VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

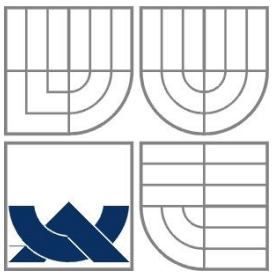
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

ANALÝZA A REKONSTRUKCE WEBOVÉHO PROVOZU S VYUŽITÍM NETMON

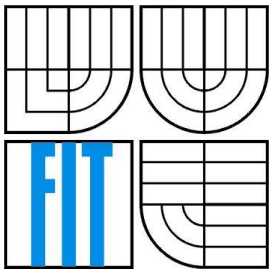
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE Patrik Weiss
AUTHOR

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

ANALÝZA A REKONSTRUKCE WEBOVÉHO PROVOZU S VYUŽITÍM NETMON

WEB TRAFFIC ANALYSIS AND RECONSTRUCTION WITH NETMON

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — Patrik Weiss
AUTHOR

VEDOUCÍ PRÁCE — Ing. Vladimír Veselý
SUPERVISOR

BRNO 2013

Abstrakt

Práce pojednává o návrhu a implementaci nástroje pro rekonstrukci webového provozu na základě HTTP proxy s pomocí aplikačního rozhraní programu Microsoft Network Monitor. Řeší taky problémy při rekonstrukci a analýze HTTP dat jako je TCP fragmentace. Výstup téhle práce je program pro rekonstrukci webové komunikace.

Abstract

This thesis discusses design and implementation of tool for reconstruction of HTTP traffic based upon HTTP proxy with Microsoft Network Monitor application interface. This thesis also solves problems with reconstruction and analysis of HTTP data such as TCP fragmentation. Output of this thesis is an application for web traffic reconstruction.

Klíčová slova

rekonstrukce, Microsoft Network Monitor, NetMon, NetMonAPI, proxy, cache, HTTP, TCP

Keywords

reconstruction, Microsoft Network Monitor, NetMon, NetMonAPI, proxy, cache, HTTP, TCP

Citace

Patrik Weiss: Analýza a rekonstrukce webového provozu s využitím NetMon, bakalářská práce, Brno, FIT VUT v Brně, 2013

Analýza a rekonstrukce webového provozu s využitím NetMon

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Vladimíra Veselýho. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Patrik Weiss
5.5.2013

Poděkování

Tímto bych chtěl poděkovat vedoucímu bakalářky Ing. Vladimírovi Veslýmu za podporu při psaní téhle práce.

Obsah

1 Úvod

Internet sa v súčasnosti denne používa k čoraz vyššiemu počtu rozmanitých činností, a to vrátane oblasti práce s citlivými údajmi. S ich rozšírením tiež prišiel rozmach využívania internetu k vykonávaniu nelegálnych aktivít, či už komunikácie s úmyslom plánovania trestných činov, alebo ich priameho vykonávania, napríklad internetové krádeže, podporovanie detskej pornografie a pod.

Pre boj s kybernetickou kriminalitou existujú dva druhy ochrany. Prvou a všeobecne účinnejšou je prevencia. Jej aplikácia je zväčša v podobe rôznych bezpečnostných nástrojov a protokolov alebo filtrovania obsahu poskytovateľmi internetového spojenia. V oblastiach, kde prvá forma boja zlyháva alebo jej aplikácia je príliš náročná na zdroje, zastáva jej miesto druhá – detekcia. Tento druh boja je reprezentovaný zväčša zachytávaním komunikácie na uzloch internetovej komunikácie a neskoršou analýzou zachytených dát automatizovanými nástrojmi, ktoré v nich vyhľadávajú podozrivú činnosť.

Úlohou tejto práce je vytvoriť nástroj detekčného typu pre HTTP protokol. Úlohou tohto nástroja je buď zachytávať webovú komunikáciu, alebo prevádzať zachytené dáta do zrozumiteľnej formy a tieto dáta neskôr interpretovať v pôvodnom vzhľade. V rámci tejto práce je nutné sa detailne zoznámiť s protokolom HTTP. Výstupom práce by mal byť už spomenutý nástroj, ktorý implementuje rekonštrukciu HTTP správ z pcap súborov, implementuje mechanizmus HTTP proxy a je schopný všetky spracované HTTP správy interpretovať.

Kapitola 2 rozoberá teóriu protokolu HTTP so zameraním na používanie HTTP proxy. Taktiež pojednáva o TCP fragmentácii, ako jednom z najväčších problémov pri implementácii. Na začiatku kapitoly je tiež spomenuté členenie proxy nástrojov. Kapitola 3 opisuje implementáciu nástroja hlavne z pohľadu základnej architektúry a hierarchie tried. Implementácia základnej architektúry je práca inšpirovaná diplomovou prácou Jakuba Olberta – *Analýza a rekonstrukce webového provozu* [1], ktorá pojednáva o rovnakom probléme. Pri implementácii je použitá knižnica Microsoft Network Monitor pre prácu s pcap súbormi. Posledná podkapitola sa zameriava na využitie nástroja. Testovanie nástroja z pohľadu funkčnosti a rýchlosti je uvedené v kapitole 4. Tiež je tam spomenuté, aké chyby testovanie odhalilo a či boli opravené. Práca sa uzatvára v kapitole 5.

2 Teória HTTP proxy

Táto kapitola sa venuje úvodu do protokolu HTTP (Hypertext Transfer Protocol) a jeho prvkov, ktoré sú nutné pre správne fungovanie rekonštrukčného nástroja. HTTP je textový protokol aplikačnej vrstvy ISO/OSI modelu, ktorý sa používa hlavne pre komunikáciu s HTML stránkami. Bol vyvinutý spoločnosťami Internet Engineering Task Force (IETF) a World Wide Web Consortium (W3C) v niekoľkých dokumentoch Requests for Comments (RFC) dokumentoch, z ktorých hlavný je RFC 2616 [2].

HTTP je protokol typu požiadavka – odpoveď (pre každú požiadavku sa vytvorí a pošle práve jedna odpoveď) založený na modeli klient – server (klient je iniciátor komunikácie, server je centralizovaný uzol, ktorý zdieľa svoje prostriedky s klientmi). Podľa svojej definície, vyžaduje k prenosu požiadaviek spoľahlivý transportný protokol, kvôli čomu sa preferuje protokol TCP [14], avšak je možné použiť i protokol UDP. Zdroje pre protokol HTTP sa identifikujú pomocou URI (Uniform Resource Identifier). Máme dve hlavné verzie protokolu, ktoré boli alebo sú používané: 1.0 a 1.1. Verzia 0.9 existovala len v štádiu návrhu a len sa z nej čerpal návrh verzie 1.0. Existuje tiež experimentálne rozšírenie verzie 1.1 – PEP (Protocol Extension Protocol), ktoré niektorí označujú ako verziu 1.2, avšak v praxi nie je veľmi rozšírená.

2.1 Proxy

Proxy [3] je server, ktorý sa správa ako prostredník medzi klientom, ktorý mu posiela požiadavky a prijíma od neho odpovede, a serverom, ktorý od neho prijíma požiadavky a odosiela mu odpovede. Keď klient potrebuje nejakú službu, pripojí sa k proxy, pričom tá sa pokúsi vyhodnotiť túto požiadavku a podľa toho s ňou ďalej zaobchádzať.

Proxy majú mnoho využití:

- ukrývanie identifikácie klientov (hlavne z bezpečnostných dôvodov)
- urýchlenie prístupu k zdrojom pomocou internej cache
- kvôli záznamom alebo odpočúvaniu, čo využívajú hlavne spoločnosti na monitorovanie stránok, ktorá zamestnanci navštívia
- na preskúmanie obsahu, či neobsahuje škodlivý software, predtým, ako sa správa doručí
- k zlepšeniu alebo zamedzeniu prístupu k istým službám:
  - aplikovaním reštrikčných pravidiel sa zamedzí prístup k nevhodným stránkam
  - pomocou proxy sa môžu obísť pravidlá poskytovateľa internetového pripojenia alebo iné reštrikčné opatrenia
  - na pripojenie k službám umiestneným na externých zdrojoch.

Existujú tri hlavné druhy proxy:

1. **brána alebo tunelovacia proxy** – preposiela požiadavky nemodifikované
2. **preposielacia** – je proxy, ktorú môže použiť klient, aby získal externé služby alebo zdroje
3. **reverzná** – používa sa ako *front-end* pre servery, môže implementovať cachovanie, rozloženie záťaže, autentizáciu, kompresiu a pod.

2.2 HTTP správa

Pod pojmom HTTP správy [2] (sekcia 4) rozumieme buď HTTP požiadavky a HTTP odpovede (HTTP request, HTTP response), pričom požiadavky chodia smerom od klienta k serveru a odpovede zo serveru ku klientovi. Oba druhy správ majú generický formát daný dokumentom RFC 822 [8]. Začínajú prvým riadkom špecifickým pre oba druhy správ, potom nasledujú hlavičky, ktoré sú ukončené prázdnym riadkom. Každý riadok je ukončený sekvenciou bajtov s hodnotou 13 a 10 (CRLF). Za hlavičkami nasleduje nepovinné telo správy.

```
generic-message = start-line
                  *(message-header CRLF)
                  CRLF
                  [ message-body ]
start-line      = Request-Line | Status-Line
```

Obrázok 1: Syntax HTTP správy v ABNF notácii

Prvý riadok pre požiadavky [2] (sekcia 5.1) obsahuje metódu, ktorá sa má použiť pre získanie zdroja (bude rozoberaná neskôr), umiestnenie zadané URI (Uniform Resource Identifier) a HTTP verziu klienta. URI sa môže líšiť podľa toho, o akú požiadavku sa jedná; ak sa jedná o požiadavku pre proxy, musí sa použiť absolútna URI, ak sa pýta priamo serveru (alebo si to aspoň myslí), môže použiť relatívnu URI alebo univerzálnu (*).

```
Request-Line   = Method SP Request-URI SP HTTP-Version CRLF
```

Obrázok 2: Syntax prvého riadku HTTP požiadavky v ABNF notácii

Prvý riadok pre odpovede [2] (sekcia 6.1) sa skladá z HTTP verzie servera, stavového kódu (bude rozoberaný neskôr) a dôvodu (správa, ktorá viac približuje stav odpovede). Tu je nutné podotknúť, že text dôvodu nie je záväzný podľa stavového kódu, a aplikácia môže použiť vlastný text vrátane prázdneho reťazca. [2] (sekcia 6.1.1)

```
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

Obrázok 3: Syntax prvého riadku HTTP odpovede v ABNF notácii

2.3 HTTP metódy

HTTP metódy [2] (sekcia 9) definujú klientom požadovanú akciu serveru pre daný zdroj. Delia sa do dvoch skupín: bezpečné (safe) a idempotentné. Bezpečné metódy by nemali vykonávať nič iné ako získanie zdrojov. Toto umožňuje *user agentom* (UA – napr. webový prehliadač) implementovať špeciálne ošetrenie idempotentných metód. Bohužiaľ i bezpečné metódy môžu prestavovať isté riziko, hlavne kvôli dynamicky generovaným zdrojom, preto jediný rozdiel medzi týmito dvoma skupinami je, že pri bezpečných metódach užívateľ nežiadal vedľajšie zdroje, a preto nemôže byť za ne zodpovedný.

Zoznam podporovaných metód sa mení pre každú verziu. V tomto dokumente sú zhrnuté len metódy najpoužívanejších verzií. Verzia 1.0 podporuje metódy *GET*, *POST* a *HEAD* a verzia 1.1 k nim pridáva metódy *OPTIONS*, *PUT*, *DELETE*, *TRACE* a *CONNECT*. HTTP server by mal minimálne podporovať metódy *HEAD* a *GET* a taktiež metóda *OPTIONS* by mala byť podporovaná tam, kde to je možné.

Názov	**Verzia HTTP**	**Idempotencia**	**Popis metódy**
GET	1.0	bezpečná	Získa informácie identifikované pomocou URI vo forme entity.
POST	1.0	idempotentná	Žiada server, aby akceptoval danú entitu a uložil ju podľa URI. Používa sa pre posielanie príspevkov (fóra a pod.) na server.
HEAD	1.0	bezpečná	Metóda *HEAD* je identická ako metóda *GET*, až na to, že klient nežiada o telo správy.
OPTIONS	1.1	bezpečná	Požiada server o zoznam HTTP metód pre zdroj identifikovaný pomocou URI. Ak je URI zastúpené znakom „*“, klient sa pýta na všeobecné metódy serveru. Ak je hlavička *Max-Forwards* rovná 0 keď dorazí na proxy, musí táto proxy odpovedať svojím vlastným zoznamom HTTP metód.
PUT	1.1	idempotentná	Žiada server aby uložil entitu identifikovanú daným URI.
DELETE	1.1	idempotentná	Žiada server aby zmazal entitu identifikovanú daným URI.
TRACE	1.1	bezpečná	Využíva sa k určeniu cesty k serveru. Ak proxy dostane túto požiadavku a hlavička *Max-Headers* je rovná 0, musí odpovedať klientovi telom požiadavku a kódom 200.
CONNECT	1.1	bezpečná	Používa sa pre vytvorenie tunela pre HTTP proxy.

Tabuľka 1: Popis metód

2.4 HTTP stavové kódy

Pre každú požiadavku môže byť odpoveď rôzna; typy odpovedí sú rozlíšené pomocou stavových kódov [2] (sekcia 10). Tie majú rozsah od 100 do 599. Delia sa do skupín po stovkách. Kódy 1xx majú informačnú funkciu; kód 100 funguje ako potvrdenie (ACK – acknowledgement) prijatých dát pri veľmi dlhých požiadavkách (napr. pri metóde *POST*). Kódy 2xx indikujú úspešné vykonanie operácie. Kódy 3xx indikujú, že je nutné vykonať ďalšiu akciu, aby sa požiadavka ukončila úspešne. Kódy 4xx indikujú, že došlo k chybe klienta (zlá URI, neplatná hlavička a pod.). Server by tiež mal odoslať entitu v ktorej vysvetľuje, k akej chybe došlo. UA je povinný túto entitu zobraziť. Kódy 5xx indikujú prípady, keď server nemôže obslúžiť valídna požiadavka, pričom server si je vedomý toho, že narazil na chybu. Okrem odpovede na metódu *HEAD* by server mal pridať i entitu vysvetľujúcu vzniknutú situáciu.

Kód	**Štandardný dôvod**	**Popis**
100	Continue	Klient by mal pokračovať v požiadavke. Server týmto dáva vedieť, že prijal iniciálnu požiadavku.
101	Switching protocols	Server je schopný a umožňuje zmenu aplikačného protokolu pre pripojenie. Protokol by mal byť zmenený, len keď je jeho použitie výhodnejšie.
200	OK	Obsluha požiadavky bola úspešná.
201	Created	Požiadavka bola obslúžená vytvorením nového zdroja, ktorý môže byť získaný na URI v hlavičke *Location*. Odpoveď s týmto kódom by nemala obsahovať entitu novovytvoreného zdroja.
202	Accepted	Požiadavka bola prijatá pre spracovanie, avšak jej obsluha sa nemusí vykonať (toto je zámer protokolu).
204	No Content	Server obslúžil požiadavku, ale nemusí vrátiť entitu a môže vrátiť aktualizované metainformácie.
206	Partial Content	Server obslúžil neúplnú *GET* požiadavku. Táto odpoveď musí obsahovať hlavičky *Content-Range, Date, ETag* alebo *Content-Location* a *Expires, Cache-Control* alebo *Vary*. Požiadavka žiada o túto odpoveď hlavičkou *Range* a voliteľne hlavičkou *If-Range*, pre podmienenú a neúplnú *GET* požiadavku.
300	Multiple Choices	Odpoveď znamená, že klient si môže vybrať medzi viacerými zdrojmi. Server by mal vložiť entitu, ktorá obsahuje zoznam týchto zdrojov tak, aby si užívateľ mohol vybrať.
301	Moved Permanently	Požadovanému zdroju bola permanentne pridelená iná URI, vložená do hlavičky *Location*. Pre metódy *GET* a *HEAD* by klient mal automaticky prejsť na danú URI. Pri ostatných metódach by mal požiadať o potvrdenie užívateľa.
304	Not Modified	Zdroj nebol od poslednej požiadavky zmenený. Táto odpoveď môže prísť len na podmienenú *GET* požiadavku (indikované

		hlavičkou). Táto odpoveď nesmie obsahovať telo, a teda je vždy ukončená hlavičkami.
307	Temporary Redirect	Zdroj je dočasne uložený pod inou URI. Táto URI sa môže kedykoľvek zmeniť, a teda klient by mal stále používať starú URI.
400	Bad Request	Server nerozumie požiadavke kvôli neplatnej syntaxi.
401	Unauthorized	Prístup k zdroju vyžaduje užívateľskú autentizáciu. Táto odpoveď musí obsahovať hlavičku *WWW-Authenticate* obsahujúcu výzvu pre autentizáciu.
403	Forbidden	Server rozumel požiadavke, ale odmieta ho obslúžiť, pričom nepomôže ani autentizácia klienta. Server tiež môže poslať kód 404, ak nechce uviesť dôvod.
404	Not Found	Server nenašiel zdroj identifikovaný danou URI. Tiež by server nemal uviesť, či je to dočasné alebo permanentné.
405	Method Not Allowed	Metóda nie je podporovaná pre zdroj identifikovaný daným URI.
407	Proxy Authentication Required	Tento kód je podobný kódu 401, akurát užívateľ musí autentizovať proxy.
408	Request Timeout	Klient nevytvoril požiadavku, ktorý by server prijal v danom čase. Túto požiadavku je možné opakovať.
500	Internal Server Error	Server nebol schopný obslúžiť požiadavku kvôli neočakávanej udalosti.
501	Not Implemented	Server nemá funkcionalitu na obslúženie požiadavky.
502	Bad Gateway	Server, ktorý sa správa ako proxy alebo brána, prijal poškodenú alebo neplatnú odpoveď od servera.
503	Service Unavailable	Server nie je schopný obslúžiť požiadavku kvôli dočasnému preťaženiu alebo údržbe.
504	Gateway Timeout	Server, ktorý sa správa ako proxy alebo brána, nedostal včas odpoveď od servera alebo od pomocnej služby ako NTP alebo DNS.
505	HTTP Version Not Supported	Server nepodporuje alebo odmieta podporovať danú verziu protokolu HTTP.

Tabuľka 2: Popis stavových kódov

2.5 HTTP hlavičky

Hlavičky HTTP [2] (sekcie 4.2 a 14) správ slúžia ako parametre HTTP transakcie. Majú pevnú syntax: Každá hlavička je ukončená sekvenciou dvoch bajtov s hodnotou 13 a 10 (CRLF). Ak je hlavička dlhá, môže byť rozdelená do viacerých riadkov tým, že ďalšie riadky začínajú znakom medzery alebo horizontálneho tabulátora. Každá hlavička sa skladá z mena a hodnoty rozdelených dvojbodkou. Hlavičky sú oddelené od tela HTTP správy prázdnym riadkom. Je tiež možné definovať hlavičky za telom správy, ak hlavička *Transfer Encoding* obsahuje hodnotu *chunked*. Tieto hlavičky by však z definície nemali mať vplyv na interpretáciu tela správy.

Protokol HTTP má definované štandardné hlavičky a ich sémantiku, je však možné pridávať vlastné hlavičky. Z tohto môžeme hlavičky rozdeliť na tri skupiny: štandardné (definované v RFC), pseudo-štandardné (nie sú definované v RFC, ale sú štandardne používané, napr. hlavička *Proxy-Connection*) a neštandardné (používajú sa interne pre konkrétne dvojice UA-server a inak môžu byť ignorované).

Hlavičky nemajú definovanú maximálnu veľkosť alebo maximálny počet, tieto limity sú však implementované vo väčšine aplikácií kvôli praktickým dôvodom a bezpečnostným rizikám. Medzi ďalšie implementačné obmedzenia patrí neštandardný formát; niektoré implementácie očakávajú štandardný formát hlavičiek, i keď syntax definovaná dokumentom RFC 2616 povoľuje i iné. Pre štandardný formát platí, že každé slovo v názve hlavičky začína veľkým písmenom a pokračuje malými písmenami (okrem hlavičky *TE*), oddeľujúca dvojbodka je hneď za názvom hlavičky a nasledovaná medzerou, po ktorej nasleduje hodnota hlavičky.

Jediná povinná hlavička v HTTP správe je *Host* pre požiadavku a to až pre HTTP verziu 1.1.

Názov	**Hodnota**	**Popis**
Cache-Control	Cache pravidlá	Používa sa na definovanie direktív, ktoré musia byť dodržané cachovacími mechanizmami pre oba smery, aby boli odpovede platné.
Connection	Typ spojenia	Definuje typ spojenia. Pri proxy, obsah tejto hlavičky by nemal byť použitý a proxy by mala vygenerovať vlastné pravidlá pre typ spojenia.
Date	HTTP dátum	Dátum, kedy bola správa vygenerovaná.
Proxy-Connection	Typ spojenia	Má tú istú sémantiku ako hlavička *Connection*, ale používa sa pre spojenie s proxy. Nie je síce súčasťou štandardu, ale veľmi často sa používa.
Via	Identifikácia proxy	Používajú ju proxy na identifikáciu protokolu. Tiež slúži pre detekciu uzavretých cyklov.

Tabuľka 3: Popis spoločných hlavičiek

Názov	**Hodnota**	**Popis**
Accept	MIME typ	Obsahuje zoznam typov médií, ktoré sú akceptovateľné pre odpoveď. Môže obsahovať zástupný znak „*“.
Accept-Charset	Znakové kódovanie	Obsahuje zoznam znakových kódovaní, ktoré je klient schopný akceptovať.
Accept-Encoding	Kódovanie obsahu	Zoznam kódovaní obsahu akceptovateľných klientom.

Host	Meno a port hostiteľa	Identifikuje doménové meno a port hostiteľa. Ak sa port neuvedie, predpokladá sa implicitný pre daný protokol, čo v prípade HTTP je 80.
If-Modified-Since	HTTP dátum	Používa sa pri podmienenej metóde *GET*; hovorí, že server môže poslať kód 304, ak nebol zdroj zmenený.
Max-Forwards	Počet preposlaní	Určuje aký počet proxy brán môže požiadavka prejsť.
TE	Kódovanie prenosu	Určuje, aké rozšírené kódovania prenosu je klient schopný prijať a či je schopný prijať hlavičky na konci kódovania prenosu *chunked*.
User-Agent	Informácie o UA	Slúži pre prenos informácii o UA. Táto hlavička slúži výhradne pre štatistické alebo diagnostické účely.

Tabuľka 4: Popis hlavičiek pre požiadavky

Názov	**Hodnota**	**Popis**
Age	Sekundy	Určuje časový odhad odosielateľa, koľko sekúnd uplynulo od vytvorenia správy. Používa sa cachovacími mechanizmami pri určovaní platnosti správy.
Content-Encoding	Kódovanie obsahu	Obsahuje definície kódovaní, ktoré boli použité na správu presne v tom poradí, ako sú uvedené.
Content-Length	Dĺžka správy	Určuje dĺžku tela správy v počte bajtov po použití všetkých kódovaní. Používa sa len v niektorých prípadoch.
Content-Type	MIME typ	Indikuje typ média entity. Ak bola požiadavka metódou *HEAD*, potom táto hlavička indikuje, aký typ média by bol pri metóde *GET*.
Expires	HTTP dátum	Určuje, kedy daná HTTP správa vyprší. Používa sa v cache mechanizmoch. Táto hlavička neznamená, že po danom čase by sa správa mala zmeniť alebo prestať existovať.
Location	Absolútna URI	Slúži k presmerovaniu klienta na server, kde by sa zdroj mohol nachádzať. Používa sa hlavne pri odpovediach so stavovými kódmi 201 a 3xx.
Server	Identifikácia serveru	Obsahuje informácie o softvéri, ktorý používa server.
Transfer-Encoding	Kódovanie prenosu	Indikuje, či boli použité nejaké transformácie správy pre jej bezpečné doručenie. Kódovania musia byť uvedené v poradí, v akom boli aplikované.

Tabuľka 5: Popis hlavičiek pre odpovede

2.6 Entita

Požiadavky a odpovede môžu prenášať i entitu [2] (sekcia 7), ak to nie je limitované metódou požiadavky alebo stavovým kódom odpovede. Entita sú efektívne dáta, ktoré sa prenášajú cez HTTP protokol, ako napríklad HTML stránka alebo JPEG obrázok. Skladá sa minimálne z hlavičiek, ku ktorým môže byť pripojené i telo.

Hlavičky obsahujú metainformácie o tele entity, v prípade entity bez tela informácie o referovanom zdroji. Medzi štandardné hlavičky entity patria *Allow*, *Content-Encoding*, *Content-Type*, *Content-Language*, *Content-Length*, *Content-Location*, *Content-MD5*, *Content-Range*, *Content-Type*, *Expires* a *Last-Modified* [2] (sekcia 7.1). Môžu sa aj definovať rozšírené hlavičky, ale odosielateľ nesmie predpokladať, že príjemca bude týmto hlavičkám rozumieť.

Telo hlavičky je vo formáte a kódovaní definovanom hlavičkami *Content-Type* a *Content-Encoding*. Prvá hlavička by mala byť prítomná, ak je v entite telo. Ak entita neobsahuje túto hlavičku, príjemca sa môže pokúsiť o odhad typu média. Ak aj napriek tomu ostane tento typ neznámy, príjemca by to mal brať ako typ *application/octet-stream* [4].

2.7 Zreťazenie požiadaviek

Verzia 1.1 protokolu HTTP podporuje aj zreťazenie HTTP požiadaviek [2] (sekcia 8.1). Redukuje to réžiu z verzie 1.0, kde pre každý pár požiadavky – odpovede sa muselo vytvoriť nové pripojenie na server. Zreťazenie je dosiahnuté hlavičkou HTTP protokolu *Connection: keep-alive*; server túto hlavičku interpretuje tak, že po odoslaní odpovede neuzavrie spojenie. Ak predsa potrebuje toto spojenie uzavrieť, musí do odpovede pridať hlavičku *Connection: close*. Taktiež odpovede zo serveru musia byť v rovnakom poradí ako prišli požiadavky.

Zreťazenie požiadaviek má však za následok, že nie je možné určiť koniec správy jednoduchým ukončením spojenia. Toto sa ešte viac komplikuje, ak dôjde k TCP fragmentácii HTTP správy, pretože pri rekonštrukcii TCP komunikácie môže jeden načítaný segment obsahovať koniec jednej HTTP správy a začiatok druhej. V tomto prípade je nutné správne určiť koniec HTTP správy podľa platnej syntaxe definovanej v RFC 2616.

2.8 Kódovanie

V záujme šetrenia sieťovej komunikácie môže byť správa zakódovaná. Rozlišujeme dva druhy kódovaného prenosu: kódovanie prenosu [2] (sekcia 3.6) a s ním súvisiaca hlavička *Transfer-Encoding* a kódovanie obsahu [2] (sekcia 3.5) a s ním súvisiaca hlavička *Content-Encoding*.

Obe hlavičky majú približne rovnaké hodnoty kódovania. Hlavička *Transfer-Encoding* sa ale používa hlavne pre zvýšenie šance, že správa bezpečne a v poriadku dorazí, a hlavička *Content-*

Encoding sa používa na kompresiu entity. Z toho vyplýva, že *Transfer-Encoding* sa aplikuje hlavne na prenos medzi dvomi bodmi, zatiaľ čo *Content-Encoding* na celú cestu medzi klientom a obslužným serverom. Transfer-Encoding tým umožňuje meniť kódovanie cachovanej správy tzv. "on-the-fly", napr. ak má server v internej cache už vygenerovanú entitu. Ak je použitý akýkoľvek typ kódovania v hlavičke *Transfer-Encoding*, posledné kódovanie musí byť vždy *chunked*.

Kódovania musia byť zaregistrované organizáciou IANA (Internet Assigned Numbers Authority) [5]. Pôvodne boli definované tieto kódovania:

- **chunked** – viac v kapitole 2.8.1,
- **gzip** – formát vytvorený programom *gzip* opísanom v dokumente RFC 1952 [11]. Používa Lempel-Ziv kódovanie (LZ77) s 32 bitovým CRC,
- **compress** – vytvorené unixovým programom *compress*. Používa Lempel-Ziv-Welch kódovanie (LZW),
- **x-gzip, x-compress** – ekvivalenty dvoch predošlých kódovaní v starších implementáciách protokolu HTTP,
- **deflate** – *zlib* formát definovaný dokumentom RFC 1950 [9] v kombinácii s *deflate* kompresným mechanizmom definovanom v dokumente RFC 1951 [10],
- **identity** – vyjadruje žiadne kódovanie. Toto kódovanie by sa malo používať len v hlavičke *Accept-Encoding*.

K tomu sa v súčasnosti pridalo:

- **exi** – W3C efektívna výmena XML (effective XML interchange) [6],
- **pack200-gzip** – kódovanie pre prenos Java archívov [7].

Všetky pôvodné kódovania sa môžu požiť v hlavičke *Transfer-Encoding* a takisto okrem *chunked* v hlavičke *Content-Encoding*. Pridané kódovania by sa mali použiť len v hlavičke *Content-Encoding*. V praxi sa však využíva pravidlo, že ak sú klient a server schopní sa dohodnúť i na inom kódovaní cez hlavičky *TE* a *Accept-Encoding*, môžu ho využiť k výmene dát.

2.8.1 Kódovanie prenosu chunked

Kódovanie prenosu typu *chunked* [2] (sekcia 3.6.1) modifikuje telo správy tak, aby mohlo byť poslané po viacerých častiach (nazývaných c*hunks* – odtiaľ názov), každý s indikáciou veľkosti tejto časti. Na konci sa tiež môžu odoslať hlavičky entity. Toto umožňuje prenášať dynamicky vytváraný obsah aj s informáciami nutnými k overeniu dĺžky správy.

Správa zakódovaná kódovaním *chunked* sa skladá z viacerých *chunk*-ov, z ktorých každý začína riadkom s indikáciou dĺžky tohto *chunk*-u v počte bajtov v hexadecimálnej sústave; riadok je ukončený sekvenciou bajtov s hodnotami 13 a 10. Správa sa potom ukončuje jedným prázdnym *chunk*-om nasledovaným hlavičkami entity a ukončenými sekvenciou bajtov 13 a 10.

```
Chunked-Body    = *chunk
                  last-chunk
                  trailer
                  CRLF

chunk           = chunk-size [ chunk-extension ] CRLF
                  chunk-data CRLF
chunk-size      = 1*HEX
last-chunk      = 1*("0") [ chunk-extension ] CRLF

chunk-extension= *( ";" chunk-ext-name [ "=" chunk-ext-val ] )
chunk-ext-name = token
chunk-ext-val  = token | quoted-string
chunk-data     = chunk-size(OCTET)
trailer        = *(entity-header CRLF)
```

Obrázok 4: Syntax kódovania chunked v ABNF notácii

Server môže použiť hlavičky entity v tomto kódovaní, len ak je jedno z nasledujúcich tvrdení pravdivé:

1. V požiadavku obsahovala hlavička *TE* hodnotu *trailers.*
2. Server obsahuje pôvodný zdroj a hlavičky obsahujú iba voliteľné metadáta. Server musí počítať s tým, že tieto hlavičky môžu byť ignorované.

Všetky HTTP/1.1 aplikácie musia podporovať toto kódovanie a musia ignorovať rozšírenia, ktorým nerozumejú.

```
HTTP/1.1 200 OK
Transfer-Encoding: chunked

10
I am chunk
9
ed encode
10
d message.
0
```

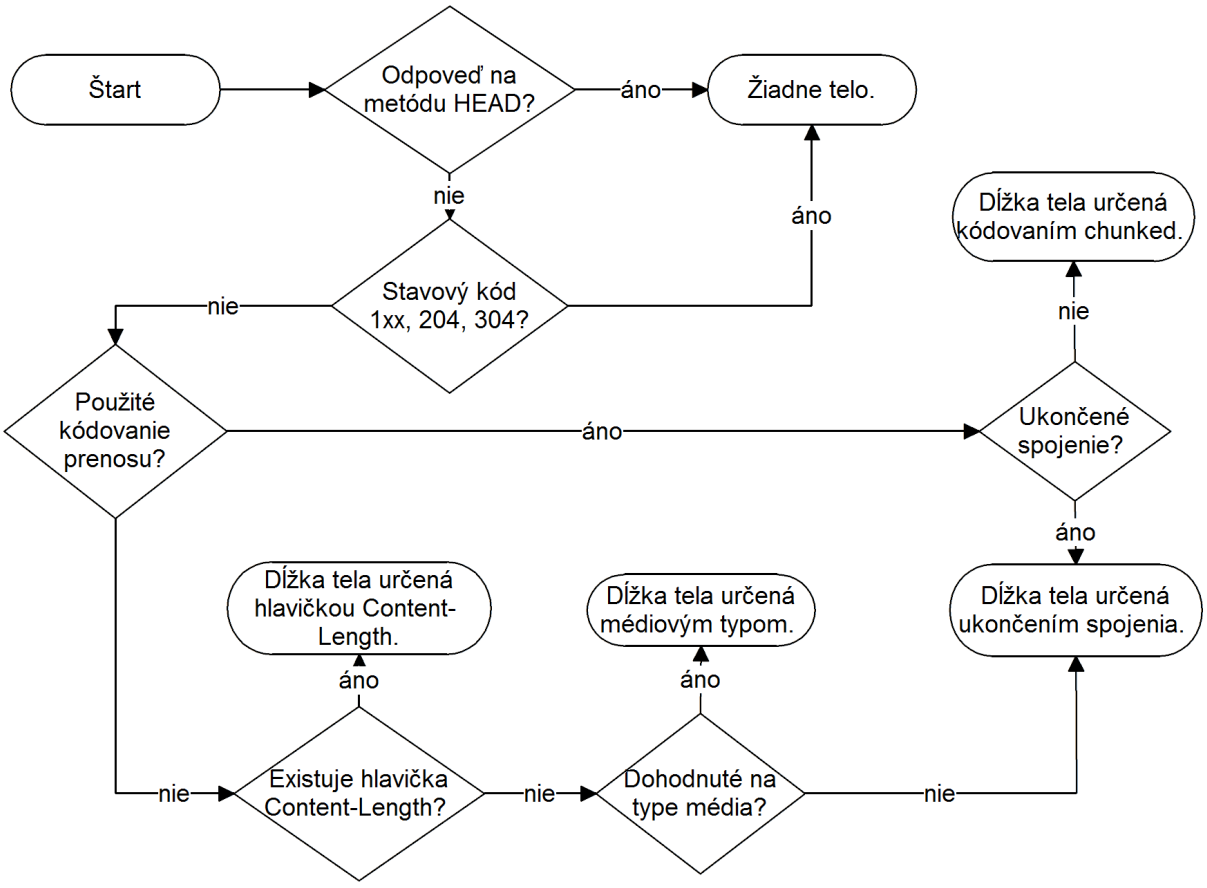
Obrázok 5: Príklad HTTP odpovede zakódovanej kódovaním chunked

2.9 TCP fragmentácia a reassembling

Pri väčších HTTP správach môže dochádzať k TCP fragmentácii. Fragmentácia TCP protokolu je súčasťou spoľahlivého prenosu; každý odoslaný bajt TCP rámcu je identifikovaný sekvenčným číslom, čo znamená, že TCP protokol berie do úvahy len bajty vyššieho protokolu, nie správy. Kontrola fragmentácie nie je teda natívne zahrnutá do TCP protokolu a preto si fragmentáciu musí ustrážiť protokol vyššej vrstvy (v našom prípade protokol HTTP).

Pravidlá pre dĺžku HTTP správy sú v tomto poradí [2] (sekcia 4.4):

1. HTTP odpovede, ktoré nesmú obsahovať telo (1xx, 204, 304), sú vždy ukončené prvým prázdnym riadkom po hlavičkách, bez ohľadu na obsah hlavičiek.
2. Správy obsahujúce hlavičku *Transfer-Encoding* inú než *identity* majú dĺžku správy definovanú práve kódovaním *chunked*, pokiaľ nie je správa ukončená ukončením spojenia.
3. Ak správa obsahuje hlavičku *Content-Length*, hodnota tejto hlavičky v dekadickej sústave predstavuje dĺžku entity aj dĺžku prenosu. Táto hlavička nesmie byť prítomná v správach ak sú tieto dve dĺžky rozdielne (napr. pri prenose *chunked*). Ak správa obsahuje obe hlavičky *Transfer-Encoding* aj *Content-Length*, tá druhá sa ignoruje.
4. Ak prenášaná správa je typu *multipart/byteranges* a dĺžka správy nie je inak definovaná, dĺžka správy je definovaná práve ukončením tohto typu. Odosielateľ sa ale musí uistiť, že príjemca je schopný načítať tento typ správy.
5. Ukončením spojenia (nemožno použiť pre požiadavky, pretože server nemôže poslať späť odpoveď).



Obrázok 6: Grafické znázornenie rozhodovania dĺžky správy

Pri fragmentácii sa teda HTTP správy musia znovu zložiť; tejto technike sa hovorí *reassembling*. Pre rekonštrukciu je nutné poskladať z bajtov prenášaných cez protokol TCP naspäť

správy protokolu HTTP. K tomu slúži syntax HTTP správ ktorej súčasťou sú aj vyššie uvedené pravidlá.

Každá HTTP správa sa skladá z troch častí: prvého riadka, hlavičiek a tela správy. Syntax prvého riadka závisí od toho, či je správa požiadavkou alebo odpoveďou, ale vždy končí sekvenciou bajtov s hodnotou 13 a 10. Druhá sekcia sú hlavičky; tie môžu byť aj na viacerých riadkoch a sú ukončené prázdnym riadkom. HTTP správa môže, ale aj nemusí obsahovať telo; to závisí na vyššie uvedených pravidlách. Všetky tieto pravidlá slúžia na *reassembling* HTTP správy po fragmentácii spôsobenej protokolom TCP.

3 Implementácia

V tejto kapitole by som chcel zhrnúť priebeh implementácie od základnej architektúry cez diagram tried až po použitie. Tiež vysvetlím akcie nástroja pri jednom páre požiadavky – odpovede. Moja implementácia je v prostredí C# .NET Framework 4.0 z dôvodu lepšej podpory vlákien.

3.1 Základná architektúra

Hlavná architektúra sa skladá z piatich modulov: rozkladu a analýzy HTTP správ, proxy, cache, načítavania HTTP správ z pcap súborov a vzdialeného ovládania. Každý z týchto modulov bude následne priblížená.

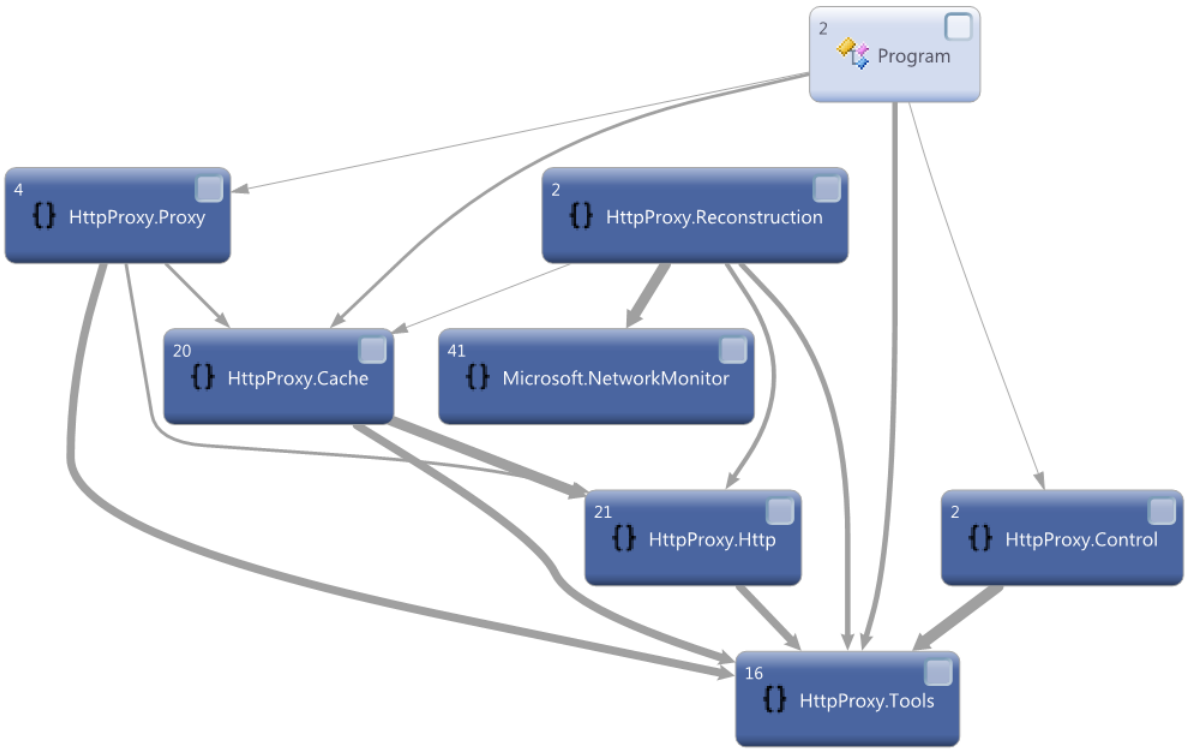
Analýza HTTP správ je implementovaná podľa RFC 2616 (a teda verzia HTTP/1.1) s podporou netradičnej notácie prvého riadka odpovede (podľa RFC za stavovým kódom nemusí byť uvedený žiaden dôvod), viacriadkových HTTP hlavičiek, neobvyklých techník zápisu HTTP hlavičiek (napr. názvy hlavičiek celé napísané malými písmenami), čiastočného kódovania obsahu a prenosu a podpory pre pripojenie proxy k inej proxy. Celý modul je rozložený do viacerých častí: načítanie prvého riadka, načítanie hlavičiek a načítanie obsahu správy. Načítanie prvého riadka a hlavičiek prebieha atomicky, ale obsah je rozdeľovaný podľa nastavenia nástroja a TCP fragmentácie, pre plynulé načítavanie väčších súborov (napr. pri načítavaní videa z *youtube.com*, nie je nutné čakať, kým sa načíta celé video aby sme si ho mohli prehrať).

Proxy časť sa stará o logiku obsluhy HTTP správ. Jej úlohou je prijímanie požiadaviek od klienta, ich preposielanie na server a následné preposielanie odpovedí zo serveru. Podporuje zreťazené požiadavky – odpovede, prepínanie medzi servermi na existujúcom klientovom spojení, obsluhu HTTP metódy *CONNECT,* obsluhu cache a pripojenie k ďalšej HTTP proxy (okrem HTTP metódy *CONNECT*).

Cache sa skladá z troch častí: databázy cache objektov, úložiska a cachovacej logiky. Databáza ukladá odpovede servera podľa metódy požiadavky, umiestnenia odpovede (získaného z prvého riadka požiadavky) a zoraďuje podľa času príchodu požiadavky. Prvý bod podporuje rozdelenie odpovedí podľa metód (napr. rôzne odpovede pre *GET* a *HEAD*), aj keď v súčasnej implementácii cache ukladá len odpovede na požiadavku *GET*. Úložisko je implementované ako perzistentné úložisko dôležitých informácií pre cache objekty. Prístup k týmto objektom je s viacnásobným prístupom pre čítanie a výhradným prístupom pre zápis. Po reštarte nástroja je teda schopné načítať objekty z predchádzajúcich spustení. Objekty sa ukladajú do hierarchie podľa kľúča *<ip_klienta>\<server>\<guid_objektu>.obj*, je teda jednoduché sledovať, kto, odkiaľ a aké objekty prezeral.

Načítavanie HTTP správ z pcap súborov sa skladá z dvoch častí: samotného načítavania HTTP dát a registrácie HTTP správ do cache. Prvá časť používa aplikačné rozhranie Microsoft Network Monitor k reassemblingu paketov, filtrovaniu HTTP správ a následnému získaniu HTTP správy z paketu. Tiež sa získavajú užitočné metainformácie ako IP adresa klienta. Druhá časť využíva modul proxy pre vytváranie párov a následné registrovanie odpovedí do cache.

Modul pre **vzdialené ovládanie** je vytvorený pre dynamickú zmenu istých parametrov nástroja. Avšak nie všetky parametre nástroja sa dajú zmeniť bez reštartovania nástroja alebo modifikácie zdrojového kódu. Vzdialené ovládanie je implementované ako prenos jednoduchého textu cez TCP protokol. Je teda možné interaktívne i automatizované vzdialené ovládanie.



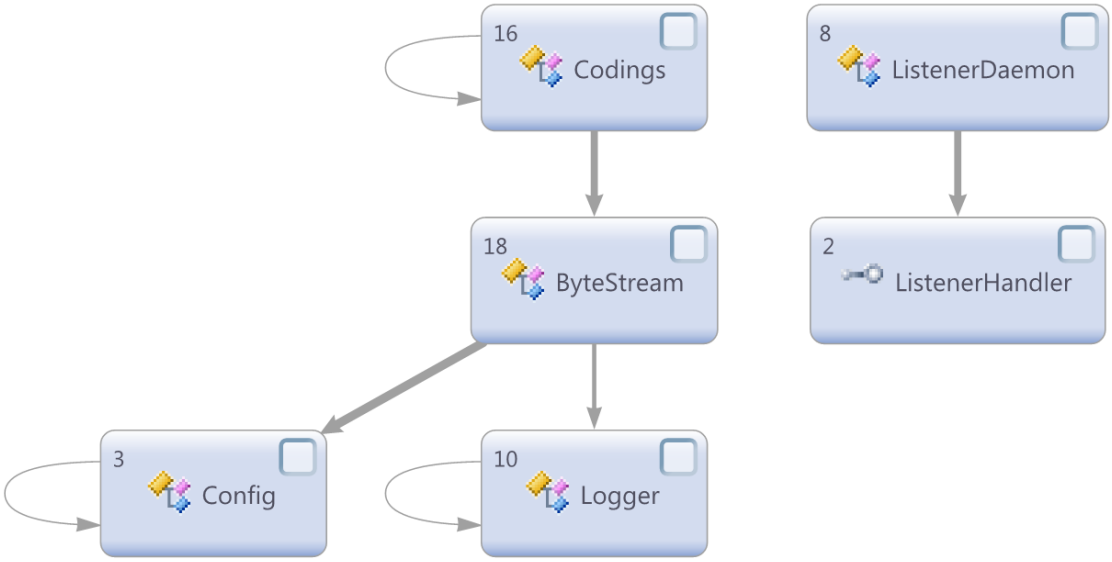
Obrázok 7: Závislosť modulov základnej architektúry

3.2 Hierarchia tried

Tak ako som v predošlej časti rozobral základnú architektúru a jej časti, v tejto podkapitole by som priblížil ich konkrétnu implementáciu. Pre každú z nich budú opísané základné triedy, ktoré sa podieľajú na funkcionalite daného modulu.

3.2.1 Pomocné triedy

Triedy spomenuté v tejto kapitole nespadajú svojou funkcionalitou ku žiadnemu modulu. Ich implementácia je zväčša zovšeobecnenie niektorých prvkov používaných v rekonštrukčnom nástroji. Patria sem triedy *Config*, *ByteStream*, *ListenerDaemon* spolu s rozhraním *ListenerHandler*, *Codings* a *Logger*.



Obrázok 8: Závislosť tried pomocných nástrojov

Trieda *Config* je typu *singleton*, ktorá zahŕňa všetky parametre, ktoré sa v nástroji používajú. Ich zhrnutie do jednej triedy umožňuje okamžitý účinok zmeny parametra pre celý nástroj. Väčšinu z týchto parametrov je možné meniť počas behu nástroja pomocou vzdialeného ovládania, ale pre zmenu niektorých je nutné nástroj zastaviť a zmeniť ho v zdrojovom kóde. Napríklad kódovanie znakov pre hlavičky HTTP správ sa v najbližšej dobe pravdepodobne nezmení, a preto nie je nutné to meniť počas behu.

Názov parametra	**Popis**
HTTP_MAX_LINE	Určuje, aká dlhý môže byť najdlhší riadok (prvý riadok i hlavičky). Tento parameter sa nedá meniť počas behu nástroja.
HTTP_HEADER_ENCODING	Určuje kódovanie znakov, ktorým sa majú odkódovať a potom zakódovať prvé riadky a hlavičky správ. Tento parameter má predvolenú hodnotu *ASCII* a nedá sa meniť počas behu programu (nemá ho zmysel meniť pokiaľ sa nezmení špecifikácia protokolu HTTP).

HTTP_MAX_BODY_LENGTH	Tento parameter slúži ako jednoduchá ochrana pred zaplňovaním pamäte veľkými správami. Jeho predvolená hodnota je 64MiB ale dá sa meniť počas behu programu (nemá zmysel meniť, ak sa už telo správy zahodilo). Jeho hodnota je v bajtoch.
FLOW_MTU	Určuje, aké maximálne veľké fragmenty správ sa majú načítavať do pamäte. Takisto ako predošlý parameter znižuje využívanie pamäte nástrojom, ale taktiež jeho nízka hodnota môže prispieť k veľkej fragmentácii správ. Jeho hodnota je v bajtoch a dá sa meniť cez vzdialené ovládanie.
PROXY_EN	Slúži len ako príznak, či je proxy zapnutá. Dá sa meniť cez vzdialené ovládanie.
PROXY_ADDR	Definuje adresu, na ktorej má proxy počúvať. Predvolene počúva na všetkých sieťových rozhraniach. Tento parameter sa dá meniť cez vzdialené ovládanie.
PROXY_PORT	Definuje port na ktorom má proxy počúvať. Tiež sa dá meniť cez vzdialené ovládanie.
PROXY_OUT_PROXY_EN	Tento parameter núti proxy pripájať sa na ďalšiu proxy namiesto na server určený HTTP správou. V tomto momente je podpora len pre HTTP proxy. Tento parameter sa dá meniť cez vzdialené ovládanie, avšak na jeho úspešné aktivovanie je nutné najprv nastaviť parametre PROXY_OUT_HOST a PROXY_OUT_PORT.
PROXY_OUT_HOST	Určuje adresu proxy, ku ktorej sa má nástroj pripájať. Dá sa meniť cez vzdialené ovládanie.
PROXY_OUT_PORT	Určuje port proxy, ku ktorej sa má nástroj pripájať. Tiež sa dá meniť cez vzdialené ovládanie.
CONTROL_EN	Indikuje, či je vzdialené ovládanie zapnuté. Z pochopiteľných dôvodov bola možnosť zmeny tohto parametra cez vzdialené ovládanie vypnutá. Namiesto toho, ak sa zmenia parametre CONTROL_ADDR alebo CONTROL_PORT, je vzdialené ovládanie automaticky reštartované.
CONTROL_ADDR	Určuje adresu na ktorej poslúcha vzdialené ovládanie. Jeho zmenou reštartuje samého seba.
CONTROL_PORT	Určuje port na ktorom poslúcha vzdialené ovládanie. Jeho zmenou reštartuje samého seba.
CONTROL_ENCODING	Definuje kódovanie znakov použité na vzdialené ovládanie. Jeho zmena môže byť dosť problematická, a preto na jeho zmenu by sme odporúčali používanie nástroja *nc* spolu s *iconv*.

CACHE_EN	Indikátor zapnutého cache. Existuje pravdepodobne len z návrhových dôvodov a do implementácie sa nedostal.
CACHE_STORAGE_ROOT	Určuje koreň úložiska. Tento parameter sa nedá meniť cez vzdialené ovládanie, pretože implementácia databáze neumožňuje vyhodenie objektov.
CACHE_LATEST_OBJECT	Súčasť funkcie na načítavanie objektov databázy do istého času príchodu odpovede. Nastavenie tohto parametra značí, že sa má brať do úvahy aktuálny čas, a teda všetky objekty databázy. Pre nastavenie iného času je nutné najprv nastaviť parameter CACHE_TIME_TOP a potom tento parameter zrušiť. Je tiež možné ho nastaviť na vyšší čas ako je aktuálny čas.
CACHE_TIME_TOP	Nastavuje vrchol databázy, z ktorého sa môžu brať objekty, a teda objekty ktoré majú neskorší príchod ako je daný čas, nie sú brané do úvahy.
CACHE_CACHE_CAPACITY	Tento parameter určuje kapacitu triedy *LRUCache*, ktorá určuje, ktoré objekty sa budú udržiavať v operačnej pamäti. Vzhľadom na implementáciu, ktorá preferuje rýchlosť vyhľadávania objektov, nie je možné meniť túto veľkosť počas behu programu.
CACHE_ONLY_IF_CACHED	Príznak vynútenej direktívy *Cache-Control: only-if-cached,* ktorá naznačuje, že sa má vrátiť požiadavka, iba ak je v cache a nemá sa pokúšať o kontaktovanie zdrojového servera. Tento parameter sa dá meniť cez vzdialené ovládanie.
TOOL_DATE_FORMAT	Jednotný formát dátumu, ktorý sa používa pre vytváranie a rozšifrovanie časových značiek v celej implementácii nástroja.
TOOL_IDENTIFICATION	Identifikácia nástroja. Používa sa pri hlavičke *Via*.

Tabuľka 6: Popis parametrov triedy Config

Trieda *ByteStream* poskytuje jednotné rozhranie pre súbory, sockety a bajtové polia, ktoré vystavuje ako prúd bajtov. Pre čítanie boli implementované dve metódy: *Read* a *ReadLine*. Obe vracajú pole načítaných bajtov do maximálneho počtu daného parametrom z triedy *Config*. Metóda *ReadLine* má tiež možnosť Zadať, aká sekvencia bajtov ukončuje riadok. Pre zápis do prúdu je vystavená metóda *Write*. Niektoré typy prúdov majú isté pravidlá pre čítanie a zápis; pre socket platí, že sa z neho môže vždy zapisovať aj čítať, pre súbor je nutné si vybrať medzi zápisom a čítaním a z poľa bajtov je možné len čítať (kvôli pamäťovým nárokom niektorých HTTP správ). Trieda *ByteStream* tiež vystavuje metódu *DataAvailable* ako test prítomnosti dát (v prípade socketu je nutné si dávať pozor, pretože dáta ešte nemuseli prísť, a teda to nie je dobrý test pri uzatváraní spojenia) a metódu *Close*, ktorá uzatvára socket alebo súbor a uvoľňuje vnútorný *buffer*.

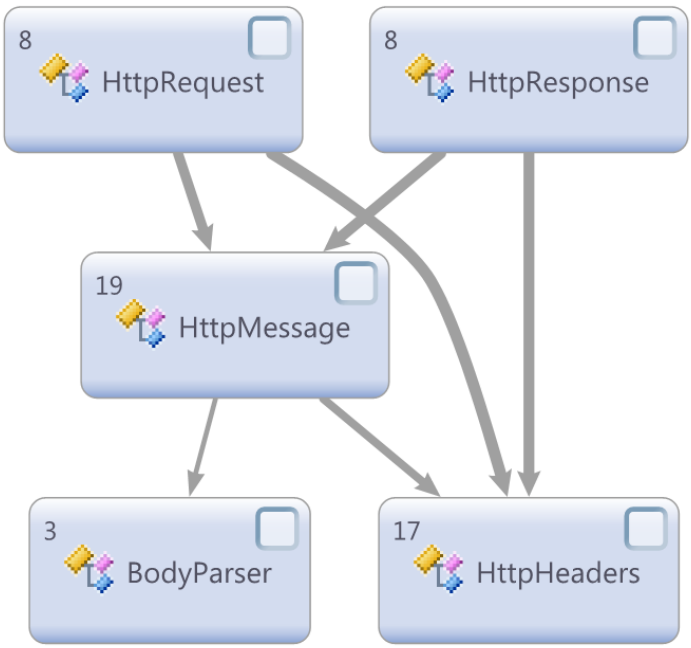
Trieda *ListenerDaemon* je abstraktná trieda, ktorá konkurentne prijíma požiadavky na spojenia na istom porte a za pomoci rozhrania *ListenerHandler* ich obsluhuje. Poskytuje metódy *Start*, ktorá vytvorí nové vlákno pre prijímanie požiadaviek a spustí ho na pozadí, *Restart*, ktorá len prehodí počúvanie na porte (ak je to nutné) a *Stop*, ktorá ukončí vlákno na pozadí. Prijaté požiadavky sa obsluhujú triedou implementujúcou rozhranie *ListenerHandler* definovanou v chránenej metóde *NewHandlerInstance*.

Trieda *Codings* je *singleton*, ktorý umožňuje odkódovanie a zakódovanie poľa bajtov. Táto trieda je implementovaná štýlom *facade*, a teda odhaľuje len dve metódy pre kódovanie: *Decode* a *Encode*. Typ kódovania sa určuje reťazcom znakov predaným v prvom parametri. Ak neexistuje implementácia kódovania pre daný reťazec, metóda vyhodí výnimku. Pre odkódovanie sú implementované kódovania *chunked*, *gzip*, *deflate* a z implementačných dôvodov i *identity*. Pre všetky kódovania okrem prvého je tiež implementované zakódovanie.

Pre triedu *Logger* je v danom momente implementácia nedokončená (tiež ale nebola vyžadovanou súčasťou nástroja). Jej myšlienka je ukladať informácie za behu programu do súboru, na výstup terminálu alebo cez sieť pomocou vzdialeného ovládania. Má štyri úrovne správ: *none*, *error*, *info* a *debug*. Pri výpise sa berie do úvahy úroveň správy; ak táto správa má nižšiu alebo rovnú úroveň, ako je maximálna povolená úroveň, vypíše sa do všetkých nakonfigurovaných výstupov (v súčasnej implementácii to je len štandardný výstup). Výstupy však môžu byť dosť chaotické, ak zoberieme do úvahy, že ide o viacvláknový nástroj. Preto sa pred samotnú správu tiež vypíše identifikácia vlákna.

3.2.2 Analýza HTTP správ

Analýzu HTTP správ vykonáva abstraktná trieda *HttpMessage*. Táto trieda poskytuje základnú analýzu a prístup k obsahu správy. K analýze správy z reťazca bajtov používa tri metódy: *ReadMessage*, *ReadHeaders* a *ReadBody*. Prvá metóda tiež zapuzdruje dve ďalšie. Metóda *ReadHeaders* načítava prvý riadok a hlavičky správy. Pre načítanie hlavičiek vytvára inštanciu triedy *HttpHeaders*. Metóda *ReadHeaders* je atomická, čiže buď sa vykoná naraz alebo vyhodí výnimku. Hlavičky sú uložené v dátovej štruktúre typu slovník, kde kľúčom je jej názov a hodnotou zoznam hodnôt všetkých jej výskytov. Tento zoznam zachováva poradie výskytu hodnôt hlavičiek [2] (sekcia 4.2). Keď sú načítané všetky, táto metóda tiež určí, ktorá implementácia triedy *BodyParser* sa má použiť. Metóda *ReadBody* má inú techniku; načíta telo správy, vyhodnotí, či nenastal koniec správy a podľa použitej implementácie vráti modifikované dáta, ktoré sa môžu poslať ďalej (od klienta na server alebo naopak). K tomu slúži abstraktná trieda *BodyParser*. V súčasnej implementácii sa však vrátia presne tie isté dáta, ktoré sa poslali na analýzu, takže trieda *BodyParser* slúži len na určenie konca správy.



Obrázok 9: Závislosť tried modulu pre analýzu HTTP správ

Abstraktná trieda *HttpMessage* má dvoch potomkov: *HttpRequest* a *HttpResponse*. Obe implementujú analýzu a prístup k dátam z prvého riadka podľa vlastných pravidiel takisto, ako určujú ktorá implementácia triedy *BodyParser* sa má použiť. Trieda *HttpResponse* má vlastný konštruktor, pri ktorom je nutné udať metódu požiadavky, na ktorý bola generovaná a je možné udať čas príchodu správy (pre uľahčenie implementácie cache mechanizmov a úložiska). Obe triedy modifikujú hlavičky tak, ako by to mala robiť proxy.

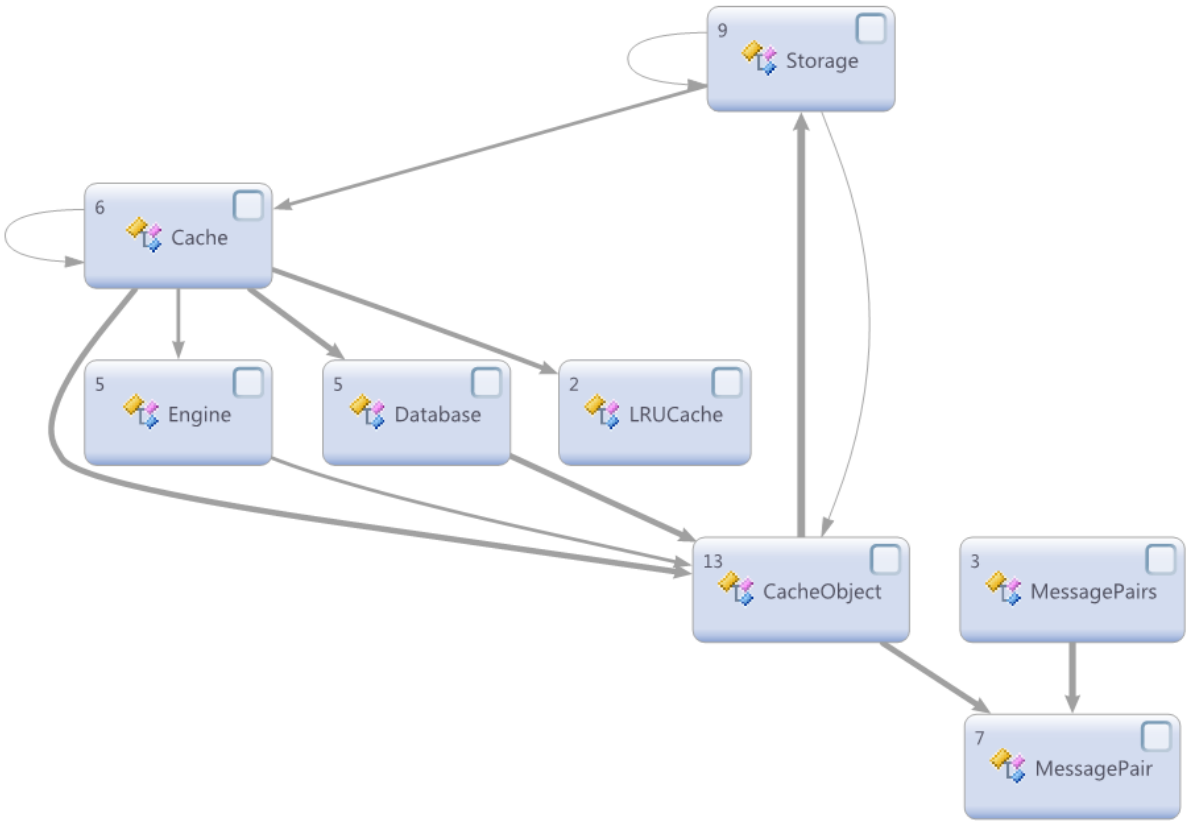
Trieda *HttpHeaders* implementuje analýzu a ukladanie hlavičiek do už spomínanej dátovej štruktúry typu slovník. Podporuje tiež načítavanie hlavičiek prenášaných na viacerých riadkoch a implementuje metódy pre bezpečný prístup na ich modifikáciu. Metóda *ReadHeaders* načítava hlavičky z bajtového prúdu, až kým nenarazí na ukončovaciu sekvenciu bajtov 2×CRLF (13, 10, 13, 10). Metódy *AddHeaders* pridávajú ďalšie hlavičky, metódy *ModifyHeaders* vymažú predošlý výskyt hlavičky a nahradia ho, ale iba v prípade, že hlavičky existovali a metóda *RemoveHeaders* zmaže všetky výskyty hlavičky, ak existovali.

Keďže každá hlavička má svoju vlastnú syntax, je lepšie mať pre každý typ hlavičky vlastný analyzátor. K tomu slúžia triedy *HeaderNumberParser*, *ListParser*, *RequestCacheControlParser*, *ResponseCacheControlParser* a *HTTPDateParser*. Trieda *HeaderNumberParser* je používaná ako rýchla konverzia medzi reťazcom znakov a číslom. Trieda *ListParser* sa používa pri analýze zoznamových hlavičiek, ako napríklad hlavičky *Content-Encoding*, ktorá obsahuje zoznam kódovaní aplikovaných na entitu. Triedy *RequestCacheControlParser* a *ResponseCacheControlParser* sú používané pre rýchle nastavenie príznakov hlavičky *Cache-Control* požiadavku, resp. odpovede, ktoré by inak bolo nutné zložito analyzovať na mieste. Trieda *HTTPDateParser* analyzuje dátum

z hlavičky *Date* alebo *Expires*. Je možné analyzovať dva typy formátov dátumu: podľa RFC 1123 [12] alebo RFC 850 [13]. Tieto triedy sa nepoužívajú pre ich meniacu sa implementáciu analýzy, a teda odpadáva povinnosť implementovať pre ne abstraktnú triedu alebo rozhranie, čo umožňuje analýzu priamo v konštruktore.

3.2.3 Cache

Modul cache predstavuje databázu všetkých odpovedí, ktoré vyhovujú istým pravidlám. Skladá sa z tried *Cache*, *CacheObject*, *MessagePairs*, *MessagePair*, *Database*, *Storage*, *Engine* a *LRUCache*.



Obrázok 10: Závislosť tried modulu cache

Trieda *Cache* slúži ako brána k objektom cache a podľa stanovených pravidiel uloží objekt do databáze alebo poskytne z nej objekt. Podľa toho vystavuje metódu *RegisterObject*, ktorá vloží objekt do databázy, ak vyhovuje kritériám, a *CacheHit*, ktorá poskytne objekt z databázy na danú požiadavku.

Trieda *CacheObject* predstavuje objekt databázy. Ten je zložený sčasti z požiadavky, času príchodu odpovede na proxy a z celej odpovede. Pre požiadavku sa ukladá metóda a absolútna URI. Pri vytvorení objektu sa odpoveď transformuje na odpoveď vo formáte *gzip* (kvôli miestu v pamäti), pokiaľ nie je definovaná hlavička *Cache-Control: no-transform*. Táto trieda poskytuje základné metódy pre prácu s objektom; uloženie objektu na perzistentné úložisko je možno vykonať metódou

Write, načítanie objektu z neho metódou *Read* a uvoľnenie pamäte po uložení metódou *DisposeFromMemory*. Taktiež poskytuje prístup k jednotlivým častiam objektu (metóda požiadavky, URI požiadavky, atď.).

Triedy *MessagePairs* a *MessagePair* vytvárajú páry požiadaviek - odpovedí spolu s dôležitými informáciami pre cache (IPv4 adresa klienta a doménové meno servera). Trieda *MessagePairs* je vytvorená ako dátová štruktúra radu, aby pomocou nej bolo možné vytvárať tieto páry pre zreťazené požiadavky.

Trieda *Database* predstavuje databázu cache objektov indexovanú podľa metódy požiadavky a URI a zoradenú podľa času príchodu odpovede. Týmto je možné obmedziť získateľné požiadavky pomocou parametra triedy *Config*. Metóda *Insert* vkladá objekt do databázy a index do triedy vracia objekt alebo vyhodí výnimku, ak objekt neexistuje.

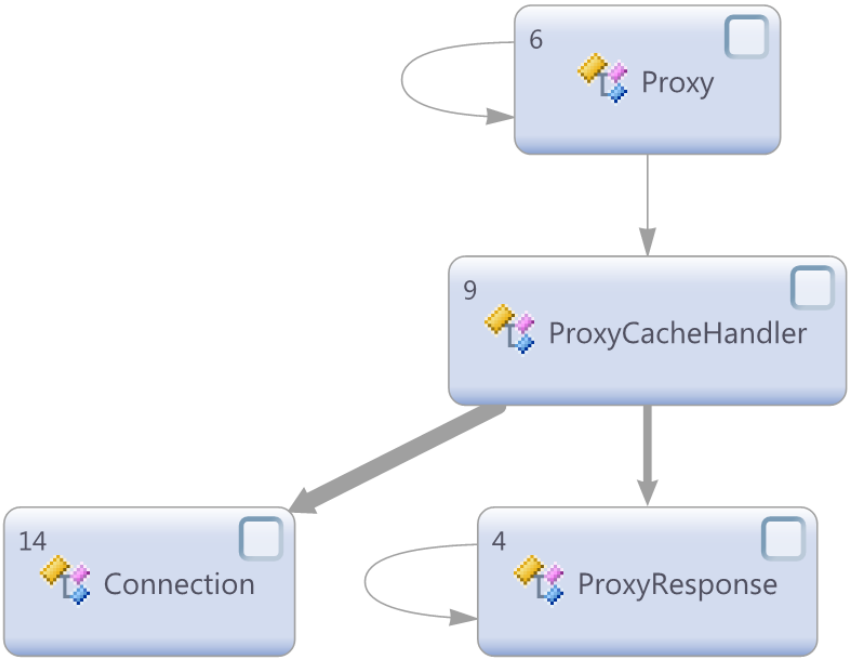
Trieda *LRUCache* je implementácia *Least-Recently-Used* cache pre cache objekty ktoré sa nachádzajú uložené v pamäti. Čím väčšiu má kapacitu, tým rýchlejšie je možné vrátiť odpoveď, ak dôjde k nájdeniu odpovede v cache. Tým sa ale kladú väčšie nároky na operačnú pamäť. Bohužiaľ, kapacita tejto cache sa nedá meniť počas behu programu, ale len zmenou zdojového kódu triedy *Config*. Poskytuje jedinú metódu *CacheObject*, ktorá buď načíta daný objekt do pamäte, vyhodí posledný objekt a uvoľní ho z pamäte a vloží nový objekt do cache, alebo presunie daný objekt v cache na prvé miesto.

Trieda *Storage* je implementácia úložiska pre viacvláknový proces. Umožňuje násobný prístup pre čítanie a výhradný prístup pre zápis. Metódy *OpenReadStream* a *OpenWriteStream* slúžia na otvorenie objektu triedy *ByteStream* súborového typu. Metódy *CloseReadStream*, *CloseWriteStream* a *RemoveFile* slúžia na jeho uzatvorenie, resp. vymazanie, ak sa detekovalo poškodenie. Vynútenie použitia rovnakého páru pre čítanie alebo zápis v danom vlákne je na úrovni výnimiek použitej triedy pre prístup k uzamknutým zdrojom. Táto trieda je schopná znovu obnoviť pôvodný stav celej cache po reštartovaní programu pomocou metódy *RestoreStorage*.

Abstraktná trieda *Engine* slúži ako vyhodnotenie objektov cache, či môžu byť uložené do cache, a k vyhodnoteniu, či daný objekt v cache sa môže vrátiť ako odpoveď, alebo je nutná jeho revalidácia. Podľa toho i poskytuje metódy *Cacheable* a *MustRevalidate*. Svoje správanie môže mať vytvorené užívateľom alebo dané dokumentom RFC. V danom momente existujú dve hlavné implementácie tejto triedy: *RFC2616Engine* a *StoreAllEngine*. Prvá implementuje do istej úrovne očakávané správanie proxy cache z dokumentu RFC 2616. Tá druhá rozširuje prvú tým, že ukladá všetky odpovede so stavovým kódom 200 alebo 203, bez ohľadu na to, aké cache mechanizmy si klient alebo server vynucujú v hlavičke *Cache-Control*, ale pri získavaní objektu z databázy tieto objekty označí ako neexistujúce. Týmto sa navonok správa ako prvá trieda, ale pritom ukladá všetky odpovede do úložiska.

3.2.4 Proxy

Modul proxy sa skladá z konkurentného servera, ktorý prijíma požiadavky klientov, a logiky pre obsluhu týchto požiadaviek. Jeho implementácia je teda odvodená od triedy *ListenerDaemon*. Tento modul obsahuje triedy *Proxy*, ktorá je potomkom už spomínanej triedy *ListenerDaemon*, *ProxyHandler*, ktorá implementuje rozhranie *ListenerHandler* a tried *Connection* a *ProxyResponse*.



Obrázok 11: Závislosť tried proxy modulu

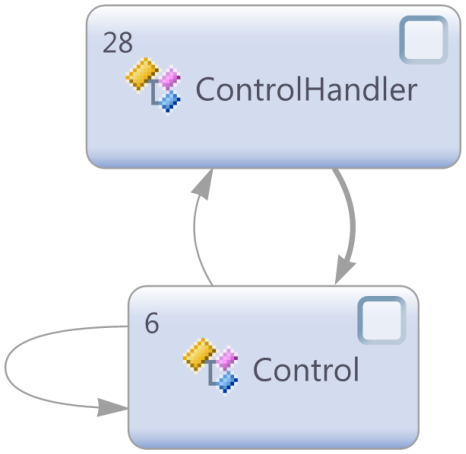
Triedy *Proxy* a *ProxyHandler* implementujú správanie triedy *ListenerDaemon*. Adresa a port na ktorom tento démon počúva je daný v triede *Config*. Trieda *ProxyHandler* používa pre obsluhu požiadaviek všetky doteraz spomenuté moduly. V metóde *Handle* si táto trieda načíta hlavičky požiadavky pomocou modulu na načítanie HTTP správ, použije metódu *CacheHit* triedy *Cache*, aby zistil prípadnú prítomnosť odpovede v cache. Podľa týchto možností sa potom rozhoduje ktorú metódu zavolať. Ak príde požiadavka na *CONNECT* použije metódu *HandleConnect*. Ak daná odpoveď existuje v cache a spĺňa požiadavky pre to, aby mohla byť použitá, zavolá sa metóda *HandleCacheHit*. V ostatných prípadoch sa volá metóda *HandleCacheMiss*. Táto metóda je tiež použitá pre revalidáciu odpovede v cache. V tomto prípade sa v objekte triedy *MessagePair* nastaví príznak, ktorý naznačuje, že ak príde odpoveď so stavovým kódom 304, majú sa aktualizovať hlavičky odpovede v objekte cache a môže sa použiť pre odpoveď klientovi. Spätný tok dát (zo serveru ku klientovi) sa obsluhuje komplementárnymi metódami s príponou *Downlink* okrem metódy *CacheHit*.

Trieda *Connection* zapuzdruje pripojenie na proxy, či už od klienta alebo na server. Udržiava informácie o tomto pripojení (hlavne perzistenciu) a poskytuje pomocné metódy pre načítanie a odoslanie hlavičiek správ. Zviditeľnené metódy pre ňu sú *ReadMessageHeaders* a *SendMessageHeaders*. V týchto metódach sa tiež upravujú hlavičky správ ktoré sú relevantné len pre jeden skok (hop-by-hop). Metóda *Connect* pripája daný objekt triedy k serveru. Je implementovaná tak, že umožňuje jej násobné volanie bez toho, aby sa viackrát za sebou pripájal k tomu istému serveru.

Trieda *ProxyResponse* je len pomocná, a používa sa pre generovanie proxy odpovedí, ako sú 200 na metódu *CONNECT*, 400, 500, 502 a 504.

3.2.5 Vzdialené ovládanie

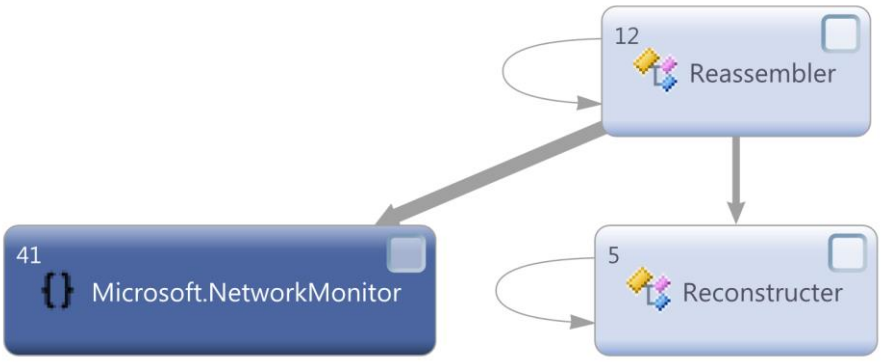
Modul pre vzdialené ovládanie je, podobne ako modul proxy, založený na abstraktnej triede *ListenerDaemon*. Triedy *Control* a *ControlHandler* sú ekvivalentom tried u proxy. Trieda *ControlHandler* však ďalej implementuje obslužné metódy pre každý podporovaný príkaz. Tento modul je síce implementovaný ako konkurentný, ale nie všetky jeho možnosti je bezpečné používať v paralelných vláknach. Riziko ale nie je až tak vysoké, keďže všetky operácie sú extrémne rýchle aj pre automatizované odosielanie príkazov. Tento modul pracuje natívne v kódovaní UTF-8, avšak je to možné tiež zmeniť na ASCII i priamo v danom nástroji. Podpora viacerých kódovaní sa dá doimplementovať.



Obrázok 12: Závislosť tried vzdialeného ovládania

3.2.6 Rekonštrukcia z pcap súborov

Rekonštrukcia pcap súborov prebieha za pomoci aplikačného rozhrania programu Microsoft Network Monitor v triedach *Reassembler* a *Reconstructer*. Využíva sa tu podobný mechanizmus ako pri module proxy.



Obrázok 13: Závislosť tried rekonštrukčného modulu a NetmonAPI

Trieda *Reassembler* ako jediná pracuje s NetmonAPI. Využíva ho k otvoreniu a analýze pcap súborov, reassemblingu paketov, filtrovaniu HTTP požiadaviek a odpovedí a načítaniu ďalších informácii, ktoré uľahčujú prácu s párovaním požiadaviek – odpovedí (HTTP conversation ID). Táto trieda vystavuje metódy pre inicializáciu NetmonAPI a pre uvoľnenie zdrojov po práci s ním, konkrétne *Init* a *Destroy*. Pre načítanie paketov z pcap súborov sa používa metóda *ParseFile*. Načítané HTTP správy potom predáva ako pole bajtov triede *Reconstructer*. Toto predávanie nie je chránené proti veľkým správam, a preto sa môže stať, že načíta správu veľkú 2GB, čo môže mať veľký vplyv na využívanie operačnej pamäte.

Na to nadväzuje trieda *Reconstructer*, ktorá používa HTTP conversation ID na identifikáciu HTTP komunikácie konkrétneho HTTP spojenia. Pri získaní správy ako poľa bajtov vytvorí nad týmto poľom inštanciu triedy *ByteStream*, z ktorej potom načítava správu pomocou tried *HttpRequest*, resp. *HttpResponse*. Pre spárovanie požiadaviek – odpovedí používa triedu *MessagePairs*, kde pre každé HTTP conversation ID je vytvorená jedna inštancia tejto triedy. Po vytvorení páru sa priamo vytvorí cache objekt. K tomu slúžia dve metódy: *AddRequest* a *AddResponse*.

3.3 Funkcie nástroja

Veľa z funkcií nástroja je roztrúsených po viacerých moduloch. V tejto kapitole by som ich chcel všetky zhrnúť a uviesť, ako ich nakonfigurovať, aby fungovali správne. Tiež by som uviedol viaceré funkcie, ktoré by sa dali doimplementovať a význam ich využitia.

3.3.1 Implementované funkcie

Jedna z najhlavnejších funkcií je implementácia reštrikcie objektov databáze na objekty, ktorých čas príchodu je skorší ako zadaný čas. Pri zadaní tohto času by sa celý nástroj mal tváriť, akoby zadaný čas bol aktuálny čas, avšak toto zasahuje do viacerých oblastí nástroja, a teda nemusí to byť úplne pravda. Tejto funkcii zodpovedajú parametre z triedy *Config* CACHE_LATEST_OBJECT

a CACHE_TIME_TOP. Prvý parameter určuje, či sa má brať do úvahy aktuálny čas, alebo čas uložený v druhom parametri. Tieto dva parametre sa v budúcnosti môžu spojiť a vytvoriť mechanizmus pre získanie pseudo-aktuálneho času, ktorý by sa mohol použiť namiesto klasického generátora aktuálneho času v .NET (*DateTime.UtcNow*).

V nástroji je tiež možné vynútiť správanie hlavičky *Cache-Control: only-if-cached*. Táto direktíva znamená, že nástroj má vrátiť odpoveď, iba ak je uložená v cache, a to aj keby jej platnosť dávno vypršala, a nepokúšať sa o kontaktovanie zdrojového servera. Využíva sa to hlavne pri pomalom spojení, keď odpoveď zo servera môže trvať veľmi dlho. Pre nás je však zaujímavá časť, ktorá umožňuje obísť pravidlá cachovacích mechanizmov pre vypršanie platnosti objektov. V našej cache je však treba rozoznávať dva typy tejto hlavičky: klasická direktíva pre cache, ktorá prišla v správe, a vynútená direktíva, nastavená v triede *Config*, pretože naša cache obsahuje i objekty, ktoré by normálne nemala. Preto v prvom prípade sa buď vráti platná odpoveď podľa cachovacích mechanizmov dokumentu RFC 2616, alebo žiadna, a v druhom prípade sa môžu vrátiť všetky objekty implementovanej cache. Je treba tiež dávať pozor, pretože objekty z druhého prípadu môžu obsahovať i citlivé informácie, preto je lepšie zastaviť proxy, aby sa ukončili všetky spojenia, a nastaviť parameter PROXY_ADDR z triedy *Config* na konkrétnu adresu vnútornej siete bez prístupu nepovolaných osôb.

Nástroj umožňuje tiež pripojenie k ďalšej proxy. V súčasnej implementácii je však podporovaná len HTTP proxy. Vďaka tejto funkcionalite je možné pripojiť nástroj k proxy s inou funkcionalitou, než je zákonné odpočúvanie, napr. rýchlejší a presnejší cachovací mechanizmus. K tejto funkcii sa viažu tri parametre triedy *Config*: PROXY_OUT_PROXY_EN, PROXY_OUT_HOST a PROXY_OUT_PORT.

Vzdialené vládanie umožňuje tiež načítavanie pcap súborov počas behu programu. Tento spôsob je však nebezpečný, pretože nie je možné zistiť, kedy správy v pcap súbore prišli, a zatiaľ sa používa čas, keď bola odpoveď načítaná, čo môže viesť k zastaraným odpovediam, podľa toho, kedy bol pcap súbor zachytený. Táto funkcia bola skôr myslená ako alternatíva nutnosti reštartovať nástroj kvôli načítaniu ďalších pcap súborov.

3.3.2 Ďalšie možné funkcie

Ako prvú funkciu, ktorá by sa dala v budúcnosti implementovať, by som uviedol obmedzenie objektov cache databázy na objekty jedného klienta. Keďže tento nástroj je vytvorený pre zákonné odpočúvanie, bolo by praktické vidieť len správy konkrétneho užívateľa v konkrétnom čase (čas je už implementovaný pomocou jeho obmedzenia v objektoch databázy). Tieto funkcie by spolu s vynúteným *only-if-cached* umožňovali odpočúvanie nielen všetkej komunikácie cez protokol HTTP, čo môže byť celkom chaotické, ale aj odpočúvanie jedného konkrétneho klienta, u ktorého sa zaznamenali podozrivé HTTP správy.

K trojici predošlých funkcií nástroja by sa tiež dala vytvoriť funkcia, ktorá by presne informovala administrátora, akí klienti sú odpočúvaní, čo navštívili a kedy. Toto by sa dalo implementovať cez vzdialené ovládanie, kam by sa pridali dva príkazy; jeden na zoznam odpočúvaných klientov a druhý na zoznam objektov, ktoré sú v databáze spolu s informáciami o klientovi, ktorý daný objekt prijal, a časom, kedy bol objekt načítaný. Druhý zoznam by sa dal ďalej filtrovať podľa klienta a času. Týmto by sa uľahčilo zadávanie správneho klienta a času pre zobrazenie istého objektu.

Ďalšou funkciou by som nadviazal na koniec predošlej kapitoly. V súčasnej implementácii existuje len jedna inštancia proxy a jedna inštancia cache (ktorá ďalej zastrešuje úložisko). Ak by sa vytvorilo viac inštancií oboch častí, pričom nad jedným úložiskom by mohli pracovať viaceré proxy, mohol by sa nástroj stať veľmi flexibilným. Napríklad by bolo možné mať jedno hlavné úložisko, nad ktorým by pracovala proxy pre obsluhu požiadaviek klientov a zároveň by nad ním mohla pracovať proxy s vynúteným *only-if-cached*. V tomto prípade by sa tiež vytvorilo ďalšie úložisko pre proxy s rekonštrukciou pcap súborov, aby neinterferovalo s úložiskom pre obsluhu klientov a oni náhodou nedostali neplatné odpovede. Problémom by bola administrácia týchto inštancií a ich vzájomné dynamické prepojovanie.

Medzi funkcie by sa dala pridať i jednoduchá ochrana pred *DoS* (denial of service) útokmi. Vytvorila by sa nárazová cache pred proxy, ktorá by kontrolovala, či daná požiadavka sa pre daného klienta už obsluhuje, pričom pri viacerých požiadavkách na ten istý zdroj od toho istého klienta sa prvý obslúži a ostatné sa zahodia. Problém pri tomto riešení je, že sa môžu generovať požiadavky na ten istý zdroj ale s inou hlavičkou *Authentication* alebo *Cookie*, čo môže viesť k odmietnutiu legitímnych požiadaviek, takže by bolo nutné vytvoriť mechanizmus, ktorý by buď veľmi rýchlo zabúdal na prichádzajúce požiadavky, čo je proti filozofii ochrany proti DoS útokom, alebo by načítaval všetky hlavičky a filtroval ich podľa relevancie (napr. vyššie spomenuté hlavičky), čo je však veľmi komplexné, náchylné na chyby a porovnávanie pravdepodobne zaberie veľa strojového času.

Posledná z väčších funkcií nástroja, ktoré by sa v budúcnosti mohli implementovať je nadviazanie kontextu pri príchode poškodených HTTP správ. V súčasnej implementácii, ak príde poškodená správa, proxy to len ohlási stavovým kódom 400, resp. 502, podľa toho či je poškodená požiadavka alebo odpoveď. Avšak vo vyrovnávacej pamäti môže ostať zvyšok správy, čo môže ovplyvniť následné správy, ak sa využije to isté spojenie. Táto funkcia by sa pokúšala nájsť začiatok ďalšej HTTP správy vo vyrovnávacej pamäti, alebo by ho vyprázdnila, čo by umožnilo znovu použiť toto spojenie.

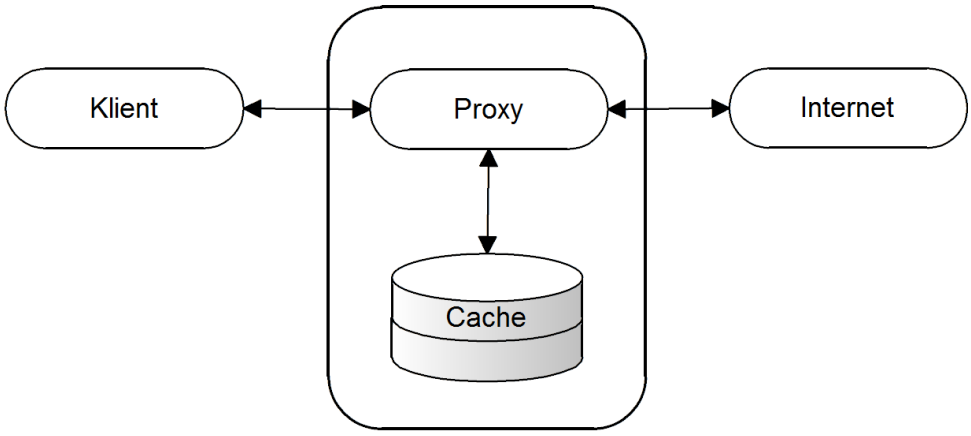
Týmto zoznam možných budúcich rozšírení nekončí. Dali by sa doň implementovať ďalšie, ako napríklad viacnásobný výstup triedy *Logger*. Táto funkcia ale získa vyššiu prioritu, keď bude rozsah implementácie väčší a hľadanie chýb v implementácii bude ťažšie. Do zoznamu môže byť uvedené tiež odstraňovanie poškodených záznamov v cache a znemožnenie ukladania duplikovaných

záznamov, najmä ak sa načíta jeden pcap súbor viackrát. Tiež by som tu mohol uviesť prevenciu sieťových slučiek, v ktorých by správa mohla cestovať donekonečna, kontrolovaním hlavičiek *Via*. Tento problém tiež nemá vysokú prioritu, keďže oba najpoužívanejšie protokoly sieťovej vrstvy, IPv4 a IPv6, majú túto podporu cez políčko TTL (time to live).

3.4 Use-case

Tento nástroj sa dá sa použiť dvomi vzájomne sa vylučujúcimi spôsobmi a jedným nezávislým spôsobom: ako obyčajná HTTP proxy v normálnom móde, v móde s vynútenou hlavičkou *Cache-Control: only-if-cached* a nástroj na rekonštrukciu webovej komunikácie z pcap súborov. Výstupmi oboch je zaplnenie modulu cache a hlavne úložiska relevantnými HTTP odpoveďami, ktoré neskôr môžu byť načítané proxy nástrojom v druhom móde.

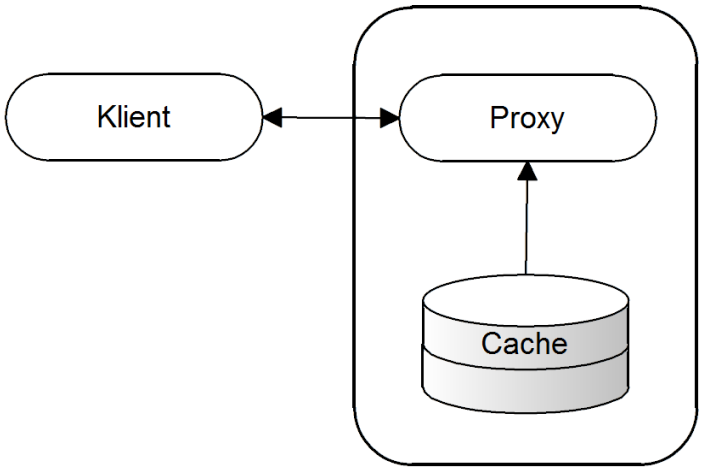
Pre prvé dva spôsoby je nutné mať aktivovanú proxy, či už pri štarte nástroja pomocou argumentov príkazového riadku alebo cez vzdialené ovládanie. Prvý spôsob predstavuje klasickú transparentnú proxy, na ktorú je nutné konfigurovať klienta, aby bol schopný pripojiť sa k serverom. Na obrázku 2 je znázornený príklad použitia ako proxy. V tomto móde klient žiada proxy aby predala požiadavky serveru. Ak bola požiadavka metódou GET a ak server odpovie stavovým kódom 200 alebo 203, proxy uloží túto odpoveď ako jednu z odpovedí, ktorú daný klient prijal. Použitá implementácia triedy *Engine* (*StoreAllEngine*) pre cache však znemožňuje použiť túto odpoveď ako odpoveď na ďalšie požiadavky v tomto móde.



Obrázok 14: Tok dát pri použití nástroja ako transparentnej HTTP proxy

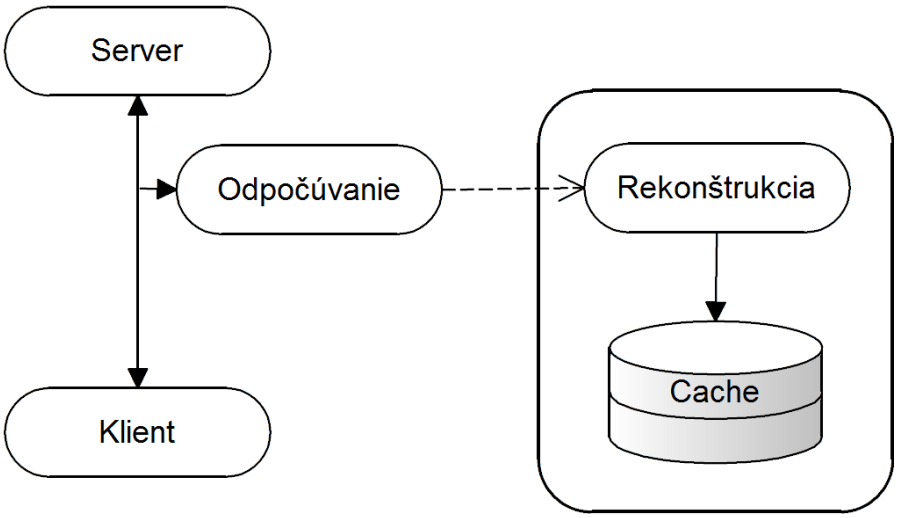
Nástroj sa dá použiť aj pre rekonštrukciu webovej komunikácie v móde s vynúteným *Cache-Control: only-if-cached*. Toto vynútenie predchádza mechanizmy použitej implementácie triedy *Engine* (*StoreAllEngine*) a tým umožňuje zobrazenie i odpovedí, ktoré by sa nemali v cache nachádzať. To tiež prináša nebezpečenstvo, že odpovede, ktoré sú v cache uložené, nemusia byť aktuálne, alebo môžu obsahovať citlivé informácie. Ak sa využije funkcia nástroja pre obmedzenie

vrcholu cache databázy, je možné získať staršie správy, než tie, ktoré boli prijaté ako posledné. V momentálnej implementácii nie je možné objekty databázy obmedziť na objekty jedného klienta, preto ak sa tento mód využije, objavia sa stránky všetkých užívateľov. Táto funkcia by tiež vyžadovala implementáciu pomocných funkcií vo vzdialenom ovládaní, a to konkrétne výpis všetkých klientov, možnosť obmedzenia databázy na konkrétneho klienta a výpis navštívených stránok pre tohto klienta najlepšie zoradených podľa času navštívenia.



Obrázok 15: Tok dát pri využití proxy v only-if-cached móde

Posledná možnosť využitia nástroja je načítavanie odpovedí do úložiska z pcap súborov. V rámci projektu *Sec6net* sa implementujú sondy, ktoré zachytávajú komunikáciu na istom mieste. Tento nástroj je schopný načítať tento výstup a rekonštruovať z neho webovú komunikáciu. Tá sa potom ukladá do cache a hlavne do úložiska. Tento mód sa dá rovnako ako predošlé dva použiť buď pri štarte nástroja, alebo pomocou vzdialeného ovládania. Je dobré podotknúť, že inicializácia rekonštrukcie trvá dlhšiu dobu hlavne kvôli načítaniu analyzátorov aplikačného rozhrania Microsoft Network Monitor, ktoré je po použití treba uvoľniť. Najrýchlejší spôsob využitia tohto módu teda je teda načítať čo najviac pcap súborov naraz.



Obrázok 16: Tok dát pri použití nástroja pre rekonštrukciu webovej komunikácie

4 Testovanie

V poslednej kapitole by sa prebralo otestovanie nástroj ako z funkcionálnej stránky tak z výkonnostnej. Pre testy funkcionality boli vytvorené sady automatických testov. Testy tejto sady sa dajú požiť aj pre budúce testovanie nástroja pri zmene implementácie ako testy, ktoré testujú, či pôvodná funkcionalita ostala nedotknutá. Výkonnostné testy sa dajú v budúcnosti použiť pre dolaďovanie rýchlosti obsluhy jednotlivých požiadaviek.

4.1 Funkcionálne testovanie

Pre otestovanie funkcionality bola vytvorená sada testov, ktoré testujú hlavne spracovávanie HTTP správ. Táto sada je priložená k nástroju. Každý test obsahuje štyri súbory:

- **desc** – súbor s popisom, čo daný test testuje,
- **req** – požiadavka, ktorá sa má poslať nástroju,
- **res** – odpoveď, ktorá sa má poslať nástroju na danú požiadavku,
- **stat** – stavový kód, ktorý sa očakáva v odpovedi od nástroja. Tento kód nemusí byť totožný so stavovým kódom v súbore *res*, ak proxy odpovedá vlastnou správou.

K odoslaniu správ je nutné mať nástroj, ktorý je schopný odoslať text cez TCP. K tomuto môže slúžiť nástroj *telnet* alebo *nc*. Pre prijatie odpovede zo súboru *res* je dobré využiť funkcie nástroja pre pripojenie k ďalšej proxy, kde nástroj môžeme presmerovať k počúvajúcemu nástroju *nc*. Príklad využitia je v pridanom skripte pre nástroj *bash* s názvom *test*. Po vykonaní testu týmto skriptom sa vytvoria ešte dva súbory:

- **req.out** – predstavuje odpoveď nástroja na požiadavku,
- **res.out** – prestavuje požiadavku klienta po prechode nástrojom.

V súčasnej implementácii je vytvorených šesť sád testov:

1. **uri** – táto sada testuje rôzne kombinácie adresy zdrojov pre požiadavky. Nástroj, aj napriek tomu, že je proxy, musí okrem absolútnej URI podporovať i formát absolútnej cesty z URI, kvôli načítavaniu pcap súborov, a formátu autority URI, pre metódu *CONNECT*. Táto sada odhalila chybu, kde adresa zdroja je vo formáte hviezdičky (*), ktorý sa používa pri adresovaní nástroja namiesto konkrétneho zdroja v metóde *OPTIONS*. Podpora pre tento formát URI nebola nakoniec ani implementovaná kvôli chýbajúcej definícii obsluhy tejto možnosti pre proxy v dokumente RFC 2616,
2. **method** – sada testuje všetky povolené kombinácie metód a HTTP verzií,
3. **status** – kontroluje, či obsluha stavových kódov funguje tak, ako by mala. Táto sada bola vyvinutá hlavne preto, lebo definícia prvého riadku odpovede z RFC 2616 povoľuje

vynechanie dôvodu, avšak je nutné ponechať medzeru medzi stavovým kódom a dôvodom,

4. **headers** - táto sada testuje rôznu syntax hlavičiek a prevod z neštandardnej notácie, ak sa vyskytne. Testy v nej testujú hlavne:
   - neštandardné vynechávania medzier alebo ich pridávanie, ktoré niektoré implementácie HTTP protokolu nemusia podporovať (je na to upozornenie v dokumente RFC 2616), a ich štandardizovanie po prechode nástrojom,
   - hlavičky na viacerých riadkoch. Priamo v RFC 2616 je uvedené, že táto možnosť je podporovaná len kvôli dlhým hlavičkám pre implementácie, ktoré nedokážu spracovať veľmi dlhý riadok (napr. pascal s 256 znakov dlhým reťazcom), a je explicitne vyjadrené, že tejto možnosti by sa malo vyhýbať, pokiaľ to ide. Táto možnosť je testovaná hlavne z pohľadu spätnej transformácie na jeden riadok a štandardného formátu,
   - neplatné hlavičky, ako napríklad hlavičky bez názvu, alebo bez oddeľujúcej dvojbodky.

   Prvé dve možnosti sú testované s rôznym rozložením bielych znakov. Tieto testy odhalili chyby v načítavaní, ako prečítanie hlavičky bez názvu, alebo chyba pri načítaní viacriadkovej hlavičky, kde oddeľujúca dvojbodka je na druhom riadku,
5. **body** – táto sada testuje, či správne funguje načítavanie tela správy podľa pravidiel v kapitole 2. Taktiež kontroluje, či správne funguje prioritizácia použitých implementácii abstraktnej triedy *BodyParser* podľa zadaných pravidiel.
6. **chunked** – táto sada slúži na otestovanie spracovania správ v kódovaní *chunked*, hlavne rozšírenia chunku a hlavičiek na konci správy.

4.2 Výkonnostné testovanie

Výkonnostné testovanie overovalo rýchlosť spracovania požiadaviek nástrojom. Pre každý test bol na porovnanie vykonaný tiež test priamym spojením., ktoré sa simulovalo *port-forwardingom* pomocou nástroja *netsh*. Toto porovnávanie je v podstate porovnanie týchto dvoch nástrojov. Spúšťanie testov a ich obsluha bola vykonávaná na dvoch rozdielnych strojoch. Nástroj bežal vo VirtualBoxe na operačnom systéme Microsoft Windows 7 32bit, so zdieľanými všetkými jadrami procesoru Intel Core i3-2310M na maximálnom výkone a so zdieľanou operačnou pamäťou o veľkosti 2500MB. Testovanie prebiehalo po samostatnom UTP kábli medzi týmito dvoma strojmi, takže interferencia s inou sieťovou komunikáciou bola minimalizovaná. V oboch strojoch boli ponechané len programy, ktoré boli nutné k úspešnému vykonaniu testov. Nástroj mal tiež vypnutý modul cache, aby nedochádzalo k odpovedaniu priamo nástrojom.

Boli vykonané 4 výkonnostné testy: 10 sériových požiadaviek s odpoveďou s veľkosťou 10MB (test č.1), 10 paralelných požiadaviek s odpoveďou s veľkosťou 10MB (test č.2), 250 sériových požiadaviek len s hlavičkami (test č.3) a 250 paralelných požiadaviek len s hlavičkami (test č.4). Odpoveď s veľkosťou 10 MB bola generovaná zariadením */dev/random*.

Testy 10 požiadaviek s odpoveďou s veľkosťou 10MB boli vykonané pre otestovanie rýchlosti načítavania veľkých HTTP správ. Obe čísla boli vybraté po opakovaných pokusoch o najlepšiu kombináciu pre obe sériové a paralelné odosielanie. Pre testy s 250 požiadavkami bolo pôvodne myslených 1000 požiadaviek, ale pri tomto čísle sa mi podarilo úspešne preťažiť stroj generujúci správy pri paralelnom spracovaní. Preto bolo toto číslo nakoniec znížené.

Paralelné testy sú svojou podstatou DoS útokmi, a keďže nástroj nemá pred nimi úplnú ochranu, mohol z času na čas spadnúť. Týmto testami sa odhalila väčšina chýb v návrhu a implementácii. Kompletné výsledky meraní sú uvedené v prílohe č.1.

Číslo testu	**Priemerné trvanie cez nástroj [s]**	**Priemerné trvanie priamo [s]**
1	82.689	5.383
2	30,530	3,018
3	4.559	24.429
4	8,694	3,402

Tabuľka 7: Priemerné trvanie testov

4.2.1 Záver testovania

Ako môžeme vidieť z nameraných dát pre veľké správy, implementovaný nástroj v tomto smere veľmi zaostáva z pohľadu výkonu, najmä sériové spracovanie. V paralelnom spracovaní dlhých správ sa to už lepší. Toto môže byť dôsledok použitých synchrónnych operácií pre načítavanie zo socketu a zapisovanie do socketu, ktoré brzdia následnú analýzu správ, alebo použitia implementácií abstraktnej triedy *BodyParser*, ktorá určuje dĺžku správy.

Pri správach bez tela sa výkon lepší, dokonca pri sériovom teste predbieha nástroj *netsh* niekoľkonásobne, čo môže byť spôsobené faktom, že implementovaný nástroj je konkurentný server, takže pri sériovom spracovaní sa nemusí čakať, aby sa rozviazalo staré spojenie predtým, ako sa nadviaže nové spojenie. Tiež to môže byť zlým spracovaním vyššieho počtu spojení nástrojom *netsh*, keďže v sériovom aj paralelnom spracovaní viacerých správ sa jeho priemerný čas zvyšuje s narastajúcim počtom spojení, pri sériovom spracovaní celkom signifikantne, a to aj napriek tomu, že v prvých dvoch prípadoch bolo prenesených omnoho viac dát ako v druhých dvoch.

Vylepšenie implementovaného nástroja podľa týchto výsledkov by malo hlavne prísť v smere rýchlosti spracovania veľkého množstva dát, zlepšeného systému ukladania cache objektov do databázy a zlepšeného systému využívania operačnej pamäte.

5 Záver

V tejto práci bol analyzovaný protokol HTTP hlavne z pohľadu proxy, o čom referuje hlavne kapitola 2. Podľa toho bol navrhnutý a implementovaný nástroj, ktorý je schopný načítavať pcap súbory, presmerovávať HTTP požiadavky ako proxy a interpretovať webové stránky pomocou prehliadača. Tento nástroj by mal byť nakoniec integrovaný ako súčasť projektu *Moderní prostředky pro boj s kybernetickou kriminalitou na Internetu nové generace – Sec6net*, ktorá zahŕňa viac nástrojov pre detekciu kybernetickej trestnej činnosti vyvíjaných na FIT VUT.

Funkcie implementovaného nástroja spĺňajú požiadavky a niektorých oblastiach ich dokonca predháňajú, ako napríklad pripájanie k ďalšej proxy, správa operačnej pamäte, limitovanie cache databázy podľa istých kritérií, ovládanie a zmena konfigurácie nástroja počas behu, štandardizovanie HTTP správ po ich prechode nástrojom, a ďalšie iné. Avšak ako bolo dokázané v kapitole o testovaní, v praxi to nebude stačiť, a je nutné implementovať ďalšie funkcie predtým, ako sa nástroj uvedie do prevádzky, k čomu patrí hlavne zrýchlenie spracovania veľkého objemu HTTP dát a zlepšenie správy pridelenej operačnej pamäte.

Nástroj bol otestovaný dvomi sadami testov vyvinutými priamo preň; jedna sada testuje jeho vnútornú funkcionalitu, zatiaľ čo tá druhá testuje jeho rýchlosť. Výsledky týchto testov overili jeho základnú funkcionalitu a schopnosť odpovedať i na väčšie množstvo požiadaviek. Tieto sady sa dajú použiť v budúcnosti pre overenie pôvodnej funkcionality nástroja alebo dolaďovanie jeho rýchlosti. Nástroj bol tiež odskúšaný na reálnej webovej komunikácií viacerých ľudí počas dvoch dní.

Nástroj sa dá v blízkej budúcnosti rozšíriť o mnohé funkcie spomenuté v kapitole 3.3.2 ako sú ďalšie limitovanie cache databázy, konkrétne na objekty konkrétneho klienta, nové príkazy do vzdialeného ovládania, ktoré by informovali administrátora o zozname objektov uložených v databáze, inštancovanie proxy a cache a ich vzájomné dynamické prepojovanie, jednoduchá DoS ochrana, ktorá by sa čiastočne mala vyriešiť zrýchlením obsluhy požiadaviek, alebo ďalšie menšie rozšírenia ako rozšírenie triedy *Logger*, odstraňovanie poškodených a zakázanie vkladania duplikátnych cache objektov do databázy alebo prevencia sieťových slučiek.

6 Bibliografia

[1] **Olbert, Jakub.** Analýza a rekonstrukce webového provozu, diplomová práce. Brno : FIT VUT v Brně, 2012.

[2] **Fielding, R.** Hypertext Transfer Protocol -- HTTP/1.1. [Online] June 1999. [Dátum: 12. 05 2013.] http://www.w3.org/Protocols/rfc2616/rfc2616.html.

[3] **Wikipedia.** Proxy server. [Online] 12. May 2013. [Dátum: 12. 05 2013.] http://en.wikipedia.org/wiki/Proxy_server.

[4] **Internet Assigned Numbers Authority.** MIME Media Types. [Online] 30. 01 2013. [Dátum: 12. 05 2013.] http://www.iana.org/assignments/media-types.

[5] **Internet Assigned Numbers Authority.** Hypertext Transfer Protocol (HTTP) Parameters. [Online] 15. 01 2013. [Dátum: 12. 05 2013.] http://www.iana.org/assignments/http-parameters/http-parameters.xml.

[6] **World Wide Web Consortium.** Efficient XML Interchange (EXI) Format 1.0. [Online] 2011. [Dátum: 12. 05 2013.] http://www.w3.org/TR/exi/.

[7] **John Rose, Kumar Srinivasan.** JSR-000200 Network Transfer Format for JavaTM Archives. [Online] 30. 09 2004. [Dátum: 12. 05 2013.] http://jcp.org/aboutJava/communityprocess/final/jsr200/index.html.

[8] **Crocker, David H.** RFC 822 - STANDARD FOR THE FORMAT OF ARPA INTERNET TEXT MESSAGES. [Online] 13. August 1982. [Dátum: 12. 05 2013.] http://tools.ietf.org/html/rfc822.

[9] **Deutsch, P.** RFC 1950 - ZLIB Compressed Data Format Specification version 3.3. [Online] May 1996. [Dátum: 12. 05 2013.] http://www.ietf.org/rfc/rfc1950.txt.

[10] **Deutsch, P.** RFC 1951 - DEFLATE Compressed Data Format Specification version 1.3. [Online] May 1996. [Dátum: 12. 05 2013.] http://www.ietf.org/rfc/rfc1951.txt.

[11] **Deutsch, P.** RFC 1952 - GZIP file format specification version 4.3. [Online] May 1996. [Dátum: 12. 05 2013.] http://www.ietf.org/rfc/rfc1952.txt.

[12] **Internet Engineering Task Force.** RFC 1123 - Requirements for Internet Hosts -- Application and Support. [Online] October 1989. [Dátum: 12. 05 2013.] http://www.ietf.org/rfc/rfc1123.txt.

[13] **Horton, Mark R.** RFC 850 - Standard for Interchange of USENET Messages. [Online] June 1983. [Dátum: 12. 05 2013.] http://www.ietf.org/rfc/rfc850.txt.

[14] **Wikipedia.** Transmission Control Protocol. [Online] 12. 05 2013. [Dátum: 12. 05 2013.] http://en.wikipedia.org/wiki/Transmission_Control_Protocol.

Zoznam príloh

Výsledky výkonnostných testov

Číslo spustenia	Trvanie cez nástroj [s]	Trvanie priamo [s]
1	105.301	5.101
2	81.250	5.018
3	91.652	5.362
4	87.527	4.936
5	69.970	5.584
6	78.639	5.858
7	71.945	6.711
8	96.643	5.179
9	81.409	5.289
10	62.558	4.790
Priemer	82.689	5.383

Tabuľka 8: Trvanie testov pre 10 sériových požiadaviek s odpoveďou 10MB

Číslo spustenia	Trvanie cez nástroj [s]	Trvanie priamo [s]
1	28.419	3.127
2	30.161	3.273
3	26.132	2.980
4	24.182	3.017
5	26.092	2.950
6	28.890	2.930
7	23.916	2.959
8	41.295	2.983
9	42.536	3.001
10	33.677	2.958
Priemer	30,530	3,018

Tabuľka 9: Trvanie testov pre 10 paralelných požiadaviek s odpoveďou 10MB

Číslo spustenia	**Trvanie cez nástroj [s]**	**Trvanie priamo [s]**
1	4.315	20.169
2	4.514	25.069
3	4.816	12.076
4	4.577	18.434
5	4.291	27.299
6	5.044	27.995
7	4.628	22.785
8	4.649	30.999
9	4.238	31.928
10	4.515	27.531
Priemer	4.559	24.429

Tabuľka 10: Trvanie testov pre 250 sériových požiadaviek bez obsahu

Číslo spustenia	**Trvanie cez nástroj [s]**	**Trvanie priamo [s]**
1	9.202	3.251
2	7.342	3.313
3	8.772	3.140
4	8.894	3.066
5	8.702	3.616
6	8.479	3.955
7	8.603	2.971
8	9.126	3.317
9	8.736	3.708
10	9.079	3.681
Priemer	8,694	3,402

Tabuľka 11: Trvanie testov pre 250 paralelných požiadaviek bez obsahu

Zoznam použitých skratiek

HTTP – Hypertext Transfer Protocol

TCP – Transmission Control Protocol

HTML – Hypertext Markup Language

IETF – Internet Engineering Trask Force

W3C – World Wide Web Consortium

WWW – World Wide Web

RFC – Requests for Comments

UDP – User Datagram Protocol

URI – Uniform Resource Identifier

PEP – Protocol Extension Protocol

CRLF – Carriage Return, Line Feed

ABNF – Augmented Backus-Naur Form

UA – User Agent

ACK - Acknowledgement

SP - Space

LWS – Linear White-space Character

HEX – Hexadecimal Number

NTP – Network Time Protocol

DNS – Domain Name System

IANA – Internet Assigned Numbers Authority

LRU – Least Recently Used

UTC – Coordinated Universal Time

(D)DoS – (Distributed) Denial of Service

TTL – Time to Live

nc - netcat