VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

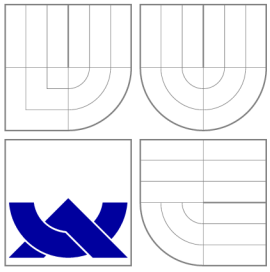
IMPLEMENTACE HTTP S LOKÁLNÍ VYROVNÁVACÍ PAMĚTÍ

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

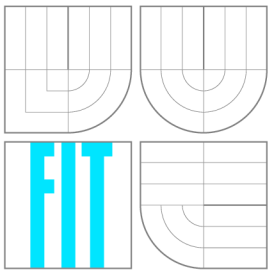
AUTOR PRÁCE
AUTHOR

MICHAL ŠVÁB

BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

IMPLEMENTACE HTTP S LOKÁLNÍ VYROVNÁVACÍ PAMĚTÍ

HTTP IMPLEMENTATION WITH A LOCAL CACHE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MICHAL ŠVÁB

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. RADEK BURGET, Ph.D.

BRNO 2009

Abstrakt

Tato práce se zabývá problematikou implementace HTTP protokolu verze 1.1 s lokální vyrovnávací pamětí. Jedná se o vytvoření aplikačního rozhraní pro odesílání HTTP požadavků a příjem následných odpovědí. Jednotlivé odpovědi jsou pokud možno uloženy na lokálním disku a poté využity při opětovných dotazech na daný dokument. Přičemž je protokol implementován v jazyce Java verze 5 a novější.

Abstract

This thesis describes problems of HTTP 1.1 protocol implementation with local cache. It describes creation of application programming interface for sending requests and receiving responses. Each response, that meets certain criteria is stored locally and then used for another requests for same document. The protocol is implemented in Java programming language, supporting version 5 and higher.

Klíčová slova

HTTP protokol, HTTP hlavičky, klient, server, lokální cache, Java 5, dotaz, odpověď.

Keywords

HTTP protocol, HTTP headers, client, server, local cache, Java 5, request, response.

Citace

Michal Šváb: Implementace HTTP s lokální vyrovnávací pamětí, bakalářská práce, Brno, FIT VUT v Brně, 2009

Implementace HTTP s lokální vyrovnávací pamětí

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Radka Burgeta

.....
Michal Šváb
20.5.2009

Obsah

Kapitola 1

Úvod

I když si to možná neuvědomujeme, tak protokol HTTP se stal nedílnou součástí našich životů. A to především díky masivnímu rozšíření internetu. HTTP protokol je základem každého prohlížeče webových stránek, ale vyskytuje se i v dalších aplikacích, které potřebují komunikovat s nějakou webovou aplikací. I když se neustále zvyšuje rychlost internetových přípojek, tak téma úspory přenosu je stále aktuální. Právě proto umožňuje protokol kombinaci klienta s lokální vyrovnávací pamětí. Ta nejen, že zamezuje zbytečnému odesílání požadavků, pokud je dokument lokálně dostupný, ale také odstraňuje nutnost čekat na příjem dokumentu ze vzdáleného serveru.

Tento dokument se zabývá vytvořením aplikačního rozhraní pro posílání požadavků a příjem následných odpovědí protokolu HTTP. Jedná se tedy o součást klientských aplikací, která je bez problémů použitelná v jakýchkoliv Javových aplikacích, které se rozhodnou využít tohoto rozhraní. Jeho součástí je i lokální vyrovnávací paměť, které je možné nastavit maximální velikost a při její dosažení se vždy odstraní ten nejméně využívaný dokument.

První částí dokumentu je teoretický úvod do problematiky. Jeho první kapitolou je popis protokolu, komunikace a datových entit protokolu HTTP. Druhá kapitola se zaobírá možnostmi cachování v protokolu HTTP a všemi podmínkami, které taková vyrovnávací paměť (cache) musí splňovat.

Druhá část dokumentu popisuje samotné řešení, tedy návrh aplikačního rozhraní a následně jeho implementaci. V závěru se nachází zhodnocení celé práce a popis možných rozšíření.

Kapitola 2

Protokol HTTP 1.1

HTTP (HyperText Transfer Protocol) je jednoduchý aplikační protokol původně určený pro přenos hypertextových dokumentů. Nyní je ale využíván i pro přenos dalších informací. Pomocí rozšíření MIME (Multipurpose Internet Mail Extensions) je možné přenášet prakticky jakákoliv data.

2.1 Historie

2.1.1 HTTP/0.9

První verze protokolu byla definována v roce 1991 asociací W3C a nyní se označuje jako HTTP 0.9 [4]. Slouží jako základ pro všechny další verze, které by měly být vždy zpětně kompatibilní s touto verzí. Protokol byl velice jednoduchý a sloužil pouze pro získání textového dokumentu ze serveru. Klient se připojil k serveru, poslal požadavek, například `GET /index.html` a server mu odeslal požadovaný dokument. Nebyly zde žádné hlavičky, ani žádné další dotazovací metody než GET a odpovědí musel být jedině dokument typu HTML.

Díky zpětné kompatibilitě těmto požadavkům rozumí i dnešní servery, ale protokol je tak jednoduchý, že dnes nemá velké využití.

2.1.2 HTTP/1.0

Díky jednoduchosti předchozí verze byl protokol rozšiřován o další možnosti a to jak ze strany klienta, tak serveru [2]. Nejvýraznějšími novými funkcemi byla možnost použití hlaviček v požadavcích a také nové druhy požadavků. Výsledkem byl protokol HTTP verze 1.0, který byl oficiálně definovaný v roce 1996 dokumentem RFC1945. Tato definice přišla až po letech úspěšného používání protokolu.

Byl vytvořený jednotný formát zpráv a jednou z hlavních změn bylo zobecnění pro podporu všemožných formátů dat, narozdíl od předchozího striktně daného hypertextového dokumentu. Toho bylo docíleno propůjčením některých koncepcí z MIME standardu definovaného pro e-mail, například některé konstrukce hlaviček.

I přes vytvoření webserverů a klientů podporujících spoustu nových vlastností, se podařilo zachovat zpětnou kompatibility s HTTP verze 0.9.

2.1.3 HTTP/1.1

S velkým rozšířením internetu od poloviny 90. let se čím dál více zatěžovaly webservery a postupem času se přišlo na následující nedostatky HTTP protokolu verze 1.0:

- Nutnost mít každou doménu na jiném serveru,
- každá HTTP session může pracovat pouze s jedním požadavkem klienta,
- chybějící podpora pro snížení zatížení serverů, jako například: vyrovnávací paměti, proxy servery a přenos pouze části dokumentu.

A proto vzniká nová verze 1.1 [5]. Tentokrát vzniká specifikace (definovaná v dokumentu RFC2068) dříve než protokol samotný. Dokument byl vydán v roce 1999, ale pracovalo se na něm již v době dokumentování verze 1.0.

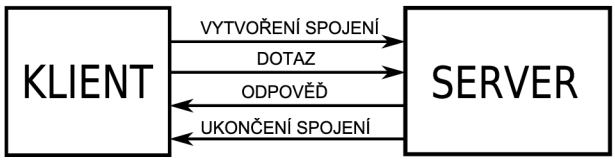
Tato verze přináší řadu vylepšení, především se jedná o:

- Podporu identifikace jména serveru,
- odeslání dat podle parametrů zadaných v hlavičce požadavku,
- stálé připojení klienta k serveru,
- přenos dat po částech,
- podporu pro proxy servery a vyrovnávací paměti.

2.2 Popis komunikace

Jedná se o aplikační protokol postavený na modelu dotazu a odpovědi. O vlastní přenos dat se stará protokol nižší vrtvy, zde se jedná o protokol TCP. Díky tomu se HTTP protokol nestará o to jak budou data doručena na druhý konec, ale pouze předpokládá, že data dorazí ve stejném stavu v jakém byla odeslána. Komunikace je iniciována klientem, který se připojí k serveru a odešle na něho požadavek. Na tento požadavek server ihned odpoví, viz obrázek 2.1.

Díky nové vlastnosti protokolu verze 1.1 ale server nemusí ihned ukončit spojení. Existuje možnost trvalého připojení, což definuje mechanismus ukončení spojení. Spojení se ukončuje poté, co klient obdrží požadavek s hlavičkou `Connection: close`. Hlavní výhodou trvalého připojení je možnost odeslání více požadavků v řadě aniž by se čekalo na odpovědi od serveru, v tomto případě je ale nutné aby server odpověděl na dotazy přesně ve stejném pořadí v jakém je obdržel.



Obrázek 2.1: Komunikace klienta se serverem

Protokol také nabízí možnosti vyjednávání o formátu odpovědi. První je *serverem řízené vyjednávání*, kde se server rozhoduje podle hlaviček dotazu. Mezi takové hlavičky patří `Accept`, `Accept-Charset`, `Accept-Language` anebo `Accept-Encoding`. Druhou možností

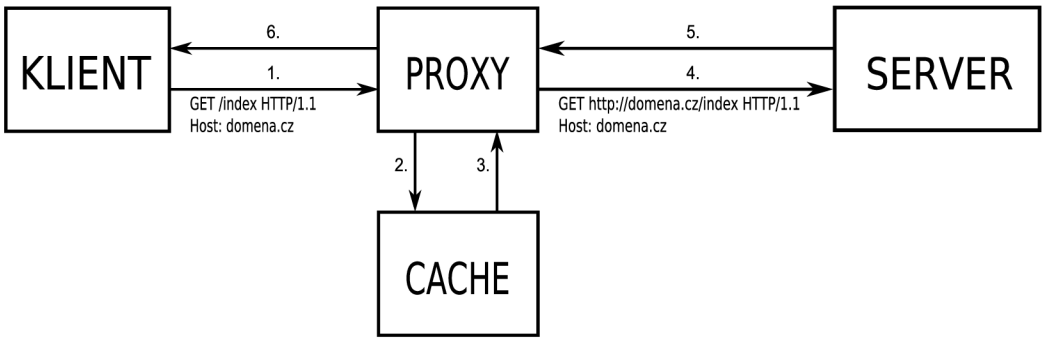
je *klientem řízené vyjednávání*. Zde se klient rozhoduje ze seznamu různých URL v těle odpovědi anebo podle obsahu hlavičky `Alternates`. Oproti serverem řízenému vyjednávání je zde ovšem nutnost použít další dotaz pro získání cílového dokumentu.

2.2.1 Proxy a Gateway

HTTP komunikace se nemusí účastnit pouze klient a server, ale i další prostředníci, jsou to proxy servery a gateway. Přičemž není limitovaný počet jednotlivých „mezistanic“ při komunikaci se serverem.

Proxy

Jedná se o program fungující zároveň jako klient a server [3]. Když se klient připojí k proxy, tak se tváří jako server. Poté proxy přepíše přijatý dotaz anebo jeho část a odešle ho na původní server a v tuto chvíli se chová jako klient. A nakonec klientovi přepošle odpověď od serveru, viz obrázek 2.2. Pokud je ovšem součástí proxy i cache, tak nemusí být původní server vůbec kontaktován, ale klientovi je ihned poslán dokument uložený lokálně na proxy serveru, viz kapitola 3.



Obrázek 2.2: Komunikace se serverem prostřednictvím proxy

Proxy má dva úkoly:

- Ukrýt za sebou cílové počítače, které se poté do internetu připojují přes tento proxy server a konkrétní počítače jsou tedy anonymní. Toto je především z bezpečnostních důvodů.
- Zrychlit přístup k dokumentům a entitám pomocí cache. Toto je nejčastější důvod pro použití proxy serveru.

Proxy servery, které zajišťují caching jsou zpravidla velmi optimalizované pro co nejrychlejší přístup k souborům, využívají tedy speciálních souborových systému a optimalizovaného TCP spojení. Často se využívají ve velkých organizacích pro snížení nákladů za internetové připojení. Zde se ale také využívá další možnosti proxy serveru a to je filtrování dotazů, díky tomu je možné zamezit přístup k některým internetovým stránkám, popřípadě celým doménám.

Proxy se dělí na dva druhy:

- **Transparentní** neupravují dotazy a odpovědi kromě údajů nutných pro autentizaci a identifikaci proxy serveru.

- **Netransparentní** upravují dotazy a odpovědi za účelem poskytnutí nějaké další funkcionality. Může to být například již zmíněná anonymita klienta.

Gateway

Na rozdíl od proxy má gateway za úkol zajistit transparentní přenos mezi různými protokoly, viz obrázek 2.3. Pokud v HTTP dotazu použijeme název jiného protokolu než HTTP, tak je to právě gateway, která zajistí transparentní přístup k takovému dokumentu. Příkladem takového dotazu může být následující.

```
GET ftp://domena.cz/soubor.txt HTTP/1.1
```

Obrázek 2.3: Komunikace se serverem prostřednictvím gateway

2.3 Formát dotazu a odpovědi

Protože každá verze protokolu zajišťuje zpětnou kompatibilitu se všemi předchozími verzemi, tak se musí vždy u požadavků dodržovat formát dané verze protokolu. Také server musí vždy odpovědět ve formátu odpovídajícímu verzi požadavku. Formáty zpráv si jsou velmi podobné mezi jednotlivými verzemi, ale na první pohled je můžeme rozlišit podle prvního řádku dotazu, ve kterém je zmíněná verze protokolu. Pokud verze není specifikovaná, tak se jedná o původní HTTP verze 0.9.

Dotaz se skládá z dotazového řádku, hlaviček blíže specifikujících požadavek, poté prázdného řádku a případně těla dotazu, to však není povinné. Příkladem může být následující dotaz.

```
GET /index.htm HTTP/1.1
Host: domena.cz
Accept: text/html
```

Odpověď od serveru má velice podobnou strukturu jako dotaz. Hlavním rozdílem je první řádek, takzvaný stavový řádek. Jako příklad je uvedená následující jednoduchá odpověď.

```
HTTP/1.1 200 OK
Server: Apache
Date: Sun, 18 Mar 2009 12:00:00 GMT
Content-Type: text/html
Content-Length: 600

<html> ...
```

2.3.1 Dotazový řádek

První částí dotazového řádku je metoda protokolu. Ta určuje službu, kterou klient od serveru požaduje. Server ovšem nemusí podporovat všechny typy těchto služeb. Pokud danou službu nepodporuje, tak je klientu odeslána odpověď s chybovou hláškou. Zde je přehled všech metod.

- **GET** Jedná se o požadavek na získání dokumentu identifikovaného specifikovanou URL.
- **POST** Používá se pro odeslání určitých dat na server. Pro odesílaná data neplatí žádná omezení a záleží pouze na serveru, respektive na cílové URL, co bude s těmito daty provedeno.
- **HEAD** Jedné se vlastně o metodu GET, ale neodesílají se žádná data v odpovědi, pouze hlavička.
- **PUT** Používá se pro uložení odesílaných dat na server, která pak budou následně dostupná například pomocí metody GET.
- **DELETE** Jedná se o metodu pro smazání dokumentu na zadané URL.
- **OPTIONS** Umožňuje získat informace o možnostech komunikace na dané URL, popřípadě pokud se jako URL zadá `*`, tak je možné získat možnosti celého serveru.
- **TRACE** Používá se k testování původního serveru, od něho by se měla vrátit kladná odpověď.

Druhou částí dotazového řádku je URL. Ta může být zadána dvěma způsoby.

1. Jako absolutní URL. Například `http://www.domena.cz/index.htm`.
2. Jako absolutní cesta. Například `/index.htm`.

Většinou se používá druhá možnost, přičemž jméno serveru, popřípadě IP adresa je odeslána v hlavičce `Host`. Absolutní cesta nesmí být nikdy prázdná, pokud se tedy dotazujeme na kořenový adresář, tak musíme uvést `/`. Absolutní URL se používá především u dotazů mezi proxy servery. Poslední možností je jako cestu uvést `*`, a tím značíme, že se jedná o dotaz na celý server.

Poslední částí dotazového řádku je označení protokolu, pokud není uvedeno, jedná se o protokol verze 0.9. V dnešní době se nejčastěji jedná o `HTTP/1.1`, případně `HTTP/1.0` u starších serverů.

2.3.2 Stavový řádek

Narozdíl od dotazového řádku je na prvním místě verze protokolu po které následuje stavový kód. Na konci ještě může být informační zpráva spojená se stavovým kódem. Stavové kódy jsou tříčíselné a jsou rozdělené do pěti skupin.

1. **informační** - oznamují, že požadavek byl přijat
   - `100 Continue` Klient by měl pokračovat v odesílání požadavku. Jde o jakousi částečnou odpověď, která značí, že zatím odeslaná část nebyla odmítnutá serverem. Na konci musí server odeslat konečný kód.

- `101 Switching Protocols` Oznamuje, že server změnil protokol podle hlavičky `Upgrade`.

2. **úspěch** - oznamují, že server přijal a akceptoval požadavek
   - `200 OK` Požadavek byl úspěšně obsloužen.
   - `201 Created` Požadavek byl obsloužen a výsledkem je nový dokument dostupný na adrese, která se nachází v těle odpovědi.
   - `202 Accepted` Požadavek byl přijat serverem, ale jeho zpracování ještě neskončilo.
   - `203 Non-Authoritative Information` Značí, že některé hlavičky nepochází z originálního serveru, ale například z cache. Jinak má ale stejný význam jako odpověď 200.
   - `204 No Content` Požadavek byl úspěšně zpracován, ale nejsou žádná data, která by mohla být odeslána. Součástí této odpovědi tedy nesmí být žádné tělo, maximálně nějaké aktualizované informace v hlavičkách.
   - `205 Reset Content` Server zpracoval daný požadavek a informuje klienta aby uvedl odeslaný dokument do původního stavu.
   - `206 Partial Content` Server zpracoval částečný požadavek typu GET, specifikovaný v hlavičce `Content-Range:`.
3. **přesměrování** - oznamují, že je zapotřebí dalších akcí pro naplnění původního požadavku
   - `300 Multiple Choices` Požadovaný dokument je dostupný na více místech a je jen na klientu, který použije.
   - `301 Moved Permanently` Dokument byl trvale přesunut na jinou URL specifikovanou v hlavičce `Location`.
   - `302 Found` Dokument by dočasně přesunut na jinou URL, ale klient by si o tom neměl ukládat záznamy, protože je to jen dočasné.
   - `303 See Other` Odpověď je dostupná na jiné URL.
   - `304 Not Modified` Klient poslal podmíněný GET požadavek na dokument, který má uložený v cache a server odpovídá, že dokument nebyl změněn.
   - `305 Use Proxy` Požadavek musí být znovu poslán prostřednictvím proxy zadaný v hlavičce `Location`.
   - `307 Temporary Redirect` Dokument by dočasně přesunut na jinou URL, ale klient by si o tom neměl ukládat záznamy, protože je to jen dočasné.
4. **chyba klienta** - oznamují, že klient odeslal chybný dotaz, anebo neměl dostatečná oprávnění
   - `400 Bad Request` Server neporozuměl požadavku z důvodu chybného zápisu.
   - `401 Unauthorized` S požadavkem nebyly odeslané správné autentizační údaje.
   - `402 Payment Required` Rezervováno pro budoucí použití.
   - `403 Forbidden` Server požadavku porozuměl, ale nesmí na něho odpovědět.
   - `404 Not Found` Dokument na zadané URL neexistuje.

- `405 Method Not Allowed` Metoda požadavku není povolena na dané URL.
- `406 Not Acceptable` Server nemůže na zadaný dotaz odpovědět ve formátu specifikovaném v dotazu.
- `407 Proxy Authentication Required` Klient se musí nejdříve autentizovat s proxy.
- `408 Request Timeout` Klient nestihl odpovědět v čase, který mu server vyhradil.
- `409 Conflict` Požadavek nemohl být obsloužen, protože daný dokument je v konfliktu. K tomu nejčastěji dochází při požadavku typu `PUT`.
- `410 Gone` Požadovaný dokument je nedostupný a nejsou na něho známé žádné odkazy.
- `411 Length Required` Server odmítá odpovědět na požadavek, který nemá hlavičku `Content-Length`.
- `412 Precondition Failed` Při testování hlaviček požadavku došlo k selhání.
- `413 Request Entity Too Large` Požadavek byl příliš velký.
- `414 Request-URI Too Long` URL v požadavku byla příliš dlouhá.
- `415 Unsupported Media Type` Tělo požadavku je v neznámém formátu.
- `416 Requested Range Not Satisfiable` V požadavku byl specifikovaný rozsah požadované odpovědi, ale dokument v tomto rozsahu není dostupný.
- `417 Expectation Failed` Server nemůže splnit požadavky specifikované v hlavičce `Expect`.

5. **chyby serveru** - oznamují, že server nemůže obsloužit požadavek

   - `500 Internal Server Error` Nastala chyba na serveru.
   - `501 Not Implemented` Server nepodporuje danou metodu požadavku.
   - `502 Bad Gateway` Server, který pracuje jako brána nebo proxy obdržel chybné informace od originálního serveru.
   - `503 Service Unavailable` Server nemůže odpovědět z důvodu přetížení nebo údržby.
   - `504 Gateway Timeout` Server pracující jako brána nebo proxy neobdržel požadavek v požadovaném čase.
   - `505 HTTP Version Not Supported` Server nepodporuje verzi HTTP protokolu požadavku.

2.3.3 Hlavičky

Hlavičky jsou ve stejném formátu jak pro dotaz, tak i odpověď. Každá hlavička je na samostatném řádku. Formát je následující.

```
název: hodnota[;parametr=hodnota] konec řádku
```

Přičemž parametry jsou nepovinné. Příklady hlaviček mohou být následující:

```
Connection: close
Accept-Charset: utf-8, iso-8859-2;q=0.9
```

Hlavičky se dělí na čtyři skupiny:

- **Obecné hlavičky** - mohou být součástí jak dotazu, tak odpovědi a poskytují obecné informace o zprávě. Například `Date`, určující datum vytvoření dokumentu, které musí být vždy ve formátu spocifikovaném v dokumentu RFC 822. Další hlavičkou je `Connection`, která umožňuje specifikovat podmínky spojení. Především se využívá k ukončení trvalého připojení k serveru. Dále třeba `Transfer-Encoding`, která určuje jakým způsobem je zpráva transformovaná.

- **Hlavičky dotazu** - některé mohou být povinné při použití určitého dotazu, případně podmíněné použitím další hlavičky. Hlavní hlavičkou je `Host`, která je povinná a určuje název ne bo IP adresu serveru na který posíláme požadavek. Dále `Accept`, kterou můžeme nastavit v jakém formátu si přejeme dostat daný dokument, `Accept-Charset`, `Accept-Encoding` umožňující použití specifikované znakové sady, respektive kódování přenosu. Nakonec to může být třeba `User-Agent` obsahující informaci o použitém software na straně klienta.

- **Hlavičky odpovědi** - `Location` oznamující lokaci URL dokumentu, který byl nedostupný na dotazované adrese. `WWW-Authenticate` oznamuje, že je požadováno ověření pro přístup k danému dokumentu, popřípadě službě.

- **Hlavičky těla** - popisují obsah těla. I zde mohou být některé hlavičky podmíněné jinými. Například je často povinná `Content-Length` popisující délku těla odpovědi nebo `Content-Type`, který se musí použít vždy když se nejedná o HTML dokument. Dále například `Content-Encoding` oznamuje jakým způsobem je tělo zakódované, přičemž hodnotou je některé z MIME kódování. Dále `Content-Range`, která umožňuje získat pouze část dokumentu.

Kapitola 3

Možnosti cachování protokolu HTTP/1.1

HTTP protokol nabízí spoustu možností pro podporu cachování a jelikož jsou úzce spjaté jak přímo s protokolem, tak i mezi sebou navzájem, bude užitečné si nejdříve popsat základní principy cachování a až poté se zaobírat konkrétními hlavičkami, metodami, atd.

Cílem cachování je snížit počet vyslaných požadavků a také nutnost odesílat celé odpovědi. Toho docílíme použitím mechanismů vypršení aktuálnosti a validace.

3.1 Správné chování cache

Cache musí vždy odpovědět na požadavek s tou nejaktuálnější odpovědí pokud požadavek splňuje jedno z následujících kritétií.

- Pokud byla aktuálnost zkontrolovaná u původního serveru.
- Pokud je dokument dostatečně „čerstvý“, to znamená, že splňuje to nejméně omezující pravidlo klienta, serveru nebo cache. V některých případech je dokument odeslán i když nesplňuje daná kritéria, v tomto případě je ale do hlavičky přidáno příslušné varování.
- Pokud je to odpověď s kódem 304 (Not Modified), 305 (Proxy Redirect) anebo jedním z chybových kódů 4xx, popřípadě 5xx.

Pokud se cache nepodaří spojit s původním serverem, musí postupovat podle předchozích pravidel pokud obsahuje daný dokument, pokud ne tak vrátí chybovou odpověď.

Pokud cache obdrží odpověď, kterou by normálně přeposlala klientovi a tato odpověď není aktuální musí jí přeposlat aniž by upravovala hlavičky této odpovědi. Jinak by mohlo dojít k zacyklení komunikace. Je totiž na klientovi, jak vyhodnotí neaktuální odpověď.

3.1.1 Varování o neaktuálnosti

Pokud odpověď od cache není aktuální, popřípadě to nejsou data přímo z dané cache, musí cache přidat k odpovědi hlavičku `Warning`. Součástí této hlavičky je třímístný kód. První číslo určuje zda musí být varování z položky cache smazáno po úspěšné revalidaci dokumentu. Pokud se jedná o cache verze HTTP/1.0, tak budou tyto hlavičky uložené vždy.

Aby proto nedocházelo ke konfliktům s novějšími cache, tak se navíc posílá v této hlavičce parametr `warning-date`.

Kódy s číslem 1xx popisují status revalidace odpovědi popřípadě její aktuálnost. Tyto kódy musí být smazány po úspěšné revalidaci. Tyto kódy také nesmí být generovány klienty, ale pouze v cache, která validuje uložený dokument.

Kódy s číslem 2xx popisují, že některé části dokumentu nemohli být zaktualizovány během revalidace a proto tyto kódy musí zůstat uložené v cache za všech okolností.

Součástí této hlavičky je také varovná zpráva. Ta může být v libovolném jazyce a proto je možné v jedné odpovědi mít i více hlaviček typu `Warning` a to i se stejnými varovnými kódy.

3.2 Model vypršení aktuálnosti

3.2.1 Vypršení specifikované serverem

Cachování funguje nejlépe, když cache může vynechat odeslání požadavku na původní server. Toho je docíleno stanovením času, jak dlouho bude dokument aktuální v odpovědi od původního serveru. Po tuto dobu tedy cache může odesílat klientům odpovědi aniž by byl kontaktován původní server. Na serverech musí být velmi opatrně nastaven čas po který zůstane dokument aktuální, aby nedošlo k nějakým problémům.

Pokud server chce, aby byl dokument vždy validován, tak stačí nastavit čas vypršení na nějaké datum v minulosti. Díky tomu si bude sémanticky transparentní cache vždy validovat dokument jako potenciálně neaktuální. Pokud ale server chce, aby byl dokument zaručeně vždy revalidován, musí se použít parametr `must-revalidate` v hlavičce typu `Cache-Control`.

Datum vypršení aktuálnosti je možné specifikovat dvěma způsoby:

1. pomocí hlavičky `Expires`,
2. pomocí parametru `max-age` v hlavičce `Cache-Control`.

3.2.2 Heuristické vypršení aktuálnosti

Ne vždy servery vrací datum, respektive čas, kdy vyprší aktuálnost dokumentu. Údajně jen 10% serverů využívá těchto možností [7]. Proto se některé cache snaží přiřadit dokumentům tyto časy heuristicky a to z informací z ostatních hlaviček, například `Last-Modified`. Součástí specifikace HTTP protokolu ale nejsou takovéto algoritmy, a proto není zaručena kvalita takovýchto „předpovědí“. Proto je preferováno, aby servery odesílaly časy vypršení aktuálnosti ve svých odpovědích.

Mnoho cache využívá tzv. *last-modified factor (LM faktor)* algoritmu. Ten je založený na relativně intuitivním principu, že stránka která se nedávno změnila se nejspíše změní zase. Naproti tomu stránka, která nebyla dlouho modifikovaná se nejspíše ani dlouho měnit nebude. LM faktor se počítá jako poměr času, kdy byla entita uložena do cache a stáří entity v době ukládání do cache. A cache předpokládá, že entita bude aktuální tak dlouho dokud LM faktor neklesne pod určitou hranici. Například pokud budeme mít hranici 0.2 a entita ukládaná do cache bude stará 10 dní, tak zůstane v cache aktuální po dobu 2 dnů. Poté se revaliduje původním serverem a pokud nebyla změněna tak se aktualizují časy v hlavičkách. Stáří se změní na 12 dní a doba od uložení do cache se vynuluje. Nyní je tedy vypočítána aktuálnost po dobu 2.4 dní.

3.2.3 Výpočet času vypršení aktuálnosti

Abychom zjistili zda je dokument aktuální je třeba porovnat jeho dobu platnosti se stářím dokumentu. Pokud byla doba vypršení stanovena v hlavičce `Cache-Control`, tak má tato hodnota přednost a výpočet je velmi jednoduchý. Doba platnosti je přímo údajem z parametru `max-age`. Pokud ale tato hlavička neexistuje, použije se datum z `Expires`. A doba platnosti se spočítá odečtení data z hlavičky `Date` od data z hlavičky `Expires`. Poté už je zjištění aktuálnosti jednoduché, pokud je doba platnosti větší než aktuální datum tak je dokument aktuální.

3.3 Model validace

Pokud je v cache požadovaný dokument, ale není aktuální tak se cache musí dotázat původního serveru, popřípadě jiné cache. Tento mechanismus nazýváme validací. Protože nechceme znovu žádat o celý dokument pokud je cachovaný dokument aktuální, tak HTTP protokol verze 1.1 podporuje možnosti podmíněných metod.

Klíčovými prvky protokolu pro podmíněné metody jsou tzv. cache validátory. Vždy když původní server odešle celou odpověď, tak přidá do hlavičky jeden z validátorů, který je uložen současně s dokumentem do cache. Pokud poté klient pošle podmíněný dotaz na dokument, který má v cache, tak je tento validátor odeslán v požadavku. Server poté porovná validátor v požadavku s validátorem v uloženém dokumentu a pokud se shodují je odeslána odpověď s kódem 304 (Not Modified) a bez těla. V opačném případě je odeslán celý dokument.

V protokolu HTTP/1.1 vypadá podmíněný dotaz stejně jako jakýkoliv jiný s tím rozdílem, že obsahuje další hlavičky, které dělají tento požadavek podmíněný.

Je možné odesílat jak pozitivní tak negativní podmínky. Můžeme tedy požadovat, aby metoda bylo vykonána pouze pokud validátory souhlasí a nebo pouze pokud nesouhlasí.

3.3.1 Datum poslední modifikace

Častým validátorem je hlavička `Last-Modified`, která nás informuje o datu poslední modifikace dokumentu. Dokument je tedy validní pouze pokud nebyl změněn od data uvedeného v této hlavičce.

3.3.2 ETag validátory

Hlavičky `ETag` představují tzv. netransparentní validátory. Mohou poskytnout spolehlivější validaci v případech, kdy nechceme ukládat data modifikace. Preferovaným způsobem je použití jak ETagu, tak i data poslední modifikace.

ETag se využívá pro porovnání dvou entit a využívá se především v hlavičkách `If-Match`, `If-None-Match` a `If-Range`. Ukázkou může být následující odpověď:

```
HTTP/1.1 200 OK
ETag: "8cac4-276e-35b36b6a"
```

Když chce cache zvalidovat takovou entitu musí poslat například následující doptaz:

```
GET /index.htm HTTP/1.1
If-None-Match: "8cac4-276e-35b36b6a"
```

V jednom požadavku může být dotaz na aktuálnost více entit, tedy pokud je v cache více dokumentů se stejnou URI, ale jinými ETagy, tak jedním dotazem můžeme zjistit který z nich je aktuální. Například pokud na následující dotaz

```
GET /index.htm HTTP/1.1
If-None-Match: "foo","bar"
```

obdržíme odpověď 304 (Not Modified) musí být její součástí ETag aktuální verze entity.

3.3.3 Silné a slabé validátory

Validátor považujeme za silný, pokud při změně dokument se změní i příslušný validátor. Někdy je ale výhodné měnit validátor pouze při významných změnách a proto se během drobných úprav nemusí měnit příslušné validátory. V tomto případě je nazýváme jako slabé validátory.

ETagy jsou považovány za silné validátory, ale obsahují i mechanismus pro označení za slabé, stačí na začátek přidat řetězec `w/`. Silný validátor můžeme definovat jak validátor, který se změní vždy když se změní alespoň jeden byte dokumentu, ale slabý se změní pouze když se změní význam dokumentu.

Výhodou silných validátorů je možnost jejich využití v jakémkoliv požadavku. Narozdíl od slabých, které není možné využít pro dotazy na rozsah bytů, protože by tak klient nemusel získat konzistentní data. Dalším omezením slabých validátorů je nemožnost použití u složitějších dotazů.

3.4 Kdy můžeme odpověď cachovat?

Odpověď je možné cachovat vždy, když se jedná o úspěšnou odpověď a cachování není zakázáno v parametrech hlavičky `Cache-Control` a zároveň pokud neobsahuje hlavičku `Authorization`. Pokud odpověď neobsahuje čas vypršení aktuálnosti ani žádný validátor, tak by neměla být cachována. Ovšem některé cache protokolu HTTP/1.0 jsou známé porušováním tohoto pravidla.

V některých případech není vhodné dokument, respektive část dokumentu ukládat do cache a právě proto existuje hlavička `Cache-Control`.

Pokud odpověď obsahuje jeden z kódů 200, 203, 206, 300, 301 nebo 410, tak je možné tuto odpověď uložit do cache, ostatní odpovědi nesmí být nikdy uloženy. Jedinou vyjímkou je odpověď s kódem 206, která nemusí být cachována pokud cache nepodporuje hlavičky `Range` a `Content-Range`.

Pravidla pro cachování autentizovaných dat nejsou nejjednodušší. Takové odpovědi mohou být cachovány pouze pokud mají hlavičku `Cache-Control` s jedním z následujících parametrů `s-maxage`, `must-revalidate` nebo `public`. Přičemž ten poslední dovoluje využít odpověď pro řadu dalších odpovědí v budoucnu. HTTP ovšem nespecifikuje jakým způsobem s těmito daty naloží nesdílená cache, tedy cache internetového prohlížeče. V podstatě by mohly být bez problému cachovány a znovu používány jelikož cache není sdílená s více uživateli. To ovšem zcela neplatí, a to například v knihovnách nebo internetových kavárnách.

Dalším problémem spojeným s autentizací je, že některé servery používají autentizaci založenou na adrese klienta místo hesla. Zda bude mít uživatel přístup nebo ne je tedy založeno na jeho IP adrese. V těchto případech by mohlo proxy umožnit přístup spoustu lidem, kteří by normálně přístup neměli.

3.5 Vynucení aktualizace entity

Jednou z nevýhod cachování je, že občas klient obdrží neaktuální entity. V takovém případě je potřeba tyto data aktualizovat anebo zvalidovat. Pro tyto situace má HTTP pár mechanismů. Nejčastěji používané jsou parametry `no-cache` a `max-age` hlavičky `Cache-Control`.

3.5.1 Parametr no-cache

Upozorňuje cache, že nemůže vrátit cachovaná data a to i pokud je v cache aktuální verze dokumentu. Klientův požadavek musí být odeslán přímo původnímu serveru. Toto se nazývá jako tzv. *end-to-end* validace. Požadavek s tímto parametrem je odeslán například po kliknutí na tlačítko *Obnovit* v internetovém prohlížeči. Protože hlavička `Cache-Control` není přítomná v předchozí verzi protokolu HTTP, ale existuje zde obdobná hlavička `Pragma`, která umožňuje totožnou funkcionalitu a i přesto, že je to hlavička protokolu HTTP/1.0 tak je velmi často používaná i dnes.

Použití tohoto parametru ovšem neznamená, že se smaže daný dokument z cache, ale že se odešle podmíněný požadavek s validátorem a pouze pokud je ze serveru přijata novější verze, tak je aktualizovaná entita v cache.

3.6 Vytváření odpovědí z cache

Cílem cache je ukládání informací z odpovědí pro použití v odpovědích na budoucí požadavky. Ve většině případů cache vezme lokálně uložený dokument a odešle ho klientovi. V některých případech ale uložené položky závisí na předchozích požadavcích a proto musí odpověď vytvořit z více částí.

3.6.1 Trvanlivost hlaviček

Abychom mohli definovat chování cache a proxy serverů rozdělujeme hlavičky do dvou skupin.

1. Hlavičky, které jsou součástí odpovědi po celou trasu komunikace. Tedy od původního serveru, přes proxy servery, cache až ke klientovi. Tyto hlavičky musí být vždy uložené společně s dokumentem v cache a zároveň musí být po odeslání z cache připojené k odpovědi.

2. Hlavičky, které mají význam pouze pro jednu komunikaci a nejsou ukládané v cache ani přeposílány proxy servery. Sem patří následující hlavičky:

   - `Connection`,
   - `Keep-Alive`,
   - `Proxy-Authenticate`,
   - `Proxy-Authorization`,
   - `TE`,
   - `Trailers`,
   - `Transfer-Encoding`,
   - `Upgrade`.

   Všechny ostatní hlavičky definované HTTP/1.1 protokolem tedy patří do první skupiny.

3.6.2 Nemodifikovatelné hlavičky

Některé vlastnosti protokolu závisí na některých hlavičkách a proto jsou hlavičky, které nesmí být modifikované pokud tak není přímo uvedeno.

Proxy servery nesmí modifikovat ani přidávat následující hlavičky:

- `Content-Location`,
- `Content-MD5`,
- `ETag`,
- `Last-Modified`,
- `Expires`.

Pokud je v odpovědi hlavička `Cache-Control` s parametrem `no-transform`, tak nesmí být modifikovány ani další hlavičky:

- `Content-Encoding`,
- `Content-Range`,
- `Content-Type`.

3.6.3 Kombinace hlaviček

Pokud cache odešle na původní server požadavek na validaci a server vrátí 304 (Not Modified) nebo 206 (Partial Content), tak cache vytvoří odpověď z dostupných entit.

V případě odpovědi 304 (Not Modified) cache vrátí kompletní dokument, který se nachází v cache. Pokud je to odpověď 206 (Partial Content) a validátory se přesně shodují, tak cache použije částečná data od serveru a zbytek odpovědi vytvoří z uložených entit. V tomto případě se použijí hlavičky uložené s dokumentem v cache se třemi úpravami:

- Hlavičky `Warning` s varovným kódem 1xx jsou smazány jak z odpovědi, tak z příslušného dokumentu v cache.
- Hlavičky `Warning` s varovným kódem 2xx zůstanou nadále v cache i v odpovědi.
- Hlavičky, které přišli v odpovědi od původního serveru nahradí ty uložené v cache.

Pokud se cache nerozhodne daný dokument smazat z cache, tak musí hlavičky daného dokumentu aktualizovat těmi, které přišly s odpovědí od původního serveru.

3.6.4 Kombinování bytových rozsahů

V odpovědi může být pouze část dokumentu, například kvůli použití hlavičky `Range` nebo kvůli chybě v připojení. Po několika takovýchto přenosech může mít cache více částí jednoho dokumentu. V tomto případě může tyto části kombinovat pokud jsou splněny následující podmínky:

- Jak příchozí odpověď, tak uložený dokument mají stejný validátor.
- Oba musí mít silný validátor.

Pokud nejsou tyto podmínky dodrženy, tak je cache povina uložit pouze nejnovější část.

3.7 Sdílené a nesdílené cache

Z důvodů bezpečnosti a důvěrnosti je nutné rozlišovat mezi sdílenými a nesdílenými cache. Nesdílená je taková, která je pouze přístupná konkrétnímu uživateli, v tomto případě by měl být přístup zabezpečen. Všechny ostatní cache jsou považovány za sdílené.

3.8 Chyby a nekompletní odpovědi

Pokud cache obdrží chybnou odpověď, například velikost odpovědi neodpovídá hlavičce `Content-Length`, tak i přesto může být tato odpověď uložena. Musí se s ní ale zacházet jako s částečnou odpovědí. Takováto odpověď musí být vždy označena kódem 203 (Partial Content).

3.9 Invalidace entit po některých metodách

Některé metody, které přímo mění obsah dokumentů mohou způsobit, že odpovídající dokumenty v cache přestanou být použitelné. I když někdy může být tento dokument v cache stále aktuální, my to nemůžeme s jistotou určit.

HTTP protokol nemůže zaručit, že se všechny příslušné dokumenty v cache invalidují. Například požadavek na původní server vůbec nešel přes danou proxy. I přesto ale existují určitá pravidla, která s tímto problémem mohou pomoci.

Některé metody musí způsobit invalidaci příslušného dokumentu v cache. Jedná se o dokument popsaný v jedné z hlaviček `Request-URI`, `Location` nebo `Content-Location`. Jedná se o metody:

- PUT,
- DELETE,
- POST.

Aby nedocházelo k DoS (Denial of Service) útokům tak cache může invalidovat dokument pouze když hlavička `Host` odpovídá upravovanému dokumentu. Pokud cache nerozumí metodě, tak by měla invalidovat všechny entity zmíněné v hlavičce `Request-URI`.

Kapitola 4

Návrh

Cílem je vytvořit aplikační rozhraní (API) zajišťující HTTP komunikaci na straně klienta. Je třeba dávat pozor na význam slova rozhraní v tomto kontextu. V tomto dokumentu se nebude jednat o rozhraní ve smyslu abstraktní třídy jazyka Java. Ale o aplikační rozhraní ve smyslu knihovny, kterou mohou využívat další aplikace pro zpřístupnění určité funkcionality, v tomto případě komunikaci pomocí protokolu HTTP. Součástí je také lokální persistentní cache, která ukládá dokumenty lokálně na disk, aby je bylo možné použít i po ukončení jedné session. To vše implementované v Javě SE 5 s použitím standartních knihoven.

4.1 Model HTTP komunikace

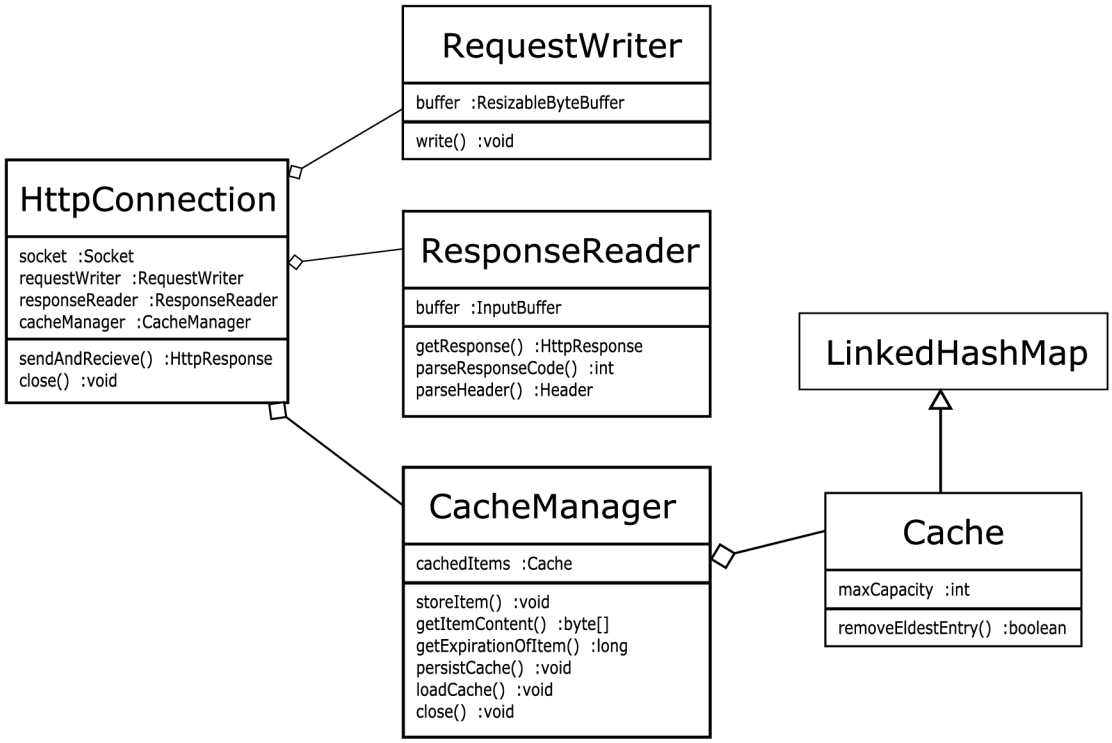
Protože se jedná o aplikační rozhraní, vstupním bodem pro ostatní aplikace bude jediná třída poskytující metody pro odeslání požadavku a přijetí odpovědi. Veškeré operace zajišťující logiku těchto metod jsou skryté před ostatními aplikacemi. Touto třídou je `HttpConnection`, která poskytuje metodu `sendAndRecieve()` jak můžeme vidět na diagramu tříd na obrázku 4.1. Této metodě pouze zadáme jaký typ požadavku se má poslat a na jakou adresu a vzápětí získáme odpověď, tato metoda se postará o veškeré formátování dotazů, vytváření odpovědí a komunikaci s vyrovnávací pamětí cache. Na tomto diagramu také můžeme vidět rozdělení logiky pro odesílání požadavků a přijímání odpovědí.

Třída `RequestWriter` zajišťující pouhé odeslání požadavku na připojený server pracuje s objekty `HttpRequest`, které reprezentují požadavky se všemi náležitostmi jak je definováno ve specifikaci protokolu HTTP/1.1.

Třídou získávající odpovědi od serveru a sestavující z těchto dat objekty `HttpResponse` je `ResponseReader`.

I přesto, že jsou dotazy a odpovědi rozdílné mají jednu věc společnou a to jsou hlavičky. Některé mohou být pouze u odpovědí, některé jen u dotazů, ale vždy mají stejný formát, viz sekce 2.3.3. Proto jsou obě zmíněné třídy potomky třídy `HttpEntity`, která obsahuje jediný atribut a to je seznam hlaviček ve formě objektů typu `Header`, viz diagram na obrázku 4.2. Objekt reprezentující hlavičku HTTP entity se skládá z dvojice jméno a hodnota hlavičky. Díky rozdělení až na tak dílčí části docílíme velmi jednoduchých operací se všemi prvky dotazů a odpovědí. U odpovědí to pak ocení zejména programátoři aplikací, kteří využívají tohoto rozhraní.

Aplikace používající toto rozhraní tedy obdrží odpověď jako instanci objektu třídy `HttpResponse`. Výhodou tohoto přístupu je, že aplikace má k dispozici jakékoliv parametry



Obrázek 4.1: Diagram tříd zobrazující hlavní třídy aplikačního rozhraní

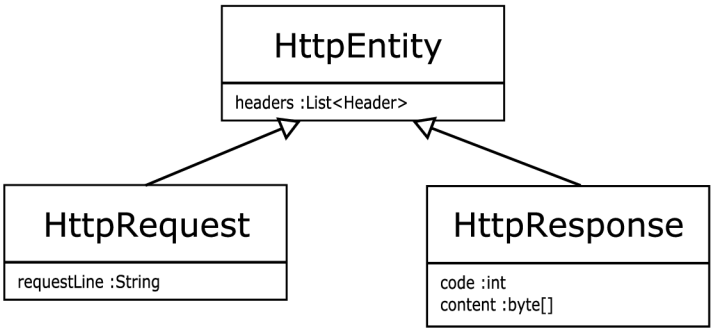
takové odpovědi ve formátu se kterým může velmi jednodušše pracovat. Jak můžeme vidět na obrázku 4.2, kód statusu odpovědi je v podobě primitivního typu *integer* a aplikace tak může ihned vyhodnotit stav odpovědi. Taktéž jsou přístupné i hlavičky jako seznam `Header` objektů, které nabízí různé funkce pro získání jak jména, tak hodnoty hlavičky, popřípadě je možné testovat zda hlavičky s daným jménem je dostupná v dané kolekci. Posledním atributem odpovědi je pole bytů obsahující datovou část. Rozhodl jsem se využít bytového pole, protože se nemusí jednat jen o textový dokument, ale i o binární soubor. I kdyby se ale jednalo jen o textové dokumenty, tak si myslím, že je bytové pole na místě, protože tak dává prostor aplikaci rozhodnout se v jaké znakové sadě bude vytvořený dokument.

4.2 Model cache

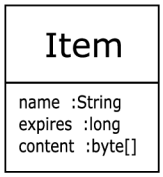
Dalším prvkem implementace je cache. Ta je rozdělená na dvě části, viz obrázek 4.1. První je tzv. manager, který zajišťuje komunikaci s ostatními třídami a poskytuje metody pro ukládání a získávání dokumentů z cache. Druhou třídou je samotná cache, která obsahuje jednotlivé dokumenty a umožňuje jejich uložení a načtení z disku. Každý dokument je reprezentován třídou `Item`, viz obrázek 4.3. Obsahuje vše co potřebujeme vědět o takovém dokumentu, tedy jeho URL, čas kdy vyprší jeho aktuálnost a jeho datový obsah. Z toho jsme bez problému schopni sestavit odpověď.

4.3 Vývojová platforma

Protože ze zadání plyne implementace celého protokolu v Javě, rozhodl jsem se využít vývojového prostředí Eclipse. Je to velmi vyspělá sada vývojových nástrojů a jako jeho



Obrázek 4.2: Diagram tříd zobrazující HTTP entity



Obrázek 4.3: Diagram třídy reprezentující dokument uložený v cache

hlavní výhodu považuji debugger, který je na velmi vysoké úrovni a díky němu je velmi snadné odhalit potencionální problémy projektu.

Jako výchozí verzi Javy jsem se rozhodl použít Javu 5 SE. Díky některým novým konstrukcím jazyka již ale starší verze nejsou podporované. Samozřejmě nejnovější verze Java 6 SE je také bez problému použitelná. Celé aplikační rozhraní je vystavěné nad standardními knihovnami Javy a proto by neměl být problém se spuštěním i v jiných než oficiálních virtuálních strojích kompatibilních s Javou 5.

4.4 Volba komunikace přes sockety

Pro komunikaci pomocí socketů máme ve standardní Javě dvě možnosti. První a také mnohem rozšířenější je použití tříd balíku `java.io` a třídy `Socket` pro připojení k serveru. Druhou možností je použití nového V/V rozhraní, tzv. NIO (New I/O), které je tvořeno balíkem `java.nio` a pro síťová připojení se využívá třída `SocketChannel`. Nyní následuje stručný popis rozdílů v jednotlivých přístupech ke komunikaci.

4.4.1 SocketChannel

Tato třída a celá knihovna NIO je součástí Javy od verze 1.4. Knihovna je vystavěná na knihovně IO. Pokud jsme se kdykoliv rozhodli něco poslat nebo přijmout přes IO, tak jsme museli pracovat s bytovými poli, zde se ale tyto dvě knihovny rozchází. NIO přichází s tzv. buffery, což jsou objektové implementace polí jednotlivých primitivních typů kromě `boolean` [6]. Tyto buffery lze alokovat jako nativní datovou strukturu, takže jejich výkonnost je vyšší než při použití pole bytů. Ovšem jakmile jednou tento buffer vytvoříme tak již nemáme šanci změnit jeho velikost.

4.4.2 Socket

Třída `Socket` a celá knihovna IO jsou součástí Javy již od počátku a proto je to nejrozšířenější knihovna pro přístup k síťové komunikaci. Pro odesílání a přijímání dat se využívají především bytová pole. Rozhodl jsem se využít tohoto přístupu zejména kvůli předem nejasné velikosti odpovědi od serveru, takže by bylo zapotřebí neustále vytvářet nové a větší buffery. A také je výhodné data z odpovědí uchovávat v bytových polích, protože se nemusí vždy jednat o textové dokumenty. Bylo by tedy zbytečné neustále převádět data z bufferů do polí.

Kapitola 5

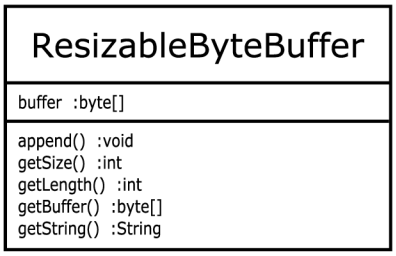
Implementace

V této kapitole je popsána samotná implementace aplikačního rozhraní. Začnu popisem HTTP komunikace a dále bude popsána implementace cache.

5.1 Pomocné třídy

5.1.1 Bytové pole s rozšířenou funkcionalitou

Při načítání dat ze serveru jsem často narážel na problémy s klasickým bytovým polem, a to především díky předem neznámé velikosti příchozího dokumentu a tak jsem musel často vytvářet nové větší pole, aby se neztratila žádná data. Tento způsob byl ale velice neefektivní a to mě přimělo k vytvoření nadstavby nad bytovým polem. Jedná se o třídu `ResizableByteBuffer`, viz obrázek 5.1. Oproti klasickému poli primitivního typu `byte[]` nabízí především metody `append()`, které připojí na konec pole zadaný znak, pole znaků nebo pole bytů. Přitom je úprava velikosti vnitřního pole upravována přímo těmito metodami, pokud je zapotřebí. Na první pohled se zdá neustálá změna velikosti neefektivní, ale je využito standardní metody `System.arraycopy`, která je vysoce optimalizována a plně využívá instrukcí moderních procesorů a je výkonější než ekvivalent jazyka C `memmove()` [1].



Obrázek 5.1: Diagram třídy reprezentující rozšířené bytové pole

Další užitečnou metodou je `getString()`, který vytvoří ze zadaného rozsahu pole řetězec typu `String`.

5.2 HTTP komunikace

Jak již bylo zmíněno v kapitole 4 hlavní třídou celé komunikace je `HttpConnection` a jediná „zvenčí“ přístupná metoda `sendAndRecieve()`. Jejím prvním parametrem je URL požadovaného dokumentu, poté metoda dotazu a nakonec je možné specifikovat další parametry dotazu pokud se jedná o metodu POST. Po zavolání této metody se provedou následující kroky.

1. Vytvoří se požadavek zadaného typu.
2. Díky podpoře *Keep-Alive* připojení, tedy možnosti získání více dokumentů ze serveru v rámci jednoho připojení, se buď použije již připojený `Socket` a nebo se vytvoří nový.
3. Zjistí se, zda je cílený dokument v cache a zda je aktuální a podle toho se provede jedna z následujících akcí:
   - Dokument není v cache, odešle se normální požadavek a vrátí se přijatý dokument.
   - Dokument je v cache, ale není aktuální. Odešle se tedy požadavek typu GET s hlavičkou `If-Modified-Since`. Pokud přijde odpověď `304 (Not Modified)`, tak se použije dokument z cache, jinak se vrátí přijatý dokument.
   - Dokument je v cache a je aktuální, použije se tento dokument a žádný požadavek není odeslán.
4. Pokud byl přijatý dokument ze serveru, tak se zjistí zda je možné ho uložit do cache a případně se uloží.

5.2.1 Odeslání požadavku

Odeslání požadavku zajišťuje třída `RequestWriter` a její metoda `write()`. Jejím jediným parametrem je objekt typu `HttpRequest`. Z tohoto objektu se vytvoří požadavek ve formátu definovaném v protokolu HTTP, ten je následující:

```
metoda URI HTTP/1.1<CR><LF>
hlavička: hodnota<CR><LF>
          ⋮
hlavička: hodnota<CR><LF>
<CR><LF>
```

Přičemž jediná chybička v odřádkování způsobí celý požadavek nečitelný pro server. Tento požadavek ve formě řetězce je poté odeslán přes připojený `Socket` na server.

5.2.2 Přijetí odpovědi

O přijetí dat ze serveru a následné vytvoření objektu `HttpResponse` obsahující odpověď se stará třída `ResponseReader`. Protože na počátku nevíme jak bude odpověď velká, tak se celá hlavička načítá po jednotlivých znacích. A protože hlavička je od dat oddělená znaky *CRLFCRLF*, tak víme, kdy je celá hlavička kompletní. V tuto chvíli z dat v hlavičce zjistíme jak velká je datová část a tu již načítáme po blocích. Jelikož mohou být dokumenty jak textové tak i binární, tak se datová část vždy nechává v bytovém poli.

Odpovědi mohou být zakódovány různými způsoby a proto přijatá data prochází různými metodami, pokud je nutné je dekódovat. První krok probíhá ihned po načtení hlavičky odpovědi a jsou zde dvě možnosti postupů.

- V případě, že je specifikována hlavička `Content-Length`, tak se načte přesný počet bytů specifikovaný v této hlavičce.
- Pokud, ale odpověď obsahuje hlavičku `Transfer-Encoding: chunked`, tak je způsob načítání mnohem složitější. Celé tělo je rozdělené do několika bloků, přičemž každý blok začíná velikostí tohoto bloku v bytech v hexadecimálním tvaru. Poté mohou, ale nemusí následovat dodatečné hlavičky a na dalším řádku začíná samotný blok. Takovýchto bloků může být libovolný počet. V tomto případě se vždy načte první řádek ze kterého se zjistí velikost bloku a tento blok je poté načten a uložen do pole s předchozími bloky. I když jsou data rozdělená do více částí, tak vždy po spojení dávají celý dokument a proto si můžeme dovolit tyto data ukládat do společného pole.

Nyní tedy jsou data stažená a uložená v bytovém poli. To ovšem neznamená, že jsou v konečném stavu. Pokud odpověď obsahuje hlavičku `Content-Encoding` s parametrem `gzip` nebo `deflate`, tak jsou data dekódována příslušnými dekodéry. Až po tomto kroku tedy získáváme požadavaná data.

V tuto chvíli přichází na řadu kontrola, zda je možné tuto odpověď uložit do cache. Prvním faktorem je stavový kód, pokud se jedná o `200 OK` nebo `201 Created` a pokud odpověď neobsahuje hlavičku `Cache-Control` s parametrem `no-cache` ani `must-revalidate` tak ji uložime do cache. Pokud se jedná o `304 Not Modified`, tak se datová část odpovědi vezme z cache. A pokud se jedná o jiný kód, tak pouze vrátíme vytvořenou odpověď.

5.3 Lokální cache

Každý dokument je reprezentován objektem třídy `Item`, která jako atributy obsahuje vše nutné pro operace s cachovaným dokumentem, tedy jeho URL, datum vypršení aktuálnosti a samozřejmě daný dokument v podobě bytového pole. Třída `Cache` představuje kolekci objektů jednotlivých dokumentů. Tato třída je rozšířením standardní `LinkedHashMap` s tím, že se vždy po dovršení maximální kapacity odstraní nejméně používaný prvek kolekce.

Samotné operace s cache zajišťuje třída `CacheManager`. Například vložení dokumentu do cache, kdyse z objektu typu `HttpResponse` vytvoří záznam v cache, přičemž z hlaviček `Cache-Control` nebo `Expires` se použije, případně vypočítá datum vypršení aktuálnosti.

Významnou vlastností této implementace lokální cache je její ukládání na disk při uzavření objektu `CacheManager` a při jeho dalších spuštěních se tyto dokumenty opět načtou z uloženého souboru *cache.dat*. Na disk se ukládá celá instance objektu `Cache` a toho je docíleno implementací standardního rozhraní `Serializable`, které umožňuje transparentní uložení objektů do souboru a jeho načtení bez ztráty nějakých atributů. Jedinou podmínkou je, aby toto rozhraní implementovaly všechny atributy dané třídy což je v našem případě splněno.

Kapitola 6

Závěr

Bylo vytvořeno aplikační rozhraní pro komunikaci klienta pomocí HTTP protokolu se serverem. Rozhraní bylo implementováno v programovacím jazyce Java za použití pouze standardních knihoven. Vše by mělo být kompatibilní s Javou verze 5 a vyšší, starší verze bohužel nejsou podporované díky novým konstrukcím Javy 5. Rozhraní je schopné odeslat jakýkoliv požadavek typu GET, HEAD nebo POST a následně aplikaci zpřístupnit získanou odpověď. Jako součást implementace byla vytvořena lokální vyrovnávací paměť cache.

Výsledky testů ukázaly, že vytvořená implementace bez problémů ukládá přijaté dokumenty do cache a při opětovném požadování daných dokumentů se vrátí ty z vyrovnávací paměti (cache) a rozhraní se ani nepokusí připojit k serveru, pokud je uložený dokument aktuální. Pokud uložený dokument není aktuální tak je správně odeslán požadavek s hlavičkou `If-Modified-Since`. Po ukončení instance objektu `HttpConnection` se obsah cache uloží na disk a při opětovném použití tohoto rozhraní se bez problémů načtou tyto dokumenty bez jakýchkoliv známek poškození nebo neúplnosti. Testy také ukázaly, že odpovědi na požadavky typu POST nejsou ukládány do cache a to splňuje požadované chování.

Protože protokol HTTP obsahuje i některé velmi pokročilé vlastnosti, tak je zde prostor pro určitá vylepšení. Prvním z nich by mohla být podpora bytových rozsahů, kdy je možné požádat o revalidaci pouze určitý bytový rozsah. V případě, že nám server pošle jen část dokumentu, tak aby rozhraní bylo schopné s tímto bytovým rozsahem správně naložit a vše odpovídalo specifikaci HTTP/1.1.

Druhým možným rozšířením by mohly být pokročilé vlastnosti cache. Jako například rozdělení HTML dokumentu do určitých fragmentů a ty by byly uložené samostatně, protože některé části stránky se nemění příliš často.

Literatura

[1] J2SE 5.0 Performance White Paper. [online] [cit 2009-04-02].
URL http://java.sun.com/performance/reference/whitepapers/5.0_performance.html

[2] HTTP/1.1. srpen 1996, [online] [cit 2009-04-15].
URL http://www.apacheweek.com/features/http11

[3] Proxy server. březen 2009, [online] [cit 2009-04-15].
URL http://en.wikipedia.org/wiki/Web_proxy

[4] BERNERS-LEE, T.: The Original HTTP as defined in 1991. 1991, [online] [cit 2009-04-15].
URL http://www.w3.org/Protocols/HTTP/AsImplemented.html

[5] FIELDING, R.: Hypertext Transfer Protocol – HTTP/1.1. červen 1999, [online] [cit 2009-04-15].
URL http://www.w3.org/Protocols/rfc2616/rfc2616.html

[6] MAJER, M.: Síťování v Javě: New I/O. květen 2006, [online] [cit 2009-04-02].
URL http://www.root.cz/clanky/sitovani-v-jave-new-io/

[7] WESSELS, D.: *Web Caching*. O'Reilly Media, Inc., červen 2001, ISBN 1-56592-536-X.

Příloha A

Obsah CD

- Složka *src* obsahující zdrojové kódy aplikačního rozhraní.
- Soubor *README* obsahující popis použití aplikačního rozhraní.
- Složka *doc* obsahující zdrojové kódy zprávy bakalářské práce.
- Soubor *bp.pdf* jako elektronickou verzi technické zprávy bakalářské práce.