struktuře threadstat globálního pole stats, indexovaného interními pořadovými čísli jednotlivých vláken. Přístup do pole je synchronizován stejně indexovaným polem mutexů z důvodů generování statistiky - pro zápis údajů přistupuje každé vlákno pouze ke svému záznamu, ale pro generování statistiky musí příslušné vlákno načíst všechny položky.

Stejně informace jsou shromažďovány také celkově pro celou aplikaci, ve stejné struktuře umístěné v globální proměnné totalstats. Přístup k této struktuře je prováděn pouze současně s přístupem k bufferu požadavků, a proto není třeba další synchronizace.

Všechny záznamy o požadavcích jsou zapisovány do logovacích souborů - zvlášť pro WWW a zvlášť pro proxy požadavky. Také tento přístup je synchronizován souběžně s bufferem požadavků.

Buffer požadavků je zásobník o pevné, při startu určené velikosti, ve kterém jsou nejstarší záznamy průběžně nahrazovány. Zásobník obsahuje struktury logentry, které obsahují detailní informace o požadavku - jeho typu, zdroji, cíli a výsledku. Pro synchronizaci bufferu je použit read-write zámek, uložený v globální proměnné logbuffer_lock. Využití read-write zámku je výhodné z důvodu odlišného přístupu k bufferu - při přidávání nového záznamu musí být přístup exkluzivní, je proto vyžadován zámek pro zápis, zatímco při samotném generování statistik je možné souběžně číst více vláken - získáním neexkluzivního zámku pro čtení, za předpokladu neexistence zamknutí pro zápis.

Pro ukládání informací slouží funkce writeLog(), která pro dané číslo vlákna zamkne jeho statistický záznam, aktualizuje jej na základě předané struktury logentry, následně provede zamčení read-write zámku logbuffer_lock pro zápis a strukturu logentry do něj vloží. Před uvolněním zámku ještě provede zápis záznamu do logovacích souborů a aktualizaci celkových statistik aplikace.

Čtení informací z bufferu a jejich prezentaci provádí funkce generateStats(). Ta využívá šablonovací subsystém aplikace k vygenerování HTML stránky s informacemi z bufferu, z pole statistik vláken a globální statistiky.

6.2.6 Šablonovací subsystém

Pro účel generování HTML stránek uživatelského rozhraní aplikace obsahuje subsystém vyplňování textových šablon konkrétními údaji. Tento systém je používán rekurzivně, tak že například

- v 1. kroku je šablona řádku tabulky vyplněna konkrétními údaji, a to opakovaně pro každý řádek tabulky, které jsou umístovány za sebe
- ve 2. kroku je vyplněná šablona těla tabulky použita jako proměnná spolu s dalšími údaji pro vyplnění šablony těla stránky, kde je umístěna na příslušné místo
- ve 3. kroku je vyplněná šablona těla stránky použita spolu s dalšími proměnnými pro vyplnění šablony celé stránky, obsahující např. společnou hlavičku a patičku všech stránek.

Samotné šablony jsou umístěny v hlavičkovém souboru zdrojových kódů a po přeložení se stávají součástí aplikace. Šablony jsou prosté texty, které mají na místech určených pro vyplnění zástupné řetězce ve tvaru {název} kde název je identifikace proměnné která má být na dané místo vložena.

Vyplňování šablon provádí funkce fillNew() která vyplní předanou šablonu předanými proměnnými do nově alokovaného paměťového prostoru, a funkce fillAppend() která vyplněnou šablonu přidá na konec předaného paměťového prostoru. Funkce fillAppend() je určena pro řetězení šablon v cyklu při výpisu neskalárních hodnot. Tato funkce si zajišťuje i případné nutné zvětšení alokovaného prostoru pro výsledek vyplňování šablony.

Hodnoty které mají tyto funkce použít pro vyplňování jsou předávány jako prosté parametry (funkce využívají funkcionalitu proměnného počtu parametrů). Samotné názvy a typy předávaných parametrů ale je nutno předat jako odkaz na pole struktur typu templatevar.

6.3 Programové celky

- circbuffer** - obsahuje implementaci kruhového bufferu - zásobníku pevné velikosti.
- constants** - obsahuje definice výchozích parametrů a konstanty užívané v HTTP
- entity** - obsahuje deklarace základních datových struktur a operace s nimi

httpin	- obsahuje metody pro zpracování příchozí HTTP komunikace
httpout	- obsahuje metody pro vytváření odchozích HTTP zpráv
inet	- obsahuje základní metody TCP komunikace
main	- obsahuje vstupní body aplikace
pages	- obsahuje šablony zabudovaných HTML stránek
proxy	- obsahuje metody specifické pro vyřizování proxy požadavků
server	- obsahuje metody s hlavními smyčkami jednotlivých vláken
template	- obsahuje šablonovací subsystém aplikace
web	- obsahuje metody specifické pro vyřizování HTTP požadavků

6.4 Globální proměnné

6.4.1 Soubor main.c

extern int debug - příznak nastavený při zpracování parametrů příkazové řádky, při nastavení aplikace vypisuje při nekritických chybách (zejména v komunikaci) ladící informace na konzoli.

6.4.2 Soubor server.c

struct task *jobqueue - fronta úloh ke zpracování

sem_t jobqueue_sem - semafor indikující počet přítomných úloh ve frontě, zároveň provádí blokování nevyužitých pracovních vláken

sem_t connection_sem - semafor indikující počet spojení která je ještě možné otevřít bez dokončení zpracování další úlohy

pthread_mutex_t jobqueue_mutex - mutex synchronizující přístup k frontě úloh

pthread_rwlock_t logbuffer_lock - read-write zámek s synchronizující přístup k bufferu

struct circularbuffer logbuffer - fixní zásobník pro informace o posledních zpracovaných požadavcích

int proxylog - filedescriptor logovacího souboru proxy subsystému

int weblog - filedescriptor logovacího souboru HTTP server subsystému

pthread_mutex_t *stats_mutex - pole mutexů synchronizující přístup ke statistikám vláken

struct threadstat *stats - pole struktur se statistikami vláken

struct threadstat totalstats - globální statistické informace aplikace

pthread_attr_t thread_attr - inicializační atributy vláken

pthread_mutex_t thread_mutex - mutex synchronizující přístup k frontě úloh

pthread_t *threads - pole identifikátorů aktivních pracovních vláken

int activethreads - počet právě běžících pracovních vláken

int maxactivethreads - nejvyšší počet aktivních pracovních vláken za běh aplikace

int maxthreads - maximální povolený počet pracovních vláken

6.4.3 Soubor constants.c

extern const char all_pages[][] - URL-cesty zabudovaných zdrojů HTTP serveru

extern const char requesttypes[][] - názvy podporovaných služeb

extern const int statuscodes[] - hodnoty stavových kódů HTTP protokolu

extern const char statustexts[][] - názvy stavových kódů HTTP protokolu

extern const char methods[][] - názvy podporovaných metod HTTP protokolu

extern const char contenttypes[][] - MIME-typy zabudovaných zdrojů HTTP serveru

extern const char all_headers[][] - názvy podporovaných hlaviček HTTP protokolu

7 UŽIVATELSKÁ DOKUMENTACE

7.1 Instalace

Aplikace je distribuována ve formě zdrojového kódu v archivu **mthps.tar.gz**. Pro její spuštění je třeba aplikaci přeložit. Podporovaný operační systém je GNU/Linux 2.6, pro překlad aplikace je potřeba nainstalované základní vývojové prostředí, tj. minimálně překladač gcc jazyka C a vývojový balík knihovny jazyka C Glibc-devel. Podporovány jsou verze GCC od 3.4 a Glibc od verze 2.3.

V prvním kroku je třeba rozbalit archiv aplikace a přejít do jeho adresáře např. příkazem

```
tar xzf mthps.tar.gz; cd mthps
```

V druhém kroku je potřeba spustit proces překladu příkazem

```
make
```

Po dokončení překladu je v aktuálním adresáři vytvořen soubor mthps který je binárním souborem aplikace.

Nyní je možné aplikaci spustit, nebo ji přesunout do příslušného adresáře, buď ručně nebo do adresáře /usr/local/bin příkazem

```
make install
```

Pro úspěšné provedení je potřeba mít oprávnění k zápisu do tohoto adresáře.

Pro zajištění automatického spouštění je potřeba toto zajistit příslušnými nástroji používané distribuce systému GNU/Linux. Pokud takové nástroje nejsou k dispozici, lze spouštění zajistit provedením příkazu

```
echo /usr/local/bin/mthps -l/var/log/proxy.log\  
-L/var/log/web.log -d >> /etc/rc.local
```

opět za předpokladu že jsou k dispozici práva k zápisu do souboru /etc/rc.local.

7.2 Spuštění aplikace

Při spuštění aplikace bez dalších voleb dojde ke spuštění s výchozími hodnotami volitelných parametrů, a aplikace zůstane běžet na popředí. Toho je možné využít k testování, neboť ji lze snadno ukončit stisknutím kláves CTRL+C.

Další možnosti nabízí využití argumentů příkazového řádku. Všechny možné volby, spolu s výchozími hodnotami volitelných parametrů, lze získat použitím volby -h:

```
[trifid@xara mthps]$ ./mthps -h
MultiThreaded HTTP Proxy Server Copyright (c) 2011-2012 Ondrej Dolezal
Usage: ./mthps
      [ -h|d|D ] [ -a address ] [ -p proxyport ] [ -P webport ]
      [ -l proxylog ] [ -L weblog ] [ -s logsize ]
      [ -t starthreads ] [ -T maxthreads ] [ -o timeout ]
-h    give this help
-d    detach from console and run in the background
-D    enable debugging output
-a    bind to this hostname or IP adress, default any
-p    port to listen on for proxy requests, default '8080'
-P    port to listen on for web requests, default '8000'
-l    proxy requests logfile pathname, default 'proxy.log'
-L    web requests logfile pathname, default 'web.log'
-s    store this many log entries in memory, default '1000'
-t    initially start this many threads, default '10'
-T    maximum number of threads, default '100'
-o    thread inactivity timeout, default '60' sec
[trifid@xara mthps]$
```

Obrázek 1: Nápověda aplikace na příkazovém řádku

Jednotlivé volby příkazového řádku jsou:

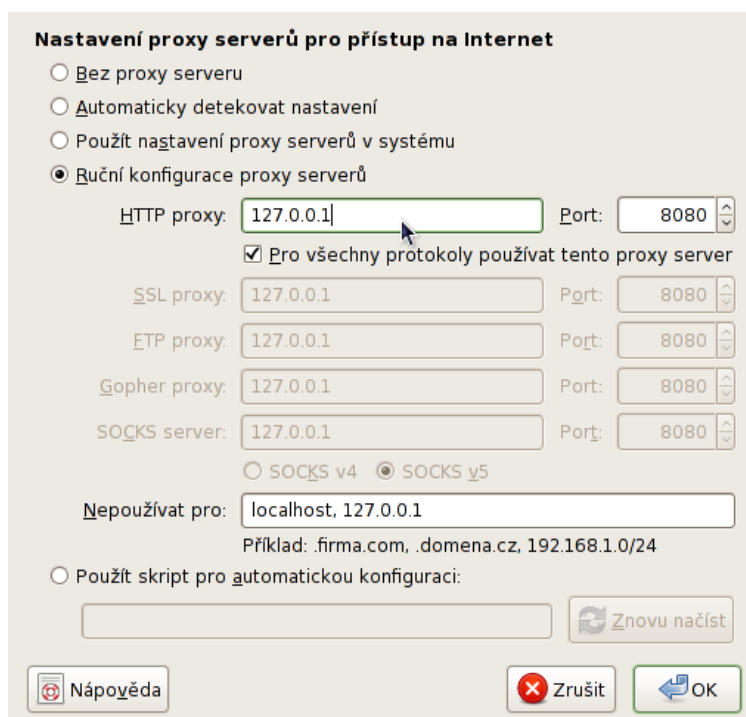
- h zobrazí všechny volby
- d po spuštění přejde do pozadí, tato volba přebíjí volbu -D
- D zobrazuje ladící hlášky (je potlačena volbou -d)
- a bude očekávat příchozí spojení na této adrese (výchozí: jakákoli)
- p použije tento port pro příchozí proxy požadavky (výchozí: 8080)
- P použije tento port pro příchozí web požadavky (výchozí: 8000)
- l použije tuto cestu pro soubor logu proxy požadavků (výchozí: proxy.log)
- L použije tuto cestu pro soubor logu web požadavků (výchozí: web.log)
- s bude ukládat a ve statistice zobrazovat tolik posledních záznamů (1000)
- t po startu spustí tolik permanentních vláken (výchozí: 10)
- T nastaví maximální počet pracovních vláken (výchozí: 100)
- o nastaví dobu po které se ukončí neaktivní dočasná vlákna (výchozí: 60s)

7.3 Použití proxyserveru

Pro použití proxy serveru je potřeba zajistit aby používaný internetový prohlížeč tento využíval. Ve většině současných prohlížečů toho lze docílit prostřednictvím nastavení aplikace, např. pro Mozilla Firefox volbami:

Úpravy -> Předvolby -> Rozšířené -> Síť -> Nastavení připojení

V zobrazeném dialogu je potřeba zvolit *Ruční konfiguraci proxy serverů*, a do políčka pro HTTP proxy uvést adresu na které proxy server běží a do políčka *Port* číslo portu na kterém naslouchá příchozím požadavkům (8080 ve výchozím nastavení).



Obrázek 2: Nastavení proxy serveru prohlížeče Firefox

7.4 Uživatelské rozhraní

Pro přístup k uživatelskému rozhraní aplikace je potřeba internetový prohlížeč. Po jeho spuštění je v něm potřeba otevřít adresu na které proxy běží, na portu na kterém naslouchá web požadavků. Ve výchozím nastavení, po spuštění na lokální stroji, je to např. adresa **http://127.0.0.1/8000** .

Po načtení stránky je zobrazeno hlavní menu obsahující identifikaci aplikace a rozcestník na jednotlivé podstránky:

- **About this software:** zobrazí stručnou informaci o softwaru a autorovi
- **User manual:** zobrazí přehled parametrů pro spuštění aplikace
- **Usage statistics:** zobrazí stránku uživatelského rozhraní pro sledování provozu proxy.

7.4.1 Uživatelské rozhraní pro sledování provozu

Na této stránce je možné zjistit údaje o provozu aplikace. Stránka sestává ze 3 tabulek.

1. Tabulka celkového provozu serveru - **Server totals**

V této tabulce je zobrazen počet aktuálně běžících pracovních vláken - **Active threads**,

- počet dosud nezpracovávaných úkolů ve frontě - **Queued jobs**,
- maximální množství pracovních vláken vytvořené za dobu běhu – **Max.concurrency**,
- celkové množství zpracovaných požadavků – **Total requests**,
- celkové množství přenesených dat v kilobytech - **Total kB**,
- celkový čas potřebný k uspokojení všech požadavků v sekundách - **Total time**.

The screenshot displays the 'Usage statistics' page with three main sections:

Usage statistics

Server totals

Active threads	Queued jobs	Max.concurrency	Total requests	Total kB	Total time (s)
10	0	10	214	1123	116

Thread states

Id	Total requests	Total kBytes	Total time (s)	Last request
1	19	201	22	sm3.sitemeter.com/meter.asp?site=sm3saturdaymorning&refer=&ip=91.228.45.6&w=1280&h=1024&clr=24&utmwv=5.3.1&utms=3&utm=1313614791&utmhn=www.macguff.fr&utmcs=UTF-8&
2	26	190	3	www.google-analytics.com/___utm.gif?utmwv=5.3.1&utms=3&utm=1313614791&utmhn=www.macguff.fr&utmcs=UTF-8&
3	22	140	13	www.google-analytics.com/___utm.gif?utmwv=1.4&utm=1743616203&utmcs=UTF-8&utmsr=1280x1024&utmsc=24-bit&u
4	19	104	11	i3.ytimg.com/crossdomain.xml
5	24	116	12	maps.google.com/maps/api/jsv2/AuthenticationService.Authenticate?1shttp%3A%2F%2Fwww.macguff.fr
6	18	127	12	pinit-cdn.pinterest.com/images/pinit6.png
7	23	49	8	www.google-analytics.com/___utm.gif?utmwv=5.3.1&utms=2&utm=267819095&utmhn=www.macguff.fr&utmcs=UTF-

Request log

Type	Date	Client IP	Method	Last request	Result	kBytes
web	2012/05/20 19:25:17	127.0.0.1	GET	127.0.0.1/stats	200	2
web	2012/05/20 19:25:17	127.0.0.1	GET	127.0.0.1/style	200	1

Obrázek 3: Uživatelské rozhraní statistiky aplikace

2. Tabulka aktivních pracovních vláken - **Thread states**

Tato tabulka obsahuje řádek pro každé v současnosti běžící pracovní vlákno aplikace.

Pro každé je zobrazeno:

- pořadové číslo - **Id**,
- celkové množství zpracovaných požadavků – **Total requests**,
- celkové množství přenesených dat v kilobytech - **Total kB**,
- celkový čas potřebný k uspokojení všech požadavků v sekundách - **Total time**,
- URL-adresa posledního zpracovaného požadavku - **Last request**.

3. Tabulka přehledu poslední požadavků - **Request log**

Tato tabulka obsahuje v jednotlivých řádcích všechny záznamy z paměti posledních požadavků:

- typ požadavku, web nebo proxy - **Type**,
- datum a čas přijetí požadavku – **Date**,
- IP adresu klienta - **Client IP**,
- metoda požadavku (GET,POST,HEAD) - **Method**,
- URL-adresa požadavku – **Request**.
- výsledný HTTP stavový kód požadavku – **Result**,
- velikost přenesených dat - **kBytes**.

ZÁVĚR

Jak tato práce ukázala, HTTP proxy server v prostředí internetu je užitečný software, který může zrychlit přístup ke zdrojům, provádět filtrování nebo sledování provozu nebo zajistit anonymitu uživatelů. K nejrozšířenějším HTTP proxy serverům v dnešní době patří zejména projekt Squid.

Dále byly diskutovány nezbytné technologie potřebné k tvorbě moderního HTTP proxy serveru - sockety a vlákna. S použitím těchto technologií byl vytvořen vícevláknový HTTP proxy server v jazyce C pro platformu GNU/Linux, který obsahuje také zabudovaný HTTP server pro přístup k provozním informacím a statistice serveru.

Zkoušky tohoto serveru potvrdily že vícevláknový návrh je velmi vhodný pro použití u serverových aplikací, a vytvořený proxy server při použití neznamená zásadní zátěž systému na kterém je provozován ani významné zpomalení WWW provozu.

V podobě jaké je proxy server prezentován, je využitelný například pro sledování WWW provozu v lokální síti, ale s dalším vývojem je možné spektrum užití relativně snadno a významně rozšířit o aplikace jako je filtrování obsahu nebo omezování přístupu ke zdrojům.

ZÁVĚR V ANGLIČTINĚ

This work proved that a HTTP proxy server is an useful piece of software. It can be used to accelerate access to resources, perform access control or traffic logging. It can be also used to enhance anonymity of its users. One of the most used HTTP proxy servers nowadays is the Squid proxy cache.

Further discussion was targeted towards technologies necessary for building current HTTP proxies - sockets and threads. A multi-threaded HTTP proxy server in C was created using discussed technologies which incorporates an embedded HTTP server used to access the runtime information and usage statistics. Target platform was GNU/Linux.

Further testing of this server proved that a multi-threaded design is very useful for server applications, and that the created proxy server neither uses significant amounts of system resources nor it degrades the WWW performance in an important way.

In its current state, the proxy server can be used to monitor the WWW traffic of a local area network but with some further development its uses can include such areas as content filtering or access control.

SEZNAM POUŽITÉ LITERATURY

- [1] RFC 791. Internet Protocol. [s.l.] : IETF, 1981. 45 s. Dostupné z WWW: <http://tools.ietf.org/html/rfc791>.
- [2] RFC 793. Transmission Control Protocol. [s.l.] : IETF, 1981. 85 s. Dostupné z WWW: <http://tools.ietf.org/html/rfc793>.
- [3] RFC 1945. Hypertext Transfer Protocol - HTTP/1.0. [s.l.] : IETF, 1996. 60 s. Dostupné z WWW: <http://tools.ietf.org/html/rfc1945>.
- [4] RFC 2616. Hypertext Transfer Protocol - HTTP/1.1. [s.l.] : IETF, 1999. 176 s. Dostupné z WWW: <http://tools.ietf.org/html/rfc2616>.
- [5] RFC 2617. HTTP Authentication: Basic and Digest Access Authentication. [s.l.] : IETF, 1999. 34 s. Dostupné z WWW: <http://tools.ietf.org/html/rfc2617>.
- [6] RFC 3143. Known HTTP Proxy/Caching Problems. [s.l.] : IETF, 2001. 32 s. Dostupné z WWW: <http://tools.ietf.org/html/rfc3143>.
- [7] GAY, Warren. Linux Socket Programming by Example. [s.l.] : Que, 2000. 557 s. ISBN 0789722410.

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

HTTP	Hyper text transfer protocol
WWW	World wide web
TCP	Transport control protocol
UDP	User datagram protocol
IP	Internet protocol
HTML	Hypertext markup language
MIME	Multipurpose Internet Mail Extensions
FTP	File transfer protocol
API	Application programming interface
SSL	Secure socket layer
TLS	Transport layer security
URL	Uniform resource locator
POSIX	Portable Operating System Interface for Unix
GPL	General public licence

SEZNAM ILUSTRACÍ

Obrázek 1: Nápověda aplikace na příkazovém řádku.....	43
Obrázek 2: Nastavení proxy serveru prohlížeče Firefox.....	44
Obrázek 3: Uživatelské rozhraní statistiky aplikace.....	45

SEZNAM TABULEK

Tabulka 1: Rozdělení procedur podle využití strojového času.....	30
Tabulka 2: Srovnání rychlosti přístupu s a bez použití proxy serveru.....	31

SEZNAM PŘÍLOH

- [1] Příloha P I: Výstup profilování aplikace
- [2] Příloha P II: Výstup benchmarkingu

PŘÍLOHA P I: VÝSTUP PROFILOVÁNÍ APLIKACE

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
16.67	0.02	0.02	139248	0.00	0.00	addHeader
16.67	0.04	0.02	19780	0.00	0.00	parseHeaders
8.33	0.05	0.01	38415	0.00	0.00	recvtimeout
8.33	0.06	0.01	20155	0.00	0.00	destroyEntity
8.33	0.07	0.01	19972	0.00	0.00	makeHeaders
8.33	0.08	0.01	10000	0.00	0.00	parseRequest
8.33	0.09	0.01	9992	0.00	0.00	getResponse
8.33	0.10	0.01	9981	0.00	0.00	cbInsert
8.33	0.11	0.01	9981	0.00	0.00	makeLogEntry
8.33	0.12	0.01	2473	0.00	0.00	fillIn
0.00	0.12	0.00	38598	0.00	0.00	sendall
0.00	0.12	0.00	30965	0.00	0.00	setHeader
0.00	0.12	0.00	19961	0.00	0.00	writeStats
0.00	0.12	0.00	19785	0.00	0.00	recvHeader
0.00	0.12	0.00	19567	0.00	0.00	passThrough
0.00	0.12	0.00	10174	0.00	0.00	destroyResponse
0.00	0.12	0.00	10008	0.00	0.00	get_in_addr
0.00	0.12	0.00	10001	0.00	0.00	sprintPeerAddr
0.00	0.12	0.00	9993	0.00	0.01	handleProxyClient
0.00	0.12	0.00	9993	0.00	0.00	makeRequest
0.00	0.12	0.00	9993	0.00	0.00	parseURL
0.00	0.12	0.00	9989	0.00	0.00	makeConnection
0.00	0.12	0.00	9981	0.00	0.00	makeResponse
0.00	0.12	0.00	9981	0.00	0.00	writeLog
0.00	0.12	0.00	9972	0.00	0.00	writeProxyLog
0.00	0.12	0.00	9781	0.00	0.00	parseResponse
0.00	0.12	0.00	2925	0.00	0.00	cbNext
0.00	0.12	0.00	2275	0.00	0.00	fillAppend
0.00	0.12	0.00	200	0.00	0.00	fillHeaders
0.00	0.12	0.00	200	0.00	0.05	getWebResponse
0.00	0.12	0.00	200	0.00	0.05	sendWebResponse
0.00	0.12	0.00	198	0.00	0.00	createHtmlPage
0.00	0.12	0.00	198	0.00	0.00	fillNew
0.00	0.12	0.00	196	0.00	0.01	createErrorPage
0.00	0.12	0.00	196	0.00	0.00	errorBody
0.00	0.12	0.00	24	0.00	0.00	startThreads
0.00	0.12	0.00	7	0.00	0.06	handleWebClient
0.00	0.12	0.00	7	0.00	0.00	matchPath
0.00	0.12	0.00	7	0.00	0.00	recvBody
0.00	0.12	0.00	7	0.00	0.00	sprintSockAddr
0.00	0.12	0.00	7	0.00	0.00	writeWebLog
0.00	0.12	0.00	4	0.00	0.00	pageBody
0.00	0.12	0.00	2	0.00	0.00	cbEmpty
0.00	0.12	0.00	2	0.00	0.00	createFixedPage
0.00	0.12	0.00	2	0.00	4.21	createStatsPage
0.00	0.12	0.00	2	0.00	4.20	generateStats
0.00	0.12	0.00	2	0.00	0.00	openListener
0.00	0.12	0.00	2	0.00	0.00	pageCLength
0.00	0.12	0.00	2	0.00	0.00	pageCType
0.00	0.12	0.00	2	0.00	0.00	pageTitle
0.00	0.12	0.00	1	0.00	0.00	cbDestroy
0.00	0.12	0.00	1	0.00	0.00	cbInit
0.00	0.12	0.00	1	0.00	0.00	deinitServer
0.00	0.12	0.00	1	0.00	0.00	initServer
0.00	0.12	0.00	1	0.00	0.00	serverStart

PŘÍLOHA P II: VÝSTUP BENCHMARKINGU

Výsledky benchmarku bez použití proxy:

This is ApacheBench, Version 2.0.41-dev <\$Revision: 1.141 \$> apache-2.0
Copyright (c) 1996 Adam Twiss, Zeus Technology Ltd, <http://www.zeustech.net/>
Copyright (c) 1998-2002 The Apache Software Foundation, <http://www.apache.org/>

Benchmarking 91.228.45.211 (be patient)

Server Software: Apache/2.0.52
Server Hostname: 91.228.45.211
Server Port: 80

Document Path: /
Document Length: 1713 bytes

Concurrency Level: 20
Time taken for tests: 0.198887 seconds
Complete requests: 1000
Failed requests: 0
Write errors: 0
Non-2xx responses: 1000
Total transferred: 1994000 bytes
HTML transferred: 1713000 bytes
Requests per second: 5027.98 [#/sec] (mean)
Time per request: 3.978 [ms] (mean)
Time per request: 0.199 [ms] (mean, across all concurrent requests)
Transfer rate: 9789.48 [Kbytes/sec] received

Connection Times (ms)

	min	mean[+/-sd]	median	max
Connect:	0	0 0.0	0	0
Processing:	1	3 1.0	3	26
Waiting:	0	3 0.9	3	26
Total:	1	3 1.0	3	26

Percentage of the requests served within a certain time (ms)

50%	3
66%	3
75%	4
80%	4
90%	4
95%	4
98%	4
99%	5
100%	26 (longest request)

Výsledky benchmarku s použitím proxy:

This is ApacheBench, Version 2.0.41-dev <\$Revision: 1.141 \$> apache-2.0
Copyright (c) 1996 Adam Twiss, Zeus Technology Ltd, <http://www.zeustech.net/>
Copyright (c) 1998-2002 The Apache Software Foundation, <http://www.apache.org/>

Benchmarking 91.228.45.211 [through 91.228.45.211:8080] (be patient)

```
Server Software:      Apache/2.0.52
Server Hostname:     91.228.45.211
Server Port:        80

Document Path:      /
Document Length:    1713 bytes

Concurrency Level:   20
Time taken for tests: 0.354959 seconds
Complete requests:   1000
Failed requests:     0
Write errors:        0
Non-2xx responses:  1008
Total transferred:  2009952 bytes
HTML transferred:   1726704 bytes
Requests per second: 2817.23 [#/sec] (mean)
Time per request:    7.099 [ms] (mean)
Time per request:    0.355 [ms] (mean, across all concurrent requests)
Transfer rate:       5527.40 [Kbytes/sec] received
```

Connection Times (ms)

	min	mean[+/-sd]	median	max
Connect:	0	0 0.5	0	5
Processing:	1	6 2.4	6	23
Waiting:	0	5 2.6	6	23
Total:	1	6 2.4	6	23

Percentage of the requests served within a certain time (ms)

50%	6
66%	7
75%	8
80%	8
90%	9
95%	11
98%	12
99%	13
100%	23 (longest request)