VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

SLUŽBA PRO ARCHIVACI WEBOVÉHO OBSAHU

WEB CONTENT ARCHIVING SERVICE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **ADAM MATUŠ**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. VILIAM SEREČUN**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Matuš Adam**
Program: Informační technologie
Název: **Služba pro archivaci webového obsahu**
Web Content Archiving Service
Kategorie: Data mining

Zadání:

1. Nastudujte si problematiku analýzy a exportu webového provozu - protokol HTTP, HTTPS, internetové proxy a exportní formáty (zejména MHTML a MAFF).
2. Seznamte se s frameworky pro archivaci webu (např. Lemmiwinks).
3. Navrhněte službu a její API, která bude archivovat webový obsah s využitím frameworku Lemmiwinks.
4. Po konzultaci s vedoucím implementujte navrženou službu.
5. Otestujte a analyzujte výsledné řešení na reprezentativním vzorku stránek, se zaměřením na stránky z Deep webu.

Literatura:

- R. Fielding, "RFC 2616 - Hypertext Transfer Protocol - HTTP/1.1", IETF, 1999.
- H. Lie, "The text/css Media Type", IETF, 1998.
- W3C, "Document Object Model (DOM)", online: http://www.w3.org/DOM/, 2011.
- Mozdev Community Organization Inc., "The MAFF Specification", online: http://maf.mozdev.org/maff-specification.html, 2014.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Serečun Viliam, Ing.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 24. října 2018

Abstrakt

Tato bakalářská práce řeší návrh a implementaci webových mikroslužeb pro archivaci obsahu se zaměřením na stránky z deep webu. Zabývá se seznámením se s principy aplikačních protokolů HTTP a HTTPS a dále bude nastíněna architektura webových služeb. Čtenář bude seznámen s existujícím řešením frameworku pro archivaci a rekonstrukci webu Lemmiwinks. S využitím tohoto frameworku je navržen a implementován systém webových služeb, které lze využít pro archivaci zvoleného webového obsahu do formátu MAFF.

Abstract

This bachelor's thesis deals with the design of web microservices for web archiving, mainly focused on deep web pages. The HTTP and HTTPS protocols will be explained in detail as well as web services design and architecture. The reader will learn about web archiving frameworks, especially the Lemmiwinks framework. This framework will be used as a basis for web services designed for archiving into the MAFF archive.

Klíčová slova

mikroslužby, služby, MAFF, MHTLM, HTTP, REST, deep web

Keywords

microservices, services, MAFF, MHTLM, HTTP, REST, deep web

Citace

MATUŠ, Adam. *Služba pro archivaci webového obsahu*. Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Viliam Serečun

Služba pro archivaci webového obsahu

Prohlášení

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

.....
Adam Matuš
13. května 2019

Poděkování

Chtěl bych poděkovat svému vedoucímu Viliamu Serečunovi za praktické rady a přátelský přístup.

Obsah

Kapitola 1

Úvod

Internet v současnosti dává možnost přistupovat k obrovskému množství informací, u kterých není zaručeno, že se v budoucnu nezmění nebo neztratí úplně. Proto se často využívají technologie pro zálohování a archivaci webového obsahu. Aplikací pro archivaci webových stránek existuje mnoho – ve formě softwarových frameworků či rozšíření do prohlížeče.

Trendem je implementovat aplikace jako webové služby, které jsou dostupné z Internetu a využívají běžné komunikační protokoly na webu. Výhodou webových služeb je snadná reprezentace dat pomocí serializace do textových formátů, jako je například JSON.

V této bakalářské práci jsem se zaměřil na transformaci archivačního frameworku Lemmiwinks do formy 2 webových služeb – archivace a stahování. Výhodou tohoto řešení je rozdělení zodpovědnosti, kdy složitější systém je rozdělen na menší části. K těmto službám je možnost přistupovat vzdáleně uživatelům nebo jiným automatizovaným nástrojům či službám pomocí aplikačního rozhraní. Implementace bude následně vyzkoušena na stránkách z Deep webu.

1.1 Přehled kapitol

V kapitole 2 jsou popsány hypertextové protokoly, jejich struktura a principy jejich využití ve webovém prostředí.

Kapitola 3 se zabývá archivačními formáty MHTML a MAFF. Dále jsou stručně popsány principy archivačních frameworků a konkrétní framework Lemmiwinks.

Následující kapitola 4 se zaměřuje na návrh webových služeb, zejména architekturu orientovanou na služby a aplikační rozhraní REST.

V kapitole 5 je zdokumentovaná implementace navržených služeb. Pro budoucí vývojáře dokumentace poskytuje přehled použitých knihoven a implementovaných funkcí.

Kapitola 6 popisuje modul pro testování a obsahuje vyhodnocení testování na vzorku stránek z Deep webu.

Kapitola 2

Hypertextové protokoly

Tato kapitola se věnuje převážně protokolům HTTP a HTTPS. Je zde popsána jejich struktura a principy fungování v rámci Internetu. Dále budou nastíněny rozdíly mezi jejich verzemi. Čtenáři bude prezentováno několik příkladů komunikace a její zabezpečení.

2.1 Charakteristika protokolu HTTP

Hypertext Transfer Protocol, dále zkráceně HTTP, je bezstavový protokol aplikační vrstvy modelu ISO/OSI. Byl navržen pro přenos dokumentů a hypertextových dat s možností rozšíření o další použití. Je postaven na komunikačním modelu klient-server, kde klient zahajuje komunikaci zasláním HTTP požadavku a server po přijetí žádosti pošle HTTP odpověď. HTTP je textový protokol, komunikace je tedy zobrazitelná v textově formě a většina přenášených informací je čitelná člověkem – v anglické literatuře existuje pojem *human-readable*.

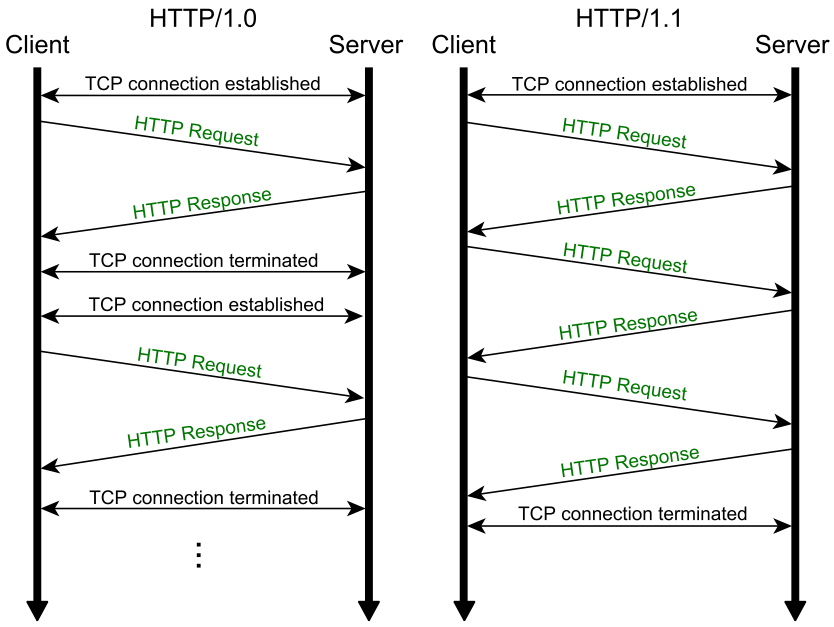
Obvyklým textovým obsahem HTTP odpovědí jsou například dokumenty v jazyce HTML nebo skripty v jazyce JavaScript. Multimediální, binární nebo jiná netextová data je možné přenášet s použitím rozšíření MIME.

Protokol HTTP je bezstavový, takže požadavky jsou obsluhovány jednotlivě bez potřeby navazovat nebo si pamatovat historii požadavků. Každý požadavek musí plně specifikovat celou akci, aby mohl server korektně odpovědět. Existují stavové mechanismy, které ale nejsou součástí specifikace protokolu HTTP [12].

Princip komunikace

O navázání spojení se stará transportní vrstva s protokolem TCP. Standardně se TCP využívá na výchozím portu 80, specifikace protokolu HTTP ale nebrání použití jiného transportního protokolu, který by měl podobné vlastnosti. Od transportní vrstvy se především předpokládá spolehlivý přenos, který protokol TCP zaručuje [11].

V rámci vytvořeného spojení se dále komunikuje na aplikační vrstvě formou HTTP zpráv. Zde je důležitý rozdíl mezi verzemi protokolů HTTP. Starší verze HTTP/1.0 dovoluje pouze jednu výměnu požadavku a odpovědi během spojení. Pro každou další konverzaci se spojení vytváří nové. Nevýhodami je velká režie a zátěž pro TCP congestion control. U novější verze HTTP/1.1 spojení existuje pro neomezený počet konverzací. Spojení se ukončuje až na žádost serveru nebo klienta. Obrázek 2.1 demonstruje rozdíl verzí.



Obrázek 2.1: Sekvence požadavků HTTP 1.0 a HTTP 1.1

2.2 Struktura protokolu

Základní datovou jednotkou protokolu HTTP je zpráva (message). Zpráva vytvořená klientem pro server se nazývá požadavek či dotaz (request) a zpráva vytvořená serverem pro klienta je odpověď (response). Každá zpráva začíná řádkem start-line. Dále následuje seznam hlaviček (headers) a tělo zprávy (body). Obsah částí protokolu se liší podle toho, zda se jedná o požadavek nebo o odpověď.

Jednotlivé elementy protokolu jsou oddělovány sekvencí CRLF, v textové reprezentaci jde o odřádkování. Ve znakové sadě ASCII této sekvenci náleží hodnoty `0x0d 0x0a`. Hlavičky a tělo zprávy jsou odděleny prázdným řádkem.

Identifikátor zdroje

Pro identifikaci zdroje na serveru se používá řetězec *Uniform Resource Identifier*, zkráceně URI. Zdrojem je obecně myšlen hypertextový dokument, multimediální obsah, služba nebo další podobné entity, které jsou serverem zpřístupněny pro dotazování. Obecný formát URI a příklad pro protokol HTTP je následující:

<scheme>://<authority>/<path>?<query>#<fragment>
http://tools.ietf.org/html/rfc2616.txt

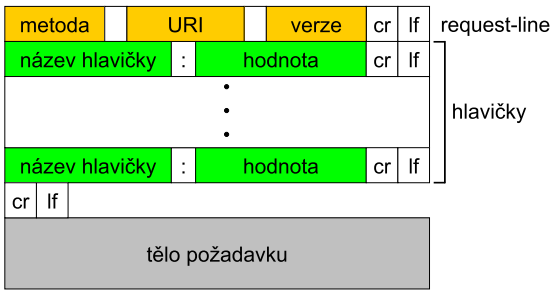
První povinná část je *schéma* (scheme) a je zde uveden komunikační protokol. Další části určují server, hierarchické umístění zdroje na serveru a informace dále upřesňující zdroj.

Formát požadavku

První řádek požadavku (request-line) se vždy skládá z názvu metody, identifikátoru zdroje a verze protokolu. Tyto položky jsou odděleny jednou mezerou a jsou očekávány v uvedeném pořadí. Verze protokolu je řetězec znaků, obvykle `HTTP/1.1` nebo starší standard `HTTP/1.0`.

Existuje celkem 9 typů metod požadavku. Název metody indikuje požadovanou akci a je povinně case-sensitive[1].

- GET metoda požaduje po serveru stáhnutí hypertextového zdroje, který se nalézá na dále uvedeném URI. Metoda byla navržena pro získávání dokumentů ze serveru, přesto ji lze nesprávně použít pro zasílání dat z formulářů od klienta.
- HEAD požaduje stejnou odpověď jako GET s tím rozdílem, že server vynechá tělo odpovědi. Metoda je vhodná pro zjištění dostupnosti zdroje a dalších informací obsažených v hlavičkách odpovědi.
- POST slouží pro odeslání dat na server. Narozdíl od metody GET jsou data obsaženy v těle požadavku, nikoliv pouze v URI. Díky této vlastnosti je pro odesílání citlivých dat na server bezpečnější než metoda GET.
- PUT odešle dokument zaslaný v těle zprávy na místo specifikované v URI. V případě existujícího dokumentu se nahradí novým.
- OPTIONS se dotáže serveru na možnosti komunikace pro konkrétní zdroj. Pro získání globálních možností serveru lze URI nahradit znakem hvězdičky.
- CONNECT metoda se pokusí zahájit obousměrné tunelované spojení. Požadavek se pošle HTTP Proxy serveru, který zprostředkuje komunikaci s cílovým serverem.
- DELETE požádá server o smazání zdroje na specifikovaném URI.
- TRACE je metoda používaná pro diagnostické účely a testování. Cílový server reflektuje přijatou zprávu zpět klientovi. Klient si může ověřit, zda požadavek dorazil s očekávaným obsahem.
- PATCH částečně modifikuje odkazovaný zdroj. Metoda umožňuje změnit část dokumentu bez nutnosti posílat celý dokument, jako tomu je u metody PUT.



Obrázek 2.2: Formát HTTP požadavku

[1]Lexikální analýza je citlivá na velikost písmen, tedy například řetězce "get"a "GET"jsou chápány jako rozdílné

Po prvním řádku následuje seznam hlaviček. Struktura hlavičky je vždy dvojce název a hodnota, tyto položky jsou oddělené dvojtečkou. Účelem hlaviček je blíže specifikovat požadavek.

- **Cache-control** upravuje výchozí caching mechanismy pro HTTP konverzaci. Pomocí klíčových slov public, private a no-cache lze nastavit možnost kešování požadavku serverem, dále je možné modifikovat expirační a revalidační mechanismy.
- **Host** obsahuje doménové jméno dotazovaného serveru a číslo portu. Není-li port uveden, předpokládá se výchozí port 80. Tato hlavička je ve verzi HTTP/1.1 povinná.
- **Cookie** hlavička slouží k předávání a uchovávání informací o probíhající konverzaci. Cookies implementuje stavový mechanismus v jinak standardně bezstavovém protokolu HTTP[1].
- **User-agent** obsahuje informace o aplikaci, která komunikuje s druhou stranou. Taková aplikace může být například běžný internetový prohlížeč nebo webový crawler[2]. Webové servery tuto hlavičku často vyžadují a v případě nepřítomnosti generují chybovou odpověď.

Existuje celá řada dalších často využívaných hlaviček, výše uvedený seznam je pouze nastíněná charakterisitka hlaviček relevantních pro tuto práci. Na obrázku 2.2 je znázorněna struktura požadavku.

Formát odpovědi

Odpověď začíná tzv. stavovým řádkem (status-line), který obsahuje verzi protokolu a stavový kód s textovou formou kódu. Stavový kód, anglicky *status code*, je tříčíselný kód indikující výsledek zpracovaného požadavku. Rozděluje se do pěti kategorií určených podle první číslice:

- **1xx** - Tato třída obsahuje informační a obvykle provizorní odpovědi. V případě kódu *100 Continue* server očekává pokračování požadavku, aby mohl zaslat finální odpověď. Kódem 101 Switching Protocols server informuje o změně komunikačního protokolu.
- **2xx** - Žádost od klienta byla úspěšně přijatá a zpracovaná. Odpověď s kódem 200 OK se liší podle požadavku, například pro požadavek GET je v těle odpovědi obsah zdroje. Kód *202 Accepted* indikuje kladné zpracování požadavku s tím, že server stále požadavek zpracovává a klient může očekávat finalní odpověď.
- **3xx** - Pokud byl požadovaný zdroj dočasně nebo permanentně přemístěn, generují se stavové kódy 301 Moved Permanently a 302 Found. Klientovi poskytují více informací o přesměrování na daný zdroj. V případě podmíněného požadavku kód 304 Not Modified dává najevo, že se zdroj od posledního získání nezměnil.
- **4xx** – Odpovědi této třídy se posílají na chybný nebo nezpracovatelný klientský požadavek. Na syntakticky chybný požadavek se odpovídá kódem *400 Bad Request*. Pokud klient není autentizován pro přístup ke zdroji, generuje se *401 Unauthorized*. Pokud server odmítá zpřístupnit zdroj třeba z jiných důvodů, používá se kód *403 Forbidden*. Pro neexistující zdroj nebo zdroj, který server nechce zviditelnit, se posílá známý kód *404 Not Found*.

[2] Web crawler je software pro automatizovaný sběr dat z webových serverů.

- **5xx** – Tyto stavové kódy značí chybu na straně serveru, příkladem může být generický *500 Internal Server Error*. Kód *502 Bad Gateway* je obvykle spojován s proxy serverem, který nemohl vyhovět klientovi z důvodu selhání některého z nadřazených (upstream) serverů.

Po stavovém řádku následují hlavičky odpovědi. Poskytují další informace o úspěšném i neúspěšném vyřízení požadavku:

- **Server** hlavička poskytuje informace o softwaru, který odpověď generoval. Obvykle se zde uvádí název, například `Apache`.
- **Content-Length** informuje o velikosti těla odpovědi v bytech.
- **Content-Type** obsahuje informaci o formátu dat a použité znakové sadě. Příkladem hodnoty pro HTML dokument je `text/html; charset=utf-8`.
- **Content-Encoding** informuje o použitém kódování těla odpovědi.

2.3 Zabezpečení protokolu HTTP

HTTP poskytuje mechanismy pro autentizaci uživatelů při posílání požadavků. Základní schéma autentizace používá hlavičku požadavku *Authorization* pro zaslání uživatelského jména a hesla. Tyto mechanismy ale neřeší šifrování komunikace. Zachycením probíhajícího přenosu lze zrekonstruovat celé bloky dat, a to i neautorizovaným útočníkem. Tato vlastnost je pro některé aplikace nebezpečná.

Hypertext Transfer Protokol over TLS, dále jen HTTPS, je protokol zajišťující šifrovanou zabezpečenou komunikaci. Před samotnou HTTP konverzací se vytvoří spojení s použitím technologie Transport Layer Security (TLS). V minulosti se využívala varianta HTTP over Secure Sockets Layer (SSL), která byla nahrazena TLS. Protokol HTTPS řeší zabezpečení autentizace a další komunikace ve většině moderních webových serverech [16].

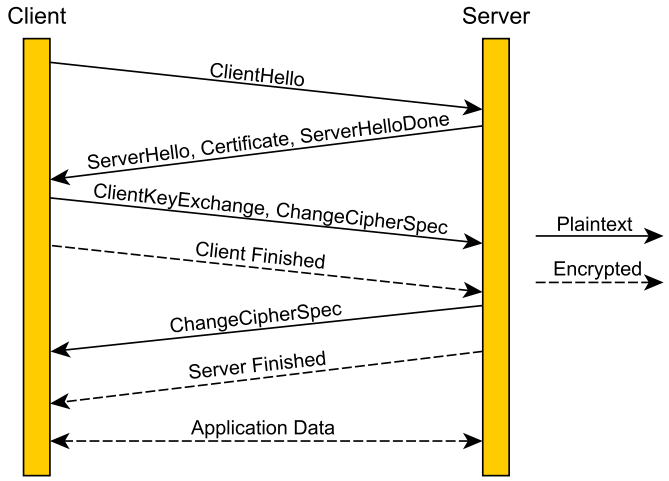
Navázání TLS spojení

Před zahájením HTTP konverzace je potřeba ustavit TLS spojení pomocí *handshake* procedury. Předpokladem je podpora TLS na obou stranách. Stručně popsaný průběh je ukázán na obrázku 2.3.

- Klient pošle serveru zprávu 'Client hello' s kryptografickými informacemi,
- Server odpoví zprávou 'Server hello' a následuje zpráva obsahující certifikát serveru. Server také může požadovat certifikát po klientovi. Server dokončí inicializaci zprávou *Server hello done*
- Klient ověří pravost certifikátu přes certifikační autoritu. Tento krok zabrání falšování certifikátu při útoku *man-in-the-middle*[3].
- Klient vygeneruje a pošle Pre-Master Secret zašifrovaný veřejným klíčem, který získal z certifikátu serveru.

[3]Man-in-the-middle je typ síťového útoku, kdy útočník odposlouchává a zasahuje do komunikace mezi účastníky.

- Na základě Pre-Master Secret jsou obě strany schopné vygenerovat svoje symetrické klíče, kterými se bude obsah HTTP zpráv šifrovat a dešifrovat.
- Ukončení handshake proběhne výměnou nešifrovaných zpráv *Change cipher spec* a šifrovaných *Client finished* a *Server finished* pro ověření správného dokončení handshake. Další přenos aplikačních dat je šifrovaný.



Obrázek 2.3: TLS handshake

2.4 Internetová proxy

Při běžné komunikaci na protokolu HTTP se předpokládají dva účastníci, tedy klient přímo požaduje po serveru webový obsah a server přímo odpovídá klientovi. Procesu předávání HTTP zpráv se může zúčastnit i další server, který se nazývá *proxy server*. Internetový proxy server zprostředkovává přeposílání zpráv mezi klientem a serverem. Stává se tak třetím síťovým prvkem, který zastupuje roli klienta nebo serveru v závislosti na směru zprávy. Mezi využití proxy serverů patří zapouzdření složitějšího systému distribuovaných serverů, kdy klient vnímá celý systém jako jeden server. Další využití je komunikace přes tunelovací proxy, který nemění obsah zpráv a pouze skrývá informace nižších protokolů (například IP adresu) původního klienta před serverem. [8]

Pro využití klientských aplikací na Internetu se používá forward proxy. Spojení s tímto proxy serverem je obvykle v zájmu klienta. V případě veřejných open proxy je možné se setkat se službou anonymizace klienta před cílovým serverem. Požadavky jsou serveru přeposílány bez informací síťového protokolu, jako například IP adresa. V této oblasti je opakem transparentní proxy, která zahrnuje IP adresu původního klienta do zprávy cílovému serveru.

Ze strany distribuovaných serverů v privátní síti jsou zajímavé tzv. *reverzní proxy*. Klientovi se reverzní proxy jeví jako cílový server, který zajišťuje generování všech odpovědí. Ve skutečnosti tento server zastiňuje síť, která svoji odpověď na původní požadavek pošle klientovi přes proxy server. Reverzní proxy se stará o výběr cílového serveru a obvykle

s nimi přímo spolupracuje na vyřízení složitějších požadavků, u kterých je nutné dešifrování, dekomprese a podobně. Při správném návrhu slouží reverzní proxy jako mechanismus vyrovnávání zátěže.[4]

Cache proxy

Webový obsah je přenášen v HTTP odpovědích, které je možné uchovávat pro více klientů. Cache proxy servery uchovávají odpovědi vzdálených serverů po určitou dobu, čímž urychlují reakci na klientův dotaz. Zároveň se snižuje zatížení sítě na straně cílového serveru. V ideálním případě se cílové servery, které generují dynamický obsah, nemusí zabývat odpovídáním na stejné požadavky různých klientů na statické zdroje.

Kapitola 3

Webové archivační formáty

Archivní formát slouží pro uložení kolekce dat ve formě datových struktur nebo celých souborů a spolu s metadaty vytváří archivní soubor. Významnou vlastností webových archivních formátů je kromě uložení textových dat také možnost znovu vizualizovat stránku v původní podobě. Výhodou archivních souborů je snadná přenositelnost – logický celek je sloučen do jednoho souboru. Archívy obsahují adresářovou strukturu včetně jednotlivých souborů a metadata pro detekci a opravu chyb. Archivační formáty obvykle využívají datové komprese pro zmenšení velikosti na disku. U některých typů archivačních formátu je možnost šifrování obsahu.

Archivace webu je proces sběru dat z webových stránek a ukládání do některého archivačního formátu, ať už za účelem uchovat stav webu pro pozdější analýzu, data mining nebo zveřejnění historického vzhledu webových stránek.[1]. Problematiky archivace webu se dotýká pojem *web crawler*, což je software nasazován pro automatizované stahování a archivaci velkého množství dat z různých webových stránek nebo služeb.

Struktura webových dokumentů v archivu

Jádrem webových archivů je webová stránka, která je založena na stromové struktuře. Document Object Model, dále jen DOM, je objektově orientovaný model webových dokumentů standardizovaný organizací W3C. DOM tvoří rozhraní pro čtení a modifikaci obsahu a stylu částí dokumentu. Dnešní webové prohlížeče využívají DOM jako vnitřní reprezentaci HTML dokumentu, který je stěžejní pro vizualizaci webového obsahu.

3.1 Formát MHTML

MIME HTML, dále jen MTHML, je webový archivační formát pro uložení HTML dokumentů včetně odkazovaných zdrojů. Webová stránka je archivovaná v jediném souboru s příponou `.mht`. Obsah archívu je kódovaný pomocí rozšíření MIME. Formát MHTML je popsán jako internetový standard RFC 2557.[5] Dnešní prohlížeče jsou schopné pracovat s formátem MHTML bez nutnosti instalování rozšíření.

Charakteristika rozšíření MIME

Multipurpose Internet Mail Extensions, dále jen MIME, je standard navržen pro posílání souborů přes email. Využívá se pro kódování souborů jako hypertextové dokumenty,

[1] Na stránkách https://archive.org/web/ můžete navštívit historii známých webů

netextové data, binární data a podobně. Obecně lze MIME uplatnit i mimo emailový protokol, například u přenosu netextových dat protokolu HTTP. Struktura MIME se dělí na hlavičku a tělo zprávy. Hlavička obsahuje položky, která pochází z původního návrhu pro email, MHTML tuto část využívá pro zaznamenání času archivace a další vhodné informace o souboru. Položka *Content-type* popisuje tělo zprávy. MHTML soubor, který se skládá z několika částí, je charakterizován MIME typem *multipart/related*. Tento typ umožňuje zřetězit části webových stránek. Mezi jednotlivé části se vkládá oddělovač – `multipart-boundary`.

Struktura MHTML

MHTML formát je strukturovaný jako MIME struktura typu `multipart/related`, jejíž části jsou potom popsány příslušným typem původního obsahu – *text/html* pro HTML dokument, *image/jpeg* pro obrázek a další. Celá struktura tvoří vázaný seznam, nebo-li sekvenci dokumentů ve formátu MIME.

Při vytváření MHTML souboru se prochází stromová struktura, přičemž navštívené uzly generují seznam MIME částí. Dalším průchodem seznamu se vytvoří příslušné vazby. Procházení stromu od kořene garantuje, že vazby na další části nikdy nebudou zpětné.

Protože je MHTML formát strukturovaný jako vázaný seznam MIME částí, je potřeba jej transformovat na stromovou strukturu webových stránek. Kořenový dokument je obvykle prvním prvkem seznamu, který odkazuje na další položky seznamu. Tyto položky jsou zpravidla uzly stromu o úroveň níže. Takto je možné projít lineární seznam a zpětně rekonstruovat stromovou strukturu webové stránky.

3.2 Mozilla Archive File Format

Mozilla Archive File Format, zkráceně MAFF, je formát pro archivaci webového obsahu. Umožňuje ukládání jedné nebo více webových stránek včetně multimediálních souborů do jednoho archivního souboru. MAFF je postaven na formátu ZIP souborů a využívá stejný princip komprese. MAFF není považován za internetový standard, formát vznikl jako rozšíření internetového prohlížeče Mozilla Firefox. Toto rozšíření již není v nových verzích podporované, přesto je specifikace MAFF udržovaná jako aktuální pro využití jinými aplikacemi.

Specifikace formátu

MAFF byl navržen pro archivaci výhradně webového obsahu – uložená metadata často určují URL webového zdroje, ze kterého byl archiv stránky vytvořen. MAFF není určen jako kompletní offline cache webového obsahu. Negarantuje, že archivované klientské skripty nebo reference na multimédia budou funkční, pokud využívají externí závislosti. Obsah webových stránek v MAFF souboru není vhodné částečně aktualizovat. Například pokud se obsah zdroje cílového serveru změní, je potřeba uložit celou stránku v aktuální podobě. Jednotlivé archivované stránky jsou proto chápány jako atomické prvky.

MAFF archiv používá stejnou metodu komprimace jako formát ZIP.[2] ZIP je běžně používaným formátem pro bezeztrátovou kompresi dat. Z tohoto pohledu se chová MAFF stejně a lze jej komprimovat a rozbalovat pomocí nástrojů pro ZIP formát. Obsah archívu využívá strukturu JAR protokolu.

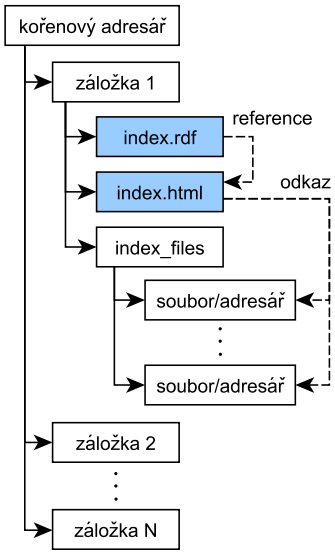
[2]ZIP je všeobecný souborový formát pro kompresi a archivaci dat

Adresářová struktura

MAFF je založen na adresářové struktuře a vychází ze struktury souborového systému, viz. obrázek 3.1. Kořenový adresář obsahuje pouze adrešáře pro uložené stránky a nesmí obsahovat žádné soubory. Jednotlivé adresáře potom obsahují metadata archivované stránky a stromovou strukturu založenou na uspořádání cílového webu.

Soubor pro uložení metadat jedné stránky se jmenuje `index.rdf`. Zde se ukládají informace o archívu jako čas uložení, znaková sada a podobně. Tento soubor může chybět v případě, že stránka obsahuje hlavní dokument `index.html` nebo index soubor jiného typu. Soubor RDF využívá značkovací jazyk XML[3]. Příklad RDF souboru je na obrázku 3.2.

Adresář pro webovou stránku dále obsahuje adresář `index_files`, ve kterém se vyskytují další dokumenty dostupné z indexového dokumentu. Při rozbalení se tento adresář zobrazí jako jedna záložka (bookmark) ve webovém prohlížeči.



Obrázek 3.1: Struktura MAF

```
<RDF:RDF>
  <RDF:Description RDF:about="urn:root">
    <MAF:title RDF:resource="mozdev.org - maf: index"/>
    <MAF:originalurl RDF:resource="http://maf.mozdev.org/"/>
    <MAF:indexfilename RDF:resource="index.htm"/>
    <MAF:archivetime RDF:resource="Sat, 24 Nov 2018 16:11:28 +0100"/>
    <MAF:charset RDF:resource=""/>
  </RDF:Description>
</RDF:RDF>
```

Obrázek 3.2: Příklad souboru index.rdf

[3]XML – Extensible Markup Language

3.3 Frameworky pro archivaci webu

Archivace webu je proces sběru dat z WWW za účelem zachování obsahu webových stránek v archívu pro pozdější analýzu či zobrazení. Vzhledem k tomu, že Internet je velmi rozsáhlá síť obsahující přibližně 1.5 miliard webových stránek[4], je výhodné využít sofistikovaný software pro archivaci webu. Snahou je vytvořit takový software, který bude automatizovat činnost stahování, analyzování a archivování stažených dat.

Aplikace pro archivaci webu často musí řešit problémy stejného typu. Základem pro takové aplikace může být framework – programová struktura, která usnadní vývoj archivačního software pro specifický účel. Cílem frameworku pro archivaci je řešení typických problémů, například zpracování HTTP odpovědí, analýza stažených dokumentů a algoritmy pro vytváření archívů. Programátoři pracující s frameworkem se potom mohou soustředit na řešení problémů specifických pro své zadání.

3.4 Framework Lemmiwinks

Lemmiwinks je framework pro automatizované stahování a archivaci webového obsahu. Je implementovaný v jazyce Python 3.6, který umožňuje využití asynchronních úloh pomocí standardní knihovny *asyncio*. Dále využívá několik externích knihoven, například *beautifulsoup4* pro analýzu HTML dokumentů nebo *tinycss* pro zpracování CSS. Framework Lemmiwinks je rozdělen do 4 samostatných modulů, které řeší specifické problémy [14].

- *httplib* se stará o klientskou část aplikace. Modul obsahuje jednoduchého klienta a asynchroního klienta.
- *parslib* řeší analýzu stažených dokumentů, zejména HTML a CSS a zajišťuje stažení všech potřebných zdrojů, které jsou v dokumentech mnohdy odkazované rekurzivně.
- *archive* je modul pro archivaci staženého obsahu do formátu MAFF
- *extractor* specifikuje rozhraní pro extrahování dat pomocí regulárních výrazů.

V implementační části této práce se bude stavět na tomto frameworku, zejména s využitím modulů *parslib* a *archive*. Návrh síťové komunikace bude inspirovaný moduluem *httplib*, navíc bude nutné navrhnout aplikační rozhraní a serverovou funkcionalitu jednotlivých webových služeb.

[4] http://www.internetlivestats.com/total-number-of-websites/

Kapitola 4

Návrh webových služeb

V softwarovém inženýrství je *servisně orientovaná architektura*, zkráceně SOA z anglického *Service-oriented architecture*, sada principů pro nasazení nezávislých komponent poskytující služby. Architektura je navržena pro fungování v počítačové síti. Jednotlivé komponenty implementují elementární služby a komunikují s okolím přes společný protokol. Tímto způsobem lze sestavit celý informační systém, jehož komponenty nejsou k tomuto systému pevně vázané [2].

4.1 Aplikační rozhraní

SOA je nejčastěji implementovaná pomocí webových služeb komunikujících pomocí protokolu HTTP. Jedním ze standardních rozhraní využívající HTTP je *Representational state transfer*, známé pod zkratkou REST. REST je datově orientované rozhraní pro distribuované hypermediální systémy a je použitelné pro jednotný přístup ke zdrojům. Zdrojem jsou data nebo stav aplikace odvoditelný z dat. Zdroje jsou identifikovány přes URI podle zvyklostí protokolu HTTP [3].

Webové službě, která splňuje a správně implementuje architekturu REST, se říká RESTful. Taková aplikace se řídí těmito omezeními:

- Oddělení zájmů klienta od zájmů serveru zajišťuje, že se komponenty systému mohou vyvíjet nezávisle.
- Server musí být bezstavový, což je přetrvávající princip z protokolu HTTP, kdy si server neudržuje kontext. Řídit stav komunikace je v možnostech klienta.
- Server je povinen klientovi sdělit, jak a která data odpovědí je možné uchovávat (vztahující se anglické pojmy *cache*, *cacheable requests*). Klient je oprávněn využít data odpovědi po určitou dobu, pokud by klient chtěl poslat ekvivalentní požadavek.
- Rozhraní mezi klienty a servery je uniformní. Všechny požadavky klienta jsou orientovány na zdroj, který odpoví reprezentací zdroje v závislosti na požadavku. Klient je potom schopený využít tuto reprezentaci k modifikaci zdroje na serveru. Každá zpráva musí obsahovat informaci, jak tuto zprávu zpracovat. Zprávy serveru musí specifikovat možnosti kešování požadavků.

Službu lze chápat jako komponentu, která se chová jako server i klient zároveň. Pro splnění požadavku, což je serverová část služby, často aplikace komunikuje s jiným serverem či službou, čímž se stane klientem jiné služby.

4.2 Návrh služeb pro archivaci

Archivace webu se dá rozdělit na 2 hlavní podúkoly, které lze implementovat jako samostatné služby:

- Stahování dat ze zdrojů – je potřeba zajistit stažení zdroje na požadovaném URL, vyhodnotit úspěšnost stažení, poskytnout informace o přesměrování a podobně. Tato služba se dále bude nazývat *stahovací služba*.
- Archivace stažených dat – služba pomocí analýzy získaných zdrojů zjistí, které další zdroje je potřeba stáhnout. V případě úspěchu stahování všech potřebných zdrojů se ze získaných dat vytvoří archív ve formátu MAFF. Tato služba bude dále označena jako *archivační služba*.

Pro přístup k systému z vnějšku se často vytváří ještě služba typu *API gateway* (brána), která funguje jako řídící jednotka při vyhodnocování požadavků u složitějších systémů. V případě této architektury je jednoduché řešení dostačující, a sice integrovat API bránu do jedné ze dvou služeb. Protože bude služba pro archivaci řídit celkový proces, je vhodné z venku komunikovat s touto službou.

Na obrázku 4.1 je zobrazen sekvenční diagram příkladu komunikace navržených služeb. Uživatel zašle archivační službě požadavek obsahující seznam URL. Archivační služba zprávu zpracuje a zahájí komunikaci se stahovací službou. Zdroj `resource1` může představovat např. soubor `index.html`, ze kterého je odkazováno na další zdroj `resource2`, což může být obrázkový soubor nebo CSS. Po získání všech potřebných souborů je vytvořen MAFF archiv na disku archivační služby. Při úspěšné archivaci uživatel dostane odpověď, která obsahuje název vytvořeného archivu. Archiv lze kdykoliv později stáhnout přímo ze serveru.

Aplikační rozhraní služby pro archivaci

Tato služba bude jádrem systému a bude sloužit jako výchozí bod pro využití aplikace. Služba implementuje tyto zprávy:

- **GET** */* – Hlavní stránka, která bude mít informativní charakter. Stránka je přístupná všem uživatelům.
- **GET** */archive* – Stránka s archívy. Obsahem bude příklad využití služby nebo odkazy na existující archívy.
- **POST** */archive* – Požadavek na vytvoření archívu MAFF z daných URL. Tělo požadavku jsou *application/json* data:
  - "URLs": [string]

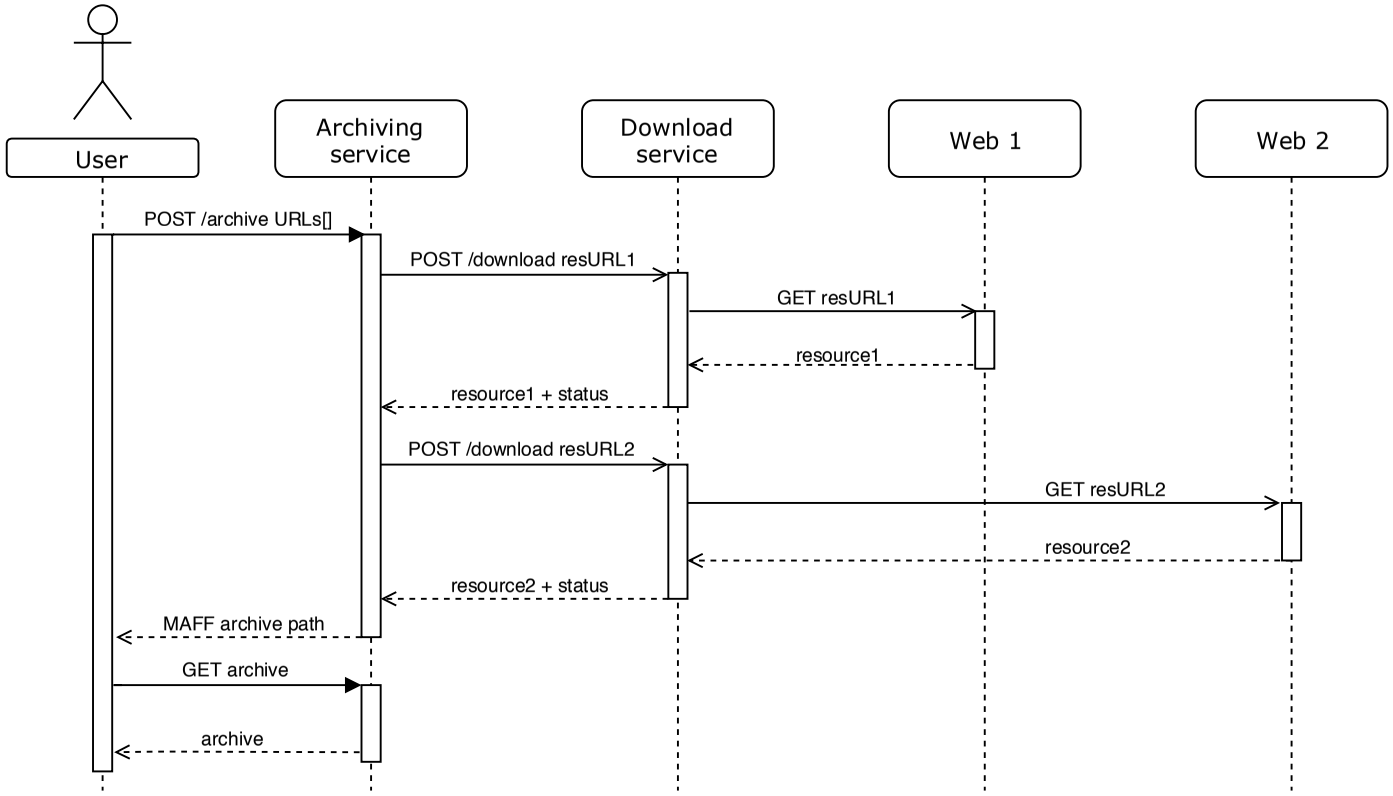
  Odpověď ve formátu *application/json*:
  - "status": string
  - "archivePath": string

Aplikační rozhraní služby pro stahování

Základem této služby je zajistit stažení požadovaného zdroje a informovat o případném neúspěchu. API stahovací služby bude využívat služba pro archivaci. Stahovací služba implementuje tyto zprávy:

- **GET** */* – Hlavní stránka, která bude mít informativní charakter. Stránka je přístupná všem uživatelům, tedy není nutná autorizace.
- **POST** */download* – Požadavek na stažení zdroje. Tělo zprávy obsahuje *application/json* data.
  - "resourceURL": string
  - "useTor": string

  Odpověď ve formátu *application/json*:
  - "status": string
  - "url_and_status": list
  - "data": base64



Obrázek 4.1: Sekvenční diagram komunikace služeb

Kapitola 5

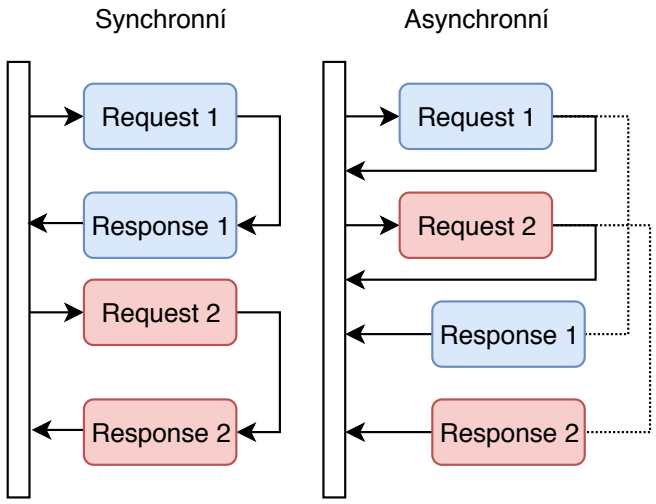
Implementace

Tato kapitola popisuje implementační detaily obou služeb a případné možnosti rozšíření. Zabývá se technikami asynchronního programování a implementací asynchronních služeb v jazyce Python 3.6, přičemž bude přiblížen způsob využití frameworku Sanic v kombinaci s asynchronním klientem. V části týkající se archivačního frameworku budou zdokumentovány modifikace frameworku Lemmiwinks a motivace těchto modifikací.

5.1 Asynchronní programování

Asynchronním programování je druh programování obecně jakékoliv aplikace, kde je kladen důraz za paralelní běh jednotek. Stále je zde k nalezení primární vlákno, kterým obvykle aplikace začíná a stará se o paralelně běžící úlohy. Po dokončení úlohy je hlavní vlákno informované o stavu proběhlé operace.

Na obrázku 5.1 je zobrazen rozdíl mezi synchronním a asynchronním vyřízením 2 požadavků. Tečkovaná čára značí vytvoření korutiny pro zpracování požadavku, která se váže ke konkretní pozdější generované odpovědi. Časový interval mezi odpověďmi může být daleko větší, protože vyřízení různých požadavků může trvat značně delší dobu.



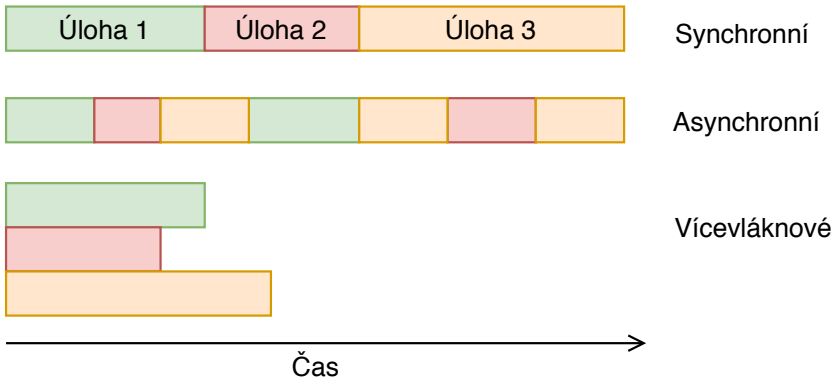
Obrázek 5.1: Rozdíl mezi synchronním a asynchroním modelem požadavek-odpověď

Paralelní běh úloh je výhodně využitelný při implementaci serverové logiky, například v rámci webových služeb, kde je důležitá rychlá reakce na požadavky, a to zejména na požadavky, které jsou snadno zpracovatelné a není důvod generovat a posílat odpověď se zpožděním. Vyřizování požadavků často obsahuje operace, kdy se čeká na stažení dat po síti a podobně. Asynchronně napsaná aplikace potom nebude v těchto případech blokující.

Podobně lze asynchronně naprogramovat klientskou logiku. Například pokud je potřeba získat N zdrojů nezávisle na sobě, je možné paralelně poslat všechny požadavky a postupně přijímat odpovědi od serveru.

Dalším důležitým rozdílem oproti například vícevláknové aplikaci je pouze 1 běžící vlákno. V asynchronním kódu Pythonu je paralelismus v režii interpretu a modulů pro řešení asynchronních operací. Využití vícevláknových aplikací v Pythonu je problematické z důvodu samotné implementace interpretu Pythonu, jenž využívá mutex *Global Interpreter Lock* pro řízení přístupu k objektům a bytekódu Pythonu [18].

Na obrázku 5.2 je ilustrace výkonávání úloh v čase pro různé varianty plánování.



Obrázek 5.2: Příklad zpracování 3 úloh synchronně, asynchronně a více vlákny

Asyncio

Od verze Python 3.4 je k dispozici modul *asyncio* a *async/await* syntaxe [15]. Modul asyncio implementuje tyto speciální konstrukce:

- *smyčka událostí* – anglicky také *event loop* – je smyčka která řídí a distribuuje vykonávání úloh. Řeší tok řízení mezi novými a existujícími úlohami. Kromě základní implementace smyčky také existuje alternativní implementace *uvloop*.
- *korutiny* jsou speciální funkce deklarované syntaxí *async*. Korutina před spuštěním musí být naplánovaná ve smyčce událostí. Korutina může předat řízení zpět pomocí *await* syntaxe.
- *futures* jsou objekty, které reprezentují výsledek asynchronní operace.

5.2 Struktura projektu

Kořenový adresář projektu se jmenuje `Lemmiwinks-services`. Tento adresář dále obsahuje podadresáře jednotlivých služeb a testovacího klienta:

- `archive_service`

- `download_service`
- `test_client`

Tyto adresáře obsahují vlastní zdrojové kódy a závislosti. Kromě těchto adresářů je zde soubor `makefile` pro automatické spuštění procesů služeb. Spuštění probíhá příkazem `make`, jenž je součástí běžných linuxových distribucí.

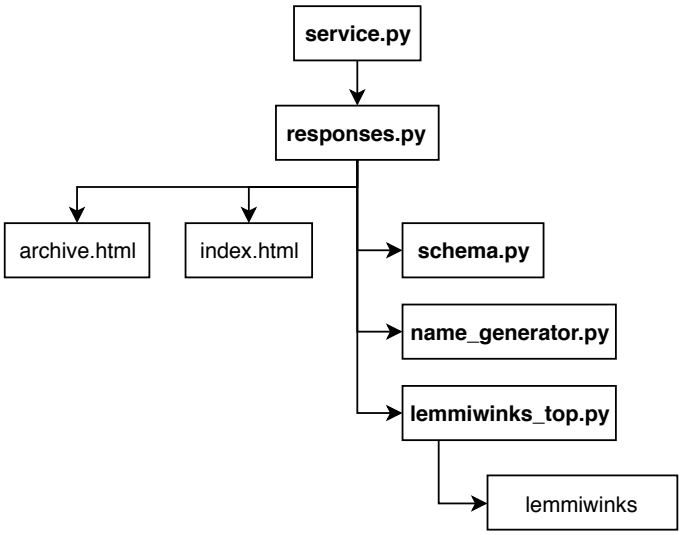
Tato struktura dovoluje systém mikroslužeb dále dělit a rozšiřovat o nezávisle vyvíjitelné služby, například službu pro uživatelské webové rozhraní či jiné implementace služeb v odlišných technologiích.

5.3 Archivační služba

Hlavní úlohou této služby je řídit proces vytváření archívu, v tomto případě archívu ve formátu MAFF. Implementace se nalézá v adresáři `archive_service` s top-level souborem `service.py`. Požadavky HTTP na tuto službu budou přicházet z vnějšku a je důležité tento fakt zohlednit v implementaci kontrolou syntaxe a schématu JSON objektu, který je zasílán v těle požadavku. Služba běží v rámci jediného procesu a lze ji lokálně spustit příkazem `python3.6 service.py`.

Archivační služba využívá modifikovanou verzi frameworku Lemmiwinks. Úpravy se týkají především HTTP klienta.

Stromová struktura zdrojových souborů je zobrazena na obrázku 5.3. Směr šipky značí závislost souborů. Struktura modulu Lemmiwinks je popsaná v kapitole 3.4.



Obrázek 5.3: Struktura zdrojových souborů archivační služby

Server

Implementace využívá web server framework Sanic verze 19.03[13]. Při spuštění se automaticky inicializuje objekt Sanic, který reprezentuje instanci serveru a běží navždy do doby, než je proces interpretu ukončen. Pro testovací účely je server spouštěn na adrese `0.0.0.0`. Server archivační služby poslouchá na portu `8080`, kde očekává příchozí HTTP požadavky.

Pro přístup ke zdrojům služby jsou implementované funkce typu *handler*. Sanic umožňuje využít dekorátor `@app.route`, kde `app` je instance třídy Sanic. Pro účely této služby stačí specifikovat cestu ke zdroji v prvním parametru, a seznam povolených metod pro dekorovanou handler funkci. Handler musí být vždy deklarován jako asynchronní pomocí syntaxe `async`.

Požadavky jsou obsluhovány těmito funkcemi:

- `post_archive_handler` zpracovává požadavky POST na cestu `\archive`. Obsluha tohoto požadavku je implementačně nejsložitější, protože se v jeho rámci proběhne proces vytvoření nového archívu. Handler generuje odpověď ve formátu JSON obsahující informace o stavu proběhlé archivace.
- `get_archive_handler` řeší požadavky GET na `\archive`. Odpovědí na tento požadavek je HTML dokument obsahující seznam existujících archívů.
- `get_root_handler` generuje HTML dokument informačního charakteru.

Validace požadavků

Příchozí požadavky a jejich data jsou v podstatě vstupní data do systému, a proto je potřeba věnovat pozornost jejich validaci. Základní kontroly syntaxe HTTP požadavků provádí již Sanic.

Služba očekává JSON data v těle požadavku s relativně striktně definovanou strukturou. *JSON Schema* je rozšířená specifikace, která využívá slovník pro anotaci a validaci JSON dat[7]. Tato specifikace je implementovaná pro jazyk Python jako modul *jsonscheme*.

Účelem třídy *ArchiveJsonSchema*, jenž je implementovaná v souboru scheme.py, je validovat JSON data požadavku POST na zdroj `/archive`. Jsou zde definované 2 schémata:

- `scheme` – validační schéma, které akceptuje i jiné vlastnosti než `urls`. Motivací je umožnit případné rozšíření.
- `scheme_strict` – zakazuje jinou vlastnost než `urls`. JSON data musí obsahovat právě tuto vlastnost.

Tabulka 5.1 obsahuje pravidla validace. Schéma je generované v závislosti na parametru `maxItems`, jehož výchozí hodnota je nastavena na 256. MaxItems určuje maximální počet zpracovaných URL. Maximální délka URL řetězce není oficiálně specifikovaná, avšak obvykle se implementuje limit 2048 znaků [10]. V případě nevalidní JSON instance je vyvolána výjimka ValidationError.

	Property	Type	Min Items	Max Items	Items		Additional Properties
Normal	urls	array	1	256	Type	Length	true
					string	-	
Strict	urls	array	1	256	Type	Length	false
					string	2048	

Tabulka 5.1: JSON schema pro POST požadavky na archivační službu

Lemmiwinks

Vstupním bodem frameworku je třída ArchiveServiceLemmiwinks, která dědí od kostry třídy Lemmiwinks. Protože se jedná o asynchronní framework, je zde deklarace veřejné asynchronní metody `task_executor`. Účelem je inicializovat seznam úloh, které budou dále naplánovány pro asynchronní vykonávání.

Lemmiwinks dále obsahuje metodu `run`, která spouští event loop. Tato metoda není v této aplikaci důležitá, protože event loop již běží od začátku spuštění serveru Sanic. Pro start archivace stačí implementovanou metodu `task_executor` evokovat s použitím klíčového slova await.

Úloha archivace je implementovaná v privátní metodě `__archive_task`. Metoda akceptuje 2 povinné argumenty:

- `urls` – seznam URL řetězců, jejichž obsah se bude archivovat
- `archive_name` – řetězec reprezentující název archívu bez přípony

Pro každou URL se následně vytvoří úloha asynchronního klienta pro zaslání požadavků na stažení. Tyto úlohy jsou spuštěny paralelně. Naplánování a synchronizaci těchto úloh obstarává metoda asyncio.gather, která vyčká na dokončení všech úloh stahování.

Stažená data, resp. deskriptory na soubory jsou dále předány úloze `__archive_responses`. Framework Lemmiwinks definuje seznam tabů v budoucím MAFF archívu třídou Envelop a jednotlivé taby jako objekt *letter*. V úloze se iterativně naplní objekt envelop a vytvoří se odpovídající MAFF archív.

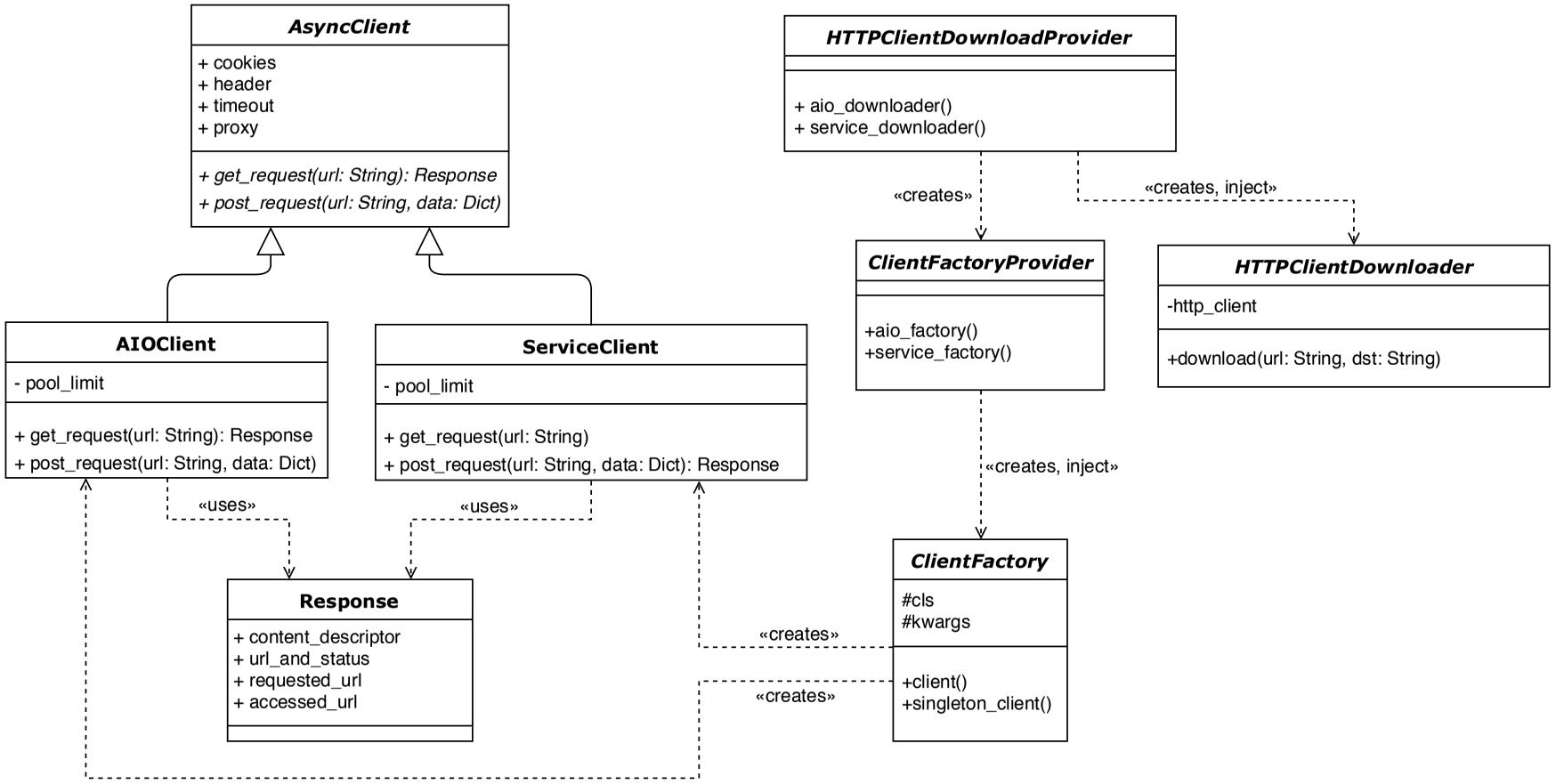
Klient

Asynchronní klient je implementovaný třídou ServiceClient abstraktní třídy AsyncClient definované frameworkem Lemmiwinks. Konkretní implementace ServiceClient je podobná již existující třídě AIOClient s těmito hlavními rozdíly:

- Třída implementuje metodu `post_request`. Pro stahování zdrojů je využíváno aplikační rozhraní stahovací služby, které zasílá požadavky metodou POST.
- Přijatá data jsou formátu JSON, který nese metadata obsahu a samotný obsah v kódování Base64. Tento obsah je potřeba dekódovat a uložit jako dočasný soubor pro potřeby frameworku.
- Metoda `get_request` není implementovaná, protože klient archivační služby přímo nestahuje žádné zdroje.

Pro klienta ServiceClient je přidána položka *service_factory* ve třídě frameworku ClientFactoryProvider pro zaregistrování nové implementace asynchronního klienta. Třída *HTTPClientDownloader* frameworku byla přepsána tak, aby komunikovala se stahovací službou přímo.

Na obrázku 5.4 je znázorněn diagram tříd upraveného modulu *httplib*. Abstraktní třída *AsyncJSClient* a třída *SeleniumClient* z původního Lemmiwinks nejsou využity a proto nejsou součástí diagramu.



Obrázek 5.4: Diagram tříd klienta archivační služby

Base64 a dekódování

Odpověď od stahovací služby s sebou nese data, které mohou být binární.*Base64* je kódování, které převádí binární data na sekvenci tisknutelných znaků [6]. Base64 je využito pro přenos binárního obsahu v textovém protokolu HTTP. Data zapouzdřena v JSON odpovědi vyžadují speciálně řetězce v kódování *UTF-8*. Platí, že data v kódování base64 jsou kompatibilní a převoditelná UTF-8.

Názvy archívů

ArchiveName je třída, která implementuje generování unikátních názvů archivů. Před uložením nového archívu se vygeneruje název bez přípony .maff, který se skládá ze 2 řetězců oddělených podtržítkem:

- Časové razítko vytvoření ve formátu `YYYYmmddTHHMMSS`. Například 20. dubna 2019 12:34:56 bude reprezentováno řetězcem `20190420T123456`.
- Identifikátor složený z 8 náhodných velkých písmen A-Z nebo číslic 0-9, například `TZEM5Y7I`

Generování náhodného identifikátoru zabrání kolizi, která by mohla nastat v případě vytvoření archívů ve stejný čas. Zároveň z identifikátoru nejde odvodit posloupnost jako u automaticky inkrementovaných identifikátorů. Identifikátor může následně být použit pro další práci a rozšíření.

Chybové stavy

Existuje několik situací, do kterých se služba při vyřizování požadavku na archivaci může dostat. Výsledkem může být vytvoření neúplného archívu nebo úplné přerušení archivace. V každém případě bude generovaná odpověď s příslušným stavovým kódem HTTP. Zde jsou některé chybové odpovědi:

- *400 Bad request* – formát požadavku nebo těla požadavku neodpovídá očekávané struktuře. Chybný JSON objekt je rozpoznán ještě před komunikací se stahovací službou. Archiv není vytvořen vůbec, a to i když některé požadované URL řetězce jsou validní.
- *404 Not Found* – API služby nezná URL požadavku.
- *405 Method Not Allowed* – použití nepovolené metody
- *408 Request Timeout* – zpracování požadavku trvalo příliš dlouho. K této chybě by teoreticky nemělo dojít, protože služba striktně neomezuje dobu zpracování požadavků na archivaci. Pokud některá součást stránky nebo obsah nelze stáhnout nebo selže požadavek na stahovací službu, archiv bude vytvořen neúplný a na požadavek bude reagováno úspěchem.
- *500 Internal Server Error* – nastala chyba z důvodu neočekávané výjimky při archivaci. Důvodem může být nedostatek paměti nebo implementační chyba. Server se z této chyby dokáže zotavit, ale výsledný archív bude v nedefinovaném stavu či nebude existovat vůbec.

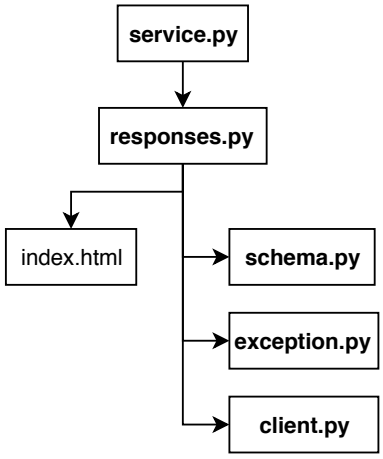
5.4 Stahovací služba

Účelem této služby je řešit komunikaci se vzdálenými servery, na kterých se nalézají požadované zdroje. Podobně jako u archivační služby je jádrem aplikace server a příslušná implementace HTTP klienta. Implementace se nalézá v adresáři `download_service` s top-level skriptem `service.py`. Tato služba očekává požadavky primárně od archivační služby. Implementační jazyk je Python 3.6.

Stahovací služba obsahuje tyto závislosti:

- Sanic
- Tor client for Ubuntu[17]
- pysocks
- aiohttp_socks
- jsonschema

Struktura zdrojových kódů je zobrazena na obrázku 5.5.



Obrázek 5.5: Struktura zdrojových souborů stahovací služby

Server

Serverová část využívá modul Sanic verze 19.03, implementace a využití je podobné archivační službě. Server je spouštěn lokálně na adrese `0.0.0.0` a poslouchá na portu `8081`.

Požadavky jsou obsluhovány těmito funkcemi:

- `post_download_handler` zpracovává požadavky POST na cestu `\download`. Během zpracování požadavku na běžnou URL adresu se využije obyčejný HTTP klient. Pokud je součástí požadavku atribut `useTor=true`, pro stažení se využije Tor klient.
- `get_root_handler` generuje hlavní stránku s informacemi o službě.

Property	Type	Required
resourceURL	string	true
useTor	boolean	false

Tabulka 5.2: JSON schema pro POST požadavky na stahovací službu

Klient

Klient stahovací služby využívá knihovnu aiohttp. Komunikuje se vzdáleným serverem, jehož zdroj je požadován k archivaci. Existují 2 různé implementace:

- **DownloaderClient** pro stahování běžným způsobem z *Visible webu*. Využívá knihovnu asyncio pro vytváření asynchronních požadavků. Z odpovědi vzdáleného serveru extrahuje historii přesměrování požadavku včetně konečné URL a vytvoří seznam `url_and_status`, který je spolu se staženým obsahem poslán zpět archivační službě.
- **TorDownloaderClient** je varianta pro anonymní stahování přes Tor. Využívá protokol SOCKS5[9] a běžící Tor-socks proxy na portu 9050. Umožňuje přistupovat na adresy .onion, které jsou využívané na deep webu.

Pro obě implementace je možnost nastavit *timeout* pro požadavky, zvlášť u Tor klienta je vhodné použít vyšší hodnotu.

Validace požadavků

Třída `DownloaderJsonSchema` v souboru `schema.py` implementuje validaci JSON struktury v požadavku POST na `/download`. Validace probíhá inicializací objektu DownloaderJsonSchema s parametrem instance, kterým je JSON objekt k validaci. V případě nevalidní JSON instance je vyvolána výjimka ValidationError. Schéma JSON zprávy je znázorněno v tabulce 5.2.

Atribut `useTor` je nepovinný a v případě jeho nepřítomnosti ve zprávě se předpokládá použití obyčejné implementace klienta.

Kódování staženého obsahu

Stažený netextový obsah, jako jsou například obrázky, je převáděn do textového kódování Base64. Tento formát lze reprezentovat řetězcem a je tedy možné jej přenést v JSON formátu. Base64 implementace v Pythonu je v modulu `base64`. Výsledný řetězec bajtů je potom převeden do UTF-8 textového kódování:

```
base64.b64encode(bytes(content)).decode('utf-8')
```

Kapitola 6

Testování

Tato kapitola se zaměřuje na testování implementovaných služeb na různých typech a skupinách stránek. Nejdříve budou ukázány testovací příklady a výsledky archivace na stránkách z tzv. *surface webu*, tedy stránky přístupné přes běžné klienty nebo prohlížeče. Další sekce se zaměřuje na testování na stránkách z *deep webu*.

V rámci testování se bude řešit především kvalita výsledných archívů, tedy kolik souborů a obsahu se podařilo stáhnout a archivovat s odhadem důležitosti obsahu, a rychlost archivace.

6.1 Testovací modul

K účelům testování služeb byl implementován modul `test_client`, který obsahuje Python 3.6 skript `test_client.py`. Testovací klient využívá knihovnu *requests* pro zasílání testovacích HTTP požadavků na archivační službu. Součástí modulu jsou soubory JSON, které jsou rozdělené na testovací vstupy podle tématických skupin webových stránek. Každý soubor obsahuje dvojce název testované stránky a seznam URL adres.

Testovací skript lze spustit jednoduše příkazem `python3.6 test_client.py`. Při testování na několika desítkách stránek může běh skriptu trvat až několik minut.

6.2 Stránky z deep webu

Tato sekce se zaměřuje na testování na stránkách z *deep webu* s využitím anonymní sítě Tor a onion routing. Vybrané adresy jsou součástí pseudodomény .onion a jsou dostupné pouze přes Tor. Bylo vybráno 50 adres z webů zaměřených na kolekci odkazů a wiki stránek, jako například *thehiddenwiki*[1]. Ačkoliv byla snaha vybrat stránky relativně slušné a prezentovatelné, stránky stále mohou obsahovat nelegální a citlivý obsah. Testované stránky z deep webu jsou rozděleny do souborů podle tématiky:

- Vyhledávače stránek deep webu a seznamy s odkazy – `test_link_sites.json` – 10 adres
- Tržní, obchodní, finanční a kryptoměnové weby – `test_finance.json` – 10 adres
- Služby – `test_commercial_services.json` - 10 adres

[1] https://thehiddenwiki.org/

- Blogy a rádia – `test_blogs.json` – 10 adres
- Politika a konspirační weby – `test_politics.json` – 10 adres

Nekteré z testovaných adres jsou již zastaralé nebo jsou nedostupné. Servery mohou také odmítnout požadavky od web crawleru. Archivace těchto webů vetšinou dopadne vypršením času na zpracování nebo vzdálený server přímo přeruší spojení. Ve vyhodnocení testů budou prezentované pouze adresy, na kterých byl server dosažitelný alespoň pro stažení hlavního dokumentu. Testování proběhlo třikrát a až na drobné odchylky v době zpracování se výsledky příliš nelišily.

6.3 Vyhodnocení

Testování služeb proběhlo na virtuálním stroji s operačním systémem Ubuntu 18.04.1 LTS s těmito prostředky:

- CPU: Intel Core i7-4700HQ CPU @ 2.40GHz
- RAM: 4 GB
- Síťová karta: Realtek PCIe GBE (1000 Mbit/s)
- Rychlost připojení: 20 Mbit/s download, 3 Mbit/s upload

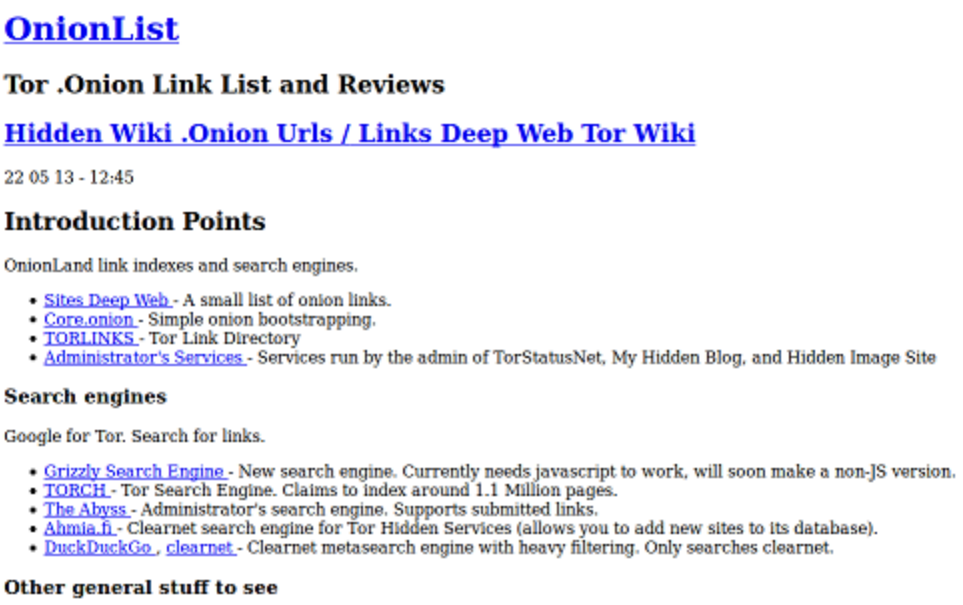
Z vybraných testovacích adres se podařilo spojit s 23 servery, z toho bylo možné úplně nebo částečně archivovat jejich obsah. Průměrná rychlost archivace je zobrazena v tabulkách pro každou skupinu stránek.

Tabulka 6.1 obsahuje výsledky rychlosti archivace z první skupiny testů. Jedná se o vyhledávače, wiki a seznamy, tedy stránky s poměrně jednoduchou dokumentovou strukturou.

Název stránky	**URL**	**Rychlost** [s]
DuckDuckGo Search Engine	`http://3g2upl4pq6kufc4m.onion/`	8.85
TORCH – Tor Search Engine	`http://xmh57jrzrnw6insl.onion/`	4.15
Uncensored Hidden Wiki	`http://zqktlwi4fecvo6ri.onion/`	1.26
Seeks Search	`http://5plvrsgydwy2sgce.onion/`	2.26
Onion Wiki	`http://wiki5kauuihowqi5.onion/`	2.79
The Hidden Wiki	`http://kpvz7ki2v5agwt35.onion/`	2.70
TorLinks	`http://torlinkbgs6aabns.onion/`	3.37
Hidden Wiki .Onion Urls	`http://jh32yv5zgayyyts3.onion/`	2.32

Tabulka 6.1: Výsledky rychlosti archivace z testové sady test_link_sites

Příklad výsledné stránky `Hidden Wiki .Onion Urls` je zachycena na snímku obrazovky 6.1. V tomto případě realná stránka používá soubory kaskádových stylů, které se nepodařilo stáhnout, přesto ale dokument obsahuje relevantní informace.



OnionList

Tor .Onion Link List and Reviews

Hidden Wiki .Onion Urls / Links Deep Web Tor Wiki

22 05 13 - 12:45

Introduction Points

OnionLand link indexes and search engines.

- Sites Deep Web - A small list of onion links.
- Core.onion - Simple onion bootstrapping.
- TORLINKS - Tor Link Directory
- Administrator's Services - Services run by the admin of TorStatusNet, My Hidden Blog, and Hidden Image Site

Search engines

Google for Tor. Search for links.

- Grizzly Search Engine - New search engine. Currently needs javascript to work, will soon make a non-JS version.
- TORCH - Tor Search Engine. Claims to index around 1.1 Million pages.
- The Abyss - Administrator's search engine. Supports submitted links.
- Ahmia.fi - Clearnet search engine for Tor Hidden Services (allows you to add new sites to its database).
- DuckDuckGo , clearnet - Clearnet metasearch engine with heavy filtering. Only searches clearnet.

Other general stuff to see

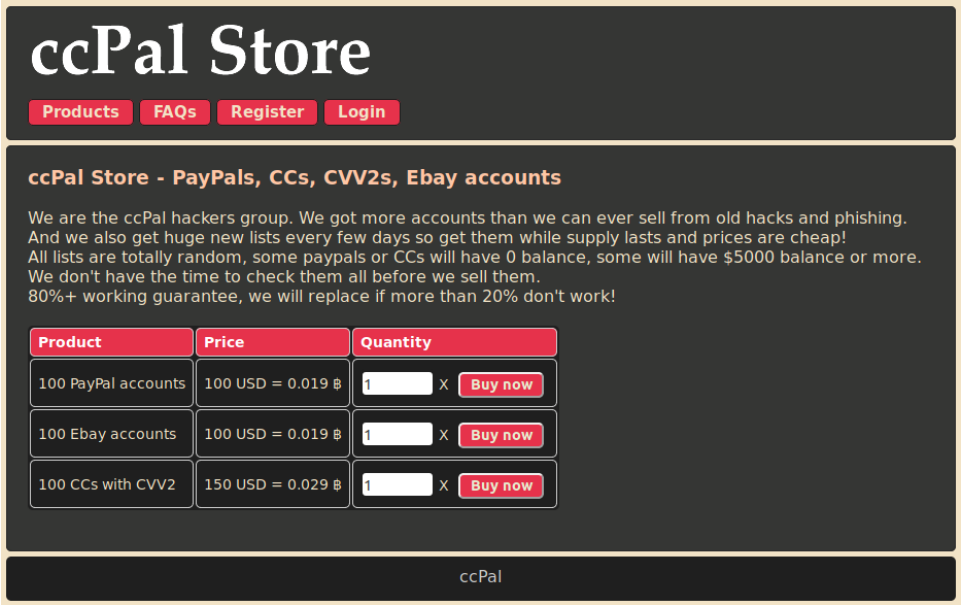
Obrázek 6.1: Snímek obrazovky stránky OnionList

Tabulka 6.2 zachycuje výsledky rychlosti archivace z druhé skupiny testů. V této skupině jsou stránky zaměřené na finance a kryptoměny.

Název stránky	URL	Rychlost [s]
EasyCoin	http://easycoinsayj7p5l.onion/	4.48
WeBuyBitcoins	http://jzn5w5pac26sqef4.onion/	1.43
OnionWallet	http://ow24et3tetp6tvmk.onion/	2.14
ccPal Store	http://3dbr5t4pygahedms.onion/	5.15
HQER	http://y3fpieiezy2sin4a.onion/	2.38
Counterfeit USD	http://qkj4drtgvpm7eecl.onion/	3.44

Tabulka 6.2: Výsledky rychlosti archivace z testové sady test_finance

Pro stránku `ccPal Store` je výsledná rekonstrukce na snímku obrazovky 6.2.

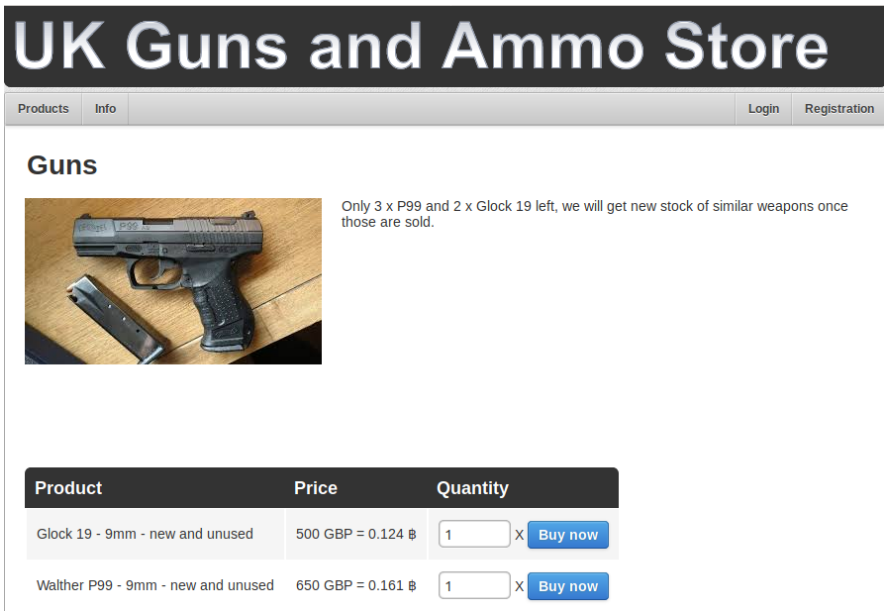


Obrázek 6.2: Snímek obrazovky stránky ccPal Store

Tabulka 6.3 obsahuje výsledky rychlosti archivace ze třetí skupiny testů. V této skupině jsou stránky zaměřené na obchod a služby. Na snímku obrazovky 6.3 je zrekonstruován online obchod se zbraněma `UK Guns and Ammo`.

Název stránky	URL	Rychlost [s]
Tor Market Board	http://6w6vcynl6dumn67c.onion/	2.12
UK Guns and Ammo	http://tuu66yxvrnn3of7l.onion/	2.75
Hidden BetCoin	http://hbetshipq5yhhrsd.onion/	7.62

Tabulka 6.3: Výsledky rychlosti archivace z testové sady test_commercial_services



Obrázek 6.3: Snímek obrazovky stránky UK Guns and Ammo

Tabulka 6.4 obsahuje výsledky rychlosti archivace ze čtvrté skupiny testů. V této skupině jsou blogy, rádia a news weby. Na snímku obrazovky 6.4 je rekonstrukce stránky rádiostanic Deep webu `Deep Web Radio`.

Název stránky	URL	Rychlost [s]
Beneath VT	http://74ypjqjwf6oejmax.onion/	3.28
Deep Web Radio	http://76qugh5bey5gum7l.onion/	5.30
All the latest news for tor	http://newsiiwanaduqpre.onion/	1.62

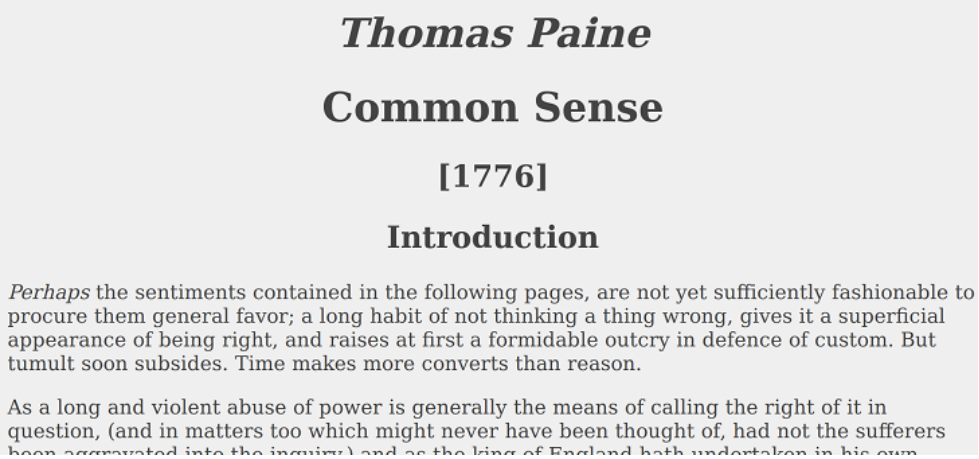
Tabulka 6.4: Výsledky rychlosti archivace z testové sady test_blogs

Obrázek 6.4: Snímek obrazovky stránky Deep Web Radio

Tabulka 6.5 obsahuje výsledky rychlosti archivace z páté skupiny testů. Tato sada testů obsahuje stránky zaměřené na politiku a informační/dezinformační weby. Na snímku obrazovky 6.5 je rekonstrukce blogového webu `Common Sense` autora Thomas Paine.

Název stránky	URL	Rychlost [s]
Bugged Planet	http://6sgjmi53igmg7fm7.onion/	4.15
Example rendezvous points page	http://duskgytldkxiuqc6.onion/	2.83
Common Sense by Thomas Paine	http://duskgytldkxiuqc6.onion/comsense.html	2.34

Tabulka 6.5: Výsledky rychlosti archivace z testové sady test_politics



Obrázek 6.5: Snímek obrazovky stránky Common Sense

Kapitola 7

Závěr

V rámci této bakalářské práce jsem si nastudoval problematiku hypertextových protokolů a exportní a archivační formáty MHTML a MAFF. Dále jsem představil principy frameworků pro archivaci webu. Pro řešení webových služeb jsem zvolil již existující archivační framework Lemmiwinks.

Protože framework řeší dílčí problémy modulárně, byla aplikace rozdělena do 2 samostatných služeb - archivace a stahování. Obě tyto webové služby jsou navrženy tak, aby mohly být vyvíjeny samostatně. Dohromady tvoří systém, který zajišťuje archivaci a rekonstrukci webu podobně jako konzolové řešení využívající Lemmiwinks. Komunikace služeb mezi sebou i okolním světem probíha pomocí aplikačního rozhraní vycházejícího z principů REST API. Byla navržena sada HTTP zpráv pro komunikaci, které tvoří aplikační rozhraní těchto služeb.

Služby byly implementované v jazyce Python 3.6 s využitím archivačního frameworku Lemmiwinks. Framework byl částečně upraven, aby vyhovoval návrhu služeb. Serverová implementace obou služeb je založena na asynchronním frameworku Sanic, který dosahuje vyššího výkonu než běžné synchronní implementace. Implementace byla otestovaná na vzorku stránek z deep webu, přičemž u úspěšných pokusů byla změřena celková doba archivace a příklady snímků obrazovky výsledné rekonstrukce.

V této práci lze dále pokračovat například implementací autentizace, rozšíření API služeb o další možnosti nastavení archivace nebo implementace mezipaměti a cachování stažených souborů.

Literatura

[1] D. Kristol, L. M.: *RFC 2109 HTTP State Management Mechanism*. 1997, [Online; navštíveno 18.10.2018].
URL https://tools.ietf.org/html/rfc2109

[2] Erl, T.: *SOA: Principles of Service design*. 2008, [Online; navštíveno 26.10.2018].
URL http://serviceorientation.com/index.php/serviceorientation/index

[3] Fielding, R. T.: *Representational State Transfer (REST)*. 2000, [Online; navštíveno 27.10.2018].
URL
https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

[4] Graham Leggett: *Apache Module mod_proxy*. [Online; navštíveno 29.1.2019].
URL http://httpd.apache.org/docs/2.0/mod/mod_proxy.html

[5] J. Palme, A. H.: *MIME Encapsulation of Aggregate Documents, such as HTML (MHTML)*. 1999, [Online; navštíveno 2.11.2018].
URL https://tools.ietf.org/html/rfc2557

[6] Josefsson, S.: *The Base16, Base32, and Base64 Data Encodings*. 2006, [Online; navštíveno 4.4.2019].
URL https://tools.ietf.org/html/rfc4648

[7] JSON Schema. [Online; navštíveno 11.4.2019].
URL https://json-schema.org/

[8] Luotonen, K., Ari a Altis: World-Wide Web proxies. *Computer Networks and ISDN Systems*, ročník 27, 11 1994: s. 147–154, doi:10.1016/0169-7552(94)90128-7.

[9] M. Leech, Y. L. R. K. D. K. L. J., M. Ganis: *SOCKS Protocol Version 5*. 1996, [Online; navštíveno 11.5.2019].
URL https://tools.ietf.org/html/rfc1928

[10] Maximum URL length. [Online; navštíveno 11.4.2019].
URL https://support.microsoft.com/en-us/help/208427/maximum-url-length-is-2-083-characters-in-internet-explorer

[11] Postel, J.: Transmission Control Protocol. RFC 793, RFC Editor, September 1981.
URL https://www.rfc-editor.org/rfc/rfc793.txt

[12] R. Fielding, J. G.: *RFC 2616 Hypertext Transfer Protocol – HTTP/1.1*. 1999, [Online; navštíveno 14.10.2018].
URL https://tools.ietf.org/html/rfc2616

[13] Routing - Sanic 19.03.1 documentation. [Online; navštíveno 11.4.2019].
URL https://sanic.readthedocs.io/en/latest/sanic/routing.html

[14] Serečun, V.: *Automatic web page reconstruction*. Diplomová práce, Brno University of Technology, 2018.
URL http://www.fit.vutbr.cz/study/DP/DP.php.cs?id=21138&file=t

[15] Solomon, B.: *Async IO in Python: A Complete Walkthrough*. 2019, [Online; navštíveno 4.4.2019].
URL https://realpython.com/async-io-python/

[16] T. Dierks, E. R.: *RFC 5246 The Transport Layer Security (TLS) Protocol*. 2008, [Online; navštíveno 14.10.2018].
URL https://tools.ietf.org/html/rfc5246

[17] *Installing Tor on Debian/Ubuntu*. 2019, [Online; navštíveno 11.5.2019].
URL https://2019.www.torproject.org/docs/debian.html.en

[18] Wouters, T.: *Global Interpreter Lock*. 2017, [Online; navštíveno 10.4.2019].
URL https://wiki.python.org/moin/GlobalInterpreterLock