

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.

3. dubna 2020

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Staněk** Jméno: **Michal** Osobní číslo: **420236**
Fakulta/ústav: **Fakulta elektrotechnická**
Zadávací katedra/ústav: **Katedra řídicí techniky**
Studijní program: **Kybernetika a robotika**
Studijní obor: **Kybernetika a robotika**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Automatizovaný systém stahování webového obsahu potřebného k doplňování cleansetu

Název diplomové práce anglicky:

Automatic system for updating cleanset by downloading web content

Pokyny pro vypracování:

Práce se zaměřuje na vylepšování vlastností množiny neškodných souborů (cleansetu) využívané při detekci škodlivých webových stránek.

1. Analyzujte možné způsoby obohacování cleansetu a vhodný způsob implementujte.
2. Navrhněte a implementujte automatizovaný systém stahování dat pro obohacování cleansetu. Systém začleňte do stávající infrastruktury.
3. Proveďte zobrazení vzniklého systému pro využití i v jiných projektech.
4. Výsledné řešení otestujte a zhodnoťte jeho přínos.

Seznam doporučené literatury:

- [1] Eric Lawrence, CreateSpace Independent Publishing Platform (2015). Debugging with Fiddler: The complete reference from the creator of the Fiddler Web Debugger, 2nd Edition. ISBN: 9781511572903.
- [2] Ryan Mitchell, O'Reilly Media, Inc. (2018). Web Scraping with Python, 2nd Edition. ISBN: 9781491985564.
- [3] Harry Colvin, CreateSpace Independent Publishing Platform (2015). VirtualBox: An Ultimate Guide Book on Virtualization with VirtualBox. ISBN: 1522769889.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Jan Kubr, Ph.D., katedra počítačové grafiky a interakce FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **06.01.2020**

Termín odevzdání diplomové práce: _____

Platnost zadání diplomové práce:

do konce letního semestru 2020/2021

Ing. Jan Kubr, Ph.D.
podpis vedoucí(ho) práce

prof. Ing. Michael Šebek, DrSc.
podpis vedoucí(ho) ústavu/katedry

prof. Mgr. Petr Páta, Ph.D.
podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

Datum převzetí zadání

Podpis studenta

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 3. dubna 2020

.....

České vysoké učení technické v Praze

Fakulta elektrotechnická

© 2020 Michal Staněk. Všechna práva vyhrazena.

Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě elektrotechnické. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

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

1	Úvod	1
1.1	Motivace	1
1.2	Cíle práce	2
2	Analýza a řešení problému	3
2.1	Výběr programovacího jazyka	3
2.2	Předzpracování zadaných adres	3
2.3	Získání čistých souborů z webových stránek	4
2.3.1	Emulace webového prohlížeče	5
2.3.2	Zachytávání komunikace Fiddlerem	6
2.4	Zabezpečení stahovacího procesu	8
2.5	Nahrání získaného obsahu do databáze cleansetu	9
2.6	Začlenění do stávající infrastruktury	10
3	Použité technologie	11
3.1	Python 3.7	11
3.2	Fiddler	11
3.3	Selenium	12
3.4	Jenkins	12
3.5	Kafka	12
3.6	Virtualbox	13
3.7	Docker	14
4	Implementace	17
4.1	Klient pro stahování webového obsahu	17

4.1.1	Skript pro ovládání pomocných programů	18
4.1.2	Skript pro nastavení Fiddleru	19
4.2	Zpracování dat a upload do databáze	21
4.2.1	Třídění dat	21
4.2.2	Upload do databáze	23
4.3	Virtualizace	25
4.4	Implementace frontového systému Kafka	29
4.4.1	Kafka producer	29
4.4.2	Kafka consumer	30
4.5	Integrace do stávající infrastruktury	32
4.5.1	Vytvoření a nastavení virtuálního stroje	33
4.5.2	Systém Jenkins	34
4.5.3	Systém Luft	37
5	Otestování a zhodnocení přínosu	39
5.1	Testovací proces	39
5.1.1	Komponenty využívající frontový systém Kafka	39
5.1.2	Zabezpečené stahování a filtrování dat	40
5.1.3	Přenos souborů do HCP úložiště	42
5.2	Zobecnění projektu a budoucí využití	43
5.3	Zhodnocení přínosu	44
	Závěr	45
	Literatura	47
	A Obsah příloženého CD	49

Seznam tabulek

4.1	Metadata stažených souborů	20
-----	--------------------------------------	----

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 zabránilo 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

1. ÚVOD

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 funkčnosti 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.

Analýza a řešení problému

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

- Výběr programovacího 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 souladu s firemní politikou.

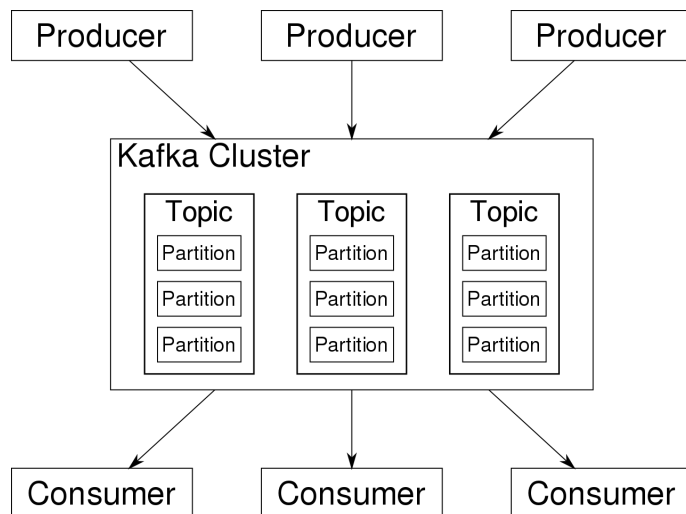
2.2 Předzpracování zadaných adres

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 *Kafka* má výhodu v již naimplementovaném řešení fronty. Jednou z funkcionalit *Kafka*, 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

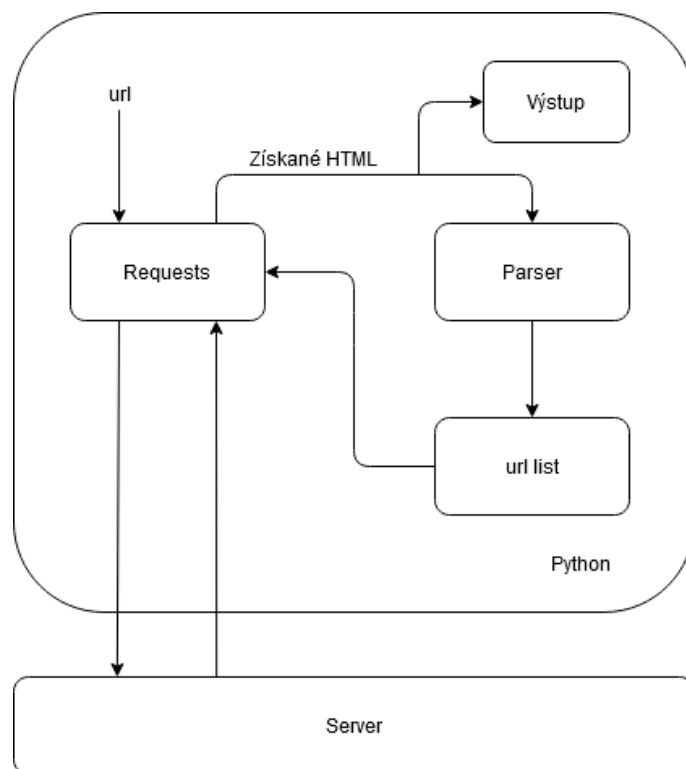
2.3.1 Emulace webového prohlížeče

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 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 provést 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 nutnosti

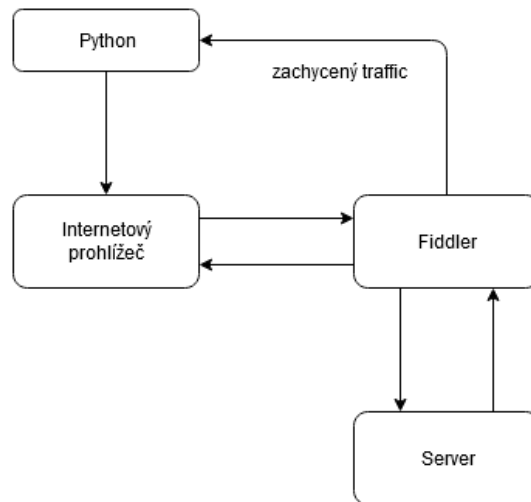


Obrázek 2.2: Diagram zachytávání komunikace pomocí web scrappingu

stažené soubory spouštět, čímž by nebezpečí infekce 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 sledování internetové komunikace (tzv. traffic), zachytává veškerý traffic, který skrze nástroj proudí (Obr.2.3). Bylo by tedy potřeba implementovat logiku pro



Obrázek 2.3: Diagram zachytávání komunikace Fiddlerem

třídění a následnou selekci HTML a .js souborů.

2.3.2.1 Selektce 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 vykreslení webové stránky, ale i režijní komunikaci mezi prohlížečem a serverem. Tento přístup je možné změnit pomocí již zmíněného inicializačního scriptu.

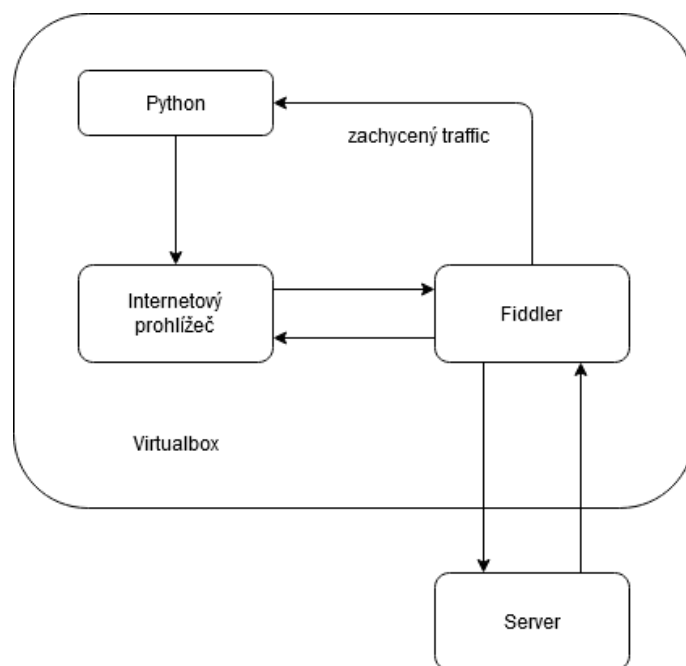
Avšak problémy této metody přináší i webový prohlížeč, kterého se zde využívá. Téměř vždy při spuštění prohlížeče probíhá nevyžádaná komunikace prohlížeče se serverem, která je potřeba vytřídit. I zde je možné využít inicializačního scriptu *Fiddleru*, implementovat třídění již při zachytávání komunikace 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 chtě-

nými 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



Obrázek 2.4: Zaobalení stahovacího procesu virtuálním strojem

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 popsány v subsekcí 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 nazývanou *vSphere*. Tato infrastruktura obsahuje vlastní SDK nástroje pro implementaci do programovacího jazyka *Python*[10], 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*[11]. Tato knihovna 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 (stejně soubory mohou mít rozdílné názvy, ale hash souboru je pro identické soubory stejná). K hashi souboru se přikládají metadata mimo jiné s informací o původu souboru, času výskytu a prevalenci.

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 používán 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 spuštěn manuálně.

Jiným přístupem je využít firemní mutaci nástroje Kubernetes interně nazývanou Luft3.7, která by umožňovala mít systém spuštěný bez přestávky. Krom neustálého spuštění skriptu je velkou výhodou Kubernetes tzv. škálování, kdy lze při velkém vytížení jednoduše navýšit výpočetní prostředky danému procesu a tím urychlit jeho běh. Pro spuštění skriptu v Luftu je nejprve potřeba tento skript dockerizovat. Dockerizace je proces, při němž je vytvořen tzv. docker image, neboli balíček, který je plně spustitelný ve virtualizovaném prostředí, například v Luftu. Při dockerizaci je nutné vytvořit **Makefile**, který obsahuje informace o vzniku balíčku.

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

Použité technologie

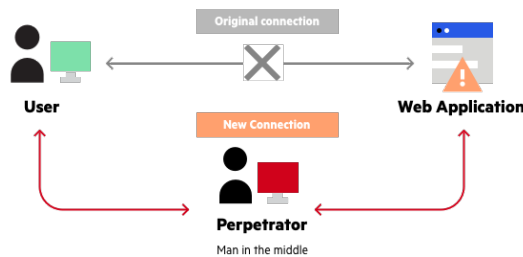
V této kapitole jsou stručně popsány všechny technologie využité při zpracování této práce.

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 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.



Obrázek 3.1: MitM útok [5]

3.3 Selenium

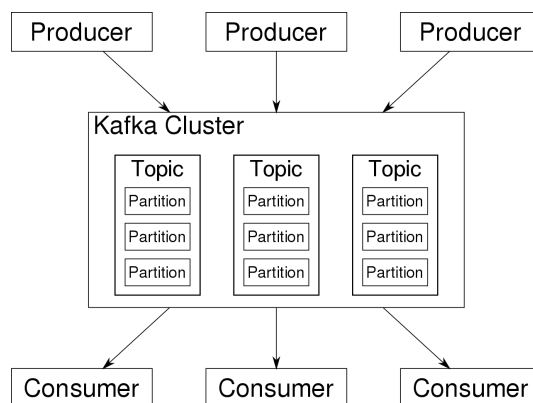
Selenium je opensource nástroj používaný k automatizovanému přístupu k webovým aplikacím. Těchto vlastností se často využívá při testování, avšak v této práci je použit pouze k obsluze webového prohlížeče. Selenium obsahuje vlastní vývojové prostředí, které lze využít bez velké znalosti programování, 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 spouštět 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, 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



Obrázek 3.2: Znázornění Kafka systému [6]

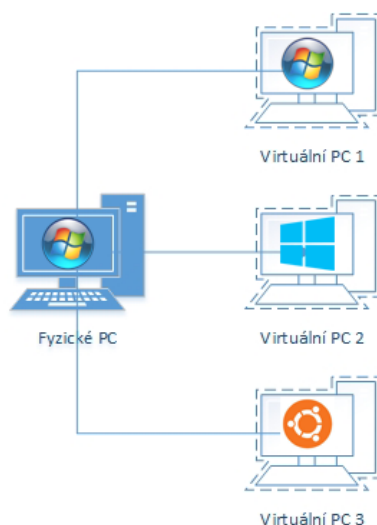
rozdělení do rozdílných skupin a každá skupina dostává identické zprávy, čím se zabrání vzájemnému čtení stejných zpráv.

Zprávy zůstávají v topicu po určitou dobu, což určuje hodnota retence. Po tuto dobu jsou zprávy přístupny pro každou skupinu konzumerů. Síla Kafka je v politice přečtených zpráv. Na rozdíl od zmiňovaného RabbitMQ, který přečtením zprávy zprávu odstraní ze své fronty, funguje v Kafka 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]

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 vrátit provedené změny zpět do bodu vytvoření snapshotu. Toho je možné využít při testování aplikací, využití je avšak možné i v oblasti bezpečnosti. Pokud je vytvořen snapshot čistého systému,



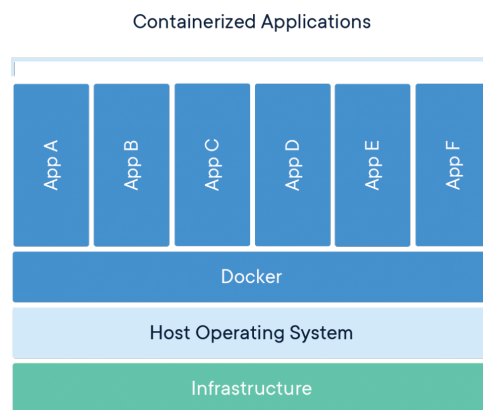
Obrázek 3.3: PC s virtuálními stroji [9]

lze se do něj vraátit v případě potenciálního nakažení virtuálního stroje. Těto funkcionality se využívá i v této práci.

3.7 Docker

Docker je souhrn produktů typu PaaS (platform as a service) vyvíjen firmou Docker, Inc. [8]. Služba využívá virtualizaci operačních systémů či jiných platforem k vytvoření tzv. kontejnerů. Tyto kontejnery obsahují vlastní knihovny a konfigurace, ke spuštění však potřebují centrální systém, nazývaný Docker Engine. V tomto je tato služba oproti virtuálním strojům (např. Virtualbox) více odlehčená, nevýhodou je však nutnost centrálního operačního systému.

K dockerizaci aplikace je potřeba vytvořit tzv. Dockerfile, který obsahuje sekvenci příkazů k sestavení kontejneru. Ten je poté možné spustit ve službě Kubernetes dostupné firmou Docker, Inc. Tato služba je tzv. Task Manager, neboli systém, v kterém je možné spravovat jednotlivé procesy, zde spuštěné instance kontejnerů.



Obrázek 3.4: Systém s dockerizovanými aplikacemi [8]

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 podsekci:

- 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[12], je určený pro komunikaci s internetovým prohlížečem Mozilla Firefox[13], naproti tomu webdriver ChromeDriver[15] je vyvinut pro komunikaci s internetovým prohlížečem Google Chrome[14]. 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 umožňuje nastavovat různé parametry. V tomto případě je zvolen přepínač `headless`, který určuje spouštění webového prohlížeče bez grafického uživatelského prostředí (tzv. 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.

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 odpovědi 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 dříve sloužil pro přesměrování dočasná (tzv. *moved temporarily*), od standardu HTTP/1.1 je však více využíván pro všeobecná přesměrování[16]. 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.

```
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

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 zapíše 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 třídění 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 metadatach obsahovat požadovanou webovou stránku v klíčovém slovu *url* (viz tabulka s metadaty 4.1). Hlavní smyčka třídícího skriptu je znázorněna na snippetu 4.3.

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:

$$< scheme > : // < netloc > / < path > \quad (4.1)$$

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 dvě datové struktury. Set `referers` bude obsahovat všechny již známé a

```
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.add(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.add(netloc_url)
```

Obrázek 4.3: Implementace filtrování stažených dat

chtěné referery (tj. netloc adresy, které si vyžádaly načtení zdrojových souborů současně url 4.1). Jak již bylo dříve zmíněno, prvním takovým refererem je původní zadaná adresa. Druhá struktura, 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 oříznou pomocí metody `netloc_from_url()` a dále se pracuje jen s jejich netloc částí. První podmínka vyhodnocuje přesměrování. Pokud je `response_code` v metadatach souboru roven 301 nebo 302, což značí přesměrování, a pokud je

zároveň **host** této adresy již v setu **referers**, nejedná se o validní soubor, jde však 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 setu **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 patří 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_for_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 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

```

for counter, file in enumerate(filenamees):
    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

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:

$$\{\text{původní_adresa}\}\{\text{pořadí_zpracovaného_souboru}\} \quad (4.2)$$

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`[11], 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 me-

toda `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 nekonečnému 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 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 parametrem `dir_to_path`. Třetím parametrem metody je konfigurační hodnota (tzv. flag), která určuje typ přenosu.

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

Obrázek 4.10: Spouštění příkazů ve virtuálním systému

4.4 Implementace frontového systému Kafka

Dalším krokem je implementace frontového systému Kafka, jehož serverová část je součástí firemní infrastruktury. K těmto účelům byly vygenerovány týmem spravující interní mutaci Kafka přihlašovací údaje pro systém Magpie (přihlašovací jméno, heslo a SSL certifikát). Pro implementaci samotného klientu byla použita Python knihovna `confluent-kafka`[17], jejíž výhodou je možnost zabezpečené komunikace klientu se serverem. Byly vytvořeny dva skripty obsahující rozdílné klienty:

- Kafka producer
- Kafka consumer

4.4.1 Kafka producer

Úkolem skriptu implementujícího klienta Kafka producer je posílání předzpracovaných url do frontového systému. Za tímto účelem byla vytvořena třída (tzv. class) `KafkaProducer` využívající import třídy `Producer` z knihovny `confluent-kafka` (snippet 4.11). Tímto způsobem lze přetížít konstruktor

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

Obrázek 4.11: Implementace třídy Kafka Producer

nadřazené třídy a obohatit ho o konfigurační informace `KAFKA_PRODUCER`, tedy o přihlašovací jméno, heslo a cestu k SSL certifikátu.

Při generování těchto informací byl vytvořen i tzv. **TOPIC**, podle kterého se na serverové části ukládají zprávy. Producer klienti produkují zprávy do určitých topiců, ke kterým jsou přihlášení konzumeři, kteří odtud zprávy stahují. Topic je tedy jakási nástěnka na serverové části Kafky. Metoda pro posílání zpráv do topicu `send_message_to_kafka()` je znázorněna na snippetu 4.12. Metoda přijímá v argumentu tělo zprávy `message` a následně produkuje

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

Obrázek 4.12: Posílání zpráv do frontového systému Kafka

zprávu voláním metody `produce()` zděděné z nadřazené třídy. Při produkování zprávy je v parametru předán název topicu, do kterého se má zpráva poslat. Zároveň je volána metoda `flush()`, která zajistí vyprázdnění lokální fronty a přenesení všech zatím nedešlaných zpráv na server.

Implementace hlavní smyčky posílání zpráv je k vidění na snippetu 4.13. Pro každou url v listu načtených adres se nejdříve kontroluje, zda se jedná o url adresu. Toho je docíleno pomocí metody `is_url()`, která využívá knihovnu `URLParser` vyvíjenou pro interní účely firmy. Pokud je vyhodnoceno, že vstupní string `url` není url adresa, je tento vstup přeskočen. Dále probíhá kontrola, zda adresa obsahuje informaci o protokolu. Tato kontrola probíhá s využitím nativní metody string objektu `startswith()`. Pokud je zjištěno, že vstupní string protokol neobsahuje (nezačíná znaky `http`), je k adrese přidán nezabezpečený protokol. Takto ošetřená adresa je následně předána metodě pro posílání zpráv `send_mesasge_to_kafka()`.

4.4.2 Kafka consumer

Implementace skriptu pro přijímání zpráv ze serverové části byla opět realizována s využitím knihovny `confuent-kafka`. Byla vytvořena třída, která dědí z importované třídy `Consumer` (snippet 4.14). I zde je konstruktor přetížen, a to z důvodu doplnění inicializace klientu o konfigurační data `KAFKA_CONSUMER`. Jedná se o přihlašovací jméno, heslo, SSL certifikát a oproti klientu producer navíc i parametr `group id`. Tento parametr určuje, do jaké skupiny konzumerů

```
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)
```

Obrázek 4.13: Implementace hlavní smyčky posílání zpráv do Kafka

```
from confluent_kafka import Consumer

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

Obrázek 4.14: Implementace třídy Kafka Consumer

tento klient spadá. Každý topic totiž může mít přihlášeno vícero skupin konzumerů. V takové situaci jsou zprávy odebírány vždy v rámci skupiny (přestože již skupina A odebrala z topicu zprávu, tato zpráva je stále viditelná pro skupinu B, dokud ji neodebere člen této skupiny). Dále při inicializaci probíhá přihlášení k odběru stejného topicu `KAFKA_TOPIC`, který je využíván klientem pro posílání. Toho je docíleno metodou `subscribe()`. Jeden konzumer může být přihlášený k odběru více než jednoho topiců. Z tohoto důvodu je jejím vstupem pole, které může obsahovat více elementů. Pro účely této práce postačuje však pouze jeden topic.

Tato třída obsahuje metodu `consume_url()`, která implementuje přijímání zpráv ze serveru (snippet 4.15). Toho je docíleno převzatou metodou `poll()`, která je volána s parametrem `timeout`. Tato metoda se připojí k serverové části a po dobu zadaného limitu (tzv. `timeout`), je připravena přijímat zprávy. Další postup následuje podle typu zprávy, která byla přijata. V první podmínce probíhá kontrola, zda se vůbec k nějaké komunikaci došlo. Pokud byla fronta prázdná po celou dobu čekání, metoda `poll()` nevrací žádnou hodnotu a `msg` tak není deklarována. V tomto případě je z metody `consume_url()`

```
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

navracen `None` (prázdná hodnota). V případě, kdy ke komunikaci došlo, avšak v průběhu nastala chyba, je tato chyba zachycena v objektu příchozí zprávy `msg.error()`. Druhá podmínka tedy kontroluje, zda byl průběh komunikace bezchybný. Pokud ne, je výstupem metody objekt s chybovou hláškou. Pokud nenastal žádný z předešlých dvou případů lze očekávat, že `msg` obsahuje objekt přijaté zprávy. V tomto případě je tedy výstupem metody dekodovaná hodnota zprávy `msg.value()`.

V hlavním programu pro příjem zpráv z Kafky dochází nejdříve k inicializaci Kafka klienta typu konzumer (snippet 4.16). Následuje nekonečná smyčka, v které se volá již popsaná metoda `consume_url()`. Pokud tato metoda vrátí `None`, tedy situace, kdy je fronta prázdná, pokračuje se v běhu cyklu. V opačné situaci je volána metoda `process_url()`, které je zpráva předána. Tato metoda je provázána se spouštěním virtuálního prostředí a následném běhu klientu pro stahování webového obsahu (o propojení více v sekci 4.5.2). Skript pro hlavní smyčku příjmu zpráv z Kafky je zdockerizován a spuštěn v systému Luft, tedy firemní mutaci systému Kubernetes (více v sekci 4.5.3).

4.5 Integrace do stávající infrastruktury

Při vývoji bylo rozhodnuto využít oba v analýze rozebírané systémy pro interní správu skriptů. Projekt je tedy rozložen, část je spouštěna v Jenkinsu a část ve firemní mutaci Kubernetes nazývané Luft. Dále bylo zapotřebí vytvořit

```
from lib.kafka_consumer import KafkaConsumer

if __name__ == "__main__":
    kafka_consumer = KafkaConsumer()

    while True:
        url_to_process = kafka_consumer.consume_url()

        if url_to_process is None:
            continue

        process_url(url_to_process)
```

Obrázek 4.16: Hlavní smyčka přijímání zpráv z Kafka

a nastavit virtuální stroj, který se bude spouštět pro stahování souborů ze zadaných url.

4.5.1 Vytvoření a nastavení virtuálního stroje

K vytvoření virtuálního stroje byl použit grafický manažer Virtualboxu. Fíremní infrastruktura obsahuje úložiště s již předvytvořené šablony operačních systémů ve formátu .ovf (open virtualization format), které se dají do Virtualboxu nahrát (tzv. import appliance). Pro potřeby projektu byla využita šablona s operačním systémem Windows 7 v 32 bitové verzi. Takto vytvořený virtuální systém je již zcela spustitelný, je potřeba jej však ještě nastavit pro potřeby běhu systému Magpie. Je potřeba nainstalovat program Fiddler, potřebné knihovny pro Python (základní verze Pythonu je již obsažena v použité šabloně operačního systému) a nástroj geckodriver, který slouží ke komunikaci python knihovny Selenium s webovým prohlížečem. Prerekvizity virtuálního prostředí jsou tedy následující:

- Fiddler
- Python knihovna selenium
- Geckodriver

Takto nastavený virtuální stroj je již připravený k využití. Pro systém byl následně vytvořen tzv. snapshot (uložení stavu virtuálního stroje). Při provozu

bude systém vždy spouštěn z daného snapshotu a po provedení akcí nad zadanou url do daného snapshotu opět uveden. Tím se vynuluje riziko spojené s nakažením systému škodlivým obsahem.

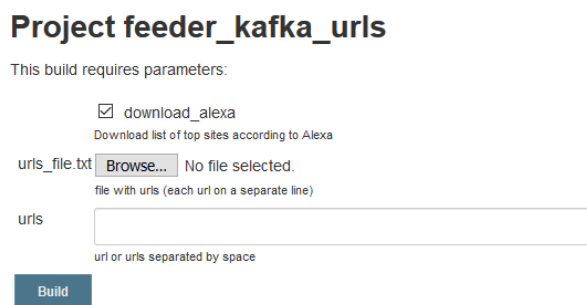
4.5.2 Systém Jenkins

Do systému Jenkins byl přesunut skript pro upload url do Kafka (Kafka producer), skript pro zabezpečené stahování dat a skript pro upload dat do databáze cleansetu. Pro každý skript byl vytvořen samostatný projekt, neboli job, který jde parametrizovat a periodicky spouštět (job je již spuštěná instance jenkins projektu, často se ale toto označení používá obecněji pro celý projekt). Jenkins joby je možné napojit přímo na git repozitář systému Magpie, takže se při jejich startu vždy stáhne nejaktuálnější verze kódu, který má být spuštěn. To jde nastavit v konfiguraci jobu, v sekci Source Code Management.

Dále bylo potřeba v prostředí každého jobu nastavit údaje, které se podle firemních nařízení z bezpečnostních důvodů nesmějí uvádět v kódu git repozitáře. Jedná se o přístupové údaje k aplikacím, tedy hesla, certifikáty, či tokeny.

4.5.2.1 Job pro nahrávání url do Kafka

Pro skript, který zajišťuje předzpracovávání zadaných adres a následný upload do Kafka byl vytvořen job **feeder_kafka_urls**. Tento job obsahuje proces rozebíraný v sekci 4.4.1. Job je nastaven tak, aby přijímal tři typy argumentů (tzv. parametrizace jobu, neboli spouštění s parametry) (viz Obr.4.17).



Obrázek 4.17: Parametry pro job **feeder_kafka_urls**

Prvním je parametr s názvem **download_alexa**, který zajišťuje, že se před nahráním url do Kafka spustí kód pro stáhnutí top url stránek z listu na adrese

<https://www.alexa.com/topsites>. Tato možnost je zaškrtnuta v základním nastavení (tzv. defaultně) a to z důvodu periodického spouštění tohoto jobu. Zbylé dva parametry jsou zde pro manuální spouštění. Prvním je možnost textového souboru `urls_file.txt`, kdy uživatel může nahrát soubor s požadovaným listem webových adres, které chce stáhnout. Zbývá textový parametr `urls`, kde se mohou adresy zadávat přímo do textového pole. Toho lze využít při potřebě stáhnout jen jednu požadovanou adresu.

Skript využívá přístup do frontového systému Kafka. Bylo tedy zapotřebí v konfiguraci jobu nastavit požadované přístupové údaje. To je možné v konfigurační sekci Bindings, kde lze vytvořit tzv. enviromentální proměnné (proměnná, která je přístupná odkudkoliv v celém prostředí) pro daný job. Takto byl nastavený SSL certifikát pro klienty Kafky a přístupové heslo pro autorizaci posílaných zpráv.

Konfigurace dále obsahuje sekci Build, v které je potřeba nastavit příkaz (tzv. command), který se má při spuštění provést. Jedná se o command příkazové řádky operačního systému Windows, který spouští skript pro upload souborů do frontového systému Kafka v prostředí pythonu se zadanými parametry.

4.5.2.2 Job pro stahování a filtraci dat

Druhým vytvořeným projektem v systému Jenkins je job pro spouštění zabezpečeného stahování a následné filtrování dat `url_processing`. K tomu je zapotřebí virtuální stroj, který byl vytvořen v sekci 4.5.1. Tento job je spuštěn samostatně pro každou url, kterou dostává parametrem při spuštění. Tento parametr je předáván ze skriptu, který dostává zprávy z frontového systému Kafka. Ten je však spuštěn v systému Luft. Proto byl v konfigurační sekci jobu vytvořen tzv. trigger token, tedy autorizační řetězec znaků (tzv. string). Pomocí tohoto autorizačního stringu je možné tento job spouštět, a to pomocí dotazu na specifickou url adresu (viz 4.3).

$$JENKINS_URL/job/url_processing/build?token = TOKEN \quad (4.3)$$

`JENKINS_URL` značí webovou adresu serveru, na kterém běží firemní verze tohoto systému a `TOKEN` je vygenerovaný autorizační řetězec znaků. Takto zadaný dotaz spustí job bez parametru. Pro předání parametru je nutné

tento dotaz upravit,

.../url_processing/BuildWithParameters?token = TOKEN&url = URL
(4.4)

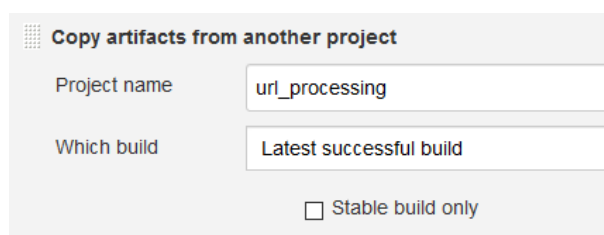
kde *URL* značí předávaný parametr *url*.

Je také nutné nastavit enviromentální proměnné pro přihlašovací údaje do virtuálního systému. Toho je opět docíleno pomocí konfigurační sekce Bindings v nastavení jenkins projektu.

Při startu jobu se spustí skript, který ovládá virtuální stroj. Toto je nastaveno v konfigurační sekci Build. Po úspěšném startu virtuálního stroje je ve virtuálním prostředí spuštěn klient pro stahování webového obsahu a následně stažené soubory přeneseny zpět do prostředí jobu v Jenkinsu. Zde dochází k filtraci dat a soubory, které jsou vyhodnocené jako relevantní pro zadanou *url* adresu jsou po doběhnutí jobu exportovány v podobě tzv. job artifactů (soubory, které jsou i po ukončení jobu dostupné k manipulaci). Tento export je proveden v konfigurační sekci Post-build Actions. Zde je také nastaveno, aby se po doběhnutí jobu spustil proces pro upload dat do databáze cleansetu (tzv. job trigger). Poněvadž jsou oba procesy v prostředí Jenkinsu, není potřeba generovat trigger token.

4.5.2.3 Job pro upload dat do databáze cleansetu

Třetím projektem vytvořeným v systému Jenkins je job pro upload do databáze cleansetu nazvaný **scav_hcp_feeder**. Tento job nemá žádné vstupní parametry, využívá však exportované artifacty z jobu pro stahování a filtraci dat. Toto je nastaveno v konfigurační sekci Build (viz Obr.4.18).



Copy artifacts from another project

Project name:

Which build:

☐ Stable build only

Obrázek 4.18: Přenos artifactů mezi joby

Parametr Project name určuje, z jakého projektu se mají exportované artifacty kopírovat, zde projekt **url_processing**. Parametrem Which build lze přímo určit specifické číslo buildu projektu, z kterého se mají exportované

artifactory získat. Lze také nastavit, aby se vždy braly z posledního úspěšného buildu, což je znázorněno na obrázku. Konfigurační sekce Build dále obsahuje příkaz pro spuštění samotného skriptu pro přenos těchto exportovaných souborů.

Poněvadž se v jobu přistupuje do HCP úložiště, je potřeba v sekci Bindings nastavit enviromentální proměnné obsahující přihlašovací údaje.

4.5.3 Systém Luft

V Luftu je spuštěn skript odebírající zprávy z frontového systému Kafka, který byl dockerizován. Dockerizace je proces vytvoření tzv. docker image (samostatně spustitelný balík v prostředí podporující funkcionalitu dockeru, např. Luft). Tento balíček byl následně nahrán (tzv. deploy) do systému Luft. Tím je docíleno neustálého běhu skriptu, což je pro tento typ programu praktické.

4.5.3.1 Dockerizace skriptu Kafka consumer

Pro vytvoření docker image je nutné nejprve vytvořit tzv. Dockerfile (obdoba Makefile pro kompilaci jakýchkoli c). Jedná se o soubor, který obsahuje sekvenci příkazů, které se mají při kompilaci docker balíčku provést. Dockerfile pro balíček obsahující skript Kafka consumer je znázorněn na snippetu 4.19.

```
FROM AVAST_DOCKER/python:3.7

COPY requirements.txt .

RUN pip install --no-cache-dir \
-r requirements.txt \
--index-url \
https://AVAST_ARTIFACTORY/pip/packages

ADD kafka_consumer_runner.py /
ADD lib/kafka_consumer.py /lib/
ADD config/config.py /config/

CMD ["python", "./kafka_consumer_runner.py"]
```

Obrázek 4.19: Hlavní smyčka přijímání zpráv z Kafky

První řádek s příkazem `FROM` určuje, odkud stáhnout tzv. image template (tj. docker balíček, který obsahuje prostředí, které se má při kompilaci docker kontejneru vytvořit). Protože je skript `kafka consumer` napsaný v Pythonu, je zapotřebí použít příslušný template. Kvůli bezpečnosti a pro zajištění integrity má firma vlastní úložiště s docker balíčky. Proto je balíček *python* : 3.7 stažen z interního úložiště.

Dalším příkazem je překopírován soubor `requirements.txt` do prostředí nově vzniklého kontejneru. Tento soubor obsahuje seznam nestandardních python knihoven, tedy knihoven, které je potřeba ručně doinstalovat a nejsou obsaženy v instalaci pythonu.

Dále je spuštěn příkaz `pip install`, tyto nestandardní knihovny nainstaluje. K tomu je z bezpečnostních důvodů opět využito interní úložiště obsahující předeschválené python knihovny.

Těmito příkazy je tedy připravené prostředí kontejneru. Zbývá přesunout zdrojové soubory skriptu, který zde poběží. Toho je docíleno příkazy `ADD`. Posledním příkazem `CMD` je spuštěn samotný skript.

Takto vytvořený Dockerfile je potřeba sestavit (tzv. build) a následně napsat (tzv. deploy). K sestavení balíčku byl využit firemní server TeamCity (server pro kontinuální integraci a správu sestavování aplikací vyvíjen firmou JetBrains [18]). Zdejší firemní struktura obsahuje šablony pro build docker balíčků, které jsou následně nahrány do již zmíněného firemního úložiště.

Při nasazení aplikace do Luftu bylo opět využito firemní šablony. Při nasazování je potřeba vytvořit `.yaml` soubor, obsahující veškerou konfiguraci vznikající aplikace. Tento soubor byl převzat, bylo zde zapotřebí pouze nastavit environmentální proměnné, které skript využívá. Jednalo se o přihlašovací údaje a SSL certifikát k frontovému serveru Kafka a token, pomocí kterého skript spouští jenkins job pro stahování souborů (viz sekce 4.5.2.2).

Otestování a zhodnocení přínosu

5.1 Testovací proces

Před nasazením aplikace do provozu bylo nutné zajistit důkladné otestování. Tento proces byl převážně realizován formou end-to-end. Tato metoda kontroluje, zda je běh testované aplikace předpokládáný a zda aplikace vrací na zadaná vstupní data očekávané výstupy. Tento typ testování byl použit nejdříve pro jednotlivé komponenty, a následně pro celý běh aplikace. Komponenty, které byly při testování monitorovány jsou následující:

- Nahrávání adres do frontového systému Kafka
- Přijímání zpráv z Kafky a následná inicializace stahovacího procesu
- Zabecečené stahování a filtrování dat
- Nahrávání validních dat do HCP úložiště

5.1.1 Komponenty využívající frontový systém Kafka

Skripty pro komunikaci se systémem Kafka byly testovány již v průběhu vývoje, neočekávaly se zde tedy rozsáhlé komplikace. Při formě end-to-end byl zlomek testovacích adres v procesu Kafka Producer nahrán do Kafky a následně byly v procesu Kafka Consumer všechny adresy úspěšně přečteny. Následně byl proveden tzv. zatěžkávací test, tedy test, jehož snahou je určit, jak náročný provoz je daný systém schopný unést bez poruchy. Předpokládaná zátěž systému Magpie je v řádu tisíců až desetitisíců zpráv týdně. V zatěžkávacím testu byl tedy proveden přenos milionu zpráv, což bylo vyhodnoceno

jako dostačující množství. Všechny přijaté zprávy byly následně uloženy do souboru, jehož velikost byla porovnána se souborem obsahujícím vstupní data. Ze shody velikosti obou souborů bylo vyhodnoceno, že byl přenos úspěšný.

5.1.2 Zabezpečené stahování a filtrování dat

Testování této komponenty bylo nejobsáhlejší a probíhalo na vícero testovacích množinách. Prvotní testovací množina dat obsahovala 1000 vzorků a jednalo se o starší seznam nejnavštěvovaných url adres shromážděný službou Alexa top sites[19]. Pro každou adresu byly staženy zdrojové soubory a následně byla provedena analýza výstupních dat. Při této analýze bylo zjištěno, že přes prvotní předpoklady není ošetření přesměrování se stavovými kódy 301 a 302 dostačující. Stavový kód 301 se používá pro trvalné přesměrování (tzv. Moved Permanently), kód 302 byl původně navržen pro dočasné přesměrování, nyní je nejčastěji používaným stavovým kódem pro obecná přesměrování. Z analýzy vyplynulo, že množství stránek používá i jiné stavové kódy pro přesměrování. Z ještě nezmiňovaných se nejčastěji objevoval kód 307, který od standardu HTTP/1.1 nahradil kód 302 pro dočasné redirecty (tzv. Temporary redirect). Proto bylo ošetření přesměrovávacích stavových kódů rozšířeno. Byl upraven inicializační skript(4.2) pro program Fiddler, který obsahuje podmínku pro zachytávané stavové kódy(5.1).

```
if (code == 200 || code == 301 || code == 302)
```

Obrázek 5.1: Původní podmínka selekce stavových kódů

Zmiňovaná podmínka byla rozšířena na zachytávání všech stavových kódů z intervalu (299, 310) (znázorněno na snippetu 5.2).

```
if (oSession.responseCode == 200 ||  
    (299 < oSession.responseCode < 310))
```

Obrázek 5.2: Upravená podmínka selekce stavových kódů

Z analýzy následně vyplynula důležitost uchovávání logu (výpis informačních zpráv) prohlížeče. Tento výpis je důležitý v situacích, kdy zadaná stránka již neexistuje, nebo se v průběhu připojení naskytly jiné komplikace a ob-

sah stránky není dostupný. V takovém případě není nutné pokračovat třídícím algoritmem, poněvadž všechny zachycené soubory obsahují pouze režijní komunikaci prohlížeče a Fiddleru. Geckodriver (webdriver používaný nástrojem Selenium k ovládání internetového prohlížeče Mozilla Firefox) ovšem tuto funkcionalitu nepodporuje. Z toho důvodu bylo od těchto nástrojů odstoupeno a byl použit Chromedriver v souvislosti s internetovým prohlížečem Google Chrome. K tomu bylo zapotřebí na virtuální stroj nainstalovat zmíněný webdriver ve verzi shodné s nainstalovaným prohlížečem Google Chrome (pro oba programy byla nainstalována nejnovější verze 80). Základní ovládání prohlížeče se liší pouze v importování a následné inicializaci rozdílné třídy reprezentující prohlížeč z knihovny nástroje Selenium. Kód pro ovládání prohlížeče nástrojem Selenium 4.1 byl tedy přepsán. Základní ovládání prohlížeče Google Chrome s využitím logování je znázorněno na snippetu 5.3.

```
import time

from selenium.webdriver import Chrome
import selenium.webdriver.common.desired_capabilities
import selenium.webdriver

chrome_options = selenium.webdriver.ChromeOptions()

d = desired_capabilities.DesiredCapabilities.CHROME
d['goog:loggingPrefs'] = {'browser': 'ALL'}

browser = Chrome(
    options=chrome_options,
    desired_capabilities=d
)

browser.get("https://www.seznam.cz")
log = browser.get_log('browser')
```

Obrázek 5.3: Ovládání prohlížeče Google Chrome pomocí nástroje Selenium

Obdobně jako u prohlížeče Firefox lze měnit nastavení prohlížeče skrze instanci dedikované třídy, zde `ChromeOptions`. Dále je potřeba vytvořit instanci třídy `DesiredCapabilities`, v které je možné nastavit logovací preference(5.4). Parametr `'ALL'` určuje úroveň logování, tedy zda se mají ukládat

```
d['goog:loggingPrefs'] = {'browser': 'ALL'}
```

Obrázek 5.4: Logovací preference prohlížeče

všechny informační zprávy (ALL), nebo jen zprávy typu ERROR či WARNING. Tato instance je předána spolu s třídou pro nastavení prohlížeče jako parametry při inicializaci prohlížeče. Po načtení stránky lze přistoupit k logu pomocí metody `get_log()`, která vrací strukturu obsahující informace v závislosti na nastavených logovacích preferencích. Při iteraci nad touto strukturou lze přistoupit k jednotlivým záznamům (snippet5.5). Tento výpis byl následně

```
for entry in log:
    print(f"{entry['message']}")
    if 'Fiddler - DNS Lookup Failed' in entry['message']:
        raise DNSLookupFailureException(entry['message'])
    if 'Fiddler - Connection Failed' in entry['message']:
        raise ConnectionFailureException(entry['message'])
    if 'Fiddler - Receive Failure' in entry['message']:
        raise ReceiveFailureException(entry['message'])
```

Obrázek 5.5: Iterace nad strukturou obsahující logování prohlížeče

použit k vytvoření podmínek kontrolujících, zda byla stránka skutečně načtena. Chybové hlášky prohlížeče nejsou nijak exaktní, bylo tedy nutné podmínky vytvořit na základě porovnávání stringů. Z analýzy vyplynulo, že se při neexistující stránce nebo při přerušeném připojení objevují tři rozdílné výpisy. Byly tedy vytvořeny podmínky, které tyto výpisy odchyťávají a vytvářejí výjimky, které se na začátku procesu filtrování dat vyhodnocují.

5.1.3 Přenos souborů do HCP úložiště

Posledním krokem testování byl klient pro HCP úložiště. Samotná komunikace s úložištěm je zajištěna pomocí knihovny vytvořené týmem spravujícím systém Scavenger, tuto funkcionalitu tedy testovat nutné nebylo. V testovacím procesu bylo převážně kontrolováno správné přiřkládání metadat k nahrávaným souborům. K testování byla opět využita starší seznam nejnavštěvovaných url adres shromážděný službou Alexa top sites[19] o velikosti 1000 vzorků. Jelikož

je přenos závislý na stažených datech, byly tyto vzorky nejdříve zpracovány komponentou pro stahování a filtraci dat. Soubory, které byly v tomto kroku vyhodnocené jako validní byly spolu s jejich metadaty nahrány do HCP úložiště a následně byla provedena kontrola nahraných dat. Metadata nahraných souborů byla porovnána se staženými metadaty a ve všech případech byla shledána shoda. Z toho bylo vyvozeno, že přiřkládání metadat k nahrávaným souborům funguje v pořádku.

Při této analýze bylo avšak zjištěno, že přejmenovávání nahraných souborů podle klíče

$$\{\text{původní_adresa}\}\{\text{pořadí_zpracovaného_souboru}\} \quad (5.1)$$

není optimální. V HCP úložišti mohou být pouze unikátní soubory (soubory s unikátním názvem). K přenosu souborů z HCP úložiště do systému Scavenger nedochází hned, avšak periodicky s časovým odstupem. V situaci, kdy by v tomto čase byla zpracována stejná url adresa vícekrát, došlo by k pokusu o nahrání souboru, který by měl stejné jméno jako soubor již přítomný v HCP úložišti a přenos by neproběhl. Bylo tedy nutné klíč přetvořit aby byla zachována unikátnost. K tomuto účelu je k názvu souboru přidáván i současný čas. Nový klíč pro přejmenovávání nahrávaných souborů tedy vypadá následovně:

$$\{\text{původní_adresa}\}\{\text{pořadí_zpracovaného_souboru}\}\{\text{aktuální_čas}\} \quad (5.2)$$

Dále byla na začátek přenosu přidána kontrola, zda při filtraci dat byly vůbec některé soubory označeny jako validní. Pokud tomu tak není, proces se automaticky ukončí.

5.2 Zobecnění projektu a budoucí využití

V současné době je projekt Magpie využit k doplňování cleansetu stahováním zdrojových souborů webových stránek s vysokou prevalencí. Při vývoji se ale již počítalo s jeho širším využitím v jiných aplikacích potřebujících zabezpečený zisk zdrojových souborů. Jedním z takových případů může být analýza phishingových stránek. Jedná se o stránky, které využívají nepozornosti uživatele k zisku citlivých informací. Při analýze těchto stránek, tedy při rozhodování zda je stránka phishingového charakteru, je často potřeba podívat se do zdrojových souborů stránky. K zisku těchto dat by bylo možné využívat systém Magpie.

Jiným využitím Magpie při analýze malwaru může být zisk infikovaných souborů s následným nahráním do systému Scavenger. Systém Scavenger obsahuje kromě databáze cleansetu mimojiné i databázi škodlivých souborů. Rozhodovací logika vložení nahrávaného souboru do příslušné databáze je lehce implementovatelná v systému Scavenger pomocí filterů založených na metadatech nahrávaných souborů. Pokud by byly tedy metadata obohaceny o tuto informaci, je Magpie schopný zpracovávat i infikovaná data.

5.3 Zhodnocení přínosu

Hlavní přínos systému Magpie ční v automatizaci doplňování cleansetu. Dříve bylo doplňování čistých či infikovaných HTML a .js souborů do databáze Scavengeru práce analytiků, kteří museli pro každý nahraný soubor manuálně zvolit, zda se jedná o soubor infikovaný či nikoliv. Tímto způsobem není možné zpracovávat obsáhlý počet čistých souborů, které je třeba nahrávat na cleanset, aby bylo zajištěno jeho funkčnosti. Díky systému Magpie je možné cleanset obohacovat o 200 000 nových souborů týdně, čímž je mnohonásobně navýšena stávající kapacita přidávání dat. V důsledku toho je cleanset obsáhlejší, což přispívá k lépe fungující automatizaci při vydávání nových detekcí. Virové definice vytvářejí nejen analytici, kteří musejí kontrolovat, zda definice neblokuje čisté soubory na cleansetu. Virové detekce zároveň vznikají i díky automatickým nástrojům, u kterých je silný cleanset neméně důležitý. Díky silnému cleansetu klesá množství špatných detekcí vytvořených automatickými nástroji a Magpie silný cleanset vytváří.

Závěr

Diplomová práce spočívala ve vytvoření automatizovaného systému pro obohacování cleansetu o čisté HTML a .js soubory stahováním těchto dat ze zadaných url adres. Prvním bodem zadání byla analýza možných způsobů realizace tohoto systému. Během této analýzy bylo zjištěno, jaké jsou nejlepší postupy při realizaci projektu.

Na základě toho byl vytvořen projekt Magpie. Projekt byl navržen v programovacím jazyku Python a je rozdělen do několika komponent. Pro předzpracování vstupních dat byl vytvořen skript využívající frontový systém Apache Kafka. Pomocí skriptu je možné zpracovávat seznamy url adres služby Alexa Top Sites, či ručně zadané adresy. Tento skript je nasazen v systému Jenkins. Dále byl vytvořen skript pro čtení zpráv z frontového systému Kafka. Tento skript byl dockerizován a nasazen v systému Kubernetes. Jádrem projektu spočívá ve stahování dat pomocí MitM klientu Fiddler v kombinaci s internetovým prohlížečem Google Chrome ovládaným pomocí nástroje Selenium. Tento proces je spouštěn na virtuálním stroji s využitím programu Oracle Virtualbox a nasazen v systému Jenkins. Stažená data jsou odesílána do HCP úložiště a následně nahrána do databáze cleansetu v systému Scavenger, k čemuž slouží poslední komponenta také nasazená v systému Jenkins.

Dalším bodem zadání bylo zobecnění vzniklého systému pro budoucí využití v jiných projektech. Se zobecněním bylo počítáno již při samotném návrhu hlavní aplikace. Systém Magpie je schopen zpracovávat nejen adresy pro stahování dat k doplňování cleansetu.

ZÁVĚR

Posledním bodem zadání bylo důsledné otestování navrženého systému. Testování proběhlo na vícero testovacích množinách s využitím seznamu adres služby Alexa Top Sites. Každá komponenta byla otestována zvlášť a nakonec bylo využito i end-to-end testování celé aplikace.

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. *Distribované 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.docker.com/resources/what-container>

- [9] <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

- [10] [<https://code.vmware.com/web/sdk/6.7/vsphere-automation-python>]
- [11] Dorman Michael. *pyvbox Documentation*. ©2017 Dostupné z: <https://buildmedia.readthedocs.org/media/pdf/pyvbox/latest/pyvbox.pdf>
- [12] <https://github.com/mozilla/geckodriver/releases>
- [13] <https://www.mozilla.org/cs/firefox/new/>
- [14] https://www.google.com/intl/cs_CZ/chrome/
- [15] <https://chromedriver.chromium.org/>
- [16] <https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>
- [17] <https://docs.confluent.io/current/clients/python.html>
- [18] <https://www.jetbrains.com/teamcity/>
- [19] <https://www.alexa.com/topsites>

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 L ^A T _E X
└─ pictures	zdrojové obrázky pro formát L ^A T _E X
video	
└─ tutorial.mp4.....	instruktační video