České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra řídicí techniky

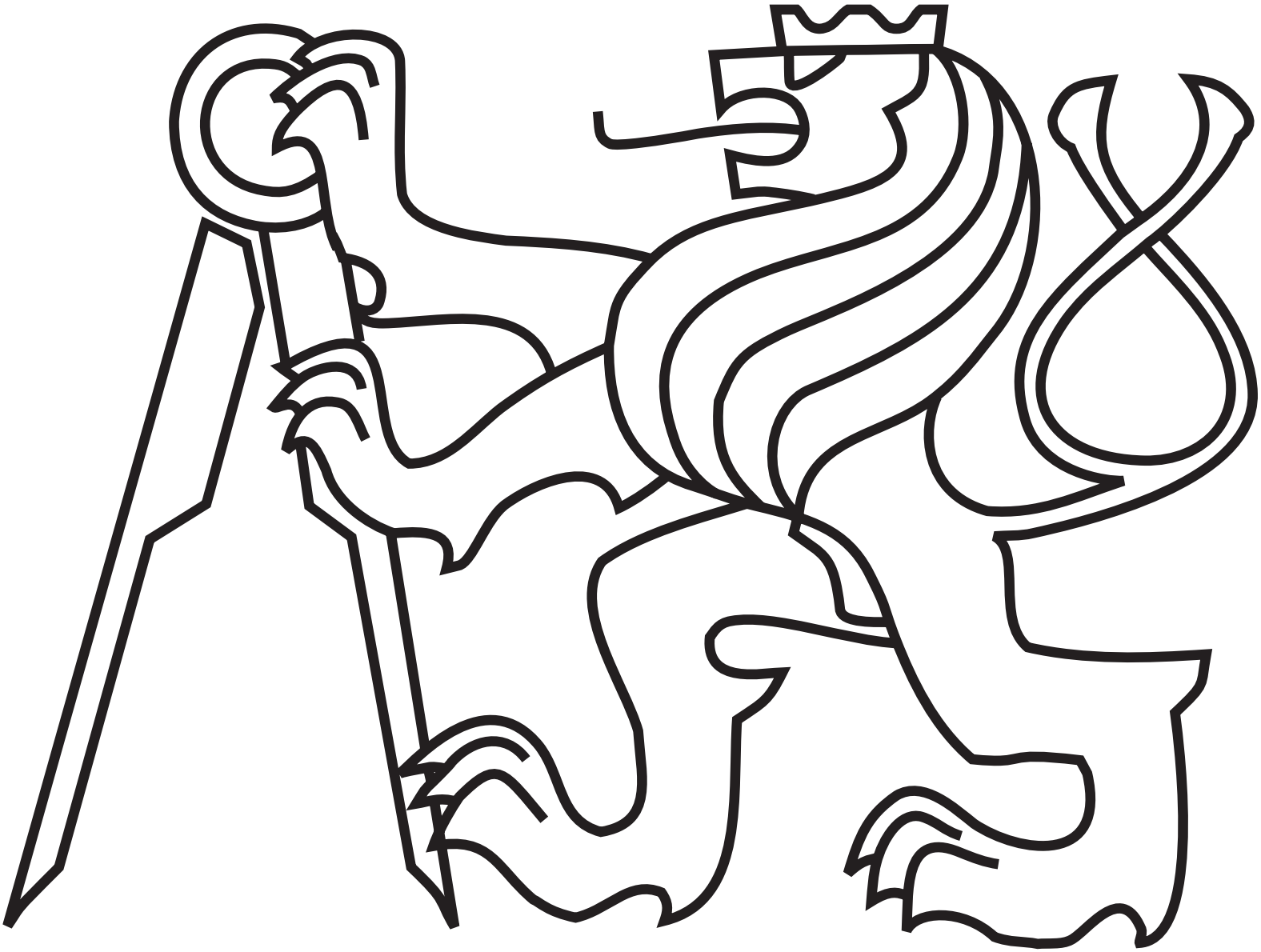
Diplomová práce

Automatizovaný systém stahování webového obsahu potřebného k doplňování cleansetu

Michal Staněk

Vedoucí práce: Ing. Jan Kubr, Ph.D.

6. března 2020



Poděkování

Chtěl bych poděkovat panu Ing. Janu Kubrovi, Ph.D. za odborné vedení mé práce, za pomoc a věcné rady při zpracování této práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

V Praze dne 6. března 2020 .....

České vysoké učení technické v Praze
Fakulta elektrotechnická

Odkaz na tuto práci

Staněk, Michal. *Automatizovaný systém stahování webového obsahu potřebného k doplňování cleansetu.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta elektrotechnická, 2020.

Abstrakt

TODO

Klíčová slova Python, Virtualbox, Fiddler, html, js

Abstract

TODO

Keywords Python, Virtualbox, Fiddler, html, js

Obsah

Seznam tabulek

Kapitola 1

Úvod

Dnešní doba je plná rizik, která představují hrozbu pro každodenního uživatele internetu. Ať už se jedná o phishing (zisk citlivých údajů pomocí podvodné internetové komunikace) či různé druhy malwaru (nežádoucí programy mající za úkol poškodit uživatele). V boji s těmito riziky je důležité chránit sebe a svoje data pomocí antivirových programů. Jedním z nejrozšířenějších je Avast, který má přes 435 milionů aktivních uživatelů a měsíčně zabrání okolo 2 miliardám útoků[1].

1.1 Motivace

Avast, stejně jako většina antivirových programů, uchovává informace o všech známých škodlivých entitách. Tato databáze se denně rozšiřuje o spousty nových záznamů, které obsahují nejen informace o celých souborech, ale i kusy kódu webového obsahu (tzv. string detekce), které jsou považovány za příznak podvodných úmyslů. Může se však stát, že je tento kus kódu moc obecný a dochází tak i k blokování čistého obsahu (tzv. false-positive detekcím). Aby se těmto situacím předcházelo, je zapotřebí udržovat i databázi s čistými záznamy (tzv. cleanset). Tyto záznamy jsou převážně HTML a .js soubory.

Dříve, než se nová string detekce začlení do jádra antiviru, je její obsah porovnán se všemi záznamy na cleansetu a pokud dojde ke shodě (tj. detekční string je součástí nějakého souboru na cleansetu), je tato detekce považována za nevalidní. Tímto dochází k zabránění fals-positive detekcím.

Ideálním stavem je tedy mít záznam o veškerém čistém obsahu internetu, což je samozřejmě nemožné. Avšak čím více záznamů cleanset obsahuje, tím kvalitnější je běh antivirového programu. V současné době dochází k doplňování cleansetu pouze občasně a to převážně manuálně za pomoci jednoduchých scriptů.

1.2 Cíle práce

Hlavním cílem této práce je vytvořit plně automatizovaný systém, který bude databázi s čistými záznamy periodicky doplňovat o nový obsah, čímž by mělo dojít ke zlepšení funčnosti antivirového programu. Dále bude potřeba systém začlenit do již stávající infrastruktury. Primárním úkolem je tedy vytvořit systém, který by modernizoval doplňování cleansetu, avšak současně je možné jej zobecnit k využití i v jiných aplikacích. Systému bylo dáno kódové označení *Magpie* (česky Straka), protože aplikace, stejně jako straky, bude shromažďovat data z různých míst a ukládat je na jedno místo. Výsledná aplikace bude řádně otestována a bude zhodnocen její přínos.

Kapitola 2

Analýza a řešení problému

Jednotlivé body práce by se daly rozdělit na vícero dílčích podproblémů:

- Výběr programovacícho jazyka
- Předzpracování zadaných adres webových stránek
- Získání čistých souborů z webových stránek
- Zabezpečení stahovacího procesu
- Nahrání získaného obsahu do databáze cleansetu
- Začlenění do stávající infrastruktury

2.1 Výběr programovacího jazyka

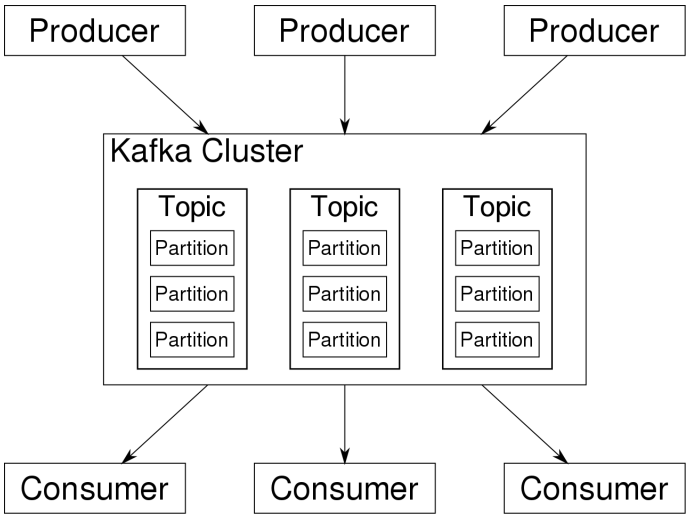
Podle prvních odhadů a požadavků bude práce obsahovat více odlišných částí, které by měly být jednoduše ovladatelné a propojitelné. K tomu by šlo využít programovacího jazyka *Python*(3.1), který obsahuje velké množství dostupných knihoven. Tato volba je i v soulady s firemní politikou.

2.2 Předzpracování zadaných adres

Předzpracování zadaných adres se bude řešit pomocí frontového systému *Kafka*(3.5).

Systém by měl reagovat na dva typy vstupů. Prvním vstupem budou adresy s vysokou prevalencí (případem takové adresy může být třeba internetový obchod `www.amazon.com`), které budou systému periodicky dodávány z externích modulů. Vstupem ale může být i ručně zadaná adresa či seznam adres v případě, kdy by operátor systému potřeboval na cleanset dodat soubory z webových stránek s menší prevalencí, či, ve více obecném řešení, by potřeboval stáhnout zdrojové soubory cílených webových stránek pro odlišné účely.

V obou případech se vložené adresy nahrají do frontového systému, odkud se budou postupně odebírat (Obr.2.1). Využití *Kafky* má výhodu v již naimplementovaném řešení fronty. Jednou z funkcionalit *Kafky*, které by zde šlo využít, je potvrzování zpracované zprávy po přijetí. Tím dojde vždy ke zpracování všech zpráv ve frontě.



Obrázek 2.1: Využití frontového systému Kafka

2.3 Získání čistých souborů z webových stránek

K získání čistých souborů je možné přistoupit dvěma způsoby. Jednou z možností je otevírat stránky ve webovém prohlížeči a k získání souborů použít nástroj *Fiddler*(3.2), druhým způsobem je emulace webového prohlížeče v *Pythonu* a stahování zdrojových kódů stránek.

Další problematikou je ošetření přesměrovávání (tzv. redirecty), které se na spoustě webových stránek používá. Jedna z možností je redirecty neřešit a zabývat se pouze obsahem dané url. To by proces získání zdrojových souborů usnadnilo. Není to ale příliš robustní řešení. Bylo by tedy rozumné s přesměrováváním webových stránek počítat.

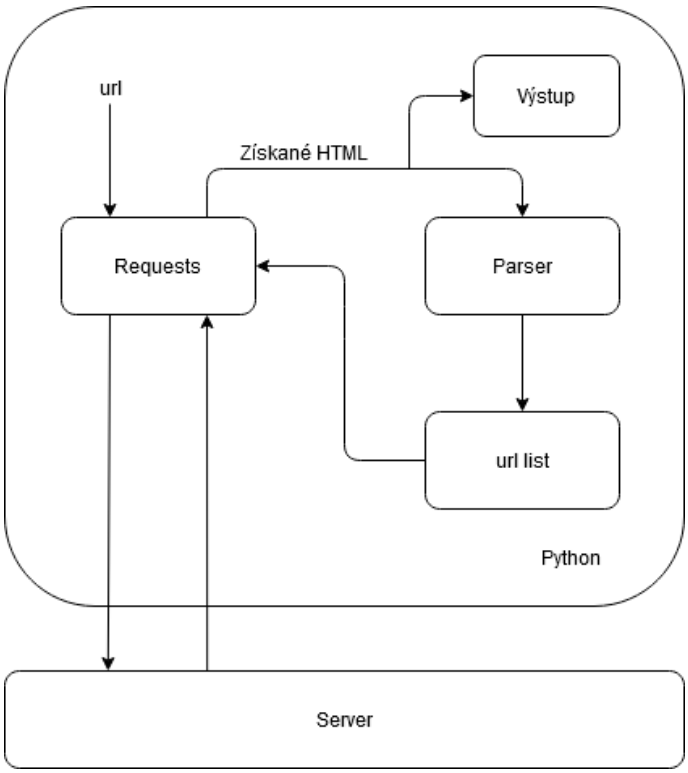
Také by zde měla být implementována logika selekce pouze HTML a .js souborů, ostatní soubory pro funkčnost cleansetu nejsou důležité. To je možné provádět přímo při stahování souborů, nebo vždy pro zadanou adresu stáhnout všechny soubory a selekci provést následně.

Jednotlivé rozebírané metody získání čistých souborů z webových stránek jsou tedy následující:

- Emulace webového prohlížeče a následné parsování webových stránek
- Spouštění stránek v prohlížeči a zachytávání komunikace Fiddlerem

2.3.1 Emulace webového prohlížeče

Oproti druhé metodě s využitím nástroje *Fiddler* by byla emulace webového prohlížeče bezesporu rychlejší. Pro práci s webovými stránkami v *Pythonu* existuje více knihoven, avšak nejčastěji se používá knihovna *Requests*. Její interface je daleko snazší na použití než u knihovny *Urllib*, jejíž výhodou je pouze fakt, že je již obsažena v základní instalaci *Pythonu* a není nutno ji doinstalovávat. Nevýhoda knihovny *Requests* je v horší práci se stránkami s



Obrázek 2.2: Diagram zachytávání komunikace pomocí web scrapperu

přesměrováváním pomocí javascriptu nebo obecně s načítáním obsahu pomocí javascriptu. Vytvořit komplexní web scrapper (tj. nástroj, který prochází obsah webových stránek), který by dokázal reagovat i na javascriptem řízený obsah, je netriviální úkol (Obr.2.2).

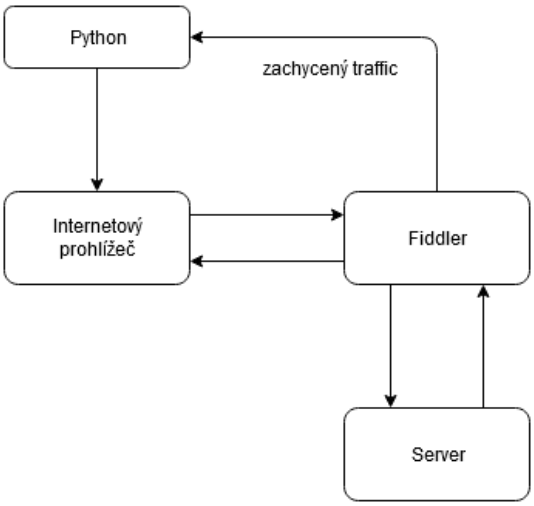
K samotnému parsování již stažené webové stránky je možné využít knihovnu BeautifulSoup[4], která implementuje HTML a XML parsery v jazyku Python. Pomocí ní je snadné procházet HTML kód a iterovat přes tagy komponentů stránky. Bylo by tedy potřeba vyhledat všechny části, které obsahují spouštění javascriptového souboru nebo přesměrování na jinou adresu, tyto soubory, respektive adresy, uložit do fronty a následně provézt stejný proces pro všechny ještě nezpracované položky fronty s postupným ukládáním již zpracovaných souborů.

Přesměrování lze ošetřit pomocí hlídání response kódů (kód, který vrací server při komunikaci s webovým prohlížečem). Při přesměrování jsou běžné kódy 301, respektive 302. Pokud tedy vrátí server kód pro přesměrování, je nutné ve zdrojovém kódu stránky najít adresu pro přesměrování, stáhnout její obsah a tento soubor přidat do fronty k zpracování.

Velkou výhodou tohoto řešení je absence nutnosti používat virtuální stroj, protože samotné stahování zdrojového kódu stránek by probíhalo bez nustnosti stažené soubory spouštět, čímž by nehrozilo nebezpeční infikace pracovního počítače škodlivým obsahem (o této problematice více v sekci 2.4). Výraznou nevýhodou je ovšem neschopnost získání obsahu webových stránek, který se načítá se zpožděním za pomoci javascriptu (tzv. lazy loading).

2.3.2 Zachytávání komunikace Fiddlerem

Z tohoto důvodu by bylo snazší použít nástroj *Selenium*(3.3), čímž už dojde k určitému zpomalení z nutnosti spouštění internetového prohlížeče, avšak odpadne nutnost implementace sledování redirectů a postupného načítání stránek pomocí javascriptu. Stále je ovšem potřeba načtené stránky nějak zpracovat, což by bylo možné pomocí *Fiddleru*(3.2). Tím sice opět vzniká další zpomalení kvůli spouštění dalšího programu, řešení ale přináší téměř kompletní implementaci rozebíraného problému s možnou variací úprav pomocí konfiguračního souboru. Protože je *Fiddler* původně vyvíjen pro testovací účely a



Obrázek 2.3: Diagram zachytávání komunikace Fiddlerem

sledování internetové komunikace (tzv. traffic), zachytává veškerý traffic, který skrze nástroj proudí (Obr.2.3). Bylo by tedy potřeba implementovat logiku pro třídení a následnou selekci HTML a .js souborů.

2.3.2.1 Selekce HTML a .js souborů

Prvním přístupem je emulace webového prohlížeče. Přináší výhodu v tom, že se při parsování HTML stránek rovnou přistupuje pouze k .js a HTML souborům a tím odpadá nutnost následně nějakou selekci provádět. Metoda s použitím *Fiddleru* je v tomto složitější. *Fiddler* je komplexní nástroj a automaticky zachytává veškerou komunikaci - nejen soubory potřebné k vykrelení webové stránky, ale i režijní komunikaci mezi prohlížečem a serverem. Tento přístup je možné změnit pomocí již zmíněného inicializačního scriptu.

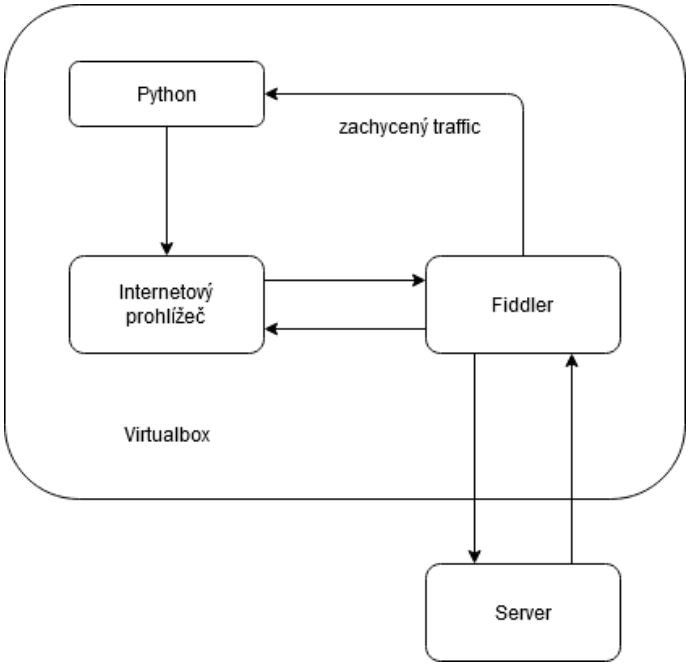
Avšak problémy této metody přináší i webový prohlížeč, kterého se zde využívá. Téměř vždy při spuštění prohlížeče probíhá nevyžádaná komunikace prohlížeče se servery, která je potřeba vytřídit. I zde je možné využít inicializačního scriptu *Fiddleru*, implementovat třídění již při zachytávání komunikace a zachytávat pouze HTML a .js soubory ze serverů, které odpovídají zpracovávané url. To ovšem přináší problémy s přesměrováním, které může seznam chtěných serverů libovolně navyšovat, což už by pro inicializační script představovalo nelehký úkol a takové řešení patrně nebude optimální. Druhý způsob je zachytávat HTML a .js komunikaci ze všech serverů a selekci řešit až později při zpracování dat v *Pythonu*. Nedostatkem této možnosti je zvýšená náročnost na paměť, kterou představuje zpracovávání více souborů. Předpokládá se ale, že tato nevyžádaná komunikace bude v poměru s chtěnými soubory minimální, tudíž by k výrazné zvýšení náročnosti na paměť dojít nemělo.

2.4 Zabezpečení stahovacího procesu

Z bezpečnostních důvodů je kladen důraz na to, aby byl veškerý proces stahování webového obsahu zabezpečen. Předpokládá se sice, že všechny stažené soubory budou nezavirované, avšak spoléhat se na to není moc bezpečné řešení. Jedním z možných způsobů jak docílit základní bezpečnosti je vytvoření a obsluha virtuálního stroje, na kterém by docházelo ke stahování souborů.

Celý koncept zabezpečení stahování pomocí virtuálního systému je ovlivněn použitím nástroje *Fiddler*(3.2) pro stahování souborů z webových stránek. Tato metoda by využívala *Fiddler*, který by běžel na pozadí, spolu s prohlížečem, v kterém by byly stránky otevírány a pomocí *Fiddleru* zachytávána komunikace (Obr.2.4). Právě otevírání stránek v prohlížeči, během čehož dochází ke spouštění javascriptových souborů, je z pohledu bezpečnosti potenciálně nebezpečné. Toho by se dalo vyvarovat emulací prohlížeče přímo v *Pythonu*, čímž by se stahovaly rovnou zdrojové kódy webových stránek bez nutnosti jejich spouštění. Tato metoda však přináší problémy, které již byly popsané v subsekci 2.3.1.

Pokud se jedná o nástroje umožňující virtualizaci systému, lze použít *VMware workstation* od firmy VMware nebo *Virtualbox* vyvíjený firmou Oracle. Firma VMware nabízí pro práci se svými virtuálními stroji infrastrukturu na-



Obrázek 2.4: Zaobalení stahovacího procesu virtuálním strojem

zývající se *vSphere*. Tato infrastruktura obsahuje vlastní SDK nástroje pro implementaci do programovacího jazyka *Python*[9], avšak celé toto řešení se nenabízí s freeware licencí. Z tohoto důvodu by bylo lepší použít virtualizační nástroj *Virtualbox*(3.6), který je zdarma. *Virtualbox* od firmy Oracle také obsahuje vlastní SDK pro podporu *Pythonu*, pro které je už napsaná knihovna *pyvbox*[10]. Tato knihova obaluje většinu metod, které SDK *Virtualboxu* obsahuje.

Z hlediska bezpečnosti by bylo rozumné spouštět virtuální stroj (neboli VM) pro každou webovou stránku zvlášť, to by ale celý proces výrazně zpomalovalo. Předpokládá se, že nejdelší dobu zabere právě startování VM. Jiné řešení by nabízelo restartovat virtuální stroj vždy po určitém čase nebo po daném množství zpracovaných url adres. Tím by se běh systému výrazně zrychlil.

2.5 Nahrání získaného obsahu do databáze cleansetu

Dalším krokem bude přesun získaného obsahu do samotné databáze cleansetu. Ta je součástí většího systému aplikací, který nese interní označení Scavenger. Zjednodušeně lze říci, že tento systém obsahuje záznamy o všech antiviru známých souborech a url adresách (nakažených i čistých). Soubory se v systému Scavenger ukládají v podobě hashe vzniklé pomocí hashovacího algoritmu sha-256. Tímto lze docílit jednoduché kontroly duplicity (stejné soubory mohou

mít rozdílné názvy, ale hash souboru je pro identické soubory stejná). K hashi souboru se přikládají metadata mimojiné s informací o původu souboru, času výskytu a prevalencí.

K nahrání souborů do Scavengeru lze využít síťový souborový systém Sambu, který implementuje přenos souborů po síti pomocí síťového protokolu SMB a to převážně v systémech Windows. Tato metoda je však z hlediska firemní infrastruktury zastaralá. Novější způsob představuje využití datové platformy HCP (Hitachi content platform). Tato platforma se specializuje na přesun a zpracování velkého množství dat z různých zdrojů. Ke komunikaci se zmíněným systémem HCP lze využít interně vyvinutý python klient, který přesun souborů usnadní.

2.6 Začlenění do stávající infrastruktury

Systém bude spouštěn periodicky, ale měl by být také spustitelný na vyžádání uživatelem. Pro takové požadavky lze použít systém Jenkins(3.4), který je firmou Avast používaný k periodickému spouštění procesů. Jednotlivé části systému Magpie je možné oddělit do samostatných procesů (v terminologii systému Jenkins tzv. jobů), které se dají sekvenčně pouštět v závislosti na úspěšném ukončení předcházejícího jobu. Tento přístup přináší přehledné rozhraní, v kterém je možné jednotlivé části samostatně monitorovat, spolu s jednoduchým přístupem k výstupům jobů. Tímto způsobem by bylo možné přistoupit k získaným datům přímo, bez nutnosti data nahrávat do databáze v systému Scavenger, v případě, kdy by byl systém spuštěný manuálně.

Jiným přístupem je využít firemní mutaci nástroje Kubernetes interně nazývanou Luft, která by umožňovala mít systém spuštěný bez přestávky. Velkou výhodou Kubernetes je jednoduché škálování, kdy lze při velkém vytížení jednoduše navýšit výpočetní prostředky danému procesu a tím urychlit jeho běh.

Je také možné pro jednotlivé části systému Magpie využít rozdílné technologie, a to kombinaci obou výše zmíněných.

Kapitola 3

Použité technologie

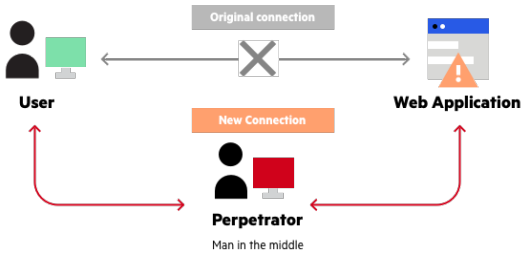
V této kapitole jsou stručně popsány všechny technologie využité při zpracovávání této práce.

3.1 Python 3.7

Python je skriptovací programovací jazyk, jehož syntaxe je lehce odlišná od konvenčních programovacích jazyků (Java, C) v tom, že nepoužívá středníky ani složené závorky. Jedná se o hybridní programovací jazyk, což znamená, že program nemusí být nutně objektově orientovaný, ale části mohou mít více procedurální charakter. Tím dochází k lepší čitelnosti kódu a celkovému zjednodušení. Síla Pythonu je i ve velkém množství balíků s knihovnami, které podporují jeho všestrannost. Kvůli těmto vlastnostem byl vybrán pro tuto diplomovou práci.

3.2 Fiddler

Fiddler[3] je nástroj vyvíjen firmou Telerik, sloužící k zachytávání internetové komunikace. Funguje na principu MitM (Man-in-the-middle) útoku, kdy se útočník vtěsná mezi dva účastníky internetového provozu a nechá je komunikovat skrz sebe. Zde je však tento útok chtěný (Obr.3.1). Jeho automatizace



Obrázek 3.1: MitM útok [5]

lze docílit inicializačním souborem, který obsahuje různá pravidla a je psaný v javascriptu. Při správném nastavení je fiddler schopný zachytávat i šifrovanou komunikaci, kvůli čemuž byl použit v této práci.

3.3 Selenium

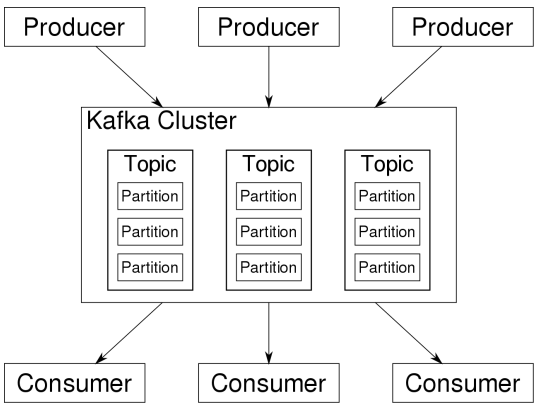
Selenium je opensource nástroj používaný k automatizovanému přístupu k webovým aplikacím. Těchto vlastností se často využívá při testování, avšak v této práci je použit pouze k obsluze webového prohlížeče. Selenium obsahuje vlastní vývojové prostředí, které lze využít bez velké znalosti programování, existují však i jeho implementace do většiny populárních programovacích jazyků.

3.4 Jenkins

Jenkins je opensource CI/CD systém (continuous integration and delivery) umožňující vykonávání automatických či periodických operací (tzv. jobů). Používá se převážně k spouštění buildů a udržování testů. Joby lze spoustět automaticky po vzniku nové verze vyvíjeného programu, lze je ale i spouštět pravidelně v předem určený čas, čehož bylo v této práci využito.

3.5 Kafka

Kafka by se dala zařadit mezi frontové systémy. Jedná se o opensource platformu sloužící ke streamování dat v reálném čase. Její zaměření je převážně na procesing velkého množství zpráv bez narůstající latence. Zprávy se však neukládají do fronty, jako je tomu třeba u RabbitMQ (tj. jiný frontový systém), avšak do tzv. topiců. Jeden Topic může obsahovat více částí (tzv. partition), do kterých se zprávy distribuují (Obr.3.2). Zprávy do topicu posílá proces,



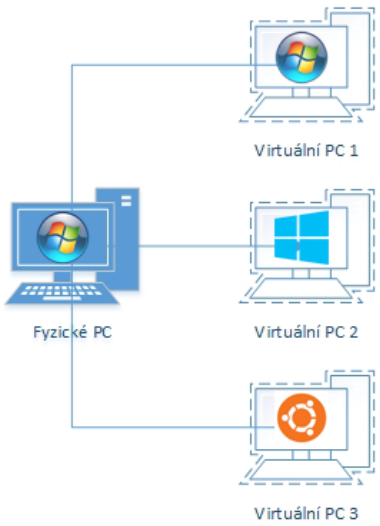
Obrázek 3.2: Znázornění Kafka systému [6]

který se nazývá Producer. Obdobně proces, který data čte, je nazýván Consumer. Ten dostává zprávy z topicu v závislosti na indexaci a časové známky. K jednomu topicu může být přihlášeno více nezávislých konzumerů, kteří jsou rozděleni do rozdílných skupin a každá skupina dostává identické zprávy, čím se zabrání vzájemnému čtení stejných zpráv.

Zprávy zůstávají v topicu po určitou dobu, což určuje hodnota retence. Po tuto dobu jsou zprávy přístupny pro každou skupinu konzumerů. Síla kafku je v politice přečtených zpráv. Na rozdíl od zmiňovaného RabbitMQ, který přečtením zprávy zprávu odstraní ze své fronty, funguje v kafce tzv. potvrzovací systém. Konzumer dostane zprávu, a po jejím zpracování vyšle tzv. commit, kterým oznámí úspěšné zpracování. Pokud by v průběhu nastala chyba, tento commit se nepošle a zpráva se vrátí zpátky, kde může být přečtena jiným konzumerem ve skupině.

3.6 Virtualbox

Virtualbox je opensource virtualizační nástroj vyvíjen firmou Oracle. Slouží k instalaci virtuálních operačních systémů na jednom fyzickém stroji. Jeho výhodou je multiplatformnost, což znamená, že je možné ho nainstalovat na MS Windows i operační systémy s unixovým jádrem (Linux, Mac OS). Tímto lze docílit například spuštění linuxového systému pod operačním systémem Windows.[7]



Obrázek 3.3: PC s virtuálními stroji [8]

Další důležitou funkcionalitou Virtualboxu jsou tzv. snapshoty. Snapshot zachycuje virtuální stroj a veškeré jeho nastavení v daném čase, v kterém je vytvořen. Tímto lze jednoduše vracet provedené změny zpět do bodu vytvo-

ření snapshotu. Toho je možné využít při testování aplikací, využití je avšak možné i v oblasti bezpečnosti. Pokud je vytvořen snapshot čistého systému, lze se do něj vraátit v případě potenciálního nakažení virtuálního stroje. Této funkcionality se využívá i v této práci.

Kapitola 4

Implementace

Z analýzy je patrné, že by zvoleným programovacím jazykem pro systém Magpie měl být Python. Prvním krokem implementace projektu je vytvořit funkční jádro, ke kterému by bylo možné přidávat ostatní komponenty a vylepšovat jeho funkcionalitu. Za toto jádro lze považovat stahování zdrojového kódu stránek. Následně je potřeba tento proces zabezpečit, zdokonalit a integrovat do stávající infrastruktury Avastu. Jednotlivé sekce implementace jsou následující:

- Klient pro stahování webového obsahu
- Zpracování dat a upload do databáze
- Virtualizace
- Implementace frontového systému Kafka
- Integrace pomocí systému Jenkins

4.1 Klient pro stahování webového obsahu

Při implementaci klientu pro stahování webového obsahu byla zvolena metoda s využitím programu Fiddler pro zachytávání internetové komunikace. Tato metoda přináší komplexnější odposlech bez nutnosti emulace webového prohlížeče. Bylo nutné vytvořit skript pro ovládání internetového prohlížeče a Fiddleru, tento skript byl napsaný v jazyku Python, a dále bylo zapotřebí nastavit Fiddler pomocí inicializačního souboru, který je napsaný v javascriptu. Sekce je tedy rozdělena do dvou podsekcí:

- Skript pro ovládání pomocných programů
- Skript pro nastavení Fiddleru

4.1.1 Skript pro ovládání pomocných programů

Tento skript je psaný v programovacím jazyku Python, stejně jako většina práce. Jeho hlavní činností je spuštění programu Fiddler a správa internetového prohlížeče. K ovládání webového prohlížeče byl použit nástroj Selenium(3.3) v podobě knihovny importované do jazyka Python. Pro komunikaci s webovým prohlížečem je nutné nainstalovat takzvaný webdriver, který zajistí správnou konfiguraci Selenia pro daný prohlížeč. Webdriver Geckodriver[11], je určený pro komunikaci s internetovým prohlížečem Mozilla Firefox[12], naproti tomu webdriver ChromeDriver[14] je vyvinut pro komunikaci s internetovým prohlížečem Google Chrome[13]. V práci byl využit internetový prohlížeč Mozilla Firefox, tudíž i nástroj Geckodriver.

Jednoduchá implementace Selenia s ovládáním webového prohlížeče v jazyku Python může vypadat jako ve výřezu kódu (tzv. snippetu) 4.1.

```
import time

from selenium.webdriver import Firefox
from selenium.webdriver.firefox.options import Options

opts = Options()
opts.headless = True

browser = Firefox(options=opts)
browser.get("https://www.seznam.cz")

time.sleep(2)
browser.quit()
```

Obrázek 4.1: Implementace nástroje selenium s ovládáním prohlížeče

Selenium nabízí možnost konfigurace webdriveru, což umožnuje nastavovat různé parametry. V tomto případě je zvolen přepínač `headless`, který určuje spouštění webového prohlížeče bez grafického uživatelského prostředí (tvz. GUI), čímž lze docílit rychlejšího průběhu. Samotné načtení webové stránky se provádí pomocí metody `get()`, kterému se do parametru dá url stránky (ve snippetu stránka `https://www.seznam.cz`). Příkazem `quit()` lze internetový prohlížeč zavřít. Metoda `get()` je implementována, aby čekala na úplné načtení stránky, avšak nepočítá s postupným načítáním javascriptových souborů. Pro kompletní načtení webové stránky je tedy potřeba přidat časovač (tzv. timer), který před zavřením prohlížeče skript pozastaví na určitou dobu (zde 2 sekundy).

4.1.2 Skript pro nastavení Fiddleru

Program Fiddler se primárně ovládá přes grafické prostředí. Je však možné změnit jeho výchozí konfiguraci tak, aby se při spuštění rovnou načetl do požadovaného nastavení. K tomuto účelu slouží inicializační skript, který nese označení `CustomRules.js`, jedná se tedy o skript psaný v javaScriptu. Fiddler při spuštění vždy načítá set pravidel, podle kterých se nastaví. Defaultně jsou tyto pravidla brána ze skriptu, který se nachází v instalačním adresáři programu. Skriptem `CustomRules.js` lze tyto pravidla nahradit.

Skript obsahuje mnoho funkcí pro různé sekce programu, z hlediska této práce je však důležitá pouze funkce `OnBeforeResponse()`, která je zavolána vždy, když přijde ze serverové části nějaká odpověď (tzv. response). Implementace této metody je znázorněna v snippetu 4.2.

```
static function OnBeforeResponse(oSession: Session) {

  var code = oSession.responseCode
  if (code == 200 || code == 301 || code == 302) {

      var path = "c:\\tmp\\fiddler_data\\"
      var filename_dat  = path + oSession.id + ".dat";
      var filename_meta = path + oSession.id + ".meta";

      System.IO.File.WriteAllBytes(
          filename_dat, oSession.responseBodyBytes);

      var url = oSession.url;
      var host = oSession.host);
      var referer = oSession.oRequest.headers["Referer"]);
      var response = oSession.responseCode);
      var redirect = oSession.oResponse.headers["Location"]);
      var time = System.DateTime.Now);

      WriteMeta(url, host, referer, response, redirect, time)
  }
}
```

Obrázek 4.2: Implementace inicializačního skriptu Fiddleru

Tělo funkce `OnBeforeRequest()` obsahuje jedinou podmínku založenou na kódu odpovědi (tzv. response kód) serverové části. Tento kód se nejdříve získá z relace oSession (relace, která obsahuje veškeré informace o komunikaci s danou url). Pokud je response kód 200, jedná se o tzv. `success` (úspěch), což

znamená, že server na daný dotaz prohlížeče vrací odpověď. Tento typ odpovědí je snaha zachytávat, neboť obsahují zdrojové soubory webových stránek. Pokud se však stránka na serveru nachází jinde než na prohlížečem dotazované url, může dojít k přesměrování. Response kódy pro přesměrování z pravidla spadají do intervalu $300 - 399$, avšak nejčastější případy přesměrování mají kódy 301, respektive 302. Kód 301 je rezervován pro trvalá přesměrování (tzv. moved permanently), kód 302 potom pro přesměrování dočasná (tzv. moved temporarily)[15]. Pokud tedy dojde k zachycení odpovědi, která obsahuje jeden z těchto tří kódů, podmínka je splněna.

Pokud se tak nastane, jsou vytvořeny dva soubory, které jsou pojmenovány podle id relace. Soubor s koncovkou `.dat` je určený pro záznam zachycených dat, soubor `.meta` slouží k uchování metadat o zachycené komunikaci. Následně se volá funkce `WriteAllBytes()`, která z relace oSession přijme parametr `responseBodyBytes`, který obsahuje tělo odpovědi ze serveru, tedy zachycená data, a zapíše je do datového souboru. Dále se shromáždí metadata potřebná pro filtraci souborů (Tabulka 4.1), která se zapíší do souboru `.meta`.

url	zdrojová stránka souboru
host	název serveru, na kterém je zdrojová stránka
referer	stránka, z které přišla žádost o načtení současné url
response kód	kód, který byl zaznamenán v hlavičce souboru
redirect	lokace, na kterou dojde k přesměrování
time	čas záznamu souboru

Tabulka 4.1: Metadata stažených souborů

4.2 Zpracování dat a upload do databáze

Protože byl ke stahování dat použit program Fiddler, který zaznamenává kompletní komunikaci prohlížeče s webovým serverem, je potřeba stažená data profiltrovat. K tomuto účelu byl vytvořen skript v programovacím jazyku Python. Následně je zapotřebí protříděná data nahrát do systému Scavenger, kde se nachází databáze cleansetu. Tato sekce tedy může být rozdělena na dvě části:

- Třídění dat
- Upload do databáze

4.2.1 Třídění dat

Pro potřeby třídení dat byl program Fiddler konfigurován, aby spolu se zdrojovým kódem webových stránek zaznamenával i pomocná metadata. Tato metadata jsou využívána v Python skriptu, který třídění dat implementuje.

Vstupem skriptu je setříděný list, který obsahuje stažená data, spolu s jejich metadaty. Tyto soubory jsou inkrementačně očíslovány, podle toho, v jakém pořadí byly Fiddlerem zaznamenány (implementováno v inicializačním skriptu Fiddleru 4.1.2).

Protože Fiddler zaznamenává veškerou komunikaci, je zapotřebí odlišit komunikaci vyvolanou přístupem na požadovanou webovou stránku od zbylé. První soubor, který je v sekvenčním průchodu důležitý, bude v metadatech obsahovat požadovanou webovou stránku v klíčovém slovu *url* (viz tabulka s metadaty 4.1). Hlavní smyčka třídícího skriptu je znázorněna na snippetu 4.3.

```
sorted_list = get_files_ids(directory)
stripped_url = netloc_from_url(url)
referers = [stripped_url]
valid_ids = []

for file_id in sorted_list:
    data = f"{directory}/{file_id}.dat"
    meta = get_meta_from_file(f"{directory}/{file_id}.meta")

    netloc_url = netloc_from_url(meta["url"])
    netloc_referer = netloc_from_url(meta["referer"])
    netloc_redirect = netloc_from_url(meta["redirect"])

    response = meta["response_code"]
    redirect_response = response == 301 or response == 302

    if redirect_response and meta["host"] in referers:
        if meta["redirect"] is not "":
            referers.append(netloc_redirect)
    elif os.path.getsize(data) < 9:
        continue
    elif netloc_url in referers or (
        netloc_referer is not "" and
        netloc_referer in referers
    ):
        valid_ids.append(file_id)
        referers.append(netloc_url)
```

Obrázek 4.3: Implementace filtrování stažených dat

Metoda `get_file_ids()` načítá id stažených souborů (očíslováno podle zaznamenaného pořadí) a ukládá je do listu, přes který iteruje hlavní smyčka skriptu. Metoda `netloc_from_url()` vrací netloc (tj. doménové jméno první

úrovně) zadané adresy. Každá url adresa se řídí daným formátem, zjednodušeně takto:

$$< scheme >: // < netloc > / < path > \tag{4.1}$$

Pro příklad `http://www.example.com/index` je tedy `http` schéma, `index` cesta a `www.example.com` hledaný netloc. Toho bylo použito při procházení setříděného listu očíslovaných souborů. Před iterací je ještě zapotřebí inicializovat dva listy. List `referers` bude obsahovat všechny již známé a chtěné referery (tj. netloc adresy, které si vyžádaly načtení zdrojových souborů současné url 4.1). Jak již bylo dříve zmíněno, prvním takovým refererem je původní zadaná adresa. Druhý list `valid_ids` bude uchovávat názvy souborů, které byly vyhodnoceny jako validní (tj. soubory zachycené komunikací s původní zadanou adresou).

V hlavní smyčce se nejprve načtou metadata k souboru pomocí metody `get_meta_from_file()` a uloží se do slovníku `meta`. Poté se tyto data ořízne pomocí metody `netloc_from_url()` a dále se pracuje jen s jejich netloc částí. První podmínka vyhodnocuje přesměrování. Pokud je `response_code` v metadatech souboru roven 301 nebo 302, což značí přesměrování, a pokud je zároveň `host` této adresy již v listu `referers`, nejedná se o validní soubor, však jde o soubor, který nese informace o přesměrování. Je tedy potřeba přidat netloc adresy, na kterou dojde k přesměrování (v metadatech klíčové slovo `redirect`).

Druhá podmínka je přidána pro optimalizaci. Fiddler zaznamenává velké množství souborů, které mají režijní charakter. Tyto soubory se vyznačují majou velikostí a jsou z hlediska filtrace nedůležité. Z toho důvodu se dál zpracovávají pouze soubory, které mají více než 8 bytů.

Poslední podmínka již kontroluje samotné validní soubory. Pokud je netloc adresy, nebo netloc referera adresy již v listu `referers`, jedná se o soubor zachycený při komunikaci s cílovou adresou a soubor je přidán do validního listu. Dále je netloc adresy přidán do `referers` z důvodu, kdy by tato adresa odkazovala na jinou url při další komunikaci. Po iteraci nad všemi soubory v setříděném listu je výstupní list `valid_ids` předán ke zpracování skriptu pro upload dat do databáze.

4.2.2 Upload do databáze

Pro nahrávání dat do databáze byl již dříve týmem, který spravuje systém Scavenger, vytvořen klient v podobě python knihovny. Tento klient používá HCP (Hitachi content platform) pro přesun dat mezi klientem a servery Scavengeru a představuje novější řešení oproti dříve používanému klientu Samba. Při uploadu dat do databáze lze tedy vycházet z tohoto kódu. V prvním kroku procesu uploadu do Scavengeru klient přesune požadovaný soubor do tzv. namespace (vyhrazený prostor na serverové části unikátní pro daného klienta) v HCP úložišti a připojí k němu požadovaná metadata. K tomuto namespace

je připojen proces (tzv. feeder), který periodicky odebírá přítomné soubory i s jejich metadaty a přesouvá je do systému Scavenger. Při tomto procesu dochází k přejmenování souborů hashováním sha256.

Implementace klientu je velmi jednoduchá (snippet 4.4). Z knihovny je

```
def upload_file(file, target_name, meta):
    client = Client(**config.HCP_CONFIG)
    client.upload_object(file, target_name, meta)
```

Obrázek 4.4: Implementace použití HCP klientu pro upload

potřeba provést import třídy `Client()` a její inicializaci, při které se předává konfigurace klientu. Konfigurace obsahuje jméno `namespace`, tedy cílový prostor v úložišti, ke kterému se připojit, přihlašovací jméno a heslo. Všechny tyto údaje byly vygenerovány týmem spravující HCP úložiště. Dále už stačí jen volat metodu `upload_object()` a předat jí potřebné parametry. Mezi tyto parametry paří cesta k nahrávanému souboru (`file`), název, jaký ponese soubor v úložišti (`target_name`) a požadovaná metadata, která se mají k souboru přiložit (`meta`). Zde není potřeba nahrávat všechna metadata získaná při stahování souborů. Ve snippetu 4.5 je kód, který tento výběr zajišťuje.

```
def set_meta_for_upload(meta_file, url):
    meta = {
    "source_url": meta_file["url"],
    "source_url_referer": meta_file["referer"],
    "trigger_url": url,
    }

    return meta
```

Obrázek 4.5: Implementace načtení metadat pro HCP upload

Metoda `set_meta_fot_upload()` má dva vstupní parametry. Prvním je slovník `meta_file`, který obsahuje všechna stažená metadata, druhým je `url`, což je původní dotazovaná adresa. Výstupem metody je slovník s metadaty určenými pro upload do HCP úložiště. Pro soubor uložený na cleansetu je důležitý údaj o adrese, na které se soubor vyskytoval - `source_url` a kdo na tuto adresu odkazoval - `source_url_referer`. Třetím důležitým údajem je původní volaná adresa `trigger_url`. Ta je důležitá v případě, že by se na cleansetu objevily dva totožné soubory. V takové situaci se nový soubor již nepřidá, aktualizují se však jeho metadata a v budoucnu je stále patrné, že byl tento soubor viděn při komunikaci s vícero rozdílnými adresami.

Snippet 4.6 znázorňuje hlavní smyčku skriptu pro upload souborů do HCP úložiště a následně do Scavengeru. V ní dochází k iteraci přes všechny validní

```
for counter, file in enumerate(filenames):
    file_name = f"{root}\\{directory}\\{file}"

    meta = hcp_feeder.set_meta_for_upload(
        files_filter.get_meta_from_file(f"{file_name}.meta"),
        directory
    )

    hcp_feeder.upload_file(
        f"{file_name}.dat",
        f"{directory}{counter}",
        meta
    )
```

Obrázek 4.6: Implementace hlavní smyčky pro upload

soubory (výstup z třídění dat 4.2.1). Pro každý soubor se nejdříve určí cesta souboru `file_name`. Následně se nastaví metadata pro upload zavoláním metody `set_meta_for_upload()`, které se předá cesta k souboru s metadaty a původní tázaná url (zde `directory` z důvodu, že se stažené soubory ukládají do složek pojmenovaných podle původní zadané adresy). Na závěr cyklu hlavní smyčky je volána funkce `upload_file`. Prvním parametrem je cesta k datovému souboru. Následuje cílový název souboru v HCP úložišti. Zde, aby se zajistila unikátnost souborů v úložišti, se soubory přejmenovávají podle klíče:

$$\{původní\_adresa\}\{pořadí\_zpracovaného\_souboru\} \tag{4.2}$$

Toto je možné z důvodu, že se následně soubory automaticky přejmenovávají při přenosu z HCP úložiště do Scavengeru pomocí hashovací funkce sha256. Posledním parametrem pro upload jsou metadata souboru.

4.3 Virtualizace

Z důvodu použití programu Fiddler pro stahování dat namísto emulace webového prohlížeče v Pythonu je nutné tento proces zabezpečit (výsledek analýzy v sekci 2.4). Toho bylo docíleno obalením stahovacího procesu do virtuálního prostředí díky využití programu Virtualbox. K tomu byla využita knihovna `pyvbox`[10], která implementuje Virtualbox API do prostředí Pythonu. Není potřeba virtualizovat celý proces, ale jen část, která se zabývá stahováním dat. Bylo tedy zapotřebí vytvořit skript, který dokáže nastartovat virtuální

stroj, v něm spustit aplikaci pro stahování souborů a tyto soubory přenést z virtuálního prostředí zpět do fyzického systému.

K tomuto účelu byla vytvořena třída `Vbox`, jejíž inicializace (tzv. konstruktor) je zobrazena ve snippetu 4.7. Instanci této třídy lze použít k ovládání

```
import virtualbox

class Vbox:
    def __init__(self, machine=""):

    # init virtualbox
    self.vbox = virtualbox.VirtualBox()

    # init session
    self.session = virtualbox.Session()

    # define machine and create a session
    machines = self.list_all_machines(self.vbox)
    if machine not in machines:
        if len(machines) == 0:
            print("No virtual machines to load")
            sys.exit()

        print(f"No or corrupted machine name, "
            f"loading machine {machines[0]}")
        machine = machines[0]

    self.machine = self.vbox.find_machine(machine)
```

Obrázek 4.7: Implementace inicializace třídy Vbox

virtuálního stroje. K tomu je nutné inicializovat tři objekty - `VirtualBox()`, `Session()` a `Machine()`. Instance třídy `VirtualBox()` slouží k ovládání Virtualbox manažeru (program, který řídí jednotlivé zaregistrované virtuální stroje). Obecné nastavování se tedy provádí přes tento objekt. Třída `Session()` je navázána na relaci konkrétního virtuálního stroje. Ovládání běhu stroje probíhá tedy s využitím tohoto objektu. Třída `Machine()` reprezentuje konkrétní stroj, tak jak je zobrazen v manažeru Virtualboxu. Při inicializaci této třídy je zapotřebí předat název stroje, což je zde řešeno pomocí převzaté metody `find_machine()`. Název stroje je možné předat konstruktoru pomocí parametru `machine`. Pokud není žádný konkrétní stroj vybrán, je proveden výčet všech dostupných strojů přihlášených pod Virtualbox manažerem (metoda `list_all_machines`) a načten první dostupný stroj.

Pro ovládání spuštění virtuálního stroje s časováním byla vytvořena metoda `boot_with_timeout()` (snippet 4.8). Tato metoda přijímá tři parametry.

```
def boot_with_timeout(self, timeout, login, password):

    self.launch_machine()
    self.wait_for_session_locked()

    t_boot = 1

    while t_boot < timeout:

        print(f"Booting... ({t_boot}/{timeout})")
        try:
            gs = self.login(login, password)
            gs.directory_exists("c:/")
            print("Machine has started.")
            return gs

        except virtualbox.library.VBoxError as e:
            print(f"Machine not started yet: {e}")

        time.sleep(1)
        t_boot += 1

    raise DidntBootInTimeException("Timeout reached. "
        "Machine didn't boot in time.")
```

Obrázek 4.8: Implementace bootovacího algoritmu pro virtualbox

Prvním parametrem je `timeout`, neboli čas, po kterém se metoda přeruší a zabrání se tak případnému nekonečného cyklu. Další dva parametry jsou přihlašovací jméno a heslo do uživatelského účtu ve virtuálním systému. Tyto údaje jsou potřeba pro inicializaci uživatele, ke které v průběhu spouštění systému (tzv. bootování) dochází. Na začátku je zavolána metoda `launch_machine()`, která zahájí proces bootování. Aby nedocházelo k nepředvídatelnému průběhu, volá se zde další metoda `wait_for_session_locked()`, která zajišťuje, že bootovací proces nebude pokračovat, dokud se neuzamkne virtuální stroj (parametr `machine` třídy `Vbox`) pro současnou relaci (parametr `session`). Při ovládání stroje skrze neuzamčenou relaci může nastat chyba a následné ukončení skriptu.

Poněvadž Virtualbox API nemá implementovanou kontrolu průběhu bootování, nelze nijak zkontrolovat, zda je už virtuální stroj načtený a připravený

k použití. Z tohoto důvodu je vytvořen `while` cyklus, kde k této kontrole periodicky dochází. V každém průběhu cyklu dochází k inicializaci uživatelského účtu a následnému testu existence kořenové složky $c : /$ metodou `directory_exists()`. Předpokládá se zde, že dojde k chybě, protože systém ještě není připravený takovéto příkazy zpracovávat. V takové situaci je výjimka odchycena a přechází se na další průběh cyklu. Pokud by již metoda `directory_exists()` proběhla bez chybové hlášky, dá se očekávat, že je systém již připravený k používání. V tomto případě dojde k ukončení metody. Výstupem je objekt s inicializovaným uživatelským účtem v případě, že bootování proběhlo bez problémů, či chybová hláška `DidntBootInTimeException` v situaci, kdy vypršel časový limit dříve, než mohl systém naběhnout.

```
commands = (
    f"cd c:/tmp & "
    f"python {config.VBOX_SCRIPT_PATH} {url}"
)
gs.execute(
    "C:\\Windows\\System32\\cmd.exe",
    ["/C", commands]
)
```

Obrázek 4.9: Spouštění příkazů ve virtuálním systému

S již vytvořeným objektem uživatelského účtu lze následně spouštět příkazy přes příkazový řádek (tzv. command line) virtuálního systému (snippet 4.9), kde parametr `commands` obsahuje jednotlivé příkazy, tedy přesun do pracovní složky a spuštění skriptu pro stahování webového obsahu (viz sekce 4.1) s parametrem požadované adresy url.

Pro přesun stažených souborů z virtuálního prostředí do reálného systému byla převzata metoda `directory_copy_from_guest()` (snippet 4.10), která je

```
gs.directory_copy_from_guest(
    dir_from_path,
    dir_to_path,
    [virtualbox.library.DirectoryCopyFlag(1)]
)
```

Obrázek 4.10: Spouštění příkazů ve virtuálním systému

implementována ve třídě pro uživatelský účet virtuálního stroje. Tato metoda kopíruje celou složku, která je předána v parametru `dir_from_path` do cílové

lokace specifikované paramterem `dir_to_path`. Třetím parametrem metody je konfigurační hodnota (tzv. flag), která určuje typ přenosu.

4.4 Implementace frontového systému Kafka

4.4.1 Kafka producer

```
class KafkaProducer(Producer):
    def __init__(self):
    super().__init__(**config.KAFKA_PRODUCER)
```

Obrázek 4.11: Implementace třídy Kafka Producer

```
def send_message_to_kafka(self, message, message_cnt):
    self.produce(config.KAFKA_TOPIC, message.encode("utf-8"))
    self.flush()
    print(f"Message sent")
```

Obrázek 4.12: Posílání zpráv do frontového systému Kafka

```
for url_count, url in enumerate(urls):
    if not tools.is_url(url):
        continue

    if not url.startswith("http"):
        url = "http://" + url

    kafka_producer.send_message_to_kafka(url, url_count)
```

Obrázek 4.13: Implementace hlavní smyčky posílání zpráv do Kafky

4.4.2 Kafka consumer

```
class KafkaConsumer(confluent_kafka.Consumer):
    def __init__(self):
        super().__init__(**config.KAFKA_CONSUMER)
        self.subscribe([config.KAFKA_TOPIC])
```

Obrázek 4.14: Implementace třídy Kafka Consumer

```
def consume_url(self):

    print("Consuming message...")
    msg = self.poll(config.KAFKA_CONSUMER_POLL_TIMEOUT)

    if msg is None:
        return None
    elif msg.error():
        print(f"Error: {msg.error().name()}")
        raise confluent_kafka.KafkaException(msg.error())
    else:
        print("Message consumed")
        return msg.value().decode('utf-8')
```

Obrázek 4.15: Přijímání zpráv z frontového systému Kafka

4.5 Integrace do stávající infrastruktury

4.5.1 Systém Jenkins

4.5.1.1 Job pro nahrávání url do Kafky

4.5.1.2 Job pro spoustění skriptu pro stahování dat

4.5.1.3 Job pro upload dat do databáze cleansetu

4.5.2 Systém Luft

4.5.2.1 Dockerizace skriptu Kafka consumer

Kapitola 5

Otestování a zhodnocení přínosu

Závěr

Literatura

[1] *Avast corporate factsheet*, Dostupné z: https://cdn2.hubspot.net/hubfs/2706737/media-materials/corporate-factsheet/Avast_corporate_factsheet_A4_en.pdf

[2] Moravec Jan. *Distribuované řízení kolon vozidel na autodráze*. ©2014, České vysoké učení technické v Praze, vedoucí práce Ing. Ivo Herman, Dostupné z: https://dspace.cvut.cz/bitstream/handle/10467/24299/F3-BP-2014-Moravec-Jan-prace.pdf

[3] https://www.telerik.com/fiddler

[4] https://www.crummy.com/software/BeautifulSoup/

[5] https://www.imperva.com/learn/application-security/man-in-the-middle-attack-mitm/

[6] https://en.wikipedia.org/wiki/Apache_Kafka#/media/File:Overview_of_Apache_Kafka.svg

[7] https://www.virtualbox.org/manual/ch01.html

[8] https://www.virtualnipc.cz/wp-content/gallery/140701-1-uvod-do-virtualizace-na-desktopu/cache/140701-uvod-do-virtualizace-na-desktopu-img-1.png-nggid013-ngg0dyn-640x480x100-00f0w010c010r110f110r010t010.png

[9] [https://code.vmware.com/web/sdk/6.7/vsphere-automation-python]

[10] Dorman Michael. *pyvbox Documentation*. ©2017 Dostupné z: https://buildmedia.readthedocs.org/media/pdf/pyvbox/latest/pyvbox.pdf

[11] https://github.com/mozilla/geckodriver/releases

[12] https://www.mozilla.org/cs/firefox/new/

[13] https://www.google.com/intl/cs_CZ/chrome/

[14] https://chromedriver.chromium.org/

[15] https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

Příloha **A**

Obsah přiloženého CD

- `slotcar-sw` ..... adresář s Java projektem
- `SimulinkControllers`
  - `SISO` ..... jednovstupový regulátor
  - `TwoInputSingleOutput` ..... dvouvstupový regulátor
  - `MISO` ..... vícevstupový regulátor
  - `transferscript.bat` ..... skript pro přenos souborů
- `text`
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `thesis.tex` ..... text práce ve formátu LaTeX
  - `pictures` ..... zdrojové obrázky pro formát LaTeX
- `video`
  - `tutorial.mp4` ..... instruktážní video