VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

PRENESENIE EXTRAKTOROV NÁSTROJA PLASO NA PLATFORMU APACHE SPARK

PORTING OF PLASO EXTRACTORS TO THE APACHE SPARK PLATFORM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. MIROSLAV BALÁŽ**
AUTHOR

VEDOUCÍ PRÁCE — **RNDr. MAREK RYCHLÝ, Ph.D.**
SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání diplomové práce

140537

Ústav:	Ústav informačních systémů (UIFS)
Student:	**Baláž Miroslav, Bc.**
Program:	Informační technologie a umělá inteligence
Specializace:	Softwarové inženýrství
Název:	**Přenesení extraktorů nástroje Plaso na platformu Apache Spark**
Kategorie:	Paralelní a distribuované výpočty
Akademický rok:	2022/23

Zadání:

1. Seznamte se s nástrojem Plaso/log2timeline pro extrakci dat ze souborových systémů za účelem forenzní analýzy. Dále se seznamte s platformou Apache Spark pro distribuované zpracování velkých dat. Prozkoumejte možnosti přenesení nástroje Plaso na platformu Apache Spark pomocí PySpark.
2. Navrhněte systém kompatibilní s nástrojem Plaso a běžící nad Apache Spark vč. způsobu využití stávajícího kódu nástroje Plaso (zejména v něm obsažených extraktorů).
3. Po konzultaci s vedoucím systém dle návrhu implementujte.
4. Řešení otestujte na vhodné datové sadě a porovnejte s původní implementací nástroje Plaso. Výsledky zveřejněte jako open-source.

Literatura:

- Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. *Learning Spark: Lightning-Fast Big Data Analysis*. First edition, 256 pp., O'Reilly Media, 2015. ISBN 978-1-449-35862-4.
- Databricks Spark Reference Applications [online]. 2017 [seen 2021-09-29]. Available at [https://databricks.gitbooks.io/databricks-spark-reference-applications]

Při obhajobě semestrální části projektu je požadováno:
Body 1 a 2.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce:	**Rychlý Marek, RNDr., Ph.D.**
Vedoucí ústavu:	Kolář Dušan, doc. Dr. Ing.
Datum zadání:	1.11.2022
Termín pro odevzdání:	17.5.2023
Datum schválení:	18.10.2022

Abstrakt

Cieľom diplomovej práce je transformácia existujúceho Plaso nástroja pre forenznú analýzu na platformu Apache Spark. Teoretická časť rozoberá fungovanie a architektúru Plaso nástroja. Práca ďalej skúma aktuálne nástroje, ktoré implementujú distribuované výpočtové modely. Popisuje ich architektúru, dátové abstrakty a spôsob ich fungovanie. Taktiež sú v práci popísané aktuálne nástroje implementujúce distribuované uložiská. V rámci práce bol vytvorený nástroj Plasospark, ktorý prevádza výpočet Plaso nástroja na Spark platformu a využíva Hadoop HDFS uložisko pre forenzné dáta.

Abstract

The theoretical part discusses the functioning and architecture of the Plaso tool. The thesis further explores current tools that implement distributed computational models. It describes their architecture, data abstracts and how they work. The thesis also describes current tools that implement distributed storage. The work includes the creation of the Plasospark tool, which converts the computation of the Plaso tool to the Spark platform and uses the Hadoop HDFS storage for forensic data.

Kľúčové slová

Plaso, Log2Timeline, Apache Spark, Hadoop HDFS, Distribuovaný výpočet, Forenzná analýza, Distribuované uložisko

Keywords

Plaso, Log2Timeline, Apache Spark, Hadoop HDFS, Distributed computing, Forensic analysis, Distributed storage

Citácia

BALÁŽ, Miroslav. *Prenesenie extraktorov nástroja Plaso na platformu Apache Spark*. Brno, 2023. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce RNDr. Marek Rychlý, Ph.D.

Prenesenie extraktorov nástroja Plaso na platformu Apache Spark

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením RNDr. Mareka Rychlého Ph.D. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

.....
Miroslav Baláž
10. mája 2023

Poďakovanie

Chcel by som sa poďakovať vedúcemu práce RNDr. Marekovi Rychlému Ph.D. za poskytnuté rady a cenné informácie počas riešenia práce. Ďalej by som sa chcel poďakovať mojej rodine, ktorá ma podporovala počas celého štúdia. Mojim kamarátom Mgr. Boris Kiška, Ing. Jaroslav Filo, Ing. Petr Backo, Ľubomír Mikula, Patrik Baláž za spríjemnenie študentských čias a taktiež Ing. Jane Surovcovej za silnú podporu v každých časoch.

Obsah

Zoznam obrázkov

Kapitola 1

Úvod

Internet sa stal neoddeliteľnou súčasťou denného života. Vďaka tomu sa exponenciálne zvýšil objem dát, s ktorým sa v dnešnom svete pracuje. Tento zvyšujúci sa trend ovplyvňuje viacero faktorov ako sociálne média, vzostup internetu vecí (IoT) a narastujúca adaptácia cloudových výpočtov.

Všetky tieto faktory spôsobili, že práca s dátami vyžaduje nové prístupy, kedže klasické prístupy sú dnes už nedostačujúce. Práve preto boli vytvorené distribuované výpočtové modely, ktoré sa snažia efektívne spracovať enormný objem dát s ktorým je potrebné v dnešných dňoch pracovať.

Odvetvie forenznej analýzy taktiež vníma tento trend zvyšujúceho sa objemu dát. V minulosti sa pri forenznej analýze pracovalo len so zlomkov objemu dát oproti dnešným dňom. Dnešný objem forenzných dát, ktoré je potrebné analyzovať na počítačoch je niekoľko násobne vyšší. K tomu je v dnešných dňoch potrebná omnoho hlbšia forenzná analýza, kedže vznikajú lepšie a komplexnejšie anti-forenzné nástroje a dát je čoraz viac.

Existuje veľká škála forenzných nástrojov, ktoré sú dnes využívané a jedným z nich je nástroj Plaso. Plaso práve kvôli anti-forenzným nástrojom poskytuje hĺbkovú analýzu skúmaného systéme a vytvára obrovské množstvo záznamov zo skúmaného systému a jeho súborov. Plaso funguje na princípe centralizovaného výpočtového modelu. Všetky jeho výpočty prebiehajú na jedno stroji, čím nie je schopný udržať krok s rastúcim objemom dát a spôsobom jeho spracovania.

Hlavným cieľom práce je transformácia Plaso nástroja do distribuovaného výpočtového modelu, ktorý bude schopný svôj výkon škálovať, čím bude možné udržať krok s rastúcim objemom dát a prispôsobiť jeho výkon tomuto trendu. Tento nástroj bude v práci vydaný ako voľne dostupný program, ktorý bude možné použiť pri forenznej analýze dát.

Pre konvertovanie nástroja Plaso do distribuovaného modelu je potrebné pochopiť ako nástroj funguje a preskúmať jeho zdrojový kód, kedže neexistuje detailná dokumentácia tohto nástroja. Po preskúmaní a pochopení nástroja je potrebné identifikovať jednotlivé komponenty, ktoré je potrebné izolovať a použiť v distribuovanom výpočte. Tento krok predstavuje kapitola 2. V tejto kapitole je popísané ako Plaso nástroj funguje, ktoré komponenty vykonávajú jednotlivé činnosti a ako je v Plaso nástroji riešenia distribúcia výpočtu.

Ďalej sa práca venuje aktuálnym výpočtovým modelom v distribuovanom prostredí. V kapitole 3 sú popísané aktuálné distribuované výpočtové modely ako Spark, Storm alebo Flink. Jednotlivé nástroje sú popísané z hladiska architektúry a dátových abstraktov, ktoré tieto modely prinášajú.

V kapitole 4 sú rozobrané aktuálne nástroje, ktoré implementujú distribuované súborové uložiská. Tieto nástroje sú v kapitole preskúmané z hladiska architektúry a princípov, ktoré sú v uložiskách využité.

Návrh distribuovaného nástroja Plaso, pomenovaný v práci ako Plasospark, je zahrnutý v kapitole 5 a jeho implementácia je popísaná v kapitole 6. V implementačnej časti sa vychádza z naštudovaných informácií o aktuálnych distribuovaných nástrojoch a ich fungovaniu v teoretickej časti. Implementácia taktiež vychádza z informácií o Plaso nástroji a jeho implementácie, ktorá bola zistená preskúmaním a analýzou tohto nástroja. Vyhodnotenie implementovaného nástroja v kapitole 7, ktoré bolo vykonané na dostupnej testovacej sade Plaso nástroja.

Posledná kapitola 8 obsahuje záverečné zhrnutie celej práce, dosiahnuté ciele a možnosti ako nástroj vylepšiť.

Kapitola 2

Digitálna forenzná analýza s Plaso

Jedna z prvých definícií digitálnej forenznej analýzi bola definovaná v [16] ako obor, ktorý má za cieľ zozbierať, validovať, identifikovať, analyzovať a prezentovať digitálne dôkazy, ktoré vychádzajú z digitálnych zdrojov pre účely rekonštrukcie udalostí, ktoré viedli ku kriminálnej aktivite. Iný zdroj ako napríklad [6] ju definuje jednoduchšie ako vedný obor, ktorého cieľom je detekcia, extrakcia a analýza dôkazov z digitálnych médií a považuje ju za kritický obor v kyberpriestore. Vo všeobecnosti sa jedná o proces, ktorý využíva vedecké a technické metódy pre skúmanie a analýzu digitálnych zariadení a médii s cieľom vytvoriť digitálne dôkazy, ktoré môžu byť použité v súdnom procese. Tento proces môže zahŕňať veľa aktivít ako napríklad opätovné získanie vymazaných alebo skrytých súborov, analýzu log súborov, preskúmanie systému a artefaktov z aplikácii.

Cieľom forenznej analýzy je poskytnúť kompletný a presný obraz o digitálnej aktivite v zariadení alebo v systéme, ktorý je možné použiť ako usvedčujúci material v právnych sporoch. Môže sa jednať o právne spory ako napríklad hacknutie, kyber šikana alebo online podvody. V korporátnej oblasti je forenzná analýza využívaná na vyšetrovanie únikov dát, krádeže intelektuálneho majetku alebo iných typov kyber zločinov spojených s majetkom spoločností. Jedným z príkladov využitia forenznej analýzy v reálnych prípadoch je hacknutie SONY spoločnosti v roku 2014[17].

Pre účely forenznej analýzy existuje veľké množstvo voľne dostupných ale aj komerčných nástrojov. Plaso (Plaso Langar Að Safna Öllu) je voľne dostupný nástroj, ktorý slúži na extrakciu udalostí s časovou pečiatkou. Bol vyvinutý Kristinnom Gudjonssonom a aktuálne je spravovaný Google Security tímom. Z udalostí vytvára super časovú os (z anglického supertimeline), ktorá poskytuje pohľad na systém v konkrétnom čase. Plaso sa často používa v spojení s ostatnými nástrojmi ako The Sleuth Kit (TSK) a Autopsy pre poskytnutie širšieho pohľadu na dôkazy v systéme. Taktiež je Plaso zahrnuté vo viacerých sadách nástrojov pre forenznú analýzu ako SIFT (SANS Investigative Forensic Toolkit) a CAINE (Computer Aided INvestigative Environment).

2.1 Log2Timeline a Plaso

Log2Timeline bol predstavený Gudjonssonom [10]. Predstavil nástroj, ktorý slúži na tvorbu a analýzu super časových osí napísaný v perl. Následne bol log2timeline prepísaný do python programovacieho jazyku a predstavený ako Plaso nástroj. Plaso predstavuje backend, ktorý využíva log2timeline ako svôj frontend. Plaso prináša framework, ktorý je plne rozšíriteľný a poskytuje dodatočné nástroje, ktoré je možné využiť vo forenznej analýze. Slúži pre

extrakciu a analýzu udalostí zo systému z jeho rôznych časti. Vytvára časovú os udalostí zo zmazaných súborov, nevyužitých pamäťových miest (z anglického file slack space) alebo živej pamäti systému. Je schopný spracovať veľkú škálu formátov ako napríklad súbory histórie prehliadačov, emaily, log súbory a taktiež je schopný spracovať mnohé typy forenzných artefaktov ako Windows Registry a tieňové kópie zväzku (z anglického Volume Shadow Copy).

Plaso je napísané v programovacom jazyku Python a využíva viaceré knižnice vrátane SleuthKit[5], libyal a pytz. Hlavnou výhodou Plaso nástroje je jeho architektúra, ktorá je plne modulárna. Modularita spočíva v jednotlivých extraktoroch (z anglického parsers) a rozširujúcich modulov (z anglického plugin), ktoré plaso používa na spracovanie súborov pri extrakcii a analýze. Extraktor predstavuje python modul, ktorý je zodpovedný za spracovanie špecifického dátového formátu. Následne je možné extraktory rozšíriť pomocou rozširujúcich modulov, ktoré využívajú logiku extraktoru a pridávajú dalšie rozšírenia tomuto základnému formátu. Ak je potrebné k existujúcemu extraktoru pridať novú logiku extrakcie, tak stačí Plaso rozšíriť len o nový modul a konkrétny formát súboru bude spracovaný s už existujúcim extraktorom. Prehľad jednotlivých extraktorov, ktoré sú dostupné v čase písania, je možné vidieť v prílohe A. Plaso nástroj je možné použiť na MAC OS, Windows a Linux operačnom systéme.

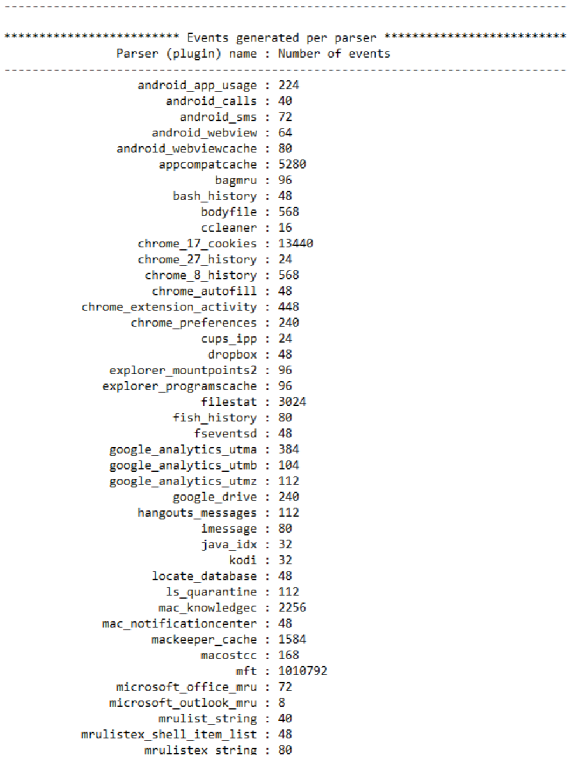
Výstup súbor extrakcie je možné spracovať s dalšími nástrojmi, ktoré plaso poskytuje vo svojom frameworku. V balíku sa nachádzajú nasledovné nástroje:

- **log2Timeline** – Ako už bolo spomenuté, log2timeline predstavuje frontend Plaso nástroja. Jedná sa o nástroj príkazového riadku, ktorý slúži na extrakciu udalostí z jednotlivých súborov, priečinkov, obrazov obsahu pamäti (z anglického storage media image) alebo zariadení. Log2Timeline vytvára výstupný plaso súbor, ktorý je možné následne analyzovať s nástrojmi `pinfo` alebo `psort`. Plaso výstupný súbor obsahuje extrahované udalosti spolu s metadátami o procese kolekcie a metadátami o získaných udalostiach. Taktiež môže obsahovať značky, ktoré boli získanie z plaso analýzy pri extrakcii.

- **pinfo** – Tento nástroj poskytuje užívateľovi informácie o informáciach uložených v plaso súbore. Napríklad poskytuje informácie o priebehu predspracovania jednotlivých súborov, metadáta o jednotlivých uložiskách, použitých extraktoroch, celkový počet vyextrahovaných udalostí a ďalšie. Výstup tohto nástroja je zobrazený na obrázku 2.1.

- **psort** – Filtrovanie a triedenie vyextrahovaných udalostí v Plaso súbore je možné pomocou tohto nástroja ako aj vyhľadávanie špecifických udalostí v konkrétnych časových rámcoch. Dodatočne poskytuje `psort` možnosť konvertovať Plaso formát do viacerých známych formátov ako napríklad CSV formát.

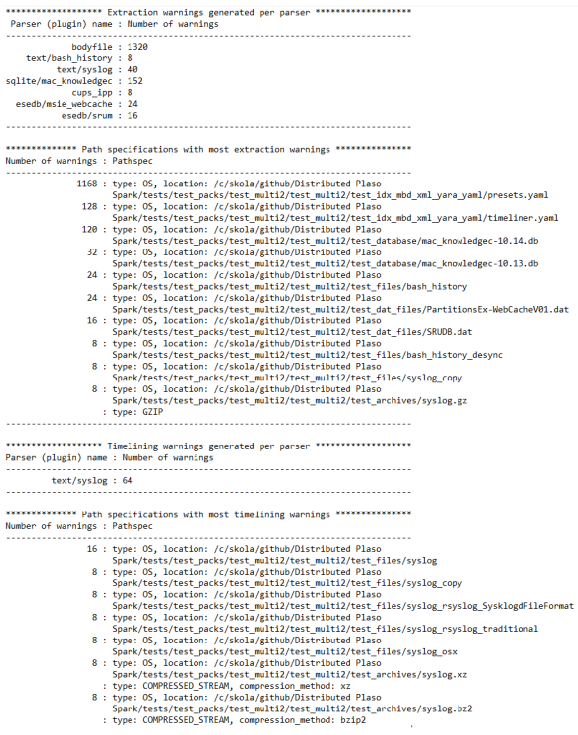
- **image_export** – Slúži na extrakciu súborov z obrazov pamäte (z anglického storage media image) alebo zariadení. Extrakcia môže mať špecifické parametre, ktoré určujú aké súbory majú byt vyextrahované napríklad podľa cesty, prípony súboru, času vytvorenia súborov a podobných parametrov.



```
------------------------------------------------------------------------------

************************* Events generated per parser *************************
             Parser (plugin) name : Number of events
------------------------------------------------------------------------------
               android_app_usage : 224
                   android_calls : 40
                     android_sms : 72
                 android_webview : 64
            android_webviewcache : 80
                   appcompatcache : 5280
                          bagmru : 96
                    bash_history : 48
                        bodyfile : 568
                       ccleaner : 16
                chrome_17_cookies : 13440
                chrome_27_history : 24
                 chrome_8_history : 568
                  chrome_autofill : 48
          chrome_extension_activity : 448
                chrome_preferences : 240
                        cups_ipp : 24
                         dropbox : 48
             explorer_mountpoints2 : 96
            explorer_programscache : 96
                        filestat : 3024
                   fish_history : 80
                       fseventsd : 48
             google_analytics_utma : 384
             google_analytics_utmb : 104
             google_analytics_utmz : 112
                    google_drive : 240
                hangouts_messages : 112
                        imessage : 80
                        java_idx : 32
                            kodi : 32
                  locate_database : 48
                    ls_quarantine : 112
                   mac_knowledgec : 2256
          mac_notificationcenter : 48
                 mackeeper_cache : 1584
                        macostcc : 168
                             mft : 1010792
                microsoft_office_mru : 72
               microsoft_outlook_mru : 8
                     mrulist_string : 40
          mrulistex_shell_item_list : 48
                     mrulistex_string : 80
```

(a) Základné informácie o výstupnom plaso súbore spolu s množstvom vygenerovaných udalostí jednotlivými extraktormi.



```
****************** Extraction warnings generated per parser ******************
 Parser (plugin) name : Number of warnings
------------------------------------------------------------------------------
                bodyfile : 1320
    text/bash_history : 8
          text/syslog : 40
sqlite/mac_knowledgec : 152
              cups_ipp : 8
  esedb/msie_webcache : 24
           esedb/srum : 16
------------------------------------------------------------------------------

************** Path specifications with most extraction warnings **************
Number of warnings : Pathspec
------------------------------------------------------------------------------
            1168 : type: OS, location: /c/skola/github/Distributed Plaso
                   Spark/tests/test_packs/test_multi2/test_multi2/test_idx_mbd_xml_yara_yaml/presets.yaml
             128 : type: OS, location: /c/skola/github/Distributed Plaso
                   Spark/tests/test_packs/test_multi2/test_multi2/test_idx_mbd_xml_yara_yaml/timeliner.yaml
             120 : type: OS, location: /c/skola/github/Distributed Plaso
                   Spark/tests/test_packs/test_multi2/test_multi2/test_database/mac_knowledgec-10.14.db
              52 : type: OS, location: /c/skola/github/Distributed Plaso
                   Spark/tests/test_packs/test_multi2/test_multi2/test_database/mac_knowledgec-10.13.db
              24 : type: OS, location: /c/skola/github/Distributed Plaso
                   Spark/tests/test_packs/test_multi2/test_multi2/test_files/bash_history
              24 : type: OS, location: /c/skola/github/Distributed Plaso
                   Spark/tests/test_packs/test_multi2/test_multi2/test_dat_files/PartitionsEx-WebCacheV01.dat
              16 : type: OS, location: /c/skola/github/Distributed Plaso
                   Spark/tests/test_packs/test_multi2/test_multi2/test_dat_files/SRUDB.dat
               8 : type: OS, location: /c/skola/github/Distributed Plaso
                   Spark/tests/test_packs/test_multi2/test_multi2/test_files/bash_history_desync
               8 : type: OS, location: /c/skola/github/Distributed Plaso
                   Spark/tests/test_packs/test_multi2/test_multi2/test_files/syslog_copy
               8 : type: OS, location: /c/skola/github/Distributed Plaso
                   Spark/tests/test_packs/test_multi2/test_multi2/test_archives/syslog.gz
                 : type: GZIP
------------------------------------------------------------------------------

****************** Timelining warnings generated per parser ******************
Parser (plugin) name : Number of warnings
------------------------------------------------------------------------------
        text/syslog : 64
------------------------------------------------------------------------------

************** Path specifications with most timelining warnings **************
Number of warnings : Pathspec
------------------------------------------------------------------------------
              16 : type: OS, location: /c/skola/github/Distributed Plaso
                   Spark/tests/test_packs/test_multi2/test_multi2/test_files/syslog
               8 : type: OS, location: /c/skola/github/Distributed Plaso
                   Spark/tests/test_packs/test_multi2/test_multi2/test_files/syslog_copy
               8 : type: OS, location: /c/skola/github/Distributed Plaso
                   Spark/tests/test_packs/test_multi2/test_multi2/test_files/syslog_rsyslog_SysklogdFileFormat
               8 : type: OS, location: /c/skola/github/Distributed Plaso
                   Spark/tests/test_packs/test_multi2/test_multi2/test_files/syslog_rsyslog_traditional
               8 : type: OS, location: /c/skola/github/Distributed Plaso
                   Spark/tests/test_packs/test_multi2/test_multi2/test_files/syslog_osx
               8 : type: OS, location: /c/skola/github/Distributed Plaso
                   Spark/tests/test_packs/test_multi2/test_multi2/test_archives/syslog.xz
                 : type: COMPRESSED_STREAM, compression_method: xz
               8 : type: OS, location: /c/skola/github/Distributed Plaso
                   Spark/tests/test_packs/test_multi2/test_multi2/test_archives/syslog.bz2
                 : type: COMPRESSED_STREAM, compression_method: bzip2
```

(b) Dodatočné informácie o procese extrakcie. Informácie ako počet varovaní ktoré vygenerovali jednotlivé extraktory pri extrakcii, počet varovaní z konkrétnych ciest extrahovaných súborov.

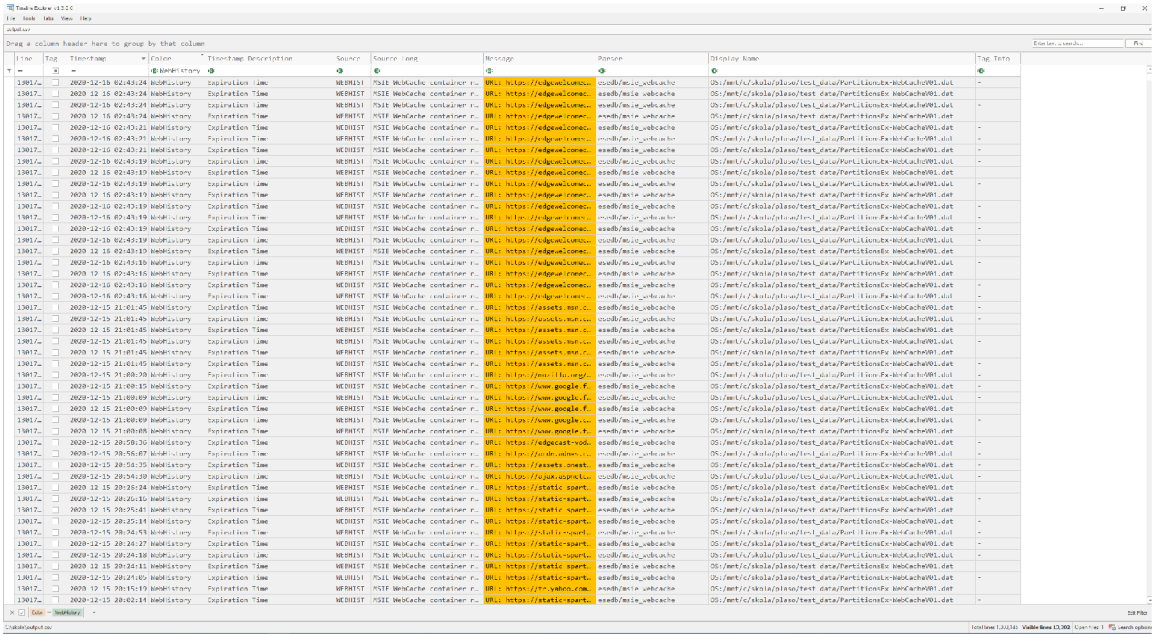
Obr. 2.1: Výstup z nástroja `pinfo`, po spracovaní testovacej sady

Super timeline

Aj ked tradičné časové osy poskytujú dobrý pohľad na skúmaný systém majú svoje nevýhody. Tradičná časová os poskytuje pohľad na udalosti len z pohľadu vytvorenia, modifikácie a zmazania súborov v rámci súborového systému. Čo prináša nevýhody, ktoré popisuje [10]. Ako jedným z hlavných nedostatkov tradičných časových ôs, ktoré sa zameriavajú výhradne na časové známky súborového systému je napríklad chýbajúci kontext. Tento kontext môže byť zapísaný v log súboroch alebo je ho možné nájst v metadátach súborov. Dalšou nevýhodou môže byť frekvencia s akou sa časové známky v súborom systéme menia. Ako príklad môžu byť časové známky poslednej zmeny súborov. Takéto známky sa menia s veľkou frekvenciou a zmenu tejto informácie môže sposobiť aj antivírus, ktorý pristupuje k súborom pri hľadaní malwaru. Potom hodnota týchto informácií s týmito časovými známkami degraduje. Dalším problémom pri tradičných časových osiach môžu byť anti-forenzné nástroje, ktorých cieľom je upraviť informácie a metadáta tak, aby nebolo možné vytvoriť vierohodné dôkazy. Jedným z týchto nástrojov je *Timestomp*[14], ktorý má za cieľ pozmeniť časové známky súborov a tým narušiť forenznú analýzu.

Plaso za týmto účelom vytvára super časovú os. Super časová os je rozšírená o informácie z viacerých zdrojov aby poskytla širší pohľad na udalosti a zároveň obmedzila využitie anti-forenzných metód. Časová os nielen, že obsahuje tradičné informácie zo súborového systému ale je v tomto prípade obohatená o záznamy z log súborov, ktoré sú v systéme, metadáta dokumentov, windows registry súbory. Tieto záznamy poskytujú väčší kontext

k udalostiam, ktoré sa v systéme udiali a môžu urýchliť celý proces vyšetrovania. Pomerne problematickou častou tvorby super časovej osi je fakt, že informácie sú získavané z vnútra rôznych zdrojov, ktoré majú svôj vlastný špecifický formát. Tento formát môže byť ako textový tak binárny a pre každý formát je potrebná samostatná logika. Ďalšou nevýhodou tohto formátu časovej osi je jej komplexnosť. Časová os síce poskytuje väčší kontext a viacej informácií o udalostiach ale s tým rastie aj komplexnosť skúmaných dát a ich vizualizácia.



Obr. 2.2: Vizualizácia super časovej osi vytvorenej plaso nástrojom v Timeline Explorer vizualizačnom nástroji pre CSV formát. Výstup bol prevedený pomocou *psort* do požadovaného formatu.

2.2 Architektúra

Plaso architektúra je postavená na využití knižnice SleuthKit. Táto knižnica je napísana v jazyku C a poskytuje nízko úrovňoví prístup k suborovým systémom a uložiskám. Plaso ho využíva na extrakciu jednotlivých súborov zo vstupu a prístupu k dátam, ktoré sú použité v procese extrakcie. Plaso využíva výhody modulárnej architekúry aby docielilo efektívnú tvorbu super časovej osi. Táto architektúra bola zvolená pre potreby využiť veľke množstvo modulov, ktoré zvládnu spracovať jednotlivé formáty dát uložených v súboroch, z ktorých sú získavané udalosti. Užívateľ môže rozšíriť funkcionalitu pomocou nových extraktorov alebo nových rozširujúich modulov pre existujúce extraktory. Celková architektúra plasa sa dá rozdeliť na základe operácií, ktoré sú v jednotlivých krokoch extrakcie vykonávané. Operácie je možné rozdeliť takto:

1. **Predspracovanie** – Zozbieranie informácií zo súboru/mount pointu/obrazu disku, ktoré môžu byť použité pre obohatenie procesu extrakcie a kolekcie.

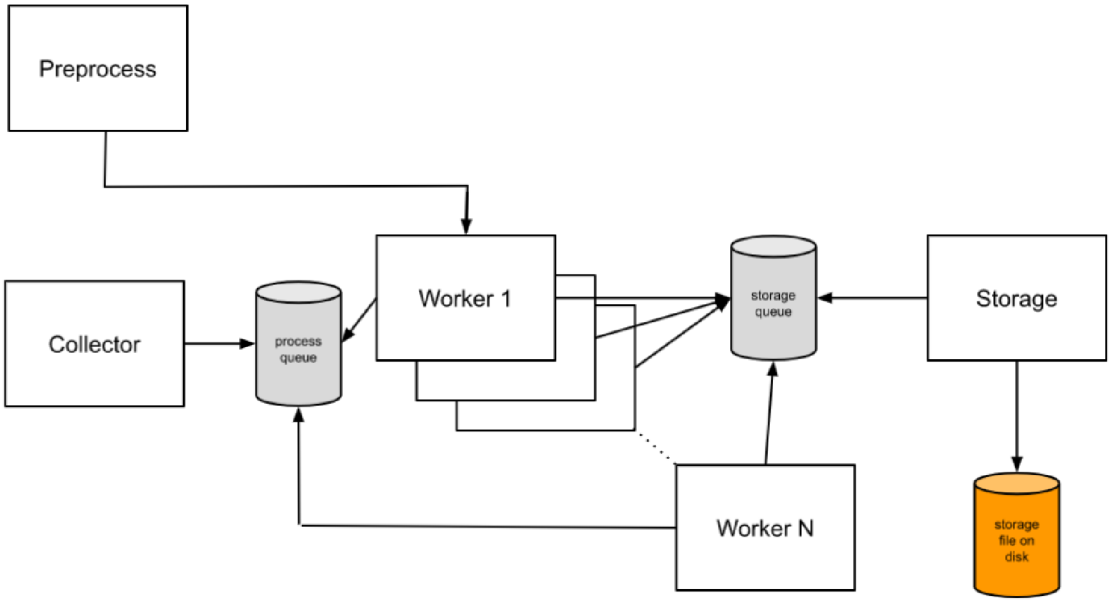
2. **Kolekcia a extrakcia** – Proces kolekcie spočíva v postupnom prechádzaní obrazu alebo mount pointu a nájdení všetkých súborov, ktoré je potrebné spracovať. Extrakcia je samotný proces, pri ktorom je súbor otvorený a spracovaný. V tomto bode sú využité jednotlivé extraktory pre extrahovanie udalostí.

3. **Analýza** – Operácie analýzy postupne iterujú cez extrahované udalosti a vytvárajú značky nad udalosťami, ktoré môžu byť dôležité.

4. **Uložisko** – Stará sa o čítanie udalostí z fronty, naplnenie vyrovnávajúcej pamäti a následným zápisom na disk.

Jednotlivé operácie môžu prebiehať ako v paralelnom spracovaní, s využitím viacerých procesov, tak aj v sekvenčnom spracovaní, kedy operácie vykonáva postupne jeden proces. Pri využití jedného procesu je poradie operácie samozrejmé a operácie prebiehajú v poradí akom sú vymenované vyššie. V prípade využitie viacerých procesov a paralelného spracovanie je poradie operácií upravené tak aby bolo možné vykonávať paralelne operácie.

Na obrazku 2.3 je znázornená architektúra nástroja.



Obr. 2.3: Architektúra základných operácií nástroj Plaso a ich vzájomné prepojenie. Šípky znázorňujú smer udalostí/dát do a z front využívaných v systéme (prevzaté a upravené z [3]).

Na začiatku celého procesu je využité predspracovanie, ktoré získa dodatočné informácie o spracovávanom vstupnom súbore a poskytne tieto informácie jednotlivým procesom. Tieto pracovné procesy (z anglického worker) následne pristupujú k fronte vstupných procesov. Do tejto fronty zapisuje proces kolekcie, ktorý postupne prechádza vstupný súbor (na vstupe môže byť súbor, obraz disku alebo mount point) a hľadá súbory, ktoré je potrebné spracovať. Jednotlivý pracovné procesy následne využívajú dostupné extraktory pre spracovanie vstupu. Vyextrahované udalosti sú zapísané do fronty uložiska, z ktorej operácie uložiška zapisujú udalosti na disk.

V nasledujúcej časti sú rozobrané jednotlivé časti do väčších detailov ako aj ich detailnejšia architektúra znázornená UML diagramami. Popis je zameraný hlavne na časti, ktoré predstavujú hlavnú časť systému, ktorý slúži na extrakciu udalostí. Tieto časti budú ďalej v práci využité pri návrhu nového systému.

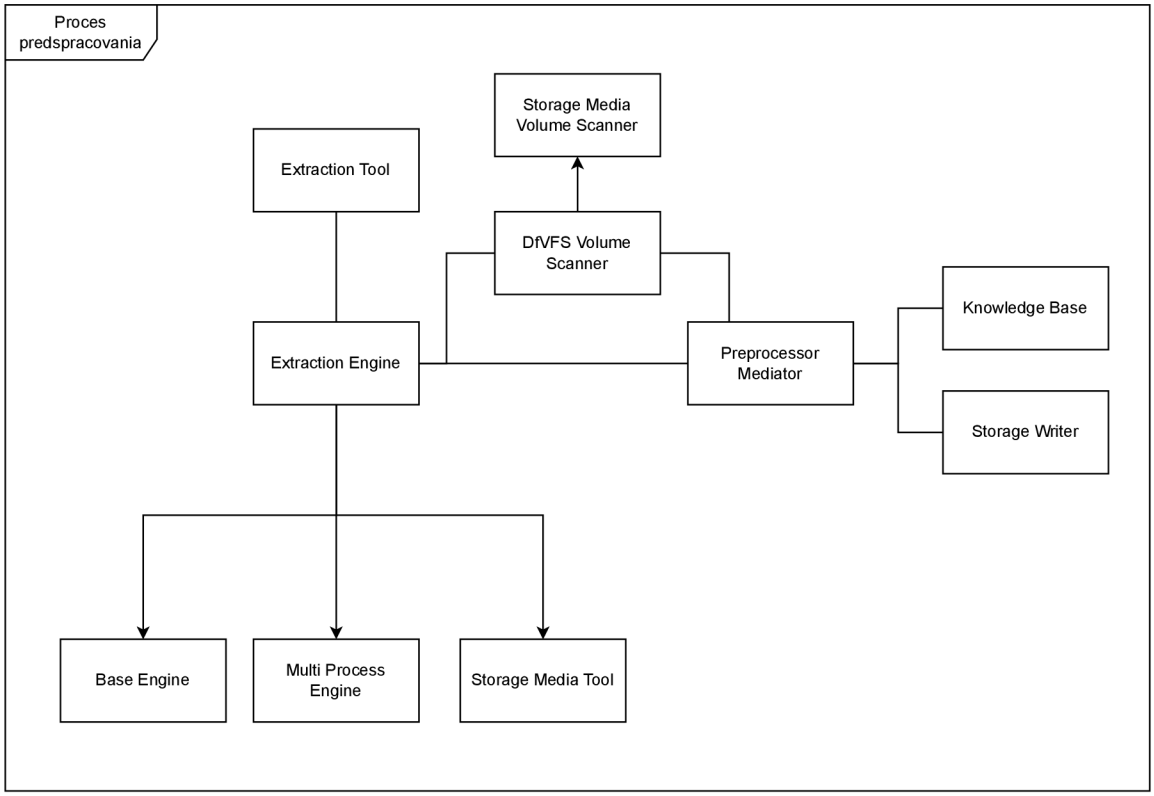
2.2.1 Predspracovanie

Jedná sa o operáciu, ktorú plaso vykonáva ešte pred zahájením celého procesu extrakcie. V tomto kroku sú z cieľovej cesty získavané informácie, ktoré neskôr v ďalších krokoch obohacujú jednotlivé operácie. Účelom predspracovania je preskúmať vstupné dáta a vyhodnotiť operačný systém, ktorý vstup používa a špecifické informácie, ktoré môžu byť použité pre zlepšie následovných operácií. Príkladom informácií, ktoré sa snaží táto operácia získať sú napríklad:

- Meno hostiteľa (z anglického hostname), ktoré obsahujú vstupné dáta. V tomto prípade môže byť vstupom napríklad obraz disku s operačným systémom.
- Informácie o časových zónach a jazyku systému.
- Lokálne užívateľské účty. V užívateľských účtoch sú zahrnuté aj informácie ako užívateľské meno a cesta v súborovom systéme k ich domovskému priečinku.
- Dalšie informácie, ktoré môžu zlepšiť výsledok ako lokálne premenné, reťazce z Windows denníka aktivít (z anglického Windows Event log)

Získané informácie sú počas tohto procesu ukladané do interného objektu, ktorý je nazývaný `báza znalostí`. Tento objekt je počas celej extrakcie dostupný všetkým komponentám a operáciam. Slúži na obohatenie kolekcie udalostí, zlepšenie extrakcie a celkového výsledky po analýze udalostí. V systéme sú vytvorené objekty mediátorov, ktoré slúžia na sprístupnenie bázy znalostí jednotlivým operáciam. Pri predspracovaní je k dispozícii mediátor predspracovania, ktorý poskytuje prístup k báze znalostí a zároveň umožnuje komunikovať s uložiskom.

Triedy, ktoré sa podielajú na tomto procese je možné vidieť na vytvorenom obrázku 2.4 spolu s ich vzájomnými závislostami.



Obr. 2.4: Diagram tried pre operáciu predspracovania a triedy, ktoré ho riadia.

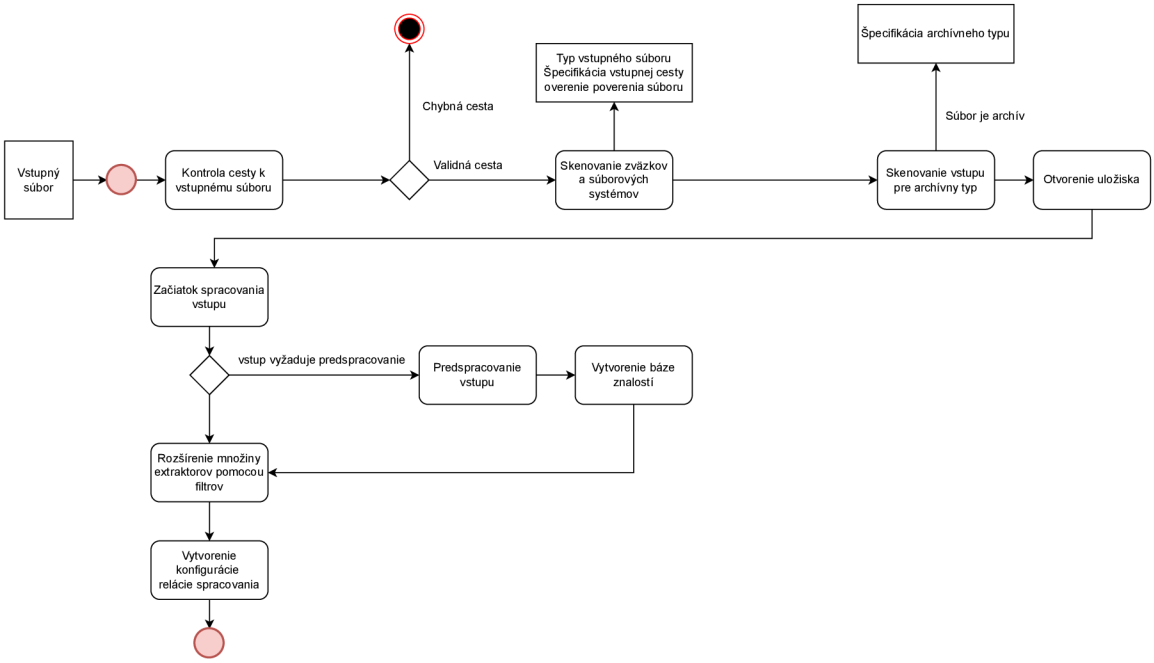
Jednotlivé triedy z obrázku 2.4 je možné popísať nasledovným spôsobom:

- **Nástroj extrakcie** (extraction tool) – Trieda predstavuje celý nástroj plaso, ktorý má na starosti hlavné spustenie engine pre extrakciu. Slúži ako vstupný bod všetkých operácií.
- **Engine extrakcie** (extraction engine) – Engine, ktorý riadi extrakciu od jej začiatku. V predspracovaní slúži na vytvorenie DfVFS skeneru zväzkov (DfVFS volume scanner). Engine môže fungovať ako jeden proces (Base engine) alebo viacero procesov (Multi process engine). Jeho schopnosti sú rozšírené pomocou nástroja pre uložisko (Storage media tool), ktoré slúžia práve pre ovládanie skeneru zväzkov.
- **DfVFS skener zdrojov** (DfVFS source scanner) – Hlavná zložka predspracovania je skener z *DfVFS* knižnicu. Táto knižnica implementuje všetky potrebné operácie pre získanie informácií zo vstupných dát, ktoré tvoria bázu znalostí. Má prístup k mediátoru, ktorý poskytuje prístup k spomínanej báze znalostí a k operáciam s uložiskom.
- **Mediátor predspracovania** (Preprocessor mediator) – Pomocou tejto triedy sú zapisované informácie do bázy znalostí. Taktiež poskytuje prístup k uložisku celého systému.
- **Uložisko** (Storage writer) – Implementuje operácie nad uložiskom pre účely zápisu nových udalostí, dát o exktracii a pod.

Celá operácia predspracovania prebieha v jednoduchých krokoch. Následujúce kroky sa vykonajú vždy na začiatky pred samotným procesom extrakcie:

1. Overenie platnosti zdrojovej cesty k vstupným dátam
2. Overenie platnosti výstupnej cesty pre extrahované dáta
3. Zistenie výskytu zväzkov alebo súborových systémov vo vstupných dátach
4. Ak vstupné dáta predstavujú archív je potrebné dodatočné skenovanie zdroja a zistenie špecifikácií pre súbory v archíve
5. Otvorenie vstupných dát pomocou triedy zapisovateľa
6. Ak vstupné dáta potrebujú predspracovať tak:
   (a) Predspracovanie vstupnych dát za pomoci mediátora, enginu extrakcie a skenera zväzkov
   (b) Vytvorenie bázy znalostí zo vstupného súboru

Tieto kroky je možné vidieť zakreslené v diagrame aktivít 2.5 pre lepšiu predstavivosť.



Obr. 2.5: Diagram aktivít znázorňujúci proces predspracoavnia v Plaso nástroji.

Po tejto fáze je vytvorená báza znalostí. Súbory zo vstupu, ktoré potrebovali predspracovanie, prešli procesom predspracovania.

2.2.2 Kolekcia a extrakcia

Kolekcia a extrakcia predstavujú hlavnú fázu celého Plaso procesu. Táto fáza prebieha v dvoch operáciach, ktoré sa navzájom dopĺňajú.

Kolekcia

Operácia kolekcia postupne prechádza vstupné súbory. Využíva `DfVFS skener zdrojov` pre určenie, či vstupné dáta predstavjú súbor, priečinok alebo obraz disku. Ak je potrebné získať dodatočné informácie o oddielu alebo tieňovom snímok (z anglického Volume shadow snapshot, tak je táto informácia získaná od užívateľa. Po získaní potrebných informácií sú vyhľadávané všetky súbory, ktoré majú byť využité pre extrakciu udalostí v dalšej fáze. Pre vyhľadávanie súborov plaso používá dva rôzne spôsoby, ktoré je možné zvoliť:

1. **Kitchen Sink** prístup – Jedná sa rekurzívny prístup, ktorý prechádza obraz disku alebo priečinku a zozbiera dáta z každého súboru, ktorý nájde. Tento prístup je vhodný, ak užívateľ nemá presnú predstavu aké súbory hľadá. Jeho nevýhoda spočíva v pomerne dlhom prehľadávaní kedže sa jedna o rekurzívne prehľadávanie všetkých súborov.

2. **Cieľený** prístup – Narozdiel od **Kitchen Sink** prístupu neprechádza všetky súbory. Využíva filter kolekcie, ktorý užívateľ zadá pri vstupe. Tento filter vymedzuje, z akých súborov sa majú zozbieravať dáta a ktoré súbory sa majú vynechať pri kolekcii. Tento prístup je vhodný, ak užívateľ dopredu vie, aké súbory chce zo vstupných dát využiť.

Filtre kolekcie

Ak užívateľ dopredu vie, aké súbory chce pri kolekcii zacieliť môže využiť filtre, ktorými vymedzí cieľené súbory. Plaso podporuje viaceré formáty filtrov:

1. **Definície forenzných artefaktov** – Projekt predstavuje všeobecný spôsob definície filtrov pre kolekciu vo forenznej analýze nielen pre Plaso projekt. Je využívaný vo viacerých forenzných nástrojoch. Viac informácií je možné získať z [2].

2. **Filtre definované súbormi** – Pôvodne riešenie filtrov, ktoré boli čisto textové súbory bolo nedostačujúce a preto bol dodatočne využitý YAML formát. YAML je odporúčaný formát samotnými tvorcami Plaso nástroja. Oba formáty definujú cesty k priečinkom, ktoré majú byť zarhnuté alebo vylúčené pri kolekcii.

   (a) Textový súbor filtrov
   (b) YAML súbor

Pri fáze kolekcie sú spracované archívy ako *.zip*, *.tar* a *.zip* a taktiež sa spracujú súbory z komprikovaných prúdov (z anglického compressed stream) ako napríklad *.log.gz*.

Extrakcia

Po získaní súborov zo vstupných dát nasleduje druhý krok tejto fázy. Extrakcia predstavuje hlavnú operáciu nástroja. Pri tejto operácií sa využívajú nájdené súbory z kolekcie a extrahujú sa z nich udalosti, ktoré sú zapísané v súboroch. Plaso využíva pri extrakcie dve komponenty:

1. **Analyzátor** – Slúži na výpočet integritného hash-u (napríklad SHA256) alebo oskenovanie obsahu súboru za pomoci YARA pravidiel. Na základe získaných vlasností analyzátor rozhodne akým spôsobom sa budú extrahovať udalosti zo súboru.

2. **Extraktor** – Slúži na extrakciu udalostí z konkrétneho formátu súboru. Jedná sa o hlavnú komponentu pri získavaní udalostí zo súboru. Celý zoznam extraktorov je v prílohe A

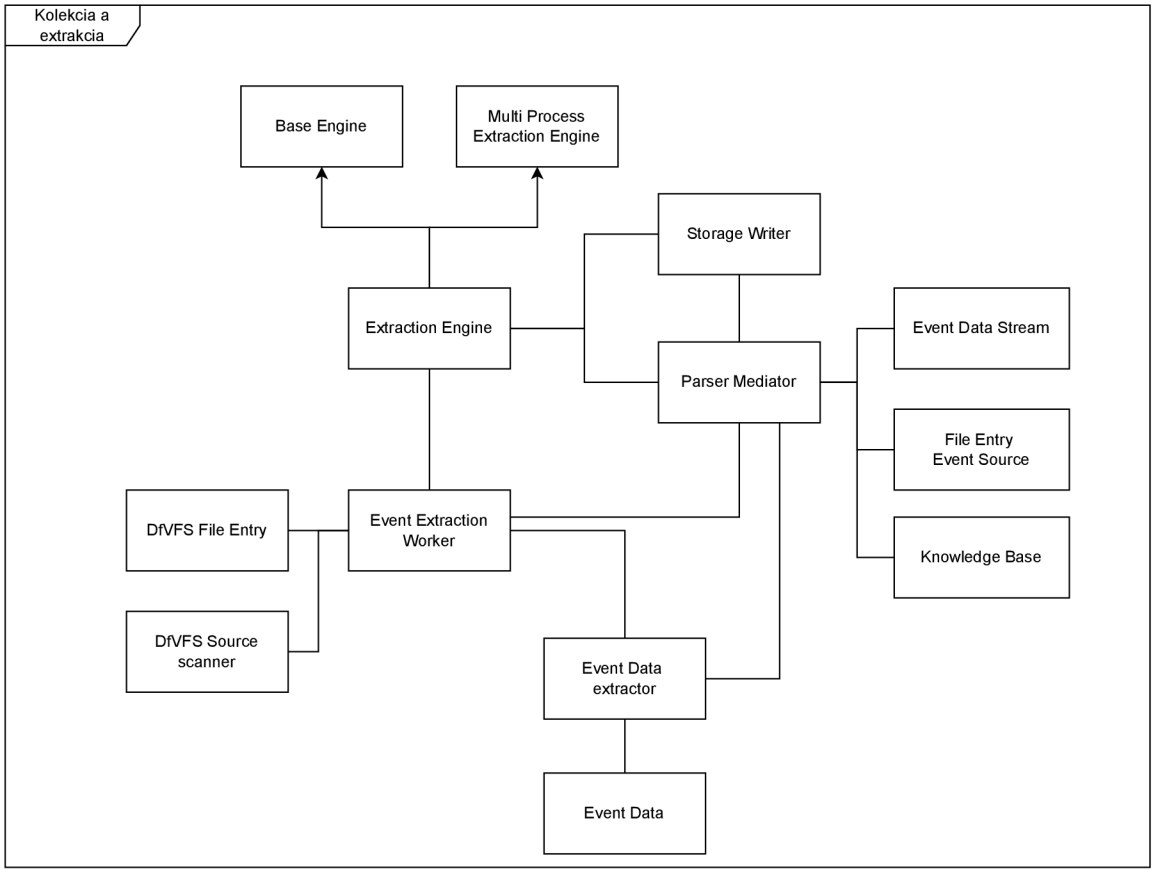
Celý spôsob extracie prebieha v dvoch krokoch. Najskôr sa využije analyzátor, ktorý rozhodne o aký druh spracovavaného súboru sa jedná. Ak vypočítaný hash alebo obsah súboru odpovedá známemu formátu, je vybraný konkrétny extrator, ktorý bude extrahovať udalosti. Samozrejme, môže nastať prípad, kedy hash ani obsah nezodpovedá žiadnemu konkrétnemu extraktoru. V tomto prípade je zvolená preddefinovaná množina všeobecných extraktorov, ktorá bude aplikovaná na súbor. Táto množina je zobrazená v tabulke 2.1.

CuppIpp	BodyFile	ChromeCache	BSM
FishHistory	Java IDX	PLS Recall	Recycler bin
UTMP	Recycler bin info 2	Android app usage	Chrome preferences
Winjob	Opera typed history	Opera global history	Bencode
Text	McAffee	Network Minner	Symantec
Trendmicro vd	JSON-L	PList	Trendmicro url
Firefox Cache	Win Restore	Firefox Cache 2	CZip

Tabuľka 2.1: Všeobecná množina extraktorov, ktorá je použitá v prípade nenájdenia zhody analyzátorom.

Po analyzátore nasleduje práca jednotlivých extraktorov. V prípade, že analyzátor našiel zhodu, je spustený konkrétny extraktor na spracovávaný súbor. Tento extrator postupne prechádza dátami v súbore a aplikuje na dáta implementovanú logiku ktorá vytvára z dát udalosti. Logika, ktorá je implementovaná v extraktoroc predstavuje spracovanie konkrétneho formátu v ktorom je súbor zapísaný. Ak analyzátor nenašiel vhodný extraktor je použitá všeobecná množina.

Nie všetky extraktory z tejto množiny vedia spracovať daný súbor. Preto je tento prípad v Plaso nástroji riešený tak, že sa použijú všetky exktraktory bez ohladu na to, či je daný extraktor schopný súbor spracovať. V prípade, že extraktor nepozná formát súboru, tak extraktor vyhlási chybu (python výnimku) a nepokračuje ďalej. Môžu nastať aj prípady, kedy súbor nebude spracovaný ani jedným extraktorom z tejto množiny. V takom prípade nie je zo súboru vyextrahovaná žiadna udalosť a extrakcia končí.



Obr. 2.6: Diagram tried pre fázu kolekcie a extrakcie zobrazujúci vztahy medzi jednotlivými triedami.

Na obrázku 2.6 je vidieť jednotlivé triedy, ktoré sú zodpovedné za proces extrakcie a kolekcie. Opäť hlavnou triedou, ktorá riadi celý proces je engine extrakcie ako tomu bolo pri fáze predspracovania na obrázku 2.4. Jednotlivé triedy kolekcie a extrakcie vykonávajú následovné činnosti:

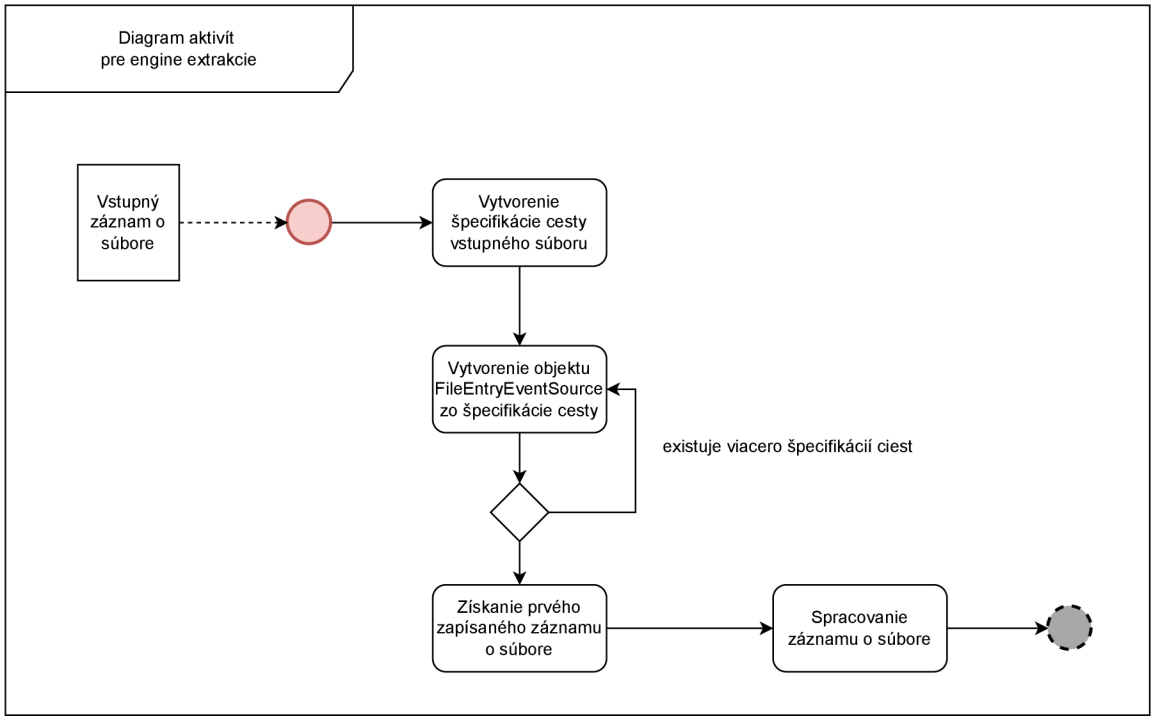
- **Engine extrakcie** (Extraction engine) – Na začiatku fázy spracuje vstupnú cestu k dátam a pomocou mediátora vytvorí počiatočné záznamy o vstupných súboroch (Source file entry event). Následné nastaví potrebné parametre pre extrakciu a kolekciu a vyžiada prvý záznam vstupného súboru pomocou triedy uloženia (storage writer). Riadi jednotlivé pracovné procesy extrakcie (event extraction worker), kde prideľuje procesom záznamy, ktoré sa nachádzajú v uložisku. Engine poskytuje informácie o progrese mediátoru extrakcie (Parser mediator). Tieto informácie sú využité pre informovanie užívateľa o jednotlivých krokoch.
- **Uložisko** (Storage writer) – Trieda, ktorá implementuje interné operácie uloženia pre uloženie vyextrahovaných udalostí, záznamov o vstupných súboroch, záznamov o prúdoch dát zo súborov (z anglického event data stream) a varovaniach vzniknutých počas extrakcie. Táto trieda je rozobratá viac podrobne v kapitole 2.2.4.
- **Mediátor** (Parser mediator) – V tejto fáze je využívaný mediátor na zapisovanie nových záznamov o vstupných súboroch, ktoré musia byť spracované. Ďalej je využívaný na zápis novo vzniknutých vyextrahovaných udalostí do uloženia a zozbieravá

informácie o progrese celého procesu. Informácie o vývoji sú následne zobrazované užívateľovi a poskytujú mu náhľad do vnútra extrakcie a kolekcie.

- **Záznam dátového toku** (Event Data Stream) – Predstavuje záznam o dátovom toku zo súboru. V súbore môže byť viacero dátových tokov a táto udalosť zaznamenáva povôd následne vyextrahovaných udalostí.
- **Záznam o vstupnom súbore** (File Entry Event Source) – Predstavuje záznam o súbore, ktorý je zdrojom vyextrahovaných udalostí.
- **Proces extrakcie udalostí** (Event Extraction Worker) – Zodpovedá za spracovanie záznamov o vstupných súboroch (File Entry Event Source). Zo záznamu o vstupnom súbore získa potrebné informácie pre otvorenie tohto súboru a následne analyzuje súbor a extrahuje metadáta z daného súboru. Vykonáva kontrolu, či daný súbor nepredstavuje uložisko alebo archív a spúšta extrakciu dátového toku pomocou extraktoru (Event data extractor).
- **Extraktor udalostí** (Event Data Extractor) – Riadi proces extrakcie udalostí z dátového toku. V prípade, že spracovávaný súbor neobsahuje žiadny dátový tok je potrebné spracovať iba metadáta daného súboru. K svojej činnosť využíva extraktory udalostí a dodatočne vytvára záznam dátového toku, z ktorého vyextrahované udalosti pochádzajú.
- **Dáta udalosti** (Event data) – Dáta o udalosti, ktoré boli vyextrahované zo súboru pomocou extraktoru.
- **Záznam o súbore** (DfVFS File Entry) – Trieda reprezentuje jednotlivé nájdené súbory v prehľadávaných vstupných dátach. Obsahuje informácie o ceste k danému súboru, jeho formát a metadáta. Jedná sa o internú reprezentáciu súboru nájdeného pri kolekcii. Využíva implementáciu z knižnice DfVFS.
- **Skener zdrojov** (DfVFS Source Scanner) – Trieda, ktorá je využíta už pri fázy predspracovania a jej využitie je popísané v kapitole 2.4.

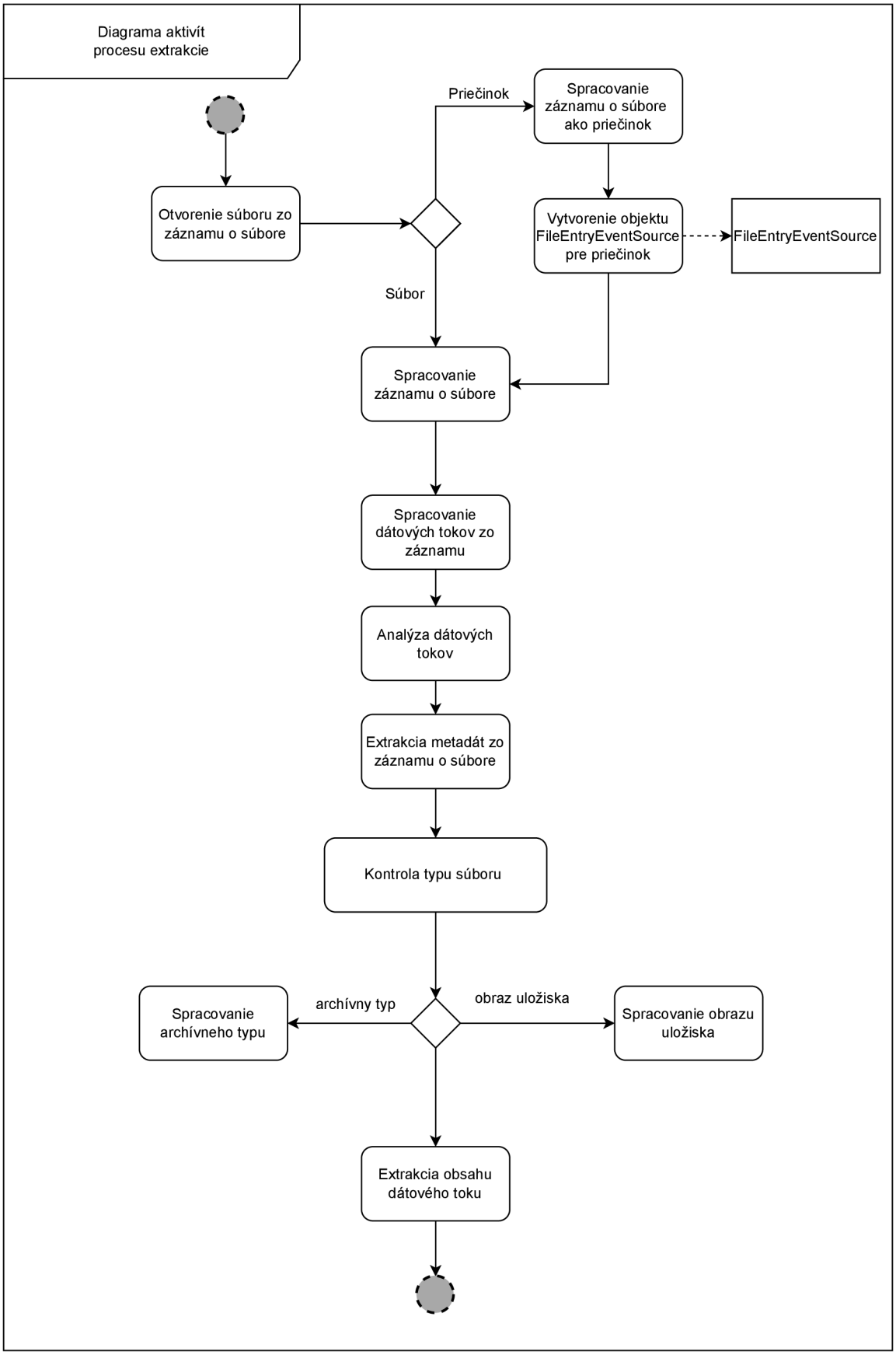
V nasledujúcej časti sú zobrazené diagramy aktivít tejto fázy. Kedže celý diagram aktivít nie je možné zobraziť v jednom obrázku sú tieto diagrami rozdelené do logických častí kvôli prehladnosti.

Prvou častou, obrázok 2.7, je diagram aktivít zobrazujúci kroky, ktoré vykonáva engine extrakcie. Engine extrakcie má na vstupe cestu k vstupným súborom. Spracuje cestu k vstupným súborom a pripraví dáta pre vytvorenie `záznamu o vstupnom súbore`. Ak je na vstupe viacero ciest, iteruje cez tieto vstupné cesty a vytvára dalšie záznamy. O vytvorení týchto záznamov je informovaný `mediátor`, ktorý ich ukladá do `uložiska`. Následne vyberie prvý záznam z fronty, ktorý bol uložený a pripraví udalosť pre `proces extrakcie`.



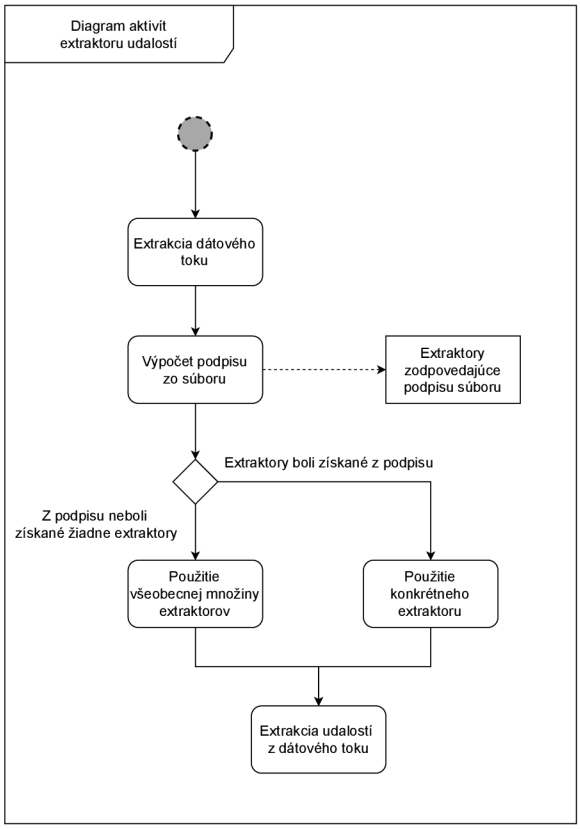
Obr. 2.7: Diagram aktivít enginu extrakcie pri kolekcii a extrakcii.

Po spracovaní vstupných dát `enginom extrakcie` následuje spracovanie záznamu vstupného súboru procesom extrakcie, obrázok 2.8. Engine extrakcie zašle prvú udalosť procesu extrakcie a predá tejto komponente riadenie. Na začiatku je otvorený vstupný súbor, ktorý je reprezentovaný triedou `FileEntry`. Ak súbor predstavuje priečinok, je ho potrebné prehľadať a vytvoriť nové záznamy o vstupných súboroch reprezentované triedou `FileEntryEventSource`. Po nájdení všetkých súborov sa pokračuje v spracovaní. Otvorí sa dátový tok súboru a začne jeho analýza. Ak súbor neobsahuje dátový tok, ukončí sa extrakcia a spracujú sa len metadáta o súbore. Ak súbor obsahuje dátový tok, vyextrahujú sa metadáta o toku a overí sa či zdroj dátového toku nie je obraz nosiča dát alebo archív. Ak sa jedná o takýto typ súboru, archiv alebo obraz sa otvorí a vytvoria sa nové záznamy o vstupných súboroch, ktoré sa uložia do fronty a pokračuje sa dalším záznamom o vstupnom súbore. V prípade, že dátový tok neobsahuje archív alebo obraz je dátový tok zaslaný do `extraktoru udalostí`.



Obr. 2.8: Diagram aktivít procesu extrakcie

Dátový tok zo súboru je otvorený, spracovaný, analyzovaný a pripravený na extrakciu udalostí zo súboru. Obrázok 2.9 zobrazuje diagram aktivít spracovania dátového toku. Vypočíta sa podpis súboru, na základe ktorého sa vytvorí zoznam extraktorov, ktoré sú schopné spracovať daný súbor s dátovým tokom. Ak podpis súboru nezodpovedá žiadnemu zo známych extraktorov, použije sa všeobecná množina extraktorov. Po získaní extraktorov sú postupne extraktory spustené nad dátovým tokom spracovávaného súboru. Jednotlivé extraktory produkujú z dátové toku `dáta udalostí`.



Obr. 2.9: Diagram aktivít extraktoru udalostí

Po skončení fáze extrakcie sú zo súborov vyextrahované všetky dáta o udalostiach a uložené uložisku systému. Posledným volitelným krokom je analýza vyextrahovaných udalostí, v ktorej sú označené udalosti.

2.2.3 Analýza

Fáza v ktorej sa postupne iteruje cez vyextrahované udalosti. Táto fáza slúži hlavne pre účely vizualizácie a zlepšenie analýzy výslednej super časovej osi. Kedže super časová os je pomerne komplexná Plaso sa snaži označiť udalosti, ktoré môžu byť dôležité alebo zaujimavé pre vyšetrovanie. Udalosti je možné označiť manuálne, ale tento proces môže byť veľmi náročný vzhľadom na množstvo vyextrahovaných udalostí. Jeden zo spôsobov akým je možné túto činnosť urýchliť je vytvorenie filtrov alebo množiny pravdiel podľa ktorých Plaso automaticky označí vytvorené udalosti. Tieto pravidlá môžu byť vytvorené napríklad na základe kľúčových slov, formátu súboru a podobných vlastností. Užívateľ môže naprík-

lad vytvoriť filter pre udalosti, ktoré boli vygenerované pod určitým užívateľským menom v systéme.

Plaso taktiež obsahuje rozširujúce moduly, ktoré sa snažia automaticky označiť udalosti. Príklad niektorých modulov, ktoré sú zahrnuté v Plaso:

- **Modul pre unikátne navštívené domény** – Modul extrahuje domény z histórie prehliadačov. Zoznam domén môže byť použitý pre rýchle porovnanie napríklad so zoznamom známych pishingových zoznamom.
- **Modul VirusTotal** – Využíva VirusTotal API pre kontrolu hash hodnoty udalosti a porovnanie s databázou vírusov.
- **Modul Chrome extension** – Analyzuje udalosti z rozšírení Chrome prehliadača a označuje udalosti spojené s vybraným rozšírením

2.2.4 Uložisko

Plaso využíva pre ukladanie a čítanie vytvorených udalostí a záznamov interné uložisko. Toto uložisko je prístupné počas celého procesu cez triedu `StorageWriter`. Táto trieda implementuje potrebné operácie pre Plaso komponenty a poskytuje systému rozhranie pre ovládanie uložiska. Komponenta uložiska sa stará o ukladanie metadát zo súborov, označovanie a zhlukovanie informácií a k uloženiu vyextrahovaných udalostí na disk.

Plaso v čase písania obsahuje dve možnosti uložiska, ktoré je možné použiť a poskytuje k nim API rozhranie:

- **Redis** – Rýchle In-memory key-value uložisko. Toto uložisko nie je plne použiteľne. V čase písania je v Plaso implementovaná len časť pre zápis záznamov a neobsahuje operácie pre čítanie z uložiska.
- **SQLite** – Klasické relačné uložisko. Do SQL súboru sú ukladané záznamy o súboroch, metadáta o extrakcii, udalostiach a informácie o analýze.

Predvolené uložisko v Plaso je SQLite. Všetky udalosti sú uložené v relačnej databáze a Plaso implementuje nad týmto uložiskom operácie ako získanie poslednej pridanej udalosti a získanie dalšej udalosti v poradí. Pre tieto operácie si rozhranie uložiska ukladá interné indexy jednotlivých pridaných udalostí kedže SQLite natívne nepodporuje tieto operácie.

2.3 Distribucia Plaso nástroja

Plaso sa využíva v spojení s Elasticsearch a Kibana nástrojov. Kedže plaso rieši vo svojej implementácií distribúciu medzi jednotlivé pracovné procesy, tak elasticsearch a kibana slúžia pre zlepšenie forenznej analýzy nad vytvorenou super časovou osou. Plaso nástroj je v tomto spojení využitý na extrakcie a spracovanie vstupných dát. Autori distribuujú výpočet plasa tak ako je to popísané v kapitole 2.2. Teda tak, že využívajú viaceré pracovné procesy, ktoré spracovávajú vstupné dáta. Réžia týchto procesov je riešená autormi plasa. Elasticsearch spolu s Kibana nástrojom využívajú výstup Plasa a poskytujú funkcie pre analýzu extrahovaných udalostí.

Pomocou *psort* nástroje ja *plaso* formát transformovaný do formátu, ktorý je možné importovať do Elasticsearch. Jeden z podporovaných súborov je napríklad L2tcsv. Import transformovaných udalostí je zabezpečený nástrojom Logstash. Logstash je nástroj, ktorý

vytvára zretazené operácie (z anglického pipeline) pre spracovanie dát, ktoré sú následne importované do Elasticsearch. Logstash importuje dáta z veľkej škály zdrojov, zahŕňajúcich textové súbory. Tieto dáta sú Logstash nástrojom zaslané do Elasticsearch, kde sú indexované a analyzované.

V momente, keď sú dáta v Elasticsearch je možné nad dátami vykonávať vyhľadávanie, analýzu a vizualizáciu pomocou vstavaných funkcií pre vyhľadávanie a analyzu. Vizualizácia dát je možná pomocou Kibana nástroja, ktorý je postavná nad Elasticsearch systémom.

Elasticsearch a Kibana dovoľujú nad vyextrahovanými udalosťami rýchly spôsob vyhľadávania aj nad takým množstvom udalostí ako vytvorí Plaso. Ďalej poskytujú vstavané funkcie, ktoré dovoľujú indetifikovať vzory, anomálie, vytvoriť vizualizácie pre lepšie pochopenie dát.

Kapitola 3

Distribuované výpočtové modely

Posledné roky je možné sledovať trend signifikantného nárastu objemu dát, ktorý je generovaný a zároveň zbieraný organizáciami. Zdrojom takéhoto veľkého objemu dát sú socialne média, senzory, systém v ktorých prebiehajú transakcie a podobné nástroje. Často sa takýto veľký objem dát nazýva big data. Big data prinášajú veľké výzvy v spracovaní, ukladaní a analýze. Kedže na objem týchto dát už nestačia bežné spôsoby spracovania bolo potrebné priniesť niečo čo bude schopné spracovať veľký objem efektívne. Big data boli motiváciou pre vývoj distribuovaných výpočtových modelov. Tieto modely dovoľujú paralelné spracovanie dát pomocou veľkého počtu počítačov. Tieto počítače (uzly) sú spojené pomocou internetu do jednej sieti, ktorá vytvára distribuovanú sieť. Distribuované sieťe pre výpočty dnes tvoria rádovo stovky počítačov, ktoré navzájom spolupracujú a rieši komplexné úlohy. Cieľom distribuovaných výpočtových modelov je zlepšiť rýchlosť a efektivitu pri spracovaní a analýze objemných dát. Tieto modely sú dizajnované tak aby boli schopné zvládnuť veľké dátove sady a komplexné výpočty. Často sú tieto modely využívané pre rôzne úlohy ako napríklad dávkové spracovanie, prúdové spracovanie alebo sú využité pri strojovom učení.

Spracovanie spočíva v rozložení výpočtových úloh medzi jednotlivé uzly v sieti čím sa docieli zrychlenie výpočtov. Jednolivé uzly komunikujú medzi sebou pomocou internetového pripojenia, vďaka čomu uzly nemusia byť na jednom mieste ale môžu byť rozmiestené po celom svete. Celkovo distribuované výpočtové modely ponúkajú efektívny a škálovateľný prístup k spracovaniu a analýze velkého objemu dát. Dajú sa rozdeliť do troch kategórií: prúdové spracovanie, dávkové spracovanie a hybridné spracovanie. Hybridné spracovanie ponúka ako dávkový spôsobo spracovania tak aj prúdový.

Medzi populárne distribuovane výpočtové modely patrí dnes Apache Spark, voľne dostupný engine pre spracovanie dát. Spark je navrhnutý tak aby bol flexibilný a rýchly a je používaný značným množstvom organizácií, ktoré ho využívajú na dávkové spracovanie, prúdové spracovanie a pri strojovom učení. Další populárnym modelom je Hadoop. Hadoop je taktiež voľne dostupný nástroj, ktorý slúži pre ukladanie a spracovanie veľký dátových sád. Jeho tvorcovia kladú dôraz na vysokú škálovateľnosť a chybovú toleranciu. Hadoop sa často využíva aj v spojení s Apache Spark. Tretím modelom, ktorý naberá v poslednej dobe na popularite je Apache Flink. Je využívana ná spracovanie dát v reálnom čase v aplikáciach, ktoré sú riadené udalosťami (z anglického event driven applications).

V kapitole je detailne rozobratý Spark výpočtový model. Je popísané na čo tento výpočtový model slúži, ako vznikol, spôsoby akým spark spracuje. Ďalej je popísaný jeho ekosystém, ktorý spark vytvára a jeho implementácie dôležitých častí. Detailne je rozobratá architektúra tohto frameworku a jeho spojenie s Apache Hadoop. V poslednej časti kapitoly je rozbor dalších technológii pre distribuované spracovanie a ich porovnanie.

3.1 Apache Spark

Apache spark predstavuje platformu pre spracovanie big data, ktorá poskytuje hybridný prístup. Ponúka možnosť dávkového aj prúdového spracovania. Spark využíva podobné princípy ako Apache Hadoop MapReduce engine. Oproti Hadoop je práve Spark nástroj, ktorý prekonáva rýchlosťou Hadoop. Ako ukazuje [18], Spark vo väčšine testovacích scenárov výrazne porazil Hadoop v rýchlosti spracovania. Je to vďaka In-memory spôsobu akým Spark vykonáva vypočty. Tento spôsob sa ukázal oveľa rýchejší ako klasické čítanie a zapisovanie na disk, ktoré používa Hadoop MapReduce. Spark je možné spustiť v samostatnom režime (z anglického standalone) alebo využiť Hadoop a nahradiť MapReduce engine Sparkom.

Hybridný spôsobom spracovania poskytuje v Spark dva rôzne režimy v akom je schopný pracovať:

- **Dávkový model spracovania** – Najväčšou výhodou Spark oproti MapReduce je jeho In-memory prístup. Spark využíva čítanie z disku len pri dvoch operácia a to pri načítani všetkých dát do pamäti a zápisom výsledných dát. Ostatné operácie vykonáva priamo v pamäti. Týmto je Spark schopný omnoho rýchlejšie vykonávať dávkové spracovanie oproti ostatným frameworkom ako je Hadoop. Spark pri spracovaní analyzuje jednotlivé operácie ktoré bude nad dátami vykonávať a je schopný ich optimalizovať tak aby boli vykonané v najoptimálnejšom poradí. Analýzu operácií vykonáva na základe orientovaných acyklických grafov (Directed Acyclic Graphs) alebo skrátene DAGs[19]. V DAGs sú zobrazené všetky operácie nad dátami a ich vzájomné vzťahy. Pre implementáciu in-memory dávkového spracovania Spark využíva dátový model, ktorý sa nazýva Resilient Distributed Datasets, skrátene RDDs. Tento model predstavuje nemennú štruktúru, ktorá existuje v pamäti a reprezentuje dáta. Operácie nad RDD vytvárajú nové RDD. Každé RDD obsahuje svôjho predchodcu až k svôjmu rodičovi. Posledným prvkom v histórii RDD predstavujú dáta priamo na disku.
- **Prúdový model spracovania** – Spark ponúka okrem dávkové spracovania aj prúdové. Kedže Spark bol navrhnutý pre pracovanie s dávkami a jeho dizajn je položený na dávkach je toto spracovanie prispôsobené tomuto dizajnu. Aby dosiahol prúdového spracovania, tak Spark využíva mikro-dávky. Jeho stratégia je spracovať prúd dát do veľmi malých dávok, čím sa snaží simulovať prúdové modely. Spark ukladá tieto mikro-dávky do vyrovnávacej pamäti a zasiela takto vytvorený malý dataset do dávkového spracovania. Aj na vzdory tomu, že toto spracovanie funguje veľmi efektívne, výkon sa môže líšiť od pravých prúdových modelov.

Vlasnosti

Spark má veľké množstvo vlastností. Ďalej sú vymenované niektoré zaujimavé vlastnosti, ktoré Spark spĺňa:

- **Rýchlosť** – Spark aplikácie je možné spustiť na Apache Hadoop clusteri. V Hadoop clusteri je nahradený MapReduce engine za Spark výpočtový model. Táto náhrada je hlavne z dôvodu, že Spark môže byť až 100 násobne rýchlejší ako Hadoop[18]. Toto zrychlenie je docielené hlavne vďaka využitiu rýchlej pamäti namiesto využitia zápisu a čítania dát z disku.
- **Použiteľnosť** – Pre vytvorenie Spark aplikácií je možné využiť známe programovacie nástroje ako Java, Scala, R alebo Python. Týmto poskytuje programátorom možnosť

využiť im známi programovací jazyk. Ďalej Spark API poskytuje veľmi jednoduchý prístup k vytvoreniu paralelných aplikácií. K dispozící je cez 80 vstavaných operácií[1].

- **Pokročilé analytické nástroje** – Okrem jednoduchých map a reduce operácií, Spark poskytuje SQL dotazy, streamovanie dát, grafové algoritmy a analytické nástroje pre strojové učenie. Všetky tieto operácie je možné skombinovať a tým vytvoriť zreťazené spracovanie (z anglického pipeline).

- **Univerzálnosť** – Spark je možné spustiť ako samostatne, tak v Hadoop clusteri, Apache Mesos alebo v cloude. Je schopný využiť rôzne dátové zdroje akými sú HDFS, Cassandra, HBase a podobné

- **In-memory vypočty** – Spark nevyužíva pre výpočty disk, ale všetky operácie prebiehajú in-memory. Vďaka tomu je schopný dosiahnuť veľmi veľkú rýchlosť. Dáta sú uložené do pamäti RAM na serveroch aby prístup k nim bol čo najrýchlejší.

Ekosystém

Spark vytvára celý ekosystém nástrojov, ktoré je možné použiť. Tento ekosystém tvorí 5 komponent spolu s hlavnou komponentou **Spark Core**. Na obrázku 3.1 je vidieť jednotlivé komponenty celého ekosystému. Jednotlivé komponenty rozširujú základnú funkcionalitu Sparku a prinášajú dalšie možnosti, kde je možné Spark využiť. Jednotlivé komponenty je možné popísať následovne[19]:

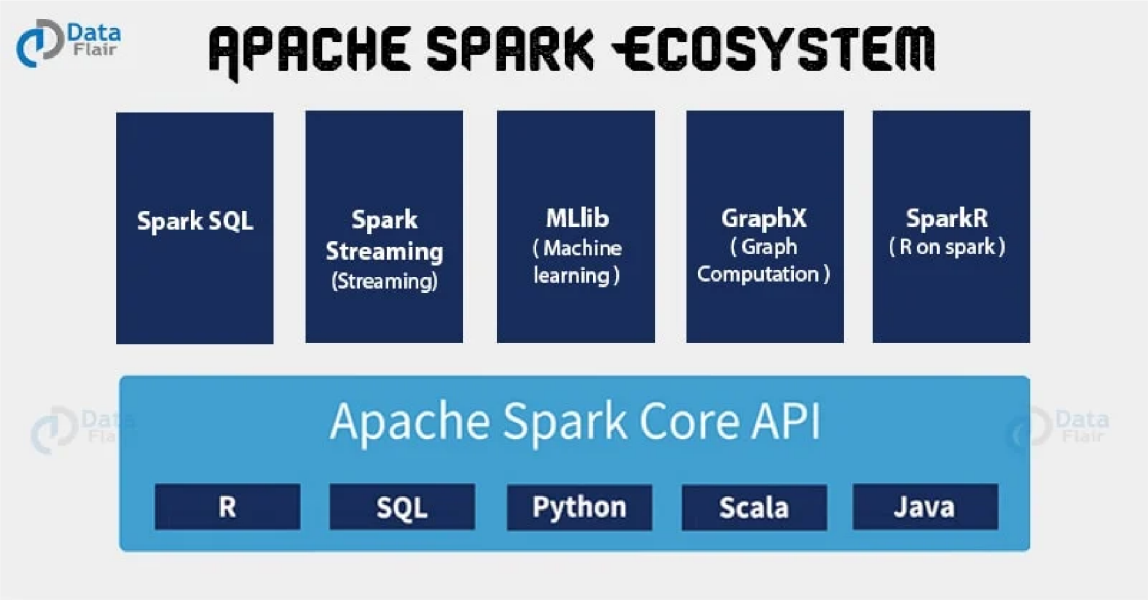
- **Spark Core** – Komponenta, ktorá implementuje hlavné vlastnosti Sparku. Poskytuje základnú funkcionalitu Sparku ako RDD, vstupno/výstupné operácie, správu pamäti, interakciu s uložiskom a DAGs. Taktiež poskytuje API, ktorá sa dá využiť s viacerými programovacími jazykmi pre ovládanie funkcionalít Spark Core.

- **Spark SQL** – Predtým nazývané ako Shark. Spark SQL je distribuovaný framework, ktorý je schopný pracovať so štruktúrovanými a semi-štruktúrovanými dátami. Poskytuje možnosť tvorby dotazov ako v SQL a HQL. Taktiež je schopný využívať viaceré zdroje, z ktorých získava dáta. Týmito zdrojmi môžu byť Hive tabuľka, Parquet alebo JSON.

- **Spark Streaming** – Dovoľuje spracovanie veľkého prúdu dát v reálnom čase. Aby spark dokázal zvládnuť spracovanie prúdu dát v reálnom čase využíva k tomu svoju schopnosť rýchleho plánovania. Toto plánovanie spočíva vo vytváraní mikro dávok, ktoré ukladá do vyrovnávacej pamäti. Následné operácie, v Spark nazývane transformácie, sú aplikované na tieto mikro dávky, ktoré môžu byť získané zo živých dátových prúdov a dátový zdrojov ako Twitter, Apache Kafka alebo senzory IoT.

- **MLlib** – Komponenta poskytujúca vysoko kvalitné algoritmy s vysokou rýchlosťou, vďaka čomu poskytujú jednoduchý prístup k strojovému učeniu a jeho škálovaniu. Implementuje niekoľko algoritmov zo strojového učenia ako regresiu, klasifikáciu, zhlukovanie a taktiež poskytuje implementáciu lineálnej algebry. Taktiež poskytuje nízko úrovňové knižnice, ktoré obsahujú algoritmy pre gradient descent optimalizáciu. Nechýbajú ani operácie pre evaluáciu, trénovanie a načítanie dátový sád využitých pri trénovaní umelej inteligencie.

- **GraphX** – Knižnica, ktorá poskytuje paralelné grafové výpočty a manipuláciu s grafmi. Dovoľuje vykonávať operácie nad grafovo štruktúrovanými dátami. Poskytuje viaceré operácie a algoritmy, ktoré dovolujú manipuláciu s grafmi ako napríklad zhlukovanie, klasifikáciu, vyhľadávanie, hľadanie cesty a podobné.
- **SparkR** – Balíček pre programovací jazyk R, ktorý dovoľuje vedcom využiť silu Spark nástroj z R shell nástroja. Poskytuje vedcom nástroj, ktorý môžu ovládať za pomoci R jazyka a vykonávať analýzy nad veľkými dátovými sadami.



Obr. 3.1: Ekosystém Spark frameworku a jeho komponenty (prevzate z [7]).

Všetky komponenty Spark ekosystému je možné využiť spolu a voliteľne ich kombinovať čo poskytuje veľkú škálu využiteľnosti. Spark spolu s týmto ekosystémom nielen, že poskytuje nástroj na distribuované výpočty ale prináša rozšírenia do oblastí ako je umelá inteligencia, SQL spôsob práce s dátami, riešenie grafových úloh a mnoho dalšieho.

Resilient Distributed Datasets

Formálne Resilient Distributed Datasets (RDDs) predstavujú dátovú štruktúru, ktorá je iba na čítanie (z anglického read-only structure)[25]. RDDs môžu byť vytvorené len deterministickymi operáciami vykonaných na:

1. Dátach, ktoré sa nachádzajú na stabilnom uložisku. Toto môže predstavovať napríklad disk alebo distribuované uložisko.
2. Z iných RDDs. Ak na jedno RDD je použitá operácia vzniká ňou dalšie RDD.

Operácie vykonané nad RDD sú prezývané transformácie. Tento názov bol zvolený preto aby sa dalo rozlíšiť medzi inými operáciami nad RDDs. Príkladom transformácie môže byť napríklad `map` alebo `filter` operácia. Každé RDD obsahuje históriu svôjho vzniku. Táto história predstavuje akýsi rodokmeň daného RDD v ktorom je možné sa dostať postupne až k jeho počiatočnému RDD (rodič). Vzhľadom na túto informáciu, nie vždy je RDD reálne

v systéme. Môže sa vyskytovať len v podobe metadát, ktoré hovoria akými transformáciami vzniklo z rodičovského RDD a v čase potreby je ho možné vytvoriť. Táto vlastnosť je využitá pri zotaveniach pri chybe. Ak nastane chyba v priebehu výpočtu v akomkoľvek bode, vďaka týmto metadátam je možné vytvoriť RDD v ktorom nastala chyba bez nutnosti vykonávať a ukladať všetky ostatné medzivýpočty.

Spark sprístupňuje RDD cez API v podporovaných jazykoch. Každá dátová sada je reprezentovaná objektom a transformácie, ktoré sa na ňom vykonávajú predstavujú metódy daného objektu. Užívateľ na začiatku vytvorí jedno alebo viacero RDD zo stabilného dátového uložiska. Následne pomocou *transformácií* (napríklad map alebo filter) vytvára ďalšie RDDs. Po vykonaní potrebných transformácií nad dátovou sadou sú použité *operácie*. Tieto operácie vracajú hodnoty aplikácií alebo ich exportujú do uložiska systému. Príkladom takýchto operácií môže byť *collect* (vracia jednotlivé elementy RDD) alebo *count* (vracia počet elementov v dátovej sade). Základú množinu *transformácií* a *operácií* je možné vidieť na obrázku 3.2.

Ďalej užívateľ je schopný definovať, ktoré RDD je potrebné zachovať v pamäti pomocou *persist* metódy. Takéto RDD ostávajú v pamäti pre účely ďalších operácií v budúcnosti. Spark automaticky necháva všetky RDD v pamäti, ale môže nastať situácia, kedy je potrebné uvoľniť pamäť RAM pre ďalšie operácie a je nútený uložiť niektoré RDD na disk. Touto metódou môže užívateľ taktiež nastaviť stratégiu perzistencie alebo replikácie medzi počítačmi. Taktiež úžívateľ môže nastaviť prioritu perzistencie pre jednotlivé RDD.

Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	*count*()	:	RDD[T] ⇒ Long
	collect()	:	RDD[T] ⇒ Seq[T]
	reduce(*f* : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, *e.g.*, HDFS

Obr. 3.2: Základné transformácie a operácie v Spark nad Resilient Distributed Datasets

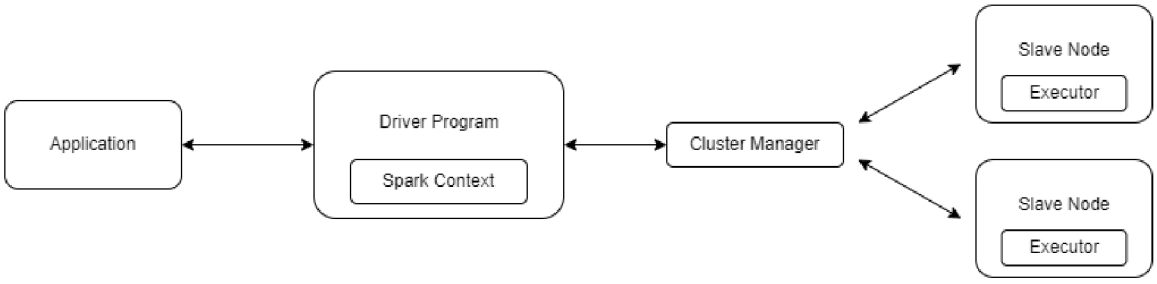
3.1.1 Architektúra

Na obrázku 3.3 je znázornená architektúra Apache Spark. Driver Program v Apache Spark architektúre volá hlavný program v aplikácií a vytvára Spark Context. Spark context pozostáva zo všetkých základných funkcií Sparku. Spark Driver obsahuje komponenty ako DAG plánovač, backend plánovač, plánovač úloh a manažéra blokov, ktorý je zodpovedný za preklad uživateľských programom do úlohy (z anglického Job), ktoré sú spúšťané v clusteri.

Spark Driver a Spark Context spoločné riadi vykonávanie úloh, ktoré sú v clusteri. Spark Driver zároveň využíva Cluster manažéra pre riadenie ostatných úloh. Cluster manažér je zodpovedný za alokáciu zdrojov potrebných pre úlohy. Následne rozdeluje úĺohy do menší častí, ktoré distribuuje do Slave uzlov. RDD, ktoré vytvorí Spark Context sú taktiež

distribuované do jednotlivých Slave uzlov a Slave uzol ich si ich môže uložiť do vyrovnávacej pamäti pre dalšie použitie. Slave uzly vykonajú priradenú úlohu Cluster manažérom a následne ju vrátia do Spark Contextu.

Exekútor v Slave uzloch zodpovedá za vykonanie a spustenie jednotlivých priradených úloh. Životnosť tejto komponenty je rovnaký ako životnosť celej Spark aplikácie. V prípade, že chceme zvýšiť rýchlosť výpočtu, môžeme zvýšiť pocet Slave uzlov, čím zvýšime možnosť väčšieho rozdelenia jednotlivých úloh.



Obr. 3.3: Apache Spark architektúra (prevzaté z [19])

V nasledujúcej časti sú popísané jednotlivé komponenty architektúry Spark aplikácie[25] z obrázka 3.3:

- **Spark Driver program** – Predstavuje centrálny bod Spark aplikácie. Je zodpovedný za vytvorenie Spark kontextu a pripojeniu ku clusteru. Taktiež plánuje úlohy, ktoré sú zadané Slave uzlom, riadi celkový priebeh aplikácie a využíva Cluster Manažéra k získaniu prostriedkov. Je zodpovedný za spustenie Exekútorov za pomoci Cluster manažéra. Po zaslaní úlohy Spark driver spustí hlavnú `main()` metódu aplikácie, vytvorí DAG graf reprezentujúci dátový tok. Na základe vytvoreného DAG požiada Cluster manažéra aby alokoval potrebné zdoje pre spracovanie úlohy. Keď sú zdroje alokované, Driver využije vytvorený Spark Context pre zaslanie úlohy (dáta spolu s kódom, ktorý má byť vykonaný) Slave uzlom, ktoré vykonajú úlohu a zašlú výsledok späť.

- **Spark Executor** – Úlohy, ktoré su zaslané Spark Driverom sú vykonávané exekútor procesom v Slave uzle. Hlavnou zodpovednosťou exekútor procesu je prijať zaslanú úlohu, spustiť kód s dátami, ktoré boli v úlohe zadané a ohlásiť výsledok, alebo neúspech, naspať Driveru. Každý Slave uzol má v Sparku jeden alebo viacero exekútor procesov a každý z týchto exekútor procesov sú zodpovedné za spustenie množiny úloh Spark aplikácie. Počet exekútorov, ktoré sú vytvorené pre aplikáciu je voliteľný a užívateľ má plnú kontrolu koľko exekútorov vo svojej aplikácií chce vyvoriť. Tento počet je nastaviteľný ako konfiguračný parameter pri vytvorení Spark Contextu[8]. Taktiež počet exekútorov je kontrolovaný Cluster manažérom, ktorý ich vytvára na základe dostupných zdrojov na danom stroji a nastavení clusteru.

- **Cluster Manager** – Komponenta zodpovedná za riadenie zdrojov počítacov v clusteri a alokovanie týchto zdrojov pre Spark Aplikáciu. Zdroj v tomto kontexte znamená CPU, pamäť a uložisko spark aplikácií, ktoré bežia v clusteri. Dostáva požiadavky na alokovanie zdrojov pre jednotlivé Slave uzly a ich exekútor procesov pre výpočet zadaných úloh. Taktiež je zodpovedný za spúštanie exekútor procesov na jednotlivých Slave

uzloch a monitorovanie stavov týchto exekútor procesov na Slave uzloch. Spark podporuje viaceré druhy cluster manažérov ako napríklad YARN, Apache Mesos a taktiež Spark Cluster manažér.

Apache Context je schopný pracovať s viacerími druhmi Cluster manažérov. Ako prvý je samostatný Spark Cluster Manager (Standalone Cluster). Ďalej je schopný spolupracovať s Hadoop YARN (Yet Another Resource Manager) alebo Mesos. V tejto časti budú rozobraté tieto tri rôzne Cluster Manager spôsoby.

Standalone Cluster

Spark poskytuje vstavaný Cluster manažer. Týmto spôsobom je veľmi jednoduché vytvoriť cluster a spustiť na ňom Spark aplikáciu. Tento spôsob je vhodný pre situácie, kedy je k dispozícii fixný počet používaných strojov, na ktorých bude vykonávaný výpočet. V tomto prípade je manažér zdrojov **Standalone Master** a uzol, ktorý vytvára Apache Standalone Cluster sa nazýva **Standalone Slave**. V tomto režíme Spark Driver beží na užívateľskom stroji a exekútor procesy sú spustené na Slave uzlov v clusteri. Driver zadáva úlohy exekútorom v uzloch a prijíma odpovede pomocou HTTP protokolu. Tento režim je vhodný pre malé až stredne veľké úlohy a zároveň je vhodný pre užívateľov, ktorý chcú rýchly a jednoduchý začiatok pri vývoji Spark aplikácií.

Hadoop Yarn (Yet Another Resource Negotiator)

Hadoop Yarn je manažér zdrojov pre cluster, ktorý poskytuje Apache Hadoop schopnosť spracovať dáta distribuovane na veľkých clusteroch. Bol predstavený v Hadoop 2.0 s cieľom vytvoriť podporu pre flexibilný spôsob spracovania dát a tvorí kľúčový komponent v Hadoop ekosystéme. Yarn je zodpovedný za správu zdrojov v clusteri a plánovanie spustenia úloh. Poskytuje centrálnu platformu pre správu, plánovanie a exekutívu distribuovaných aplikácií v Hadoop clusteri. Yarn dovoľuje užívateľovi spustiť viaceré aplikácie na rovnakom clusteri a zdielať jeho zdroje naprieč viacerím aplikáciam, čo prináša efektívnú správu zdrojov a flexibílne plánovanie úloh naprieč clusterom a rôznymi aplikáciami.

Yarn pozostáva z troch hlavných komponent: Resource manager a Node manager. Resource manager predstavuje centrálnu autoritu, ktorej sa zodpovedajú všetky uzly v clusteri. Zatiaľ co Node manager spravuje konkrétny uzol v clusteri a vykonáva operácie, ktoré zasiela Resource Manager[21].

- **Resource Manager** – Spravuje zdroje pre všetky aplikácie v systéme. Pozostáva z plánovača (z anglického scheduler) a manažéra aplikácie (z anglického application manager). Plánovač sleduje, aktualizuje a uspokojuje požiadavky na alokovanie zdrojov. Tieto požiadavky sú zasielané v podobe `ResourceRequests` správy. Tento požiadavok je možné vidieť v tabuľke 3.1. Plánovač dostáva požiadavky od aplikácie na alokáciu zdrojov, ktoré potrebuje pre spustenie úlohy. Dostupnosť zdrojov plánovač získavá z pravidelného signálu (heartbeat), ktorý zasiela Node Manager uzlu. Manažér aplikácie je zodpovedný za správu životného cyklu aplikácií, ktoré sú spustené v clusteri. Overuje či je možné prichádzajúce požiadavky od aplikácií splniť a spolu s plánovačom alokuje dostupné zdroje pre prichádzajúce požiadavky.

- **Node Manager** – Je zodpovedný za správu zdrojov v konkrétnom uzle. Komunikuje s Resource Manager pomocou signálu (heartbeat), ktorým informuje o svojej

dostupnosti, živosti a dostupných zdrojoch. Taktiež informuje Resource Manager objekt o stave vykonávaných úloh v uzle. Node Manager je zodpovedný za spúšťanie a zastavenie úloh na uzle a monitorovanie zdrojov a ich využitia.

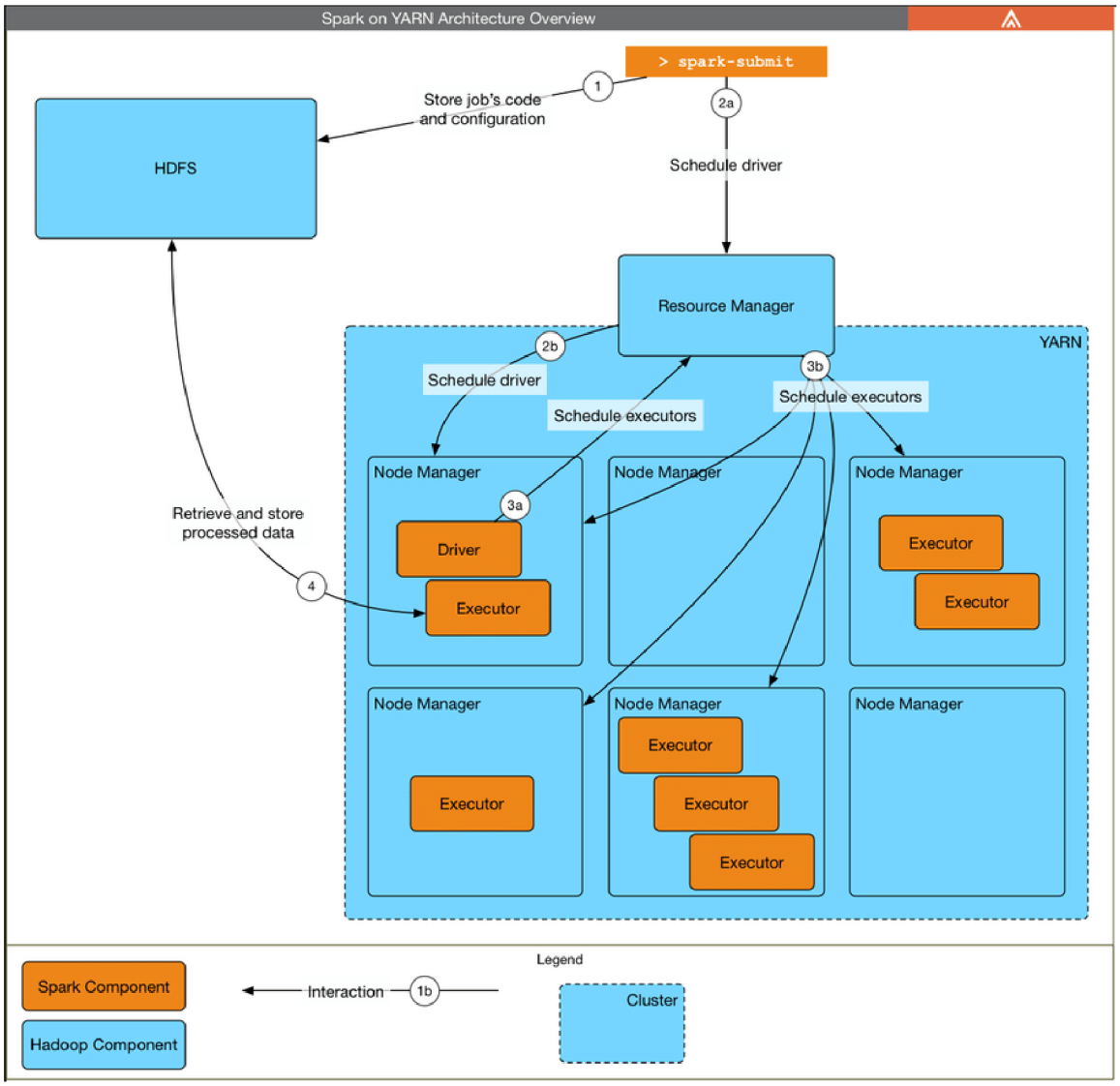
- **Application Master** – Komponenta zodpovedná za dohodnutie zdrojov pomocou `ResourceRequests` správy, ktorú zasiela Resource Manager komponente. Application Master je špecifický pre jednotlivé aplikácie. Predstavuje spustenú aplikáciu v clusteri. Application Master je taktiež spustený na jednom z uzlov v clusteri. Po zaslaní aplikácie do Yarn prostredia, Resource Manager spustí inštanciu tejto aplikácie v podobe Application Master triedy. Komponenta je zodpovedná za vykonanie úloh aplikácie a monitorovanie celkového progresu aplikácie.

1.	Počet kontajnerov (napríklad 20 kontajnerov)
2.	Zdroje pre kontajner <2GB RAM, 1 CPU>
3.	Preferencia lokality
4.	Priorita požiadavku

Tabuľka 3.1: Požiadavka na alokáciu zdrojov pre Resource Manager komponentu v YARN

V nasledujúcom obrázku 3.4 je vidieť prepojenie Spark komponent zobrazených na obrázku 3.3 a komponent Hadoop yarn popísaných vyššie. Modré objekty na obrázku predstavujú Hadoop Yarn komponenty zatiaľ čo oranžové predstavujú komponenty Spark aplikácie. Jednotlivé kroky, ktoré sú vykonané po odoslaní aplikácie a spustení úlohy je možne z obrázku popísať následovne:

1. Kód, ktorý ma byť vykonaný v úlohe spolu s konfiguráciou pre úlohu je uložený do distribuovaného uložiska HDFS.
2. Vytovrenie aplikácie v Yarn prostredí.
   (a) Aplikácia je odoslaná *Resource Manager* komponente, ktorá je zodpovedná za alokáciu prostriedkov a vytvorenia uzlu aplikácie v Yarn prostredí.
   (b) *Application Master* uzol je vytvorený v Yarn prostredí spolu s alokovanými prostriedkami.
3. Proces plánovania jednotlivých úloh a rozdelnie medzi *Slave* uzly v clusteri.
   (a) *Application Master* požiada *Rource Manager* komponentu pre naplánovanie jednotlivých úloh, ktoré aplikácie chce spustiť.
   (b) *Resource Manager* za pomoci *Plánovača* a *Application Manager* komponenty naplánuje jednotlivé úlohy pre uzly a alokuje pre nich potrebné zdroje.
4. Posledným krokom je uloženie a zozbieranie výsledkov do uložiska.



Obr. 3.4: Architektúra pre Spark s využitím Yarn Cluster manažéra (prevzané z [26]).

3.2 Apache Flink, Hadoop a Storm

V nasledujúcej časti sú popísané ďalšie nástroje, ktoré implementujú distribuované výpočtové modely. Dalšie frameworky, ktoré vytvárajú distribuovaný výpočtový model môže byť napríklad Apache Hadoop, ktorý je predchodcom spomínaného Apache Spark. Ďalší framework je Apache Storm, ktorý je navrhnutý pre spracovanie dátových prúdov. Posledný nástroj, ktorý je v tejto časti popísaný je Apache flink.

Apache Hadoop

Hadoop patrí medzi veľmi populárne voľne dostupné nástroje, ktorý sprístupňujú distribuované ulozisko, ktoré je schopné ukladanie veľké objemu dátových sád naprieč viacerími clustermi. Okrem uložiška ponúka MapReduce engine, ktorý je využívaný na distribuované spracovanie uložených dát. Hadoop je navrhnutý tak, aby ho bolo možné škálovať. To zna-

mená, že je možné tento výpočtový model spustiť ako na jednom stroji alebo na tisícoch uzlov v clusteri. Hadoop spracováva veľké množstvo dát súbežne a produkuje výsledky veľmi rýchlo. Základné časti Hadoop tvoria distribuované Hadoop uložisko (HDFS) a MapReduce engine.

Narozdiel od Apache Spark, tento nástroj prináša nielen výpočtový engine, ale taktiež poskytuje uložiska, koordinátor prvky a manažéra clusterov. Práve preto Spark využíva tento ekosystém, kedže Spark neobsahuje žiadne dalšie komponenty na správu clusterov a riadenie jednotlivých distribuovaných operácií.

Hadoop distribuované uložisko (HDFS) rozdeľuje jednotlivé súbory na menšie časti do blokov, ktoré ukladá na rôzne uzly v clusteri. V cluster existujú dva typy uzlov:

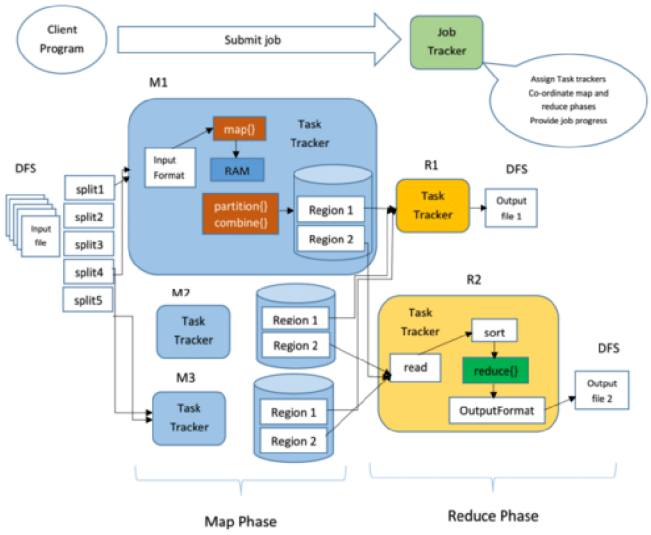
- **Master uzol** – Tento uzol sa taktiež nazýva menný uzol (z anglického namenode). Tento uzol má na starosti správu jednotlivých worker uzlov. Spravuje metadáta o súborovom systéme, čo zahŕňa informácie o umiestnení jednotlivých súborov alebo priečinkov v clusteri na worker uzloch. Predstavuje prístupový bod do HDFS uložiska a všetky požiadavky najskôr smerujú do tohto uzlu, ktorý následne poskytuje informácie kde vyžiadané dáta hladať (na ktorom dátovom uzle).

- **Worker** – Uzol sa prezýva tiež dátovým uzlom. Predstavuje uzol v HDFS clusteri, ktorý ukladá a sprístupňuje dáta. Dátový uzol komunikuje s master uzlom, ktorý ho prístupňuje a reguluje prístup klientov k dátam. Dátové uzly periodicky informujú svôj master uzol o stave a životnosti. Taktiež majú na starosť replikáciu dát. Táto replikácia je daná menným uzlom a dátové uzly sa starajú o dodržanie pravidiel nastavených menným uzlom.

Tieto dva uzly vytvárajú distribuované uložisko. Postup, ktorý sa uplatňuje v HDFS pri prístupe k dátam je možné popísať následovne. Na začiatku, užívatel zašle *master uzlu* požiadavok o konkrétny súbor uložený v HDFS. Menný uzol požiada o práva k prístupu daného súboru. Ak sú práva dostatočné menný uzol vráti zoznam HDFS blokov spolu s ich identifikačným číslom. Ďalej k tomuto zoznamu pridá zoznam dátových uzlov, ktoré majú tieto bloky uložené. Tento zoznam je následne vrátený užívatelovi, ktorý môže vykonávať potrebné operacie s týmito dátami.

MapReduce je výpočtový engine, ktorý pozostáva z dvoch operácií: Map a Reduce. Obrázok 3.5 popisuje fázy týchto dvoch operácií. Prvým krokom tohto výpočtového modeluje je operácia *map* (fáza mapovania). Map predstavuje užívatelom definovanú funkciu, ktorá je aplikovaná na vstupné dáta. Táto operácia vytvára množinu nových hodnôt, ktoré sú v tvare kľúč-hodnota (z anglického key-value). Mapovacia funkcia je aplikovaná paralelne na jednotlivých uzloch na dátach z HDFS (bloky dát, ktoré poskytuje HDFS). Novo vytvoreným hodnotám je priradená nová partícia a sú zoradené podľa kľúča. Po tomto kroku nasleduje operácia *reduce* (fáza redukovania). *Reduce* je taktiež užívateľom definovaná funkcie, ktorá je aplikovaná na všetky medzivýsledky vytvorené operáciou *map*. Táto funkcia premení medzivýsledky na výstupný formát kľúč-hodnota.

Pre správne fungovanie MapReduce bolo potrebné implementovať operáciu *Shuffle*. Táto operácia slúži na prenášanie výstupných dát z *map* operácie do jednotlivých *reduce* operácií. Operácia *shuffle* predstavuje takzvané úzke hrdlo (z anglického bottleneck) celej MapReduce operácie. Kedže *shuffle* je zodpovedný pre prenos dát, ktoré sa môžu vyskytovať na viacerých partíciach rozmiestnených v rôznych uzloch. Tento proces môže byť zdĺhavý a pomerne náročný na vstupno/výstupné operácie. Pre zlepšenie *shuffle* operácie slúži komponenta *combiner*. Combiner predstavuje lokálnu reduce operáciu pred samotným prenosom



Obr. 3.5: Architektúra MapReduce engine v Hadoop ekosystéme (prevzané z [13]).

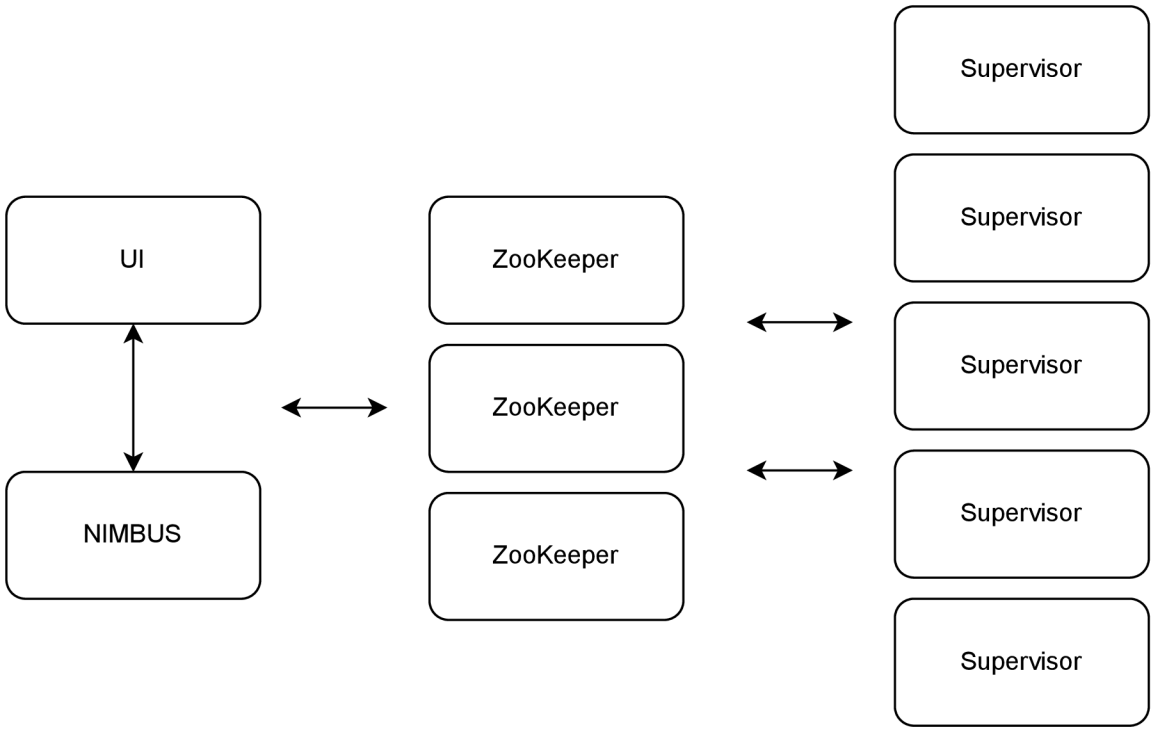
výstupných dát map funkcie. Vykoná lokálnu *reduce* operáciu a tým zmenší objem dát, ktorý musí operácia *shuffle* preniesť pre *reduce* operáciu.

Hlavný rozdielom medzi Hadoop a Spark predstavuje práve MapReduce engine. Hadoop pracuje s operáciami na disku, zatiaľ čo Spark pracuje v in-memory. Vstuno výstupné operácie su pri in-memory prístupe oveľa rýchlejšie čo je jasne vidieť na rýchlosti výpočtu Spark nástroja. Dalším hlavným rozdielom je spôsob, akým Spark vykonáva MapReduce výpočet. Spark pomocou DAGs optimalizuje výpočet tak, že všetky operácie, ktoré môžu byť vykonané nad dátami z jednej partície ukladá do fázy, čím predchádza náročnému presúvaniu jendotlivých dát medzi fázami.

Apache Storm

Predstavuje distribuovaný systém pre spracovanie prúdu dát (z anglického data stream) v reálnom čase. Vďaka Storm nástroju je možné spracovávať neviazaný prúd dát v reálnom čase a vykonávať nad týmito dátami operácie podobným spôsobom ako Hadoop alebo Spark pracuje nad dávkami dát. Storm má veľkú škálu použití ako napríklad analýzy v realnom čase, online strojové učenie, nepretržité výpočty a ďalšie.

Architektúra Storm nástroja je veľmi podobná architektúre Spark na obrázku 3.3. Aj v Storm architektúru existujú slave uzly, manager uzly a samotná aplikáca spolu s rozhraním pre Storm API, ktoré využíva aplikácia. Na obrázku 3.6 je vidieť znázornenie implementovanej architektúry Strom modelu.



Obr. 3.6: Architektúra nástroja Apache Storm (prevzané z [12]).

Jednotlivé komponenty z obrázka 3.6 majú konkrétne úlohy:

- **UI** – Webové rozhranie pre ovládanie a spúštanie jednotlivých úloh v systéme Storm.
- **NIMBUS** – Komponenta zodpovedná za spúštanie a správu jednotlivých topológii (topológia je podobná úlohe v Sparku) naprieč clusterom. Funkcionalita tejto komponenty je podobná Spark Driver komponente.
- **ZooKeeper** – Zodpovedá za správu konfigurácií a koordináciu činností jednotlivých časti Storm clusteru. Táto komponenta je využitá taktiež pri Hadoop ekosystéme a Spark konfigurácií, ktorá využíva Hadoop.
- **Supervisor** – Spravuje jednotlivé výpočtové uzly. Dohliada na správne fungovanie slave uzlu a jeho výpočet. V spark architekúre je táto komponenta najviac podobná *Exekútor* časti v slave uzle.

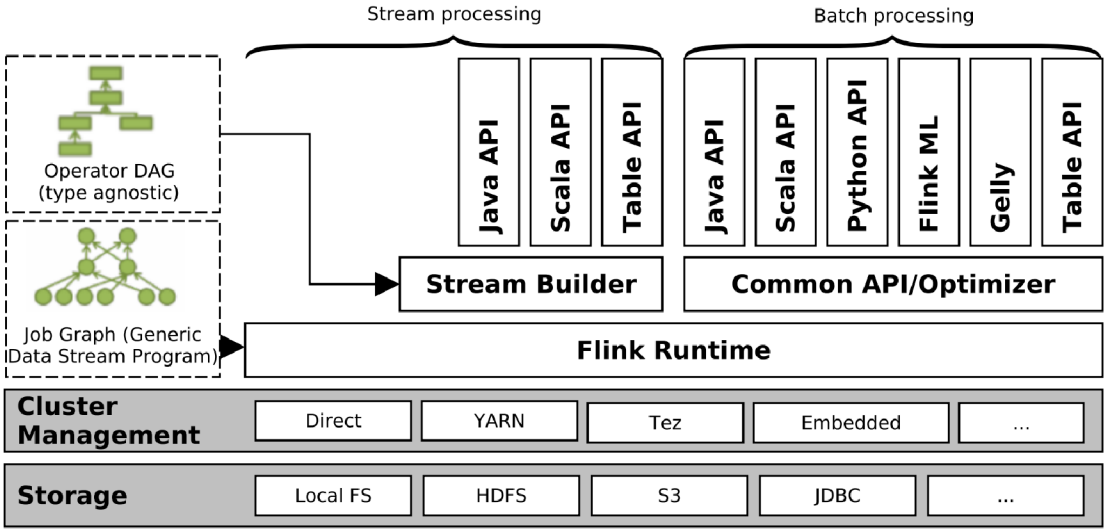
Ako je vidieť, architektúra sa podobá ako Apache Hadoop frameworku tak aj Spark modelu. Narozdiel od Spark alebo Hadoop kde sú spúšťané na výpočtových uzlov úlohy v Storm nástroji sa tieto úlohy nazývajú topológie. Topológia sa od úlohy líši jednou zásadnou vecou. Zatiaľ čo úloha v Hadoop alebo v Spark vždy skončí, topológia spracováva jednotlivé dáta nekonečne dlho (alebo dokým topológia nie je ukončená).

Hlavným rozdielom oproti Spark nástroji je, že Storm nie je natívne možné spustiť nad Hadoop clusterom. Aj napriek tomu, že Storm nie je možné natívne spustiť nad Hadoop clusterom je stále schopný využívať HDFS pre zápis alebo čítanie súborov. Druhým hlavný rozdielom oproti Sparku je spôsob prúdového spracovania. Spark vytvára mikro dávky z prúdu dát, ktoré spracováva dávkovo. Storm spracováva prúd dát postupne záznam za záznamom bez akéhokoľvek vytvárania dávok.

Apache Flink

Flink je voľne dostupný, distribuovaný nástroj pre spracovanie big data. Vie pracovať ako v dávkovom režime tak aj prúdovom. Pre tieto dva režimy Flink obsahuje veľké množstvo knižníc ktoré sú dostupné v tomto frameworku. Flink v porovnaní s ostatnými modelmi distribuovaného výpočtu prináša redukovanú úroveň komplexnosti vďaka integrovaniu tradičný databázových konceptov ako deklaratívny dotazovací jazyk a automatickej optimalizácií dotazov. Je plne kompatibilný s Apache Hadoop tak isto ako Apache Spark. Flink ako aj Spark využíva pre reprezentáciu a optimalizáciu výpočtov orientované acyklické grafy. Tieto grafy reprezentujú tok operácií a dát v spúštanom výpočte a slúžia na optimalizáciu jednotlivých krokov a plánovanie úloh[15].

Tak isto ako Spark, Flink je kompatibilný s veľkou škálou dostupných distribuovaných uložísk. Taktiež je možné spolu s Flink využiť dostupných cluster managerov. Na obrázku 3.7 je zobrazená architektúra Flink nástroja.



Obr. 3.7: Architektúra Apache flink spolu s jeho komponentami (prevzané z [15])

Ako je na obrázku 3.7 vidieť architektúra je veľmi podobná Apache Spark. Flink môže taktiež využívať pre správu clusteru napríklad Hadoop Yarn a ako uložisko môže využiť HDFS podobne ako Spark. Základom Flink nástroja predstavuje *Flink Runtime*. Jedná sa o zjednotené behové prostredie, kde sú spúšťané všetky programy. Programy vo Flink sú štrukturované ako orientovaný graf (JobGraphs), ktorý obsahuje paralelné operácie. JobGraph obsahuje uzly a hrany. Narozdiel od Hadoop, Flink nerozdeluje programy do jednotlivých individuálnych fáz. Všetky operácie v programe sú spustené paralelne. Výsledky jednotlivých operácií sú následne priamo zasielané dalším operátorom. Takéto spracovanie vytvára postupnú retaz spracovaní (z anglického pipeline execution). *Stream Builder* a *Common API* slúžia na preklad programov medzi behovým prostredím (Flink Runtime) a jednotlivými rozhraniami API. Transformujú orientované grafy logických operáci jednotlivých programov do generických programov, ktoré sú spúšťané v behovom prostredí Flink. Automatická optimalizácia toku dát v programoch je vykonávaná v tomto procese.

Kapitola 4

Distribuované súborové systémy

Distribuované súborové systémy (skrátene DFS, z anglického Distributed File Systems) predstavujú typ súborových systémov, ktoré sprístupňujú viacerým užívatelom súbory, ktoré sú uložené na viacerých zariadeniach spojených s internetom. Hlavným cieľom vývoja distribuovaných súborových systémov bolo riešenie limitácií centralizovaných súborových systémov.

Jedna z hlavným motivácií k vývoju distribuovaným súborovým systémom bola núdza o zdieľanie súborov medzi viacerými užívatelmi a zariadeniami v internetovom prostredí. Centralizované súborové systémy, v ktorých sú súbory uložené na jednom stroji, neboli dostatočne efektívne a nezvládali narastajúcu požiadavku zdielania a kolaborácie so súbormi. Distribuované súborové systémy na druhú stranu vytvárajú decentralizované uložisko a správu súborov uložených na viacerých zariadeniach. Toto poskytuje lepšiu škálovateľnosť, spolahlivosť a dostupnosť súborov.

Druhou motiváciou pre vývoj takýchto súborových systémov bola potreba pre správu a spracovanie veľkého objemu dát, nazývaných big data, ktoré sú používané napríklad vo vedeckých výpočtových aplikáciach. Distribuované súborové systémy sú schopné veľký objem dát spracovať efektívnejšie a rýchlejšie oproti klasickým centralizovaným systémom.

Medzi aktuálne najviac populárne riešenia DFS patrí Hadoop Distributed File System (skrátene HDFS). HDFS je voľne dostupné riešenie, ktore patrí do Apache Hadoop ekosystému, ktorý je rozobratý v 3. Ďalším aktuálne populárnym distribuovaným uložiskom je Ceph. Síce sa nejedná o súborový systém, ale predstavuje populárnu náhradu za HDFS.

Kedže dáta, ktoré sú použité pri forenznej analýze sú v dnešných dňoch čoraz väčšie, je potrebné využiť efektívnejšie systémy ako centralizované. Práve preto sa využitie distribuovaných súborových systémov javí ako dobrý spôsob ako tieto dáta ukladať. V nasledujúcej kapitole sú rozobraté aktuálne systémy, ktoré poskytujú distribuované súborové uložiská. Taktiež bude priblížená problematika ukladania forenzných dát do týchto súborových systémov.

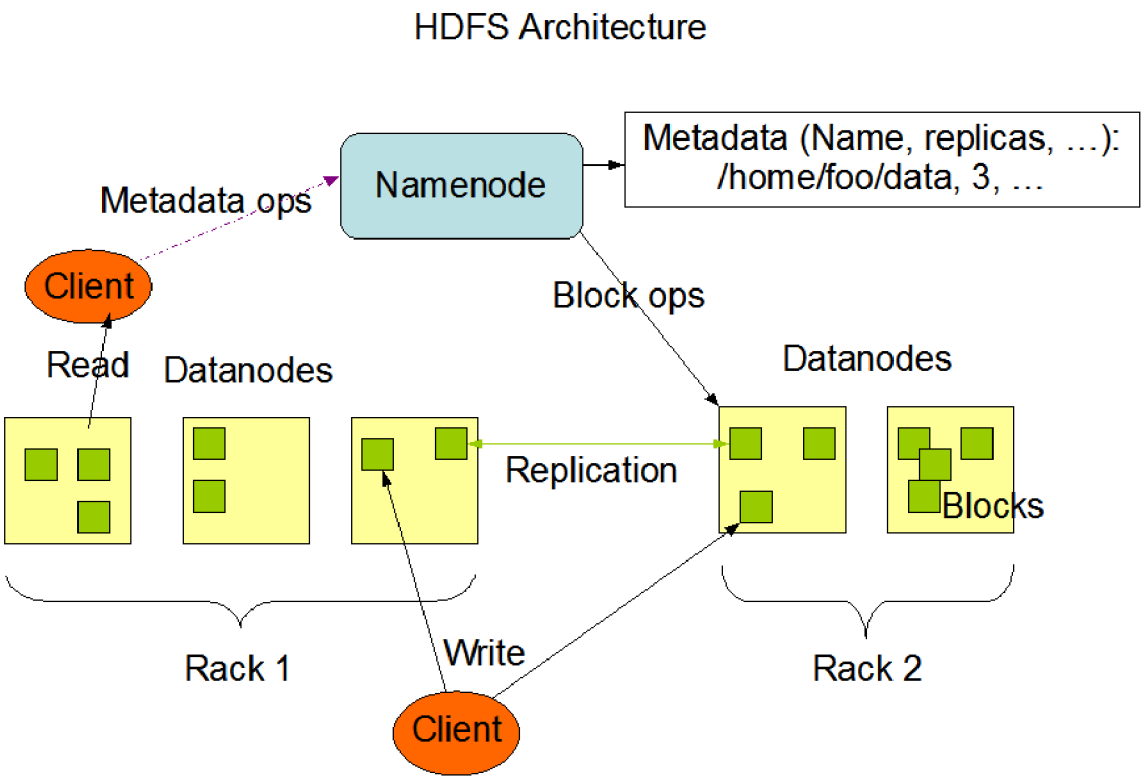
4.1 Hadoop Distributed File System

Hadoop vo svojom ekosystéme okrem MapReduce engine ponúka aj systém distribuovaného súborového systému. Tento súborový systém je využívaný v Hadoop ekosystéme pre analýzu a transformáciu veľkých dátových sád, ktoré sú spracovávané pomocou MapReduce paradigmatu. Hlavnou charakteristikou HDFS je rozdelenie dát a výpočtu naprieč veľkým množstvom strojov. Hadoop cluster je možné škálovať pridávaním nových uzlov do clusteru, čím sa zvyšuje kapacita uložiska a taktiež kapacita vstupno výstupných operácií.

HDFS predstavuje komponentu Hadoop ekosystému, ktorá zabezpečuje ukladanie a správu dát. Aj keď HDFS je podobné UNIX súborovému systemu veľa jeho štandardov je odlišných pre zlepšenie výkonnosti. HDFS ukladá metadáta o súborovom systéme a aplikačné dáta oddelene. Tak isto ako mnoho ďalších distribuovaných súbotových systémov ako Lustre[4], GFS[9] tak aj HDFS ukladá metadáta do dedikovaných serverov, ktoré sa nazývajú NameNode. Aplikačné dáta sú uložené na ostatných serveroch, ktoré sú prezývané DataNodes. Všetky uzly sú navzájom prepojené a komunikujú medzi sebou na základe TCP protokolu. HDFS nevyužíva RAID mechanizmus pre ochranu dát ako Lustre, ale spolieha na replikáciu obsahu do viacerých dátových uzlov. Štandardne je replikácia dát v HDFS do troch uzlov, ale parameter replikácie je samozrejme možné zmeniť.

4.1.1 Architektútra

Architektúra HDFS pozostáva z troch základných komponent. Prvou z komponent je menný uzol (NameNode). Menný uzol je zodpovedný za ukladanie a správu metadát, ktoré sú využívané pre správu dát, ktoré sú v súborovom systéme uložené. Druhou komponentou je dátový uzol (DataNode). Dátový uzol predstavuje komponentu, ktorá zodpovedá za dáta a ich správu. Za tretiu komponentu HDFS architektúry je možné považovať klienta, ktorý komunikuje so súborovým systémom a žiada o akcie, ktoré sa v systéme majú vykonať. Na obrázku 4.1 je možné vidieť zobrazenú architektúru HDFS systému.



Obr. 4.1: Architektúra HDFS a jeho komponenty. Menný uzol (Namenode) spravuje metadáta o súborovom systéme, ktoré popisujú uloženie súborov v dátových uzloch (Datanodes). Bloky (blocks) dát sú replikované do viacerých dátových uzlov, ku ktorým pristupuje klient (Client) (prevzané z [11]).

Menný uzol

HDFS menný priestor predstavuje hierarchiu jednotlivých súborov a priečinkov. Súbory a priečinky sú reprezentované v mennom uzle ako *inodes*. *Inodes* obsahujú záznamy o atribútoch jednotlivých súborov alebo priečinkov. Tieto atribúty môžu byť práva k súboru, modifikácie, prístupový čas a diskové kvóty. Uložené súbory v systéme sú rozdelené do blokov, typicky 128 megabajtov, a každý blok súboru je nezávisle replikovaný na viaceré dátové uzly. Úlohou menného uzlu je správa menného priestoru a správa jednotlivých blokov. Spravuje ich tak, že vytvára mapovanie jednotlivých blokov na konkrétne dátové uzly, v ktorých sú uložené (fyzická poloha dát). V prípade, že klient chce čitať súbor, ako prvé kontaktuje menný uzol pre zistenie polohy jednolivých blokov daného súboru. Následne klient využije tieto informácie per získanie jednotlivých blokov súbor od dátových uzlov, ktoré sú ku klientovi najbližsie. V prípade, že klient chce súbor zapisovať, vyžiada menný uzol aby vybral najvhodnejšie tri uzly, ktoré budú ukladať blok dát a jeho repliky.

HDFS ukladá celý menný priestor do pamäte RAM. Kedže pamäť RAM nie je trvalá, menný uzol vytvára následujúce záznamy aby bol schopný udržať menný priestor aj po reštarte alebo zlyhaní:

- **Obraz** (z anglického image) – Predstavuje dáta o inodes, ktoré zahrňajú metadáta o súborov systéme a zoznam všektých blokov, ktoré náležia jednotlivým súborom, ktoré sú v systéme uložené.

- **Kontrolný bod** (z anglickeho Checkpoint) – Trvalý záznam o obraze, ktorý je uložený v natívnom súborovom systéme menného uzlu. Kontrolné bodu sú periodicky vytvárané z menného priestoru HDFS. Po reštarte je kontrolný bod spolu so žurnálom využívaný pre obnovenie aktuálneho stavu menného priestoru. Vytvorenie nového kontrolného priestoru znamená zmenšenie aktuálneho žurnálu. Všetky zmeny sú zapísané do kontrolného bodu a je vytvorený nový prázdny žurnál. Tvorba kontrolných bodov v častejších intervaloch zabezpečuje, že žurnál nie je príliš dlhý a tým pádom nehrozí stráta veľkej časti histórie zmien.

- **Žurnál** (z anglického Journal) – Súbor, ktorý obsahuje log informácie o modifikáciach obrazu. Tento súbor je taktiež ukladaný do natívneho súborového systému menného uzlu. Žurnál je takzvaný write-ahead commit log súbor pre všetky zmeny v súborovom systéme. To znamená, že všetky transakcie od klientov sú uložené v tomto súbore ešte pred potvrdením zmien v HDFS. Toto opatrenie dovoľuje obnovenie HDFS menného priestoru do stavu pred jeho vypnutím alebo zlyhaním. Pri štarte, menný uzol inicializuje menný priestor HDFS z kontrolného bodu. Následne na posledný kontrolný bod aplikuje uložené zmeny v žurnáli a dostáva aktuálny menný priestor systému.

Pre zlepšenie odolnosti voči zlyhaniu je možné ukladať redundantné kópie kontrolných bodov a žurnálov na dalšie servery, z ktorých v prípade zlyhanie hlavného serveru môže nastať obnova. Počas reštartu menného serveru, menný server využije žurnál pre obnovenie menného priestoru. To znamená, že nad menným priestorom vykonáva jednotlivé modifikácie v takom poradí v akom boli uložené do žurnálu. Poloha jednotlivých blokov sa môže postupom času meniť a preto nie je poloha ukladaná do kontrolných bodov.

Datový uzol

Každá replika bloku na dátovom uzle je rezprezentovaná dvoma súbormi uložených na hosťovskom natívnom súborovom systéme. Jeden súbor obsahuje konkrétne dáta daného súboru a druhý obsahuje metadáta o danom bloku. Tieto metadáta obsahujú kontrolný súčet a časovú známky vygenerovania bloku. Veľkosť súboru, ktorý ukladá blok je rovnaká ako veľkosť konkrétneho bloku a tým aj v prípade menšej veľkosti bloku súbor nezaberá dodatočné miesto na disku ako je tomu pri tradičnom súborovom systéme.

Pri štarte dátového uzlu je vytvorené spojenie s menným uzlom. Pri tomto spojení je vykonaný *handshake*. *Handshake* slúži na overenie identifikačného čísla menného pristoru a taktiež na overenie verzie programu, ktorý využíva dátový uzol. Identifikačné číslo menného priestoru je priradené každému dátovému uzlu pri jeho inicializácii. Identifikačné čislo je nemenné a je trvalo uložené do každého uzlu v clusteri. Ak sa do clusteru prihlási novo incializovaný dátový úzol, korý nemá identifikačné číslo, tak je mu menným uzlom priradené. V prípade, že uzol má iné identifikačné číslo ako mu bolo priradené pri inicializácii, menný uzol takýto uzol odmietne. Odmietnutie je z dôvodu zachovania integrity súborového systému.

Druhým bezpečnostným prvkom je verzia programu spustená na dátovom uzly. Konzistencia verzií je veľmi dôležitá pretože môže spôsobiť stratu dát alebo poškodenie dát. Vzhľadom na to, že cluster často obsahuje veľký počet dátových uzlov môže nastať situácia, že pri aktualizacií a zmene verzií systému môže byť niektorý uzol nedostupný a nemusí sa prejaviť zmena verzie. Preto sú takéto uzly odmietnuté, čím je vynútená dodatočná zmena verzie systému.

Dátový uzol informuje menný uzol o vlastníctve blokov pomocou správ *block report*. Takáto správa obsahuje informácie o identifikačných číslach blokov, ktoré má uložené. Ďalej obsahuje časové známky ich vygenerovania a taktiež dĺžku každého bloku. Prvá správa je zaslaná okamžite po pripojení k mennému uzlu. Dalšie správy su zasielané v hodinových intervaloch a poskytujú mennému uzlu aktuálny prehľad o tom, kde sú všetky bloky uložené.

Dalším mechanizmom, ktorým dátové uzly informujú menný uzol je takzvaný *heartbeat*. Jedná sa o opakujúci sa signal, ktorý dátové bloky zasielajú mennému uzlu v 3 sekundovom intervale (3s interval predstavuje predvolenú hodnotu). Menný uzol považuje uzly za nefunkčné, keď v priebehu 10 minút nezašlú ani raz tento signál. V tomto prípade označí daný uzol a jeho bloky za nedostupné. Heartbeat signál slúži taktiež na prenášanie informácií a pravidelné informovanie menného uzlu o nasledujúcich hodnotách: celková kapacita uloziska, zlomok využitého uloziska a množstvo dát, ktoré je aktuálne prenášané. Tieto informácie slúžia mennému uzlu k alokácií ďalšieho miesta a vyvažovaniu záťaze medzi uzlami. Signál je taktiež využívaný menným uzlom ku komunikácií s dátovými uzlami. Menný uzol nekomunikuje priamo s dátovými uzlami, ale využíva tento signál, na ktorý zasiela odpovede. Tieto odpovede predstavujú inštrukcie dátovým uzlom. Inštrukcie môžu byť

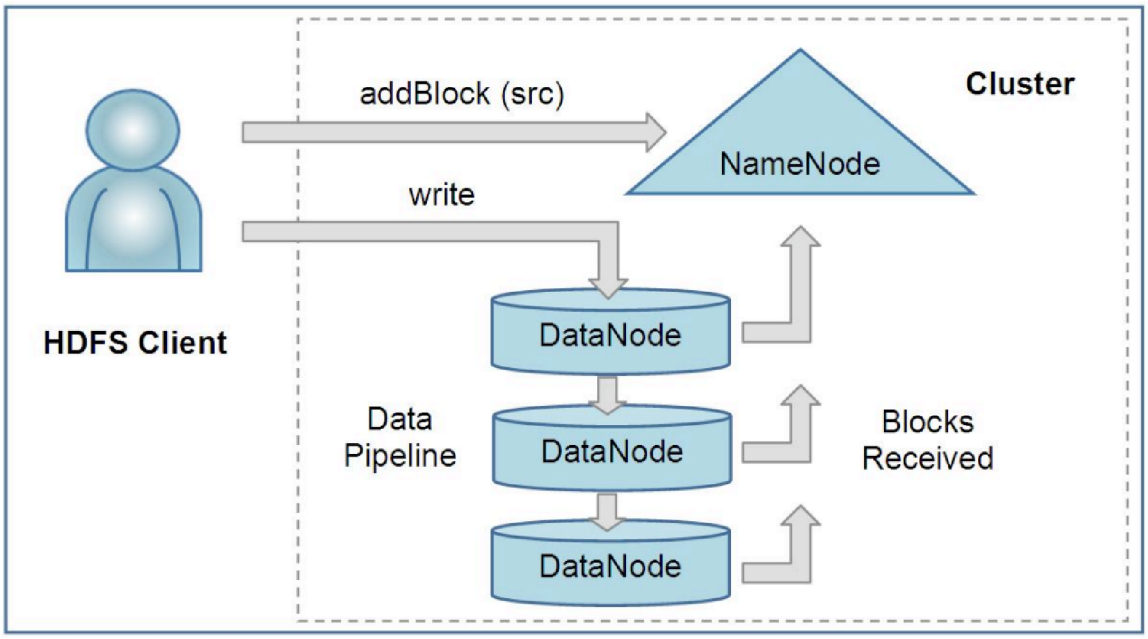
- Replikácia bloku na druhý dátový uzol.
- Odstránenie lokálnej repliky bloku.
- Znovupripojenie alebo vypnutie uzlu.
- Zaslanie okamžitého hlásenia o blokoch.

Tieto inštrukcie sú dôležité k správe a celkovej integrite systému. Preto je dôležité aby bol heartbeat signál pravidelný.

HDFS klient

Uživatelské aplikácie pristupujú k HDFS pomocou HDFS klienta, ktorý predstavuje rozhranie nad súborovým systémom. Ako všetky klasické súborové systémy aj HDFS podporuje operácie čítania, zápisu a odstránenie súboru. Taktiež podporuje operácie pre vytvorenie a odstránenie priečinkov. Aplikácie pristupujú k uloženým súborom na základe cesty k súborom z menného priestoru. Klient nepotrebuje vedieť, kde sa nachádzajú jednotlivé metadáta a konkrétne súbory. Taktiež nemá povedomie o tom, že súbory majú viaceré repliky v systéme.

Pri operácií čítania súboru, klient požiada menný uzol o zoznam dátových uzlov, ktoré majú uložené repliky blokov požadovaného uzlu. Klient po získaní zoznamu priamo kontaktuje dátové uzly a vyžiada si od nich jednotlivé bloky prisluchajúce požadovanému súboru. V prípade, že HDFS klient chce súbor zapísať, požiada menný uzol o výber dátových uzlov, ktoré budú ukladať daný blok súboru a jeho repliky. Klient vytvorí zreťazené operácie (z anglického pipeline), ktoré postupne zapisujú repliky blokov do vybraných uzlov. Keď je prvý blok zapísaný, klient vyžiada další zoznam vhodných dátových uzlov od menného uzlu pre zápis dalšieho bloku súboru. Popísaná interakcia zápisu medzi klientom a uzlami je zobrazená na obrázku 4.2



Obr. 4.2: Interakcia medzi HDFS klientom, menným uzlom a dátovými uzlami pri zápise súboru (prevzané z [20]).

Rozdiel medzi tradičnými súborovými systémami je, že HDFS odhaluje klientovi polohu blokov jednotlivých súborov. Toto odhalenie dovoľuje napríklad MapReduce alebo Spark enginu lepšie plánovanie úloh. Toto plánovanie je lepšie z dôvodu, že výpočetný engine vie o polohách konkrétnych blokov a vyberá tie, ktoré sú najbližšie k výpočetným uzlom. Taktiež HDFS prispôsobuje svôj replikačný faktor na základe dôležitosti dát. Súbor, ktorý je veľmi vyžadovaný klientom je dôležitejší a jeho replikačný faktor sa automaticky nastaví na vyššie číslo. Toto je z dôvodu odolnosti voči chybám a taktiež aby bolo v systéme viacej replík tohto súboru čím sa zvýši rýchlosť čítania, kedže bude možné tento súbor čítat z viacerých miest.

Záložný uzol a Kontrolný uzol

Menný uzol v HDFS môže okrem svojej primárnej úlohy vykonávať dalšie dve voliteľné funkcie. Môže slúžiť ako uzol, ktorý zodpovedá za vytváranie záloh alebo môže slúžiť ako kontrolný uzol, ktorý vytvára kontrolné body. Popis týchto dvoch funkcií je následovný:

- **Kontrolný uzol** (z anglického Checkpoint Node) – Periodicky kombinuje existujúce kontrolné body a žurnály pre vytvorenie nových kontrolných bodov a nových prázdnych žurnálov. Aktuálné kontrolné body a žurnály si sťahuje z menného uzlu. Následne ich spojí do nového kontrolného bodu a vytvorí nový žurnál. Novo vytvorený žurnál a kontrolný bod zašle naspäť mennému uzlu. Funkcie kontrolného uzlu sú často oddelené od menného uzlu a je vytvorený samostatný kontrolný uzol v clusteri. Toto je hlavne z dôvodu, že kontrolný uzol má rovnaké pamäťové požiadavky ako menný uzol. Preto je vhodné využiť samostatný uzol s oddelenými prostriedkami.
- **Záložný uzol** (z anglického Backup Node) – Záložný uzol je možné popísať ako menný uzol, ktorý je iba na čítanie. Záložný uzol príjma prúd žurnálov menného priestoru z menného uzlu. Tieto žurnáli si ukladá do svojej pamäti a aplikuje ich na

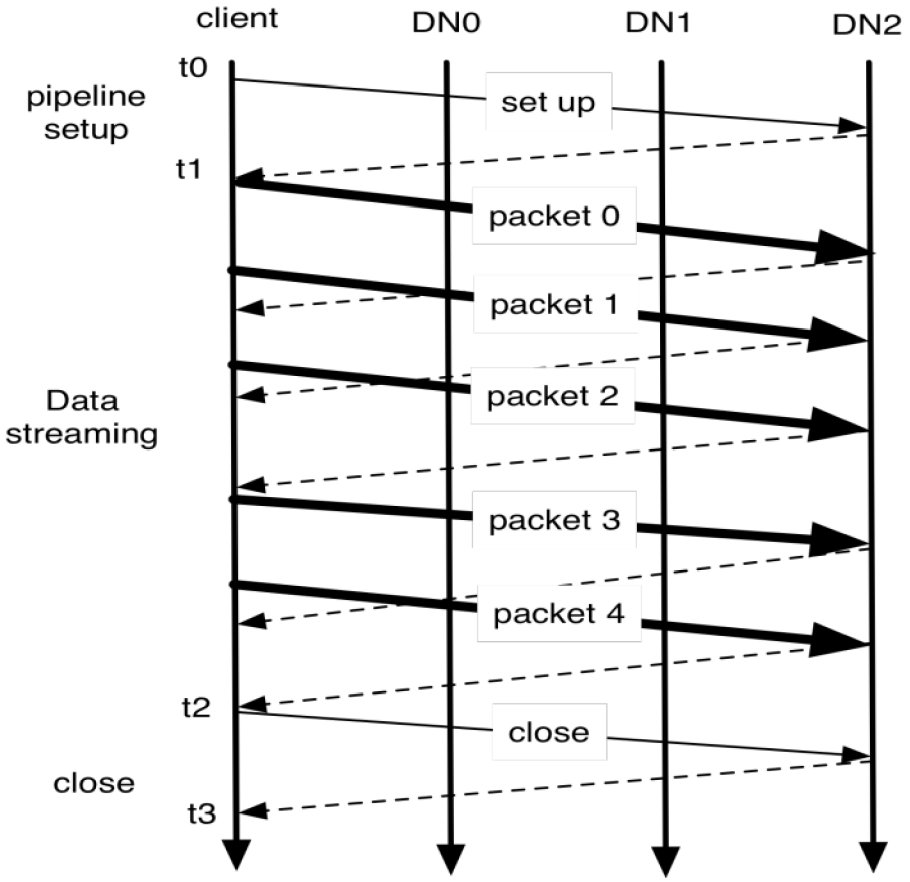
svôj vastný obraz menného priestoru. Záložný uzol si vytvára vlastne kontrolné body bez nutnosti ich sťahovania od menného uzlu, kedže vlastní aktuálny menný priestor vo svojej pamäti. Menný uzol zaobchádza s uloženými žurnálmi na záložnom uzle rovnako ako so svojimi. V prípade zlyhania menného uzlu je použitý záložný uzol, ktorý má uložené obrazy a kontrolné body, ktoré predstavujú aktuálny stavny stav menného priestoru.

4.1.2 Vstupno výstupné operácie

Do HDFS systému je aplikácia schopná pridávať dáta tak, že vytvorí nový súbor a zapíše do nového súboru dáta. Po tom, čo je nový súbor zavretý, nie je možné meniť alebo odstrániť zapísané bajty v tomto súbore. Je možné len pridávať nové dáta do súboru a to tak, že sa súbor otvorí a sú do neho pridané dáta. HDFS implementuje model jedného zapisovateľa (z anglického single-writer) a viacerých čitateľov (z anglického multiple-readers).

HDFS klient pri otvorení súboru pre zápis získava dočasný prenájom na tento súbor. To znamená, že žiadny iný klient nemôže do tohto súboru zapisovať po dobu aktívneho prenájmu. Po zavretí súboru je prenájom odstranení a sprístupnení ostatným klientom pre zápis. Ak klient potrebuje prenájom predĺžiť, využiva na tento účel už spomenutý signál *heartbeat*. Dĺžka prenájmu súboru je obmedzená dvomi druhmi limitov. Prvý limit sa nazýva jemný limit (z anglického soft limit). Po dobu jemného limitu je zaručené, že klient ma výhradný prístup k zápisu súboru. Ak po jeho uplynutí klient nezavrel súbor alebo nepožiadal o predĺženie prenájmu, iný klient môže získať výhradné právo na zápis do tohto súboru. V prípade, že uplynul tvrdý limit (jedna hodina) a klient nepožiadal o obnovenie prenájmu, tak HDFS predpokladá, že klient skončil a automaticky zavrie daný súbor v mene klienta a získa späť prenájom. Prenájom súboru na čítanie neovplyvňuje klientov, ktorý chcú zo súboru čítať. Viacero klientov môže naraz čítať zo súboru aj v prípade, že existuje aktívny prenájom na zápis.

Pri nahrávaní súboru menný súbor rozhodne o dátových uzloch, do ktorého bude uložené repliky blokov. Následne je vytvorená dátová pipeline, ktorá zabezpečí uloženie blokov do všetkých replík. Dátové uzly sú do pipeline vybrané tak aby boli čo najbližšie ku klientovi, ktorý zapisuje. Bajty sú posielane v pipeline ako sekvencia paketov. Obrázok 4.3 znázorňuje vytvorenie dátového bloku. Pipeline je vytvorená z 3 dátových uzlov (DN0, DN1, DN2) a bloku, ktorý tvorí 5 paketov.



Obr. 4.3: Dátová pipeline počas vytvárania nového bloku. Vertikálne čiary predstavujú aktivity klienta a dátových uzlov, tenké čiary na obrázku znázorňujú kontrolné správy pre nastavenie a uzavretie pipeline. Hrubé čiary predstavujú zasielané pakety, čiarkované čiary potvrdzovacie správy (prevzané z [20]).

Medzi časmi t_0 a t_1 prebieha nastavenie dátovej pipeline. Interval medzi t_1 a t_2 je fáza, kedy sa zasielajú jednotlivé pakety medzi klientom a dátovým uzlom. V čase t_1 je zaslaný prvý dátový paket a v čase t_2 je potvrdené prijatie posledného paketu z bloku. Posledný interval t_2 po t_3 je finálna fáza, v ktorej sa blok uzavrie a je zapísaný na dátovom uzle.

Ak chce klient čítať dáta zo súboru, tak požiada menný uzol o zoznam dátových uzlov, ktoré majú uložené bloky tohto súboru a zašle zoznam klientovi. Menný uzol zašle klientovi polohy všetkých replikácií blokov a zoradí ich podľa polohy klienta. Klient sa snaží čítať bloky dát z uzol, ktoré sú k nemu najbližšie. Ak čítanie bloku z najbližšieho uzlu nie je možné (napríklad nastal výpadok dátového uzlu) využije druhý najbližši uzol. Pri čítaní jednotlivých blokov klient obdrží spolu s dátami aj kontrolný súčet, ktorý náleží danému bloku. Ak vypočítaný kontrolný súčet nie je rovnaký ako zaslaný, tak klient považuje súbor za poškodený a skúsi získať súbor z iného dátového uzlu.

4.1.3 Umiestnenie blokov

Bežnú praxou pri distribuovaných systémoch je, že jednotlivý uzly sú rozložené do viacerých serverových rackov. V takomto rozložení je dôležité aby jednotlivé uzly vedeli o svojich

vzdialenostiach medzi ostatnými uzlami. Na umiestnení zaleží šírka prenosového pásma, ktoré sú schopné uzly medzi sebou dosiahnuť. Vo väčšine pripadov platí, že širka pásma je vyššia pri uzloch, ktoré sú v rovnakom racku ako pri uzloch v odlišných rackoch. HDFS odhaduje internetovú šírku prenosového pásma dvoch uzlov na základe ich vzdialenosti. Vzdialenosť medzi uzlom a jeho rodičom je definovaná ako 1. Vzdialenosť medzi dvoma uzlami je potom vypočítaná vypočítaná ako súčet vzdialeností k ich predchodcom. Čím menšia vzdialenosť medzi dvoma uzlami tým je predpokladná väčšia šírka internetového prenosového pásma.

HDFS dovoľuje administrátorom konfigurovať skript, ktorý vracia identifikačné číslo racku v ktorom sa daný uzol nachádza. Keď sa dátový uzol prihlási ku mennému uzlu je na ňom spustený tento skript, aby získal identifikačné číslo svojej polohy. V prípade, že tento skript nie je konfigurovaný, tak menný uzol predpokladá, že všetky uzly sú v spoločnom racku.

Poloha a určovanie polohy replík je pre HDFS kľúčovým prvkom. Zaleží od neho spolahlivosť a rýchlosť zápisu/čítania. Dobrá poloha repliky by mala zaručovať spolahlivosť, optimálnu rýchlosť a taktiež dostupnosť. HDFS poskytuje konfigurovateľnú politiku rozmiestňovania replík, čo poskytuje možnosť experimentovať s umiestnením replík a dosiahnuť čo najviac optimálnu stratégiu pre konkrétnu architektúru.

Predvolenú stratégiu umiestňovania replík, ktorú má HDFS nastavenú sa dá zhrnúť následovne:

- Žiadny dátový uzol by nemal mať viac ako jednu repliku žiadneho bloku dát
- Žiadny rack by nemal obsahovať viac ako 2 repliky toho istého dátovéh bloku. Samozrejme v prípade, že cluster má dostatočný počet samostatných rackov.

4.2 Ceph

Ceph je systém pre distribuované uložisko, ktorý poskytuje vysoký výkon, škálovatelnosť a spolahlivosť. Je navrhnutý tak, aby poskytoval vysokú mieru dostupnosti, čo znamená, že dáta sú vždy k dispozícií aj v prípade zlyhania viacerích uložísk. Ceph využíva replikáciu dát naprieč viacerími uzlami, aby dosiahol požadovanú dostupnosť.

Ceph implementuje RADOS algoritmus (Reliable Autonomic Distributed Object Store), ktorý slúži na správu uložených dát. RADOS je navrhnutý tak, aby bol škálovatelný horizontálne, čo uľahčuje pridávanie nových uložísk podľa potreby Ceph clusteru. RADOS taktiež spravuje replikáciu dát a poskytuje samo uzdravovací mechanizmus, ktorý zaručuje, že dáta sú vždy dostupné aj v prípade zlyhania uložísk.

Ceph vo svojom ekosystéme obsahuje taktiež distribuovaný súborový systém, CephFS. CephFS vytvára klasický súborový systém nad dátami uložených v RADOS. CephFS je implenetovaný nad RADOS a poskytuje funkcionalitu nad súbormi a priečinkami. Taktiež je možné využiť RBD (RADOS Block Device), ktorý sprístupňuje dáta uložené v RADOS v tradičnom blokovom prístupe.

Ceph je možné intergrovať s ostatným softwarom, ktorý rozširuje jeho funkcionalitu ako dátová kompresia, šifrovanie a dalšie. Taktiež podporuje protokoly pre viacnásobné prístupové protokoly (z anglického multiple access protocols) ako S3, Switf, NFS, SMB a iSCSI.

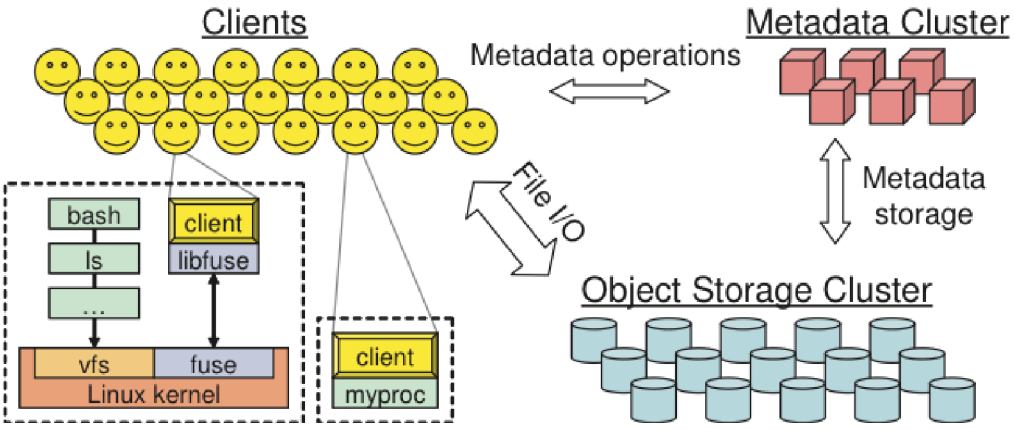
Ceph vo svojom systéme nevyužíva konvenčné pevné disky, ale využíva prístup OSD (Object-based storage). Predstavuje úložné zariadenie pre ukladanie dát podobne ako disk, ale pracuje na vyššej úrovni. Namiesto blokovo orientovaného rozhrania, ktoré zapisuje a číta

dáta v blokoch pevnej dĺžky, organizuje OSD dáta do flexibilných dátový kontajnerov, ktoré sa nazývajú objekty. Ceph využíva OSD prístup v kombinácií s CPU, internetovým rozhraním, lokálnou cache pamäťou s využitím disku alebo RAID. OSDs nahrádzajú tradičné blokové rozhranie s rozhraním, ktoré klientovi dovoľuje čítať alebo zapisovať rozsah bajtov do omnoho väčších pomenovaných objektov. Klient typicky komunikuje s metadáta serverom aby mohol vykonať metadáta operácie (napríklad otvorenie súboru, premenovanie a podobné) zatiaľ čo s OSD komunije pre vykonanie vstupno/výstupných (ďalej ako I/O operácie) operácií (čítanie a zapisovanie). Narozdiel od ostatných distribuovaných súborový uložísk ako napríklad spomenutý HDFS v kapitole 4.1, Ceph nevyužíva len jeden server pre metadáta, ale obsahuje cluster metadát serverov[23].

4.2.1 Architektúra

Obrazok 4.4 zobrazuje architektúru celého systému a taktiež jednotlivé komponenty. Ceph súborový systém má 3 hlavné komponenty:

- **Klient** – Sprístupňuje skoro POSIX súborové rozhranie. Komunikuje s metadáta servermi alebo priamo s OSD v clusteri. Môže využiť doplňujúci software, napríklad FUSE, ktorý umožnuje pripojit Ceph súborový systém.
- **Metadáta server** – V Ceph sa jedná o cluster metadáta serverov. Spravujú menný priestor súborového systému a poskytujú metadáta operácie spolu s metadátami o súboroch.
- **OSD server** – Cluster OSD uložísk, ktoré spolu ukladajú dáta uložené v systéme.



Obr. 4.4: Architektúra Ceph systému. Klient komunikuje priamo s OSDs pre vykonanie I/O operácií. Pre metadáta operácie komunikuje s metadáta servermi. (prevzané z [23]).

Architektúra bola navrhnutá s cieľom škálovateľnosti (až ku stovkám petabajtov a viac), výkonnosti a spolahlivosti. Architektúra poskytuje možnosť škálovateľnosti vo viacerích dimenziach zahŕňajúc celkovú veľkosť uloziska, priepustnosť a výkonnosť. Architektúra je schopná dosiahnuť súčasné čítanie a zapisovanie do rovnakého súboru, čo je využiteľné vo vedeckých aplikáciach, ktoré sú spustené na super clusteroch. Ceph dosahuje vysokej škálovateľnosti a zároveň dosahuje vysokého výkonu, spolahlivosti a dostupnosti pomocou 3 základných vlastností:

- **Oddelenie dát a metadát** – Ceph sa snaží čo najviac oddeliť správu súborového systému metadát a uložiska dát. Operácie nad metadátami (otvorenie, premenovanie a podobné) sú spoločne spravované metadáta servermi v clusteri. Samotné dáta má na starosti cluster OSDs, s ktorými klient komunikuje priamo pre účely I/O operácií. Ceph využíva schopnosť nízko úrovňovej alokácie samotných OSD a necháva túto réžiu na samotných zariadeniach v clusteri. Ceph narozdiel od ostatných OSD súborových systémov [23] nevyužíva žiadny zoznam objektov, ktorý by obsahoval informácie o alokovaných blokoch a ich umiestneniu. Namiesto toho, využíva jednoduchú funkciu, ktorá pomenováva blok dát na základe inode čísla, rozsahu bajtov a stratégie rozkladania dát. Miesto, kam sa objekt má uložiť je následne vypočítané na základe špecialne distribučnej funkcie CRUSH[24]. CRUSH dovoľuje každej strane v systéme vypočítať umiestnenie a polohu uloženého objektu. Toto odstraňuje nutnosť spravovania , distribúce zoznamu uložených objektov, zjednodušuje dizajn systému a znižijue záťaž na metadáta cluster.

- **Autonómne distribuovaný object storage** – Veľké systémy, ktoré pozostávajú z tisícov zariadení sú veľmi dynamické. Nové zariadenia sú pridávané pre potreby väčšieho uložiska, niektoré zariadenia sú vyradené, iné sa pokazia a podobne. Všetky tieto faktory ovplyvňujú distribúciu dát v clusteri. Ceph ponecháva distribúciu dát medzi jednotlivými uzlami na OSD. Využíva ich inteligencie (CPU a pamäť), ktorá je dostupná na každom OSD.

- **Dynamicky distribuovaná správa metadát** – Operácie spojené s metadátami v súborovom systéme tvoria skoro polovicu celkovej záťaže na systém. Preto Ceph implementuje nielen jeden prístup k metadátam, ale využíva cluster servorov pre správu metadát. Zapracováva Dynamic Subtree Partitioning architektúru z [22], ktorá dynamicky a inteligentne distribuuje zodpovednosť za správu hierarchie súborového systému medzi metadáta servery.

4.2.2 Dynamicky distribuované metadáta

Ceph distribuuje nielen uložené dáta, ale aj metadáta. Metadáta sú distribuované naprieč viacerými metadáta servermi, ktoré vytvárajú hierarchiu súborového systému. Metadáta súborov a priečinkov sú v Ceph systéme veľmi malé. Pozostávajú len zo záznamov priečinkov a inodes záznamov (80 bajtov). Narozdiel od ostatných súborových systémov, Ceph neukladá metadáta o alokácií súborov, kedže názvy súborov a ich umiestnenie je možné vypočítať pomocou CRUSH algoritmu. Dizajn distribuovaných metadát sa snaží minimalizovať I/O operácie spojené a využitie cache pamäti pomocou Dynamic Subtree Partitioning[22].

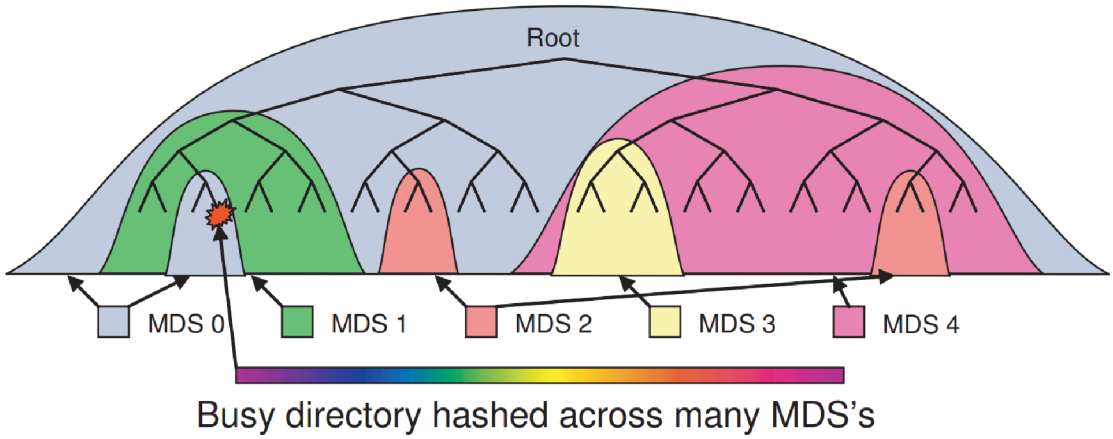
Dynamic Subtree Partitioning

Cluster metadáta serverov (skrátene MDS) v Ceph je založený na Dynamic Subtree Partitioning stratégii, ktorá adaptívne distribuuje uložené metadáta vo vyrovnávacej pamäti do hierarchie naprieč uzlami. Na obrázku 4.5 je zobrazené rozdelenie a hierarchie priečinkov. Jednotlivé MDS majú uložené časti stromu a spravujú len konkrétnu časť stromu.

Jednotlivé servery merajú popularitu metadát, ktoré spravujú vo svojej hierarchii. Popularita je meraná na základe počítadla prístupov k metadátam, ktorý má nastavený exponenciálny rozklad. Každá operácia zvýšuje počítadlo prístupu k danému inode a jeho predchodcom v stromovej hierarchii až ku koreňu. Na základe počítadla, ktoré meria popularitu

jednotlivých inodes v systéme môže MDS vytvoriť takzvané hotspoty. Hotspot predstavuje priečinok alebo súbor, ktorý je veľmi často žiadaný. Pre takéto prípady, Ceph distribuuje tento hotspot naprieč viacerými MDS servermi pre zlepšenie rýchlosti a dostupnosti takéhoto súboru.

Keď sú metadáta replikované naprieč viacerými MDS uzlami, obsah inode je oddelený do 3 skupín: bezpečnostná skupina (vlastník, mód súboru), súborová skupina (veľkosť, čas modifikácie) a nemnná skupina (inode číslo, rozloženie). Zatiaľ čo nemenná skupina sa nikdy nemení, ostatné skupiny sú ovládané konečným stavovým automatom s rôznymi stavmi a prechodmi navrhnuté a prispôsobené k rôznym prístupom a aktualizáciam.

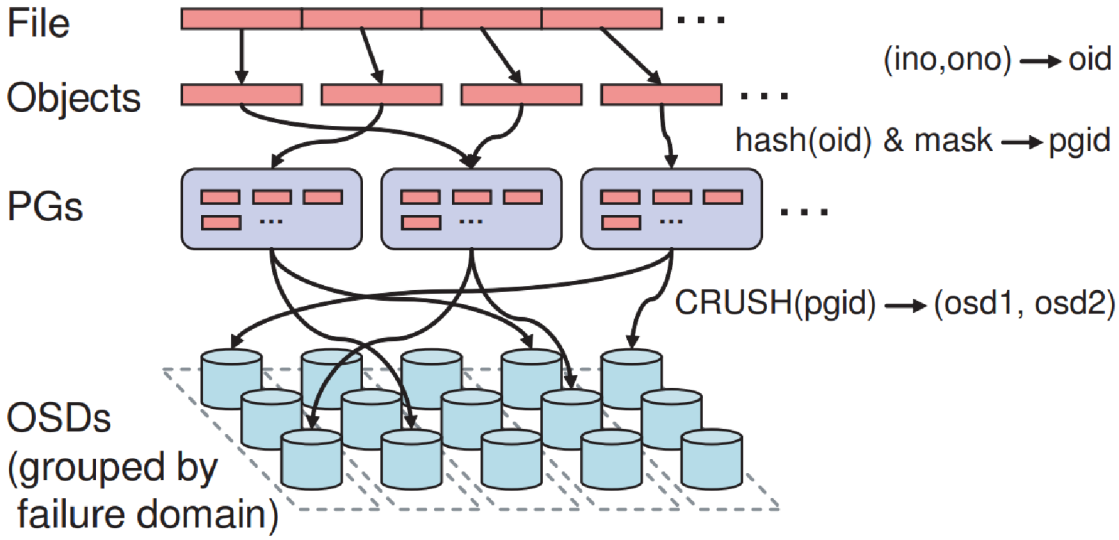


Obr. 4.5: Mapa hierarchie podstromov a rozdelenie naprieč jednotlivými MDS. Farby reprezentujú jednotlivé MDS v clusteri a podstromy, ktoré spravujú. Mapa obsahuje aj Hotspot priečinok, ktorý je distribuovaný naprieč viacerými MDS. (prevzané z [23])

4.2.3 Distribuovaný objekt uložiska

Z vyššieho abstraktného pohľadu klient a metadáta servery vidia OSD cluster ako jedno logické objektové uložisko s jedným menným priestorom. RADOS dosahuje lineárnej škálovateľnosti vďaka delegovaniu správy replikácie objektov, rozširovaniu clusteru, detekcie chýb a obnove na OSD servery.

Ceph sa pri distribúcii dát medzi uzli snaží čo najefektívnejšie využiť uložisko a internetovú prenosovú šírku pásma každého uzlu. Pre zabránenie nerovnováhy (napríklad najnovšie pridaný uzol je prázdny a nepoužívany) a záťažovej asymetrie (napríklad nové dáta sú len na jednom zariadení) vyvinul Ceph CRUSH algoritmus. CRUSH distribuuje dáta pseudo náhodne, migruje časti dát medzi uzlami náhodným spôsobom a rovnomerne distribuuje dáta z odstránených zariadení. Tento náhodný prístup spĺňa všetky požiadavky Ceph a je schopný rovnomerne využiť potencial všetkých uzlov v clusteri.



Obr. 4.6: Súbory sú rozdelené do viacerých objektov, zoskupené do PGs a distribuované do OSDs na základe CRUSH (prevzané z [23]).

Ceph najskor namapuje objekt do umiestňovacej skupiny (z anglického placement groups, PGs) pomocou jednoduchej hash funckie s konfigurovateľnou bitovou maskou pre kontrolu počtu umiestňovacích skupín (ďalej ako PGs). Ceph zvolil ako predvolenú hodnotu 1000 PGs. PGs sú následne priradené jednotlivý OSD na základe CRUSH (Controlled Replication Under Scalable Hashing). Tento proces je vidieť na obrázku 4.6. CRUSH predstavuje psuedo náhodnú distribučnu funkciu, ktorá efektívne mapuje každé PGs do zoradeného zoznamu OSD, do ktorých sú následne uložené repliky objektov. Narozdiel od konvenčných distribuovaných uložísk, umiestnenie dát nespolieha na žiadny zoznam metadát blokov alebo objektov. V porovnaní s vyhľadávaním objektu, CRUSH potrebuje iba PGs a OSD cluster mapu (kompaktný, hierarchický popis cluster uložiska). Tento prístup ma 2 výhody:

1. Každá komponenta systému je schopná nezávisle na druhej vypočítat lokáciu každého uloženého objektu
2. Cluster mapa sa zriedkavo aktualizuje, čo prakticky vylučuje akúkoľvek výmenu metadát súvisiacich s distribúciou.

4.3 Ukladanie forenzných dát

Distribuované uložiská budú v práci využité pre ukladanie forenzných dát, ktoré budú spracovávané Plaso nástrojom z kapitoly 2, ktorý bude prevedený pomocou Spark výpočetného modelu do distribuovaného modelu. Forenzné dáta, ktoré budú novým nástrojom spracovávané je potrebné uložiť do distribuovaného uložiska. Spracovávané forenzné dáta, ktoré Plaso vie spracovať je veľke množstvo. Plaso vie spracovať veľké mnoštvo formátov ako napríklad jsonl, asl, winreg a podobné. Tieto súbory sú veľmi vhodné pre uloženie na distribuovanom uložisku, kedže sa jedná o súbory, ktoré v sebe obsahujú dáta. Tieto súbory sú v distribuovanom uložisku rozdelené na viaceré bloky a uložené do dátových uzlov systému. Problém nastáva, keď do distribuovaného prostredia je nutné uložiť obraz súborového systému, ktoré Plaso taktiež vie spracovať (viď. 2). Problém obrazu je, že predstavuje externý

súborový systém, ktorý obsahuje hierarchiu súborov a priečinkov. Distribuované systémy nevedia tento typ súborov spracovať takým spôsobom ako napríklad UNIXové súborové systém, kedy stačí jednoduché pripojenie obrazu k natívnemu súborovému systému a pracovať snim bežným spôsobom.

Distribuovaný súborový systém ako HDFS nemá možnosť pripojiť tento súborový systém do svôjho systému, čo v tejto práci predstavuje prekážku pre tento typ forenzných dát. Preto je nutné prispôsobiť proces zápisu týchto dát do distribuovaného uložiska. Spôsob akým je možné odstrániť tento problém predstavuje predspracovanie obrazu súborového systému pred samotným zápisom do distribuovaného prostredia.

Obraz súborového systému je nutné pred samotným zápisom rozbaliť (pripojiť) na natívny súborový systém klienta. Po otvorení obrazu je nutné preskúmať hierarchiu celého súborového systému a získať informácie o súboroch, ktoré sa v ňom nachádzajú. Po analýze nasleduje proces vyextrahovanie všetkých súborov, ktoré obraz obsahuje a postupne jeho hierarchiu zapísať do distribuovaného súborového systému. Obraz je následne nahraný ako jednoduchý priečinok, ktorý obsahuje uloženú hierarchiu a nie je potrebné obraz pripájať alebo sním špecialne pracovať.

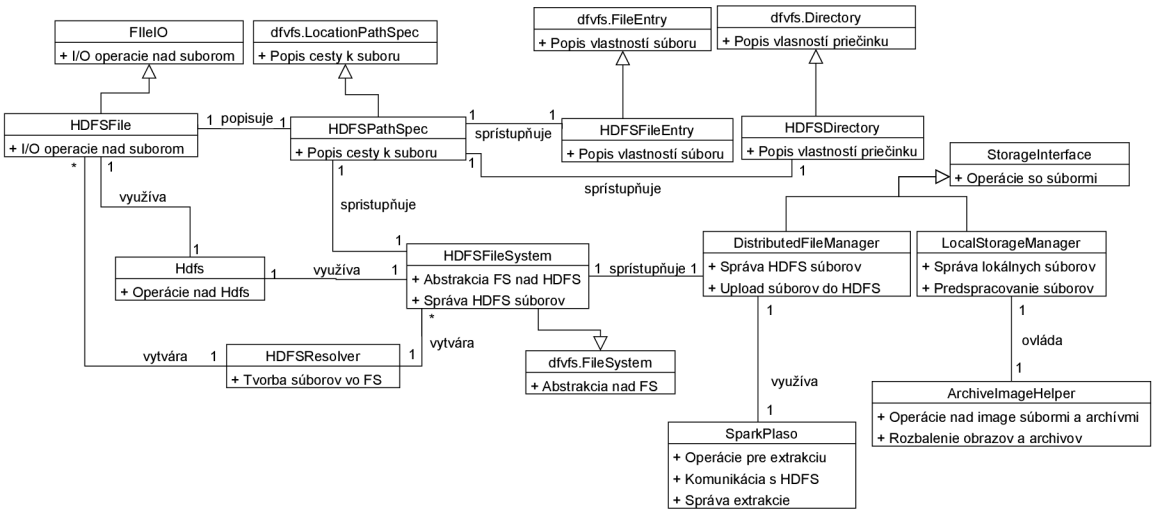
Druhým komplikovaný forenzným druhom dát sú kompresné a archívne súbory. HDFS nepodporuje natívne operácie nad týmito súbormi a nie je možné snimi pracovať ako v bežných súborových systémoch. Pre takéto dáta je nutné aplikovať rovnaký postup ako pri obrazoch súborového systému. Archívny alebo kompresný formát je nutné predspracovať pred samotným zápisom a vyextrahovať z neho všetky súbory a následne ich zapísať ako priečinok obsahujúci ich štruktúru.

Ostatné forezné dáta sú veľmi vhodné pre uloženie do distribuovaného prostredia. Všetky ostatné súbory, ktoré Plaso analyzuje sú jednoducho rozložiteľné do blokov a pre ich zápis nie je potrebné žiadne predspracovanie. Uloženie forenzných dát do distribuovaných súborových systémov má okrem spomenutých dvoch nevýhod veľkú výhodu. Kedže forezne dáta sú distribuované je možné nad týmito dátami spustiť viaceré analýzy. Distribuované systémy ponúkajú veľmi dobrú dostupnosť vďaka svojim vlastnostiam a analýzy nie sú limitované jedným strojom a jeho rýchlosťou súborového systému spolu s diskom.

Kapitola 5

Návrh riešenia

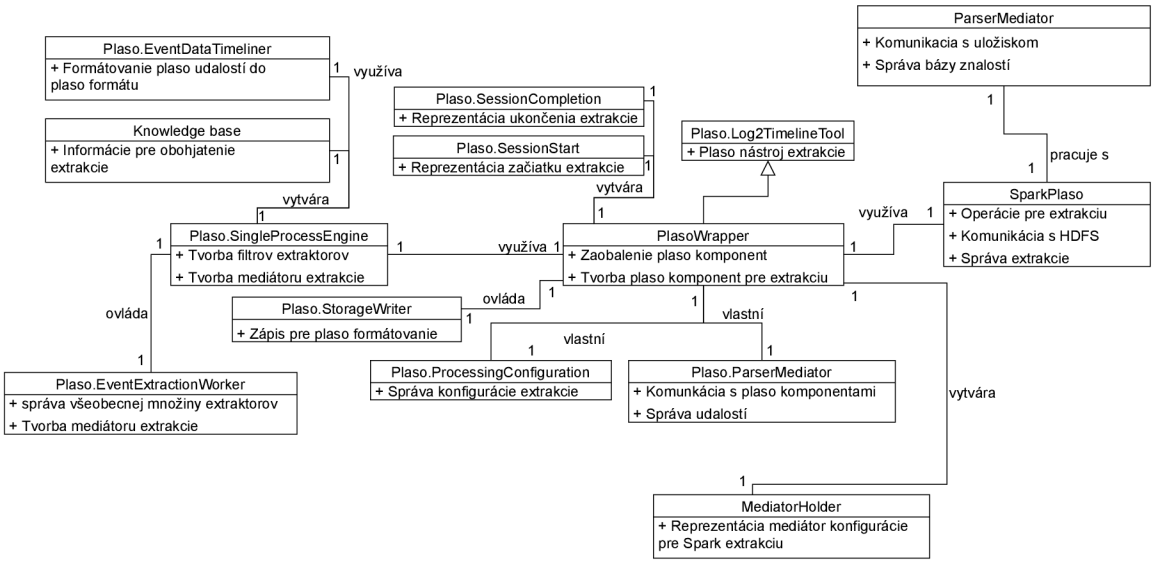
Navrhnuté riešenie, **Plasospark**, pre implementáciu distribuovanej verzie Plaso nástroja využíva Spark výpočetný model spolu s využitím HDFS distribuovaného uložiska. Riešenie sa snaží čo najviac využiť existujúce Plaso triedy, ktoré sú v riešení prispôsobené pre využitie HDFS a Spark modelu. Celý diagram tried je priložený v prílohe B. Diagram zobrazuje navrhnuté triedy a ich vzájomné vzťahy, ktoré sú v riešení využité. V tejto kapitole je diagram rozdelený do troch častí pre lepšie zobrazenie a prehľadnosť.



Obr. 5.1: Diagram tried zobrazujúci uložiskovú časť navrhnutého Plasospark nástroja.

Plaso využíva vo svojej implementácií knižnicu DFVS, ktorá vytvára virtuálny súborový systém nad vstupnými dátami. V navrhnutom riešení je táto knižnica rozšírená pre vytvorenie kompatiblity medzi HDFS súbormi a Plaso nástrojom. Knižnica je rožšírená o abstrakciu nad HDFS súbormi, ktorá implementuje operácie nad HDFS súbormi potrebné pre Plaso. V riešení sú navrhnuté triedy `HDFSFile`, `HDFSPathSpec`, `HDFSFileEntry` a `HDFSDirectory`, ktoré využijú knižnicu DFVFS a implementujú operácie nad týmito súbormi v HDFS prostredí. `LocalStorageManager` implementuje operácie nad lokálnymi súbormi, ktoré sú lokálne spracované pred uložením do HDFS uloziska. Táto trieda využíva *ImageArchiveHelper* triedu, ktorá vykonáva predspracovanie obrazov súborových systémov a archívov. Táto trieda vychádza z nedostatkov distribuovaných uložísk popísaných v 4.3.

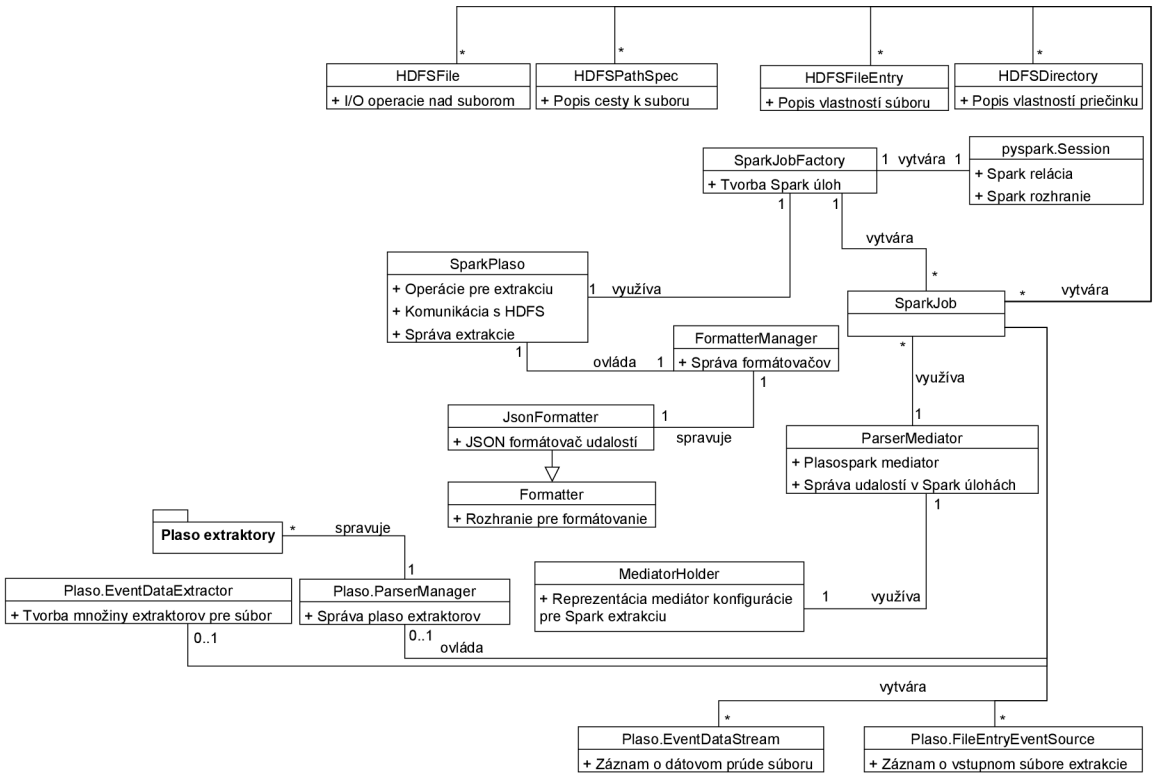
Trieda `DistributedFileManager` je zodpovedná za správu súborov uložených v HDFS systéme a poskytuje uložené súbory triede `SparkPlaso` pri extrakcii.



Obr. 5.2: Diagram tried zobrazujúci využitie Plaso komponent v navrhnutom Plasospark nástroja.

`SparkPlaso` trieda je hlavnou triedou, ktorá je zodpovedná za riadenie celého procesu extrakcie. K procesu extrakcie využíva triedy `PlasoWrapper`, `SparkJobFactory` a `DistributedFileManager`. Triedu `PlasoWrapper` využíva k prístupu k Plaso komponentám a Plaso konfiguráciam, ktoré definujú extrakciu v Plaso. `PlasoWrapper` spravuje všetky Plaso komponenty, ktoré sú využité pri extrakcii a formátovaní výstupu. `SparkJobFactory` vytvára Spark úlohy, ktoré vykonávajú v Spark Plaso operácie pre analýzu a extrakciu. Taktiž vytvárá Plaso záznamy a objekty vytvorené v priebehu Plaso extrakcie.

Pre správu extraktorov a ich aktualizáciu v prípade vydania novej verzie Plaso nástroja je využitá natívna trieda Plaso nástroja `ParserManager`. Táto trieda spravuje všetky zaregistrované extraktory z Plaso `parsers` balíčku nástroja.



Obr. 5.3: Diagram tried zobrazujúci Spark časť navrhnutého Plasospark nástroja.

V Spark úlohách reprezentované triedou `SparkJob` sú využité Plaso komponenty pre tvorbu Plaso záznamov a udalostí. Ako je z diagramu vidieť, nie všetky vytvorené Spark úlohy vytvárajú všetky Plaso objekty, ale úlohy sú rozdelené do viacerých kategórií, ktoré tvoria jednotlivé Plaso objekty.

Pre formátovanie vyextrahovaných je možné využiť natívny Plaso formát, ktorý je vytvorený za pomoci Plaso triedy `EventDataTimeliner`. Táto trieda vyextrahované dáta sformátuje do Plaso kompatibilného formátu, ktorý bude možné využiť s dalšimi Plaso nástrojmi.

Druhou možnosťou formátovania je využitie nových formátovačov, ktoré implementujú rozhranie `Formatter`. V návrhuje je vytvorený jeden formátovač `JsonFormatte`, ktorý vyextrahované udalosti formátu do JSON formátu. Pridanie nového formátovača je možné na základe poskytnutého rozhrania a registrácie formátovača pomocou triedy `FormatterManager`.

Kapitola 6

Implementácia

Nasledujúca kapitola rozoberá implementáciu nástroja **Plasospark**, ktorý využíva Spark výpočetný model pre distribúciu výpočtu Plaso nástroja. Implementácia odpovedá návrhu, ktorý je rozobratý v predchádzajúcej kapitole 5. Systém je implementovaný v jazyku python s využitím knižníc spomenutých v nasledujúcich kapitolách. Je spustiteľný na bežných linuxových distribúciach (testovaný na Ubuntu 20). K spusteniu nástroja je dostupný docker obraz, ktorý obsahuje všetky potrebné závislosti. Docker obraz je odporúčaný hlavne pre množstvo knižníc, ktoré je potrebné nainštalovať v závislosti na využitie Plaso nástroja. Systém je voľne dostupný na github[1] spolu s predvytvorenými (z anglického prebuilt) docker obrazmi, ktoré stačí stiahnuť[2].

Prvá časť kapitolu popisuje v skratke ovládanie systému pomocou dostupného API rozhrania. V časti sú rozobraté jednotlivé API koncové body (z anglického endpoints) a operácie, ktoré poskytujú.

V druhej časti je rozobratá implementácia ukladania vstupných súborov do systému. Popísané sú jednotlivé triedy, ktoré sú zodpovedné za uloženie dat do lokálneho uložiska systému, predspracovanie uložených súborov a ich následné uloženie do HDFS uložiska.

Poslednou časťou je popis implementácie extrakčnej časti systému. Extrakčný systém je popísaný pomocou jednotlivých tried, ktoré ovládajú proces extrakcie a vytvárajú Spark úlohy. V tejto časti je taktiež popísané využitie existujúceho Spark nástroj a jeho komponent a extraktorov.

6.1 Rozhranie Plasospark

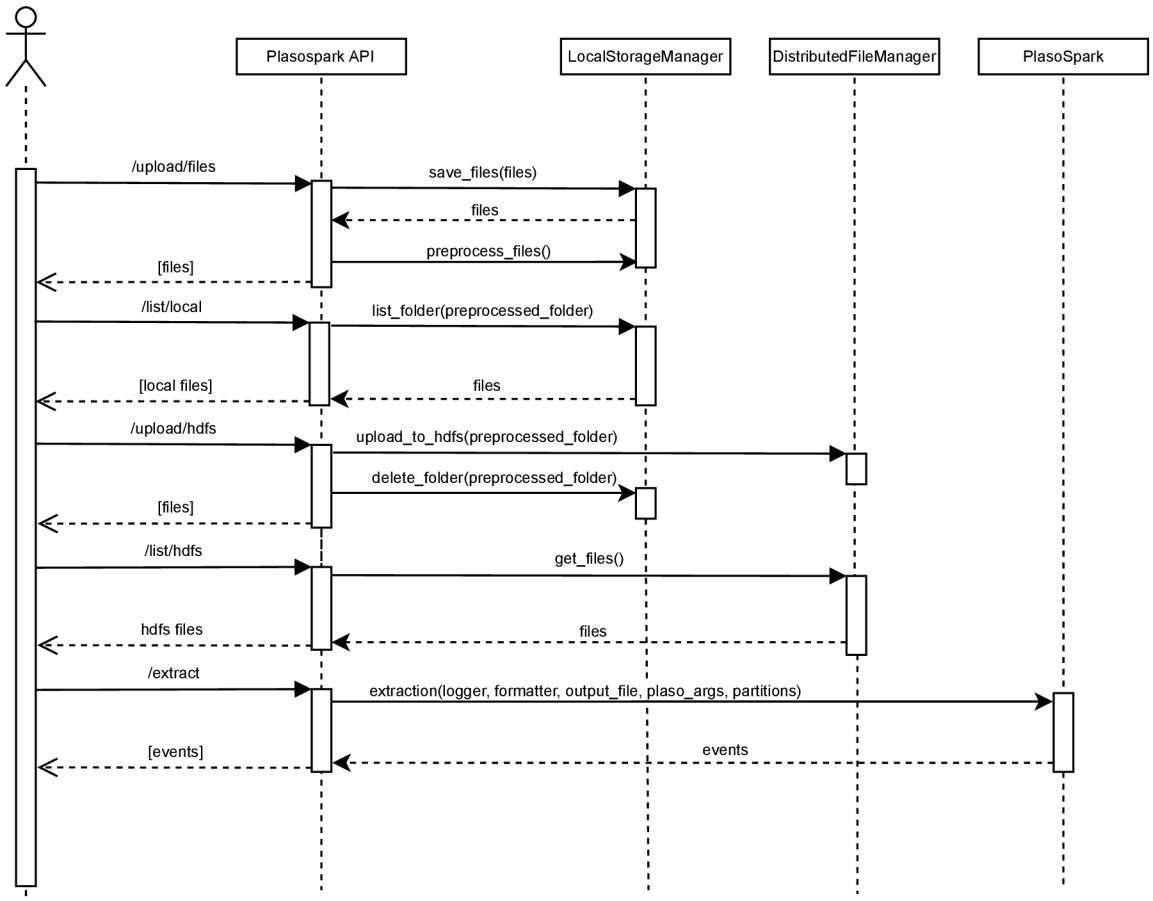
Pre ovládanie systému je dostupné API rozhranie. Implementácia rozhrania využíva python knižnicu `Flask` a zahŕňa všetky potrebné operácie pre nahratie, spracovanie a následnú extrakciu udalostí zo vstupných súborov. Pre ovládanie je dostupná Postman kolekcia spolu s popisom jednotlivých parametrov a taktiež sú k programu dostupné skripty, ktoré slúžia pre ovládanie systému na serveroch, kde nie je dostupné grafické rozhranie. Popis jednotlivých koncové body:

- **/upload/files** – Systém uloží nahrané súbory do lokálnej pamäti a predspracuje uložené súbory pred uložením do HDFS uložiska.

[1] Plasospark github repozitár `https://github.com/MiroslaviS/Distributed-Plaso-Spark`
[2] Repozitár docker obrazov `https://github.com/MiroslaviS?tab=packages`

- **/upload/hdfs** – Spustenie procesu nahrávania spracovaných súborov do HDFS uložiska.
- **/delete/hdfs** – Vymazanie všetkých súborov z HDFS uložiska.
- **/list/hdfs** – Výpis všetkých uložených súborov v HDFS uložisku.
- **/list/local** – Výpis súborov, ktoré sú lokálne uložené v systému a sú pripravené na nahratie do HDFS uložiska.
- **/extract** – Spustenie procesu extrakcie zo súborov uložených v HDFS. Výsledok extrakcie je zaslaný API rozhraním ako JSON odpoveď. V prípade využitia Plaso formátovania je výsledok uložený na disk v `.plaso` formáte. Naopak pri využití formátovania udalostí do JSON formátu sú udalosti zaslané v odpovedi z API rozhrania.

Na obrázku 6.1 je znázornený postup a využitie API rozhrania k uloženiu dát do systému. Následné uloženie dát do HDFS a spustenie extrakcie.



Obr. 6.1: Sekvenčný diagram znázorňuje využitie API rozhrania systému k príprave vstupných dát a ich uloženie. Po uložení je nad dátami spustená extrakcia.

6.2 Ukladanie a spracovanie súborov

Nasledujúca kapitola popisuje časť systému pre ukladanie a spracovanie dát. Ukladanie súborov a ich spracovanie najskôr prebieha lokálne v systéme a následne systém uloží súbory

do distribuovaného HDFS uložiska. Nasledujúca kapitola je preto rozdelená do dvoch časti popisujúce lokálne a distribuované uloženie.

Lokálne uloženie

Za lokálne spracovanie vstupnćh súborov je zodpovedná trieda `LocalStorageManager`. Táto trieda implementuje operácie, ktoré ukladajú vstupné súbory do lokálneho uložiska a predspracováva súbory pred ich uložením do HDFS uložiska.

Proces nahratia a uloženia súborov je veľmi jednoduchý. Trieda `LocalStorageManager` uloží všetky zaslané súbory pomocou API rozhrania na disk. Cestu k uloženým súborom si uloží do fronty, ktorú využíva pri nasledujúcom predspracovaní.

Pri predspracovaní sú z fronty postupne vyberané jednotlivé cesty. V prípade, že spracovávaná cesta predstavuje priečinok, tak je získaný jeho obsah a jednotlivé cesty k jeho súborom sú vložené do fronty `uploaded_files`. V opačnom prípade, kedy cesta predstavuje súbor, je riadenie predané triede `ArchiveImageHelper`. V tomto kroku trieda `ArchiveImageHelper` využíva knižnicu `dfvfs` za účelom získania typu vstupného súboru. Zo vstupnej cesty je vytvorená dfvfs špecifkácia cesty (trieda `OSPathSpec`), ktorú využíva Plaso trieda `SourceScanner` pre získanie typu spracovávaného súboru. Samotný plaso nástroj využíva rovnakú triedu pri jeho procese predspracovania na detekciu typov vstupných súborov. Týmto je docielená čo najväčšia kompatibilita medzi nástrojmi z hladiska typov súborov, ktoré môžu vstúpiť do extrakcie.

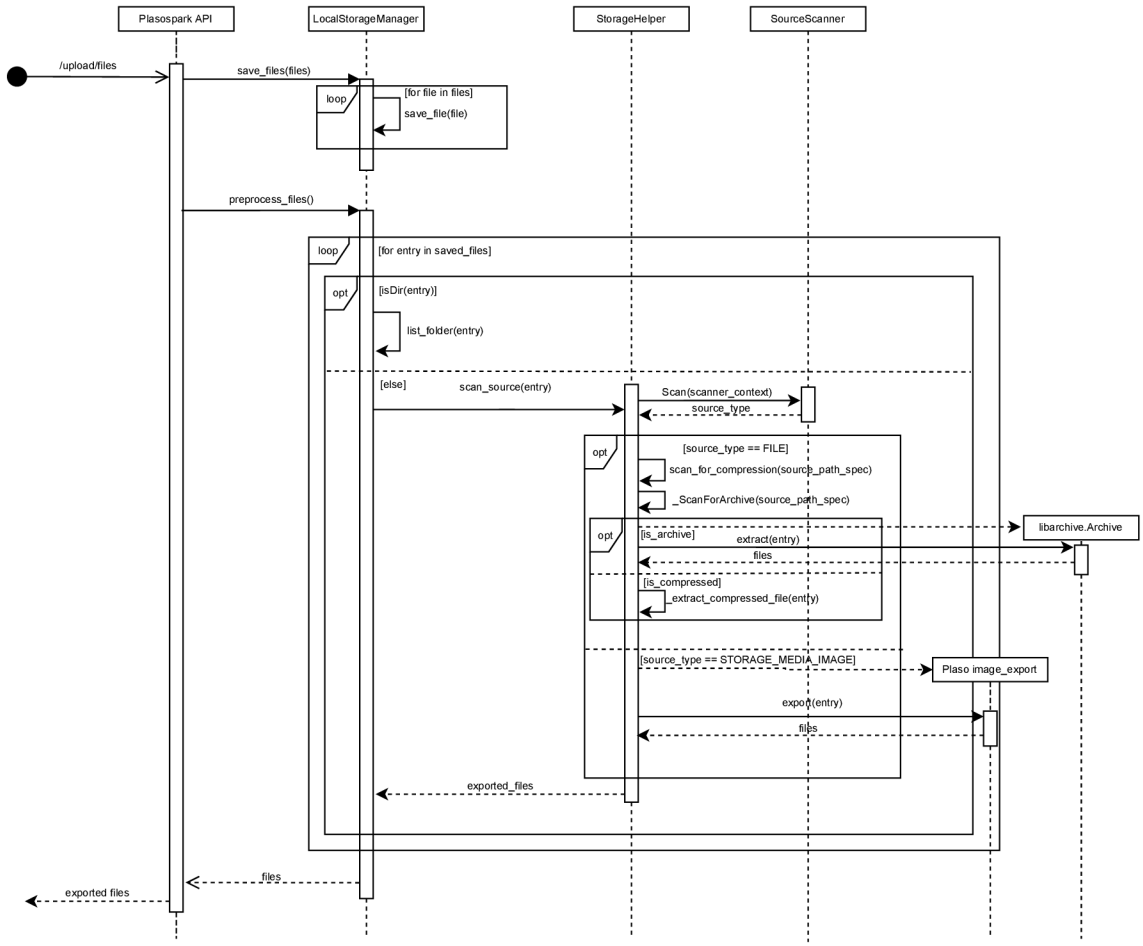
Pomocou tejto detekcie sú súbory rozdelené na dva typy: **typ súbor** alebo **typ obraz uložiska**. V pripade, že vstupný súbor je obrazom uložiska, je vyuzitý dostupný Plaso nástroj `image_export.py` z kapitoly 2.1. Pomocou tohto nástroja sú vyextrahované všetky súbory, ktoré sa nachádzajú v obraze. Cesty k súborom sú opäť uložené do spomínanej fronty pre dalšie spracovanie. Do tejto fronty sú jednotlivé súbory zaradené z dôvodu, že v obraze sa môžu nachádzať súbory, ktoré je potrebné predspracovať. Napríklad obraz môže obsahovať dalšie archívy alebo dalšie obrazy.

Ak sa nejedná o obraz uložiska, tak súbor môže stále predstavovať archív alebo komprimované dáta, ktoré je potrebné taktiež predspracovať. Na detekciu archívov a komprimovaných dát je využitá trieda `Analyzer`, ktorá je dostupná v knižnici `dfvfs`. Súbory sú z archívu vyextrahované pomocou knižnice `Libarchive` a ich obsah je uložený do fronty pre dalšie spracovanie. Tak ako pri obrazoch, tak aj pri archívoch je potrebné ich obsah znovu zaradiť do tejto fronty, kedže archív môže obsahovať dalšie archívy alebo súbory, ktoré je nutné ďalej predspracovať.

`Analyzer` trieda v prípade komprimovaných dát analyzuje vstupný súbor a detekuje typ kompresie súboru. Knižnica `dfvfs` podporuje 3 typy kompresných súborov: **bzip2**, **gzip** a **xz**. Pre všetky typy podporovaných typov súborov je implementovaná metóda pre dekomprimáciu. Nad dekomprimovaným súborom je zistené znakové kódovanie (z anglického encoding) pomocou knižnice `chardet` a dáta sú uložené do súboru. Súbor je opäť zaradený do fronty predspracovania. Dalšie spracovanie je potrebné napríklad v prípade, že dekomprimovaný súbor predstavuje archív, ktorý je nutné ďalej rozbaliť.

V prípade, že spracovávaný súbor nie je archív, komprimovaný súbor alebo obraz uložiska nie je nepotrebné dalšie spracovanie a do fronty sa tento súbor nepridáva. Predspracovanie nahraných súborov je ukončené v momente, kedy sa vo fronte pre spracovanie nenachádza žiadny súbor. Všetky predspracované súbory sú uložené do oddeleného priečinku aby bolo možné rozoznať súbory pripravené pre HDFS a súbory, ktoré je ešte potrebné pred-

spracovať. Pre ušetrenie pamäti sú všetky súbory, ktoré prejdú predspracovaním vymazané z lokálneho uložiska a sú uchované len ich spracované verzie.
Sekvenčný diagram na obrázku 6.2 znázorňuje popísaný proces predspracovania a komunikáciu s využitými Plaso komponentami.



Obr. 6.2: Sekvenčný diagram popisuje proces predspracovania nahraných súborov do lokálneho uložiska systému. Nahraté súbory sú uložené lokálne a následne sú vyexportované súbory z archívov, obrazov uložísk a komprimovaných súborov.

V prípade, že vstupný súbor nie je možné predspracovať a nastala chyba v priebehu jeho predspracovania, je takýto súbor preskočený a systém pokračuje s dalšími súbormi. Takýto scenár môže nastať napríklad v prípade, že súbor je poškodený, trieda `Analyzer` zistí nepodporovaný typ kompresie alebo nie je možné zistiť kódovanie znakov v súbore.

Distribuované uloženie

Predspracované súbory sú uložené do HDFS uložiska pomocou triedy `DistributedFileManager`. Trieda využíva implementáciu HDFS virtualného súborového systému a taktiež pomocnú triedu `Hdfs` implementujúcu všetky operácie nad HDFS uložiskom. Implementácia HDFS virtuálneho súborového systému a tried `Hdfs` je popísaná v nasledujúcej kapitole 6.3. `DistributedFileManager` pri nahrávaní súborov do HDFS uložiska postupne prechádza priečinok s uloženými predspracovanými súbormi a jednotlivé

súbory ukladá do HDFS uložiska. Pri ukladaní je zachovaná hierarchia priečinkov a súborov.

6.3 Extrakcia udalostí

Proces extrakcie využíva existujúcu implementáciu Plaso nástroja, ktorá je prevedená do Spark prostredia. Z jednotlivých Plaso funkcií sú vytvorené Spark úlohy, ktoré distribuovane vykonávajú Plaso operácie. Celá extrakcia je riadena triedou `SparkPlaso`. Táto trieda spúšťa proces celej extrakcie. Počas tohto procesu trieda vytvára potrebnú konfiguráciu Plaso nástroja, vytvára a spúšťa spark úlohy a riadi formátovanie vyextrahovaných udalostí do finálnej formy. K týmto operáciam využíva triedu `PlasoWrapper` a `SparkJobFactory`, ktoré zaobalujú operácie nad Plaso nástrojom a Spark systémom.

PlasoWrapper

Kedže systém využíva existujúcu Plaso implementáciu je potrebné pred samotnou extrakciou vytvoriť komponenty, ktoré Plaso nástroj potrebuje k extrakcii. Táto trieda je odvodená od základnej Plaso komponenty `Log2TimelineTool`. Zdedením tejto triedy je v systéme dosiahnutá kompatibilita s Plaso nástrojom a zabezpečuje zapracovanie argumentov, s ktorými je možné spustiť Plaso extrakciu. Týmto je možné spustiť extrakciu iba s definovanou sadou extraktorov rovnako ako pri využití originálneho nástroja. Po spracovaní argumentov nasleduje vytvorenie konfigurácie všetkých interných komponent Plaso . V tomto kroku sa vytvoria následovné Plaso konfigurácie a komponenty:

- **Konfigurácia spracovania** – Trieda `ProcessingConfiguration` obsahuje informácie, ktoré definujú extrakciu. Medzi tieto informácie patrí napríklad umiestnenie plaso dát potrebných pre extrakciu, filter plaso artefaktov, preferovanú časovú zónu výstupných časov udalostí, preferované kódovanie stránok (z anglického codepage). Konfigurácia je využitá ďalej pri vytváraní objektov `ParserMediator`, `EventExtractionWorker`, `EventDataTimeliner` a obsahuje výraz pre definovanie množiny extraktorov.

- **Extraktor udalostí** – Objekt `EventExtractionWorker` je vytvorený pre získanie dostupných extraktorov z Plaso nástroja. Pre jeho vytvorenie je využitá konfigurácia spracovania, ktorá popisuje ktoré extraktory môžu byť použité. Na základe tejto konfigurácie si objekt vytvorí množinu `non_sig_parsers`. Táto množina predstavuje názvy všeobecných extraktorov z 2.1. Využítím tejto triedy nie je potrebné implementovať žiadne dalšie operácie pre získanie extraktorov, ktoré môžu byť pridané do Plaso nástroja v aktualizáciach. Taktiež vďaka konfigurácií spracovania, ktorá je vytvorená zo vstupných parametrov pre Plaso je možné kontrolovať využitú množinu extraktorov.

- **Storage writer** – Zabezpečuje zápis vyextrahovaných udalostí na disk. Trieda `SQLiteStorageFileWriter` je využitá pri vytváraní kompatibilného výstupného formátu vyextrahovaných udalostí. Jednotlivé vyextrahované udalosti sú touto triedou zapisované do SQL súboru vo formáte `.plaso`, ktorý je kompatibilný s ostatnými Plaso nástrojmi z 2.1.

- **Event Timeliner** – Ďalšou použitou Plaso komponentou je trieda `EventDataTimeliner`. Spolu s triedou `SQLiteStorageFileWriter` vytvárajú

výstupný `.plaso` súbor. Úlohou tejto triedy je previesť vyextrahované surové dáta do formátu, ktorý reprezentuje trieda `EventObject`. Na základe typu jednotlivých vyextrahovaných dát sú dáta namapované do špecifikovaných atribútov, z dát je získaný extraktor, ktorý udalosť vytvoril, identifikátor udalosti a časové známky. Tieto informácie sú uložené do `EventObejct` objektu, ktorý je uložený následne do výstupného súboru pomocou `Storage Writerom`. V prípade, že systém nevyužíva Plaso výstupný format, tak táto trieda nie je použitá pri extrakcii.

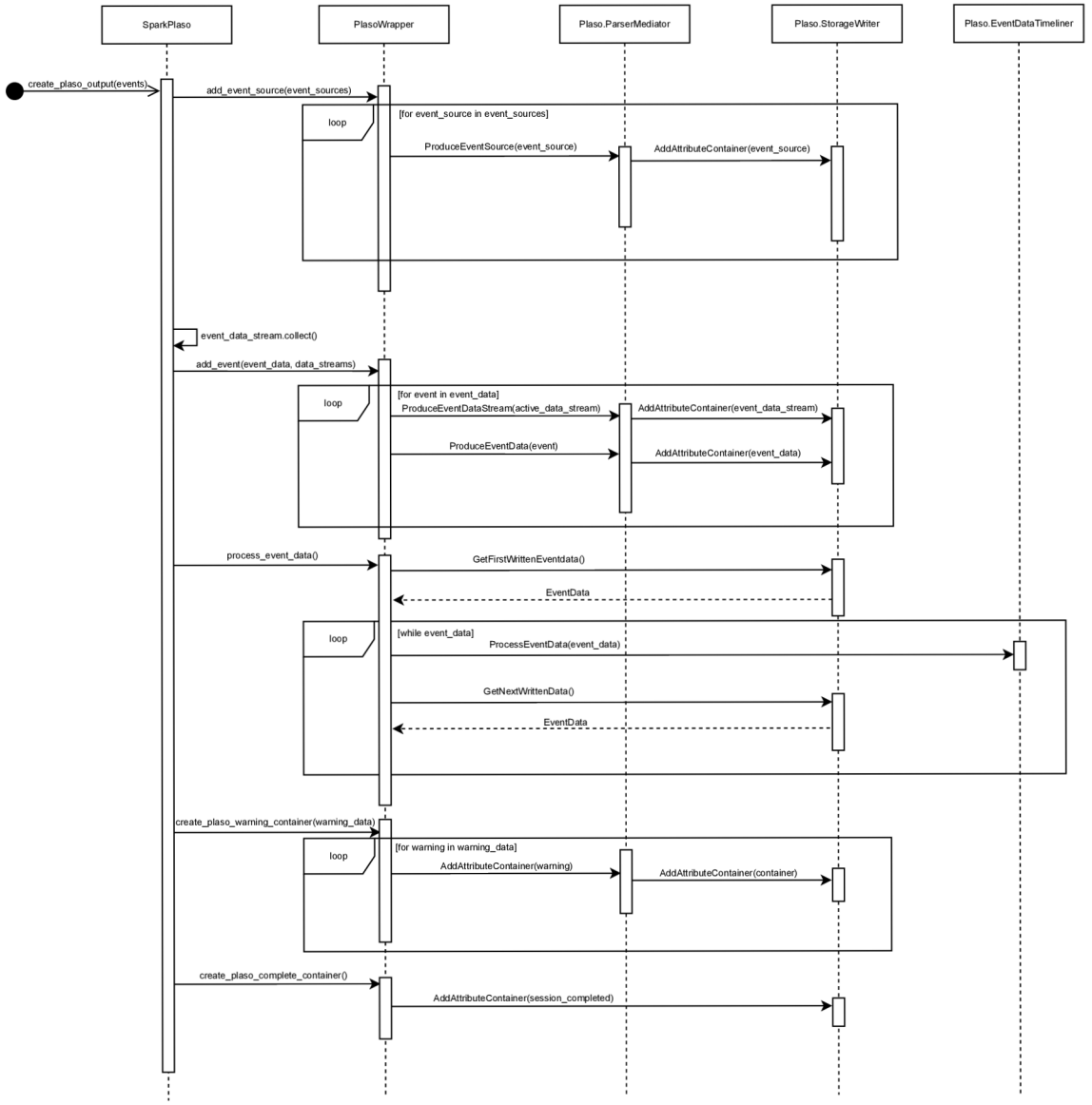
- **Mediator Holder** – Extraktory Plaso nástroja potrebujú pre správne nastavenie a fungovanie vždy k dispozícii mediator extrakcie. Preto bolo potrebné vytvoriť triedu `MediatorHolder`, ktorú je možne deserializovať a zaslať spolu so Spark úlohou. Originálna trieda `ParserMediator` v Plaso nie je deserializovateľná. Dôvodom je, že trieda `ParserMediator` napríklad obsahuje referencie na otvorený výstupný súbor. V systéme sa preto vytvorí odľahčená verzia tohto objektu, ktorý obsahuje len konfigurácie potrebné pre samotnú extrakciu. Obsahuje bázu znalostí, konfiguráciu spracovania a pomocné filtre pre proces extrakcie. Na základe týchto dát je neskôr vytvorená upravená Plasospark verzia objektu `ParserMediator` priamo v Spark úlohe.

- **Mediátor extrakcie** – Originálna verzia `ParserMediator` triedy, ktorá v systéme slúži na vytvorenie vyššie spomenutej odlahčenej verzie tejto triedy. Jej dalšie využitie je v procese formátovania výstupu do plaso formátu. Trieda slúži na komunikáciu so `Storage Writer` triedou pri zápise a vytváraní dát o extrakcii. Plasospark verzia mediátoru, ktorá je použitá v Spark extrakcii, prepisuje metódy pre zápis novo vzniknutých udalostí z extraktorov. Nezapisuje udalosti do výstupného súboru počas extrakcie, ale ukladá si všetky udalosti do fronty. Fronta obsahuje vyextrahované udalosti a je výsledkom extrakcie v Spark úlohách.

Okrem vytváraní a správy konfigurácií trieda `PlasoWrapper` slúži na formátovanie výstupných dát do Plaso formátu. Ak je systém nastavený na výstupný formát `.plaso`, táto trieda využíva spomenutý `Event Timeliner` a `Storage Writer` pre vytvorenie SQL súboru s vyextrahovanými udalosťami.Najskôr je potrebné surové dáta z extrakcie zapísať do SQL súboru.

Na začiatku celej extrakcie si trieda vytvorí plaso kontajner, do ktorého budú postupne vkladané udalosti. Pôvodne, Plaso tieto udalosti vkladá do kontajneru priamo pri extrakcií. To ale v prípade využitia Plaso nie je možné, kedže extrakcia neprebieha na rovnakom stroji ako beží samotný nástroj a prístup k tomu súboru nie je možný. V Plasospark systéme sú tieto udalosti vkladané do výstupného SQL súboru až po samotnom procese extrakcie. Po extrakcii je nutné dodatočne vytvoriť všetky objekty, ktoré by boli vytvorené už počas procesu extrakcie a zapísané do SQL súboru. Ako prvé sa vytvorí Plaso objekt `SessionStart`, ktorý značí začiatok procesu extrakcie. Tento objekt nesie informácie ako identifikátor extrakcie, verziu produktu a časovú značku začiatku. Ďalej sú vytvorené objekty `EventSource`. Tieto objekty popisujú jednotlivé zdroje udalostí v extrakcii, teda jednotlivé súbory, z ktorých boli udalosti vyextrahované. Pre zápis týchto udalostí je využitý `ParserMediator`, ktorý poskytuje `ProduceEventSource` metódu. Ďalej je potrebné dodatočne vytvoriť udalosti o dátových prúdoch (z anglického event data stream) a vyprodukovať všetky dáta o udalostiach do SQL súboru. Opäť sú použité existujúce metódy z `ParserMediator` triedy – `ProduceEventDataStream` a `ProduceEventData`. Posledné sú do SQL súboru zapísané varovania vzniknuté pri extrakciach.

Po zapísaní vyextrahovaných dát do SQL súboru nasleduje proces formátovania. Trieda `EventDataTimeliner` pri formátovaní získava dáta o udalostiach z toto súboru. Vďaka využitiu Plaso SQL súboru sú vyextrahované dáta v kompatibilnom formáte a pri formátovaní je možné využiť metódy poskytované `Storage writer` triedou. Konkrétne sa jednotlivé dáta o udalostiach získavajú pomocou metódy `GetFirstWrittenEventData` a `GetNextWrittenEventData`. Následne sú dáta sformátované pomocou Plaso tried a uložené do výstupného SQL (.plaso) formátu na disk. Po skončení formátovania je vytvorený Plaso objekt `SessionCompletion`, ktorý označuje ukončenie celého procesu extrakcie a je zapísaný do výstupného súboru. Sekvenčný diagram 6.3 znázorňuje popísaný postup jednotlivých operácií, ktoré vytvárajú Plaso výstupný súbor.



Obr. 6.3: Sekvenčný diagram popisuje proces formátovania výstupu do Plaso kompatibilného formátu. Využité sú Plaso komponenty, ktoré vytvárajú SQL výstupný súbor.

Plaso nástroj využíva virtuálny súborový systém implementovaný knižnicou `dfvfs`. Táto trieda poskytuje informácie a operácie nad jednotlivými súbormi v súborovom systéme. Aby bolo možné využiť v Plaso súbory uložené v HDFS bolo potrebné rozšíriť túto knižnicu o HDFS súborový systém. Rozšírením tejto knižnice je dosiahnutá kompatibilita Plaso komponent s novým typom súborom dostupných z HDFS. HDFS súborový systém je implementovaný do knižnice `dfvfs` pomocou následujúcich tried:

- **HDFSPathSpec** – Definícia špecifikácie cesty v HDFS. Obsahuje umiestnenie konkrétneho súboru v HDFS a slúži pre vytváranie nasledujúcich objektov súborového systému.
- **HDFSFileEntry/HDFSDirectory** – Triedy, ktoré popisujú jednotlivé súbory/priečinky v súborovom systéme. Poskytujú informácie o umiestnení súboru v súborovom systéme, typ súboru (súbor alebo priečinok), informácie o veľkosti súborov, prístupovom čase k súborom a názve súboru. Knižnica `dfvfs` poskytuje o súbore aj dodatočné informácie ako čas vytvorenia súboru, čas zmeny súboru alebo čase modifikácie súboru. Tieto informácie pre HDFS nie sú k dispozícii, kedže knižnica `pyarrow`, ktorá je využitá pre prístup k HDFS súborom, tieto informácie neposkytuje.
- **HDFSFile** – Implementuje operácie nad súborom v HDFS. Operácie, ktoré poskytuje `dfvfs` nad súborom sú otvorenie a zavretie súboru. Ďalej operácie pre čítanie súboru a operáciu vyhľadávania (z anglického seek).
- **HDFSFileSystem** – Popisuje celý súborový HDFS systém. Implementuje operácie pre otvorenie a zavretie súborového systém, čo v HDFS prípade znamená vytvorenie spojenia pomocou `pyarrow` a jeho ukončenie. Ďalej poskytuje operácie pre zistenie existencie súboru podľa cesty, získanie objektu `HDFSFileEntry` na základne špecifikácie cesty `HDFSPathSpec`. Poskytuje taktiež základne operácie vytvorenie súboru, vymazanie alebo vytvorenie zoznamu so všetkými súbormi v HDFS súborovom systéme.
- **HDFSResolverHelper** – Slúži ako pomocná trieda pre súborový systém, ktorej hlavná úloha je vytváranie `HDFSFile` objektov z HDFS súborového systému a vytváranie nového objektu HDFS súborového systému `HDFSFileSystem`.

Spolu s týmito trieda je vytvorená pomocná trieda `Hdfs`, ktorá implementuje operácie nad HDFS uložiskom pomocou knižnice `pyarrow`. Pomocou tejto triedy je vytvorené spojenie s HDFS uložiskom a implementuje operácie nad týmto uložiskom potrebné pre virtuálny súborový systém `HDFSFileSystem`.

SparkJobFactory

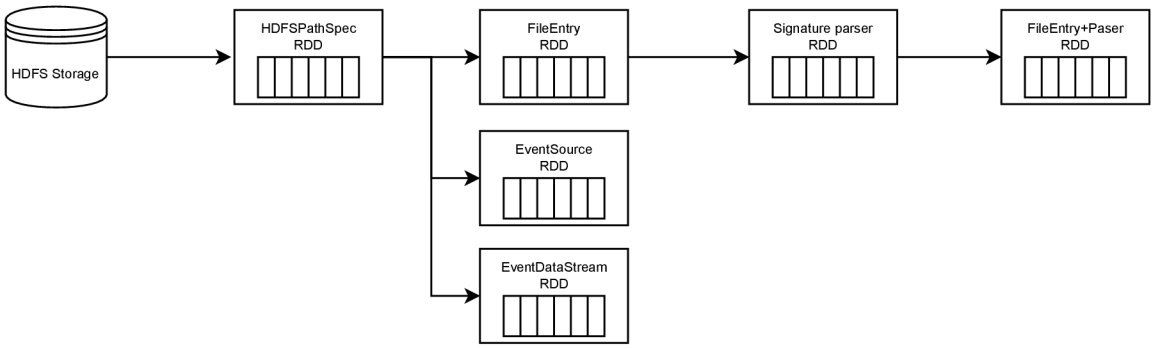
Systém komunikuje so Spark uzlami za pomoci triedy `SparkJobFactory`. Táto trieda zabezpečuje vytváranie výpočetných úloh a zasielanie úloh Spark Master uzlu. Pre ovládani a komunikáciu so Spark prostredím je využitá knižnica `pyspark`, ktorá sprístupňuje rozhranie k Spark komponentom. `SparkJobFactory` vytvára na začiatku Spark kontext, kory slúži na komunikáciu so Spark clusterom. Po vytvorení kontextu sú do Spark clusteru zaslané potrebné závisloti, ktoré sú využité pri extrakcii. Závilosti predstavujú nasledujúce balíčky (z anglického packages), ktoré sú zaslané v `zip` formáte:

- **Helpers** – Sprístupňuje jednotlivé spark skripty, ktoré sú spúštané pri procese extrakcie.

- **Mediators** – Obsahuje upravené triedy mediátorov pre extrakciu, ktoré využívajú Spark úlohy.
- **Formatters** – Zahŕňa dostupné triedy pre formátovanie výstupov v Spark prostredí.

Po zaslaní závilostí trieda sprístupňuje operácie, ktoré využíva trieda `SparkPlaso` v procese extrakcie. Trieda zabezpečuje vytvorenie mapovacích funkcií nad `RDD` štuktúrami. Trieda nespravuje RDD štruktúry, ale slúži na zreťazovanie RDD štruktúr pomocou mapovacích funkcií, v ktorých sa využívajú Spark skripty z `Helpers` balíčku.

Základné operácie, ktoré trieda sprístupňuje, predstavujú vytvorenie jednoduchých mapovacích funkcií, ktoré transformujú postupne Plaso objekty do dalších Plaso objektov spojených s extrakciou a spustenie samotného procesu extrakcie s konkrétnym extraktorom a súborom. Na obrázku 6.4 je vidieť postupnú transformáciu Plaso objektov pri extrakcii v RDD štruktúrach.



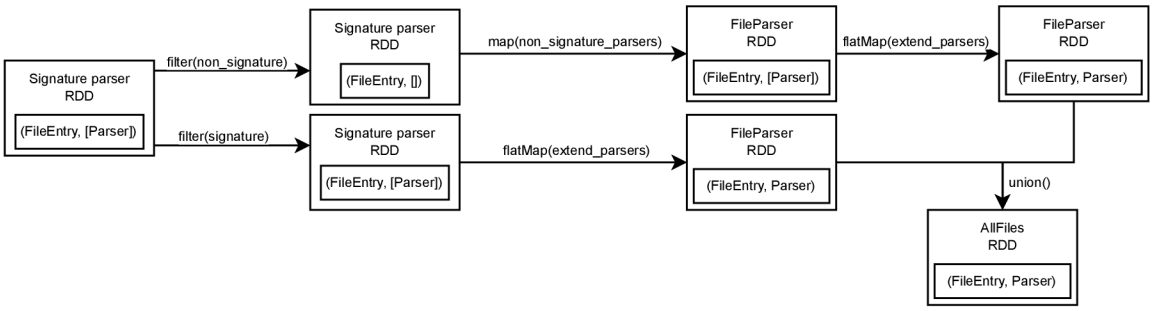
Obr. 6.4: Diagram znázorňuje postupnú transformáciu jednotlivých Plaso objektov počas celej extrakcie. Na začiatku sú zo vstupných súborov vytvorené objekty `HDFSPathSpec`. Tieto objekty tvoria základnú komponentu pre vytvorenie dalších Plaso objektov ako `FileEntry`, `EventSource`, `EventDataStream`. Ďalej z `FileEntry` sú vypočítané podpisy jednotlivých súborov a vytvorené príslušné extraktory.

Medzi zložitejšie operácie, ktoré poskytuje táto trieda patrí napríklad operácia `create_signature_parsers`. Táto metóda slúži na vytvorenie dvojíc tried `(FileEntry, [ParserName])`. Dvojica popisujuje vstupný súbor a množinu názvov jednotlivých extraktorov, ktoré je možné na tento súbor aplikovať.

Z vytvorených Plaso konfigurácií je získaná konfigurácia extraktorov, ktorá je zaslaná do všetkých Spark procesov (z anglického worker) pomocou `broadcast` metódy. Následne je nad RDD, ktoré obsahuje Plaso objekty `FileEntry`, spustená Spark úloha so skriptom `get_signature_parser`. Skript pozostáva z vytvorenia `FileObject` objektu zo vstupného `FileEntry` a vytvorenia objektu extraktoru dát udalostí. Extraktor dát udalostí predstavuje Plaso objekt `EventDataExtractor`. Extraktor udalostí na základe podpisu z daného súboru vytvorí množinu možných extraktorov, ktoré je možné aplikovať na súbor pre získanie udalostí. V prípade, že na základe podpisu súboru nie je možné získať žiadny konkrétny extraktor je množina `[ParseName]` prázdna.

Ako už bolo spomenuté v 2.2.2, Plaso nemusí vždy nájsť vhodný extraktor pre súbor na základe podpisu. V takom prípade je využitá všeobecná množina extraktorov. Všeobecná množina extraktorov nie je získaná z predchádzajúcej operácie a je potrebné túto množinu priradiť k jednotlivým súborom. `SparkJobFactory` implementuje operáciu `filter_signature_parsers`, ktorá vykoná priradenie všeobecnej množiny k súboru. Vše-

obecná množina extraktorov je získaná z Plaso konfigurácie a táto množina je rozoslaná do Spark clusteru opäť pomocou `broadcast` metódy. Ďalej sa vytvorí filter nad RDD s dvojicami `(FileEntry, [ParserName])`, ktorý získa vstupné súbory, na ktoré je nutné aplikovať všeobecnú množinu extraktorov a priradí týmto dvojiciam všeobecné extraktory. Následne je nad týmito dvojicami spustená `flatMap` operácia so skriptom `expand_file_parsers`. Skript transformuje vstupnú dvojicu do množiny `(FileEntry, ParserName)`. Táto transformácia je využitá pre účely správneho paralelizmu pri extrakcii. Keby táto transformácia nebola vykonaná, tak by jednotlivé Spark procesy spúštali extrakciu s množinou extraktorov namiesto jedného extraktoru pre súbor, čo by ovplyňovalo schopnosť paralelizmu v Spark prostredí. Pre lepšiu predstavuje je na obrázku 6.5 znázornená transformácia získaných extraktorov podľa podpisu súboru do dvojíc `(FileEntry, Parser)`.



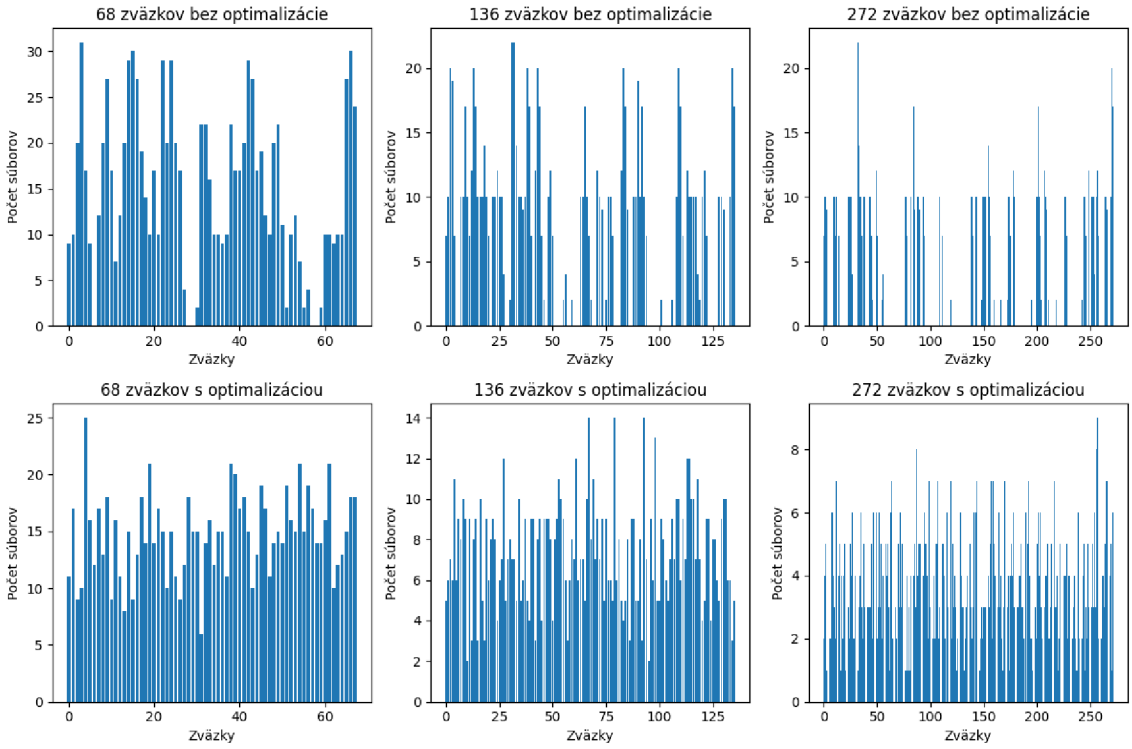
Obr. 6.5: Transformácia RDD obsahujúcich výsledok operácie pre získanie extraktorov na základe podpisu do koncového formátu pred extrakciou.

`SparkPlaso` využíva tieto triedy k riadeniu celého procesu extrakcie. Celý proces začína získaním súborov, ktoré sú uložené v `HDFS` uložisku. Prístup k distribuovanému uložisku mu umožňuje trieda `DistributedFileManager`. Vstupné súbory sú transformované do objektov `HDFSPathSpec`. Následne sú zo špecifikácií ciest vytvorené objekty `HDFSFileEntry`. V prípade, že je zvolený Plaso výstupný formát sú špecifikácie ciest využité na vytvorenie dodatočných objektov `FileEntryEventSource` a `EventDataStream`. Tieto objektu sú ďalej použité pri formátovaní a vytváraní výstupného Plaso formátu.

Po transformácií vstupných súborov následuje získanie extraktorov, ktoré budú ďalej aplikované na jednotlivé súbory a vytvorenie RDD s dvojicami `(Parser, FileEntry)`. V systéme je možné nastaviť voliteľný parameter `partitions`, ktorý pred samotným krokom extrakcie zmení počet zväzkov (z anglického partitions), do ktorých je rozdelené RDD so spomenutými dvojicami. Spark automaticky vytvára počet zväzkov podľa dostupných zdrojov v clusteri. Avšak počas implementácia a testovania sa ukázalo, že Spark vstupné dáta neefektívne rozdeluje do zväzkov a preto bol pridaný parameter, ktorý dodatočne rozdelí dáta do špecifikovaného počtu zväzkov. Spôsob, akým tento parameter ovplyvňuje efektívnosť extrakcie je zobrazené v kapitole 7.1. V prípade, že parameter nie je nastavený je extrakcia spustená s počtom zväzkom, ktorý automaticky nastavil Spark.

Okrem zmeny počtu zväzkov je v systéme implementovaná dalšia optimalizácia pre optimálne rozdelenie počtu dvojíc v zväzkoch. Spark sa automaticky snaží rozdeliť vstupné dáta do zväzkov na základe vytvoreného kľúča. V prípade dvojice `(Parser, FileEntry)` je tento kľúč názov extraktoru. Toto rozdelenie sa ukázalo ako veľmi neefektívne z dôvodu, že veľmi často tento kľúč nebol dostatočne unikátny a veľká časť zväzkov neobsahovala ani jednu dvojicu a iné zväzky obsahovali veľký počet vstupných dvojíc. Tento efekt spôsobil,

že extrakcia neprebiehala rovnomerne na každom Spark procese, ale často úlohy extrakcie vykonávalo len niekoľko Spark procesov. Preto systém implementuje vlastnú metódu tvorby kľúča pri zmene zväzkov. Kľúč sa netvorí z názvu extraktoru, ale kľúč predstavujú pseudo náhodné číslo. Toto náhodné číslo je vybrané z rovnomerného rozloženia, čím sa zabezpečí, že počet vstupných dvojíc je rovnomerne rozložený medzi všetky zväzky. Takto generované kľúče zaručia, že extrakcia nebude prebiehať len v niektorých úlohách, ale úlohy budú mať rovnomernú náročnosť výpočtu. Na obrázku 6.6 je vidieť rozloženie vstupných súborov pred optimalizáciou a po optimalizácií s rôznym počtom rozdelenia dát do zväzkov. Ďalšie vyhodnotenie tejto optimalizácie je zhodnotené v kapitole 7.1.



Obr. 6.6: Počet súborov v jednotlivých zväzkoch pred a po využití optimalizácie. Najmä s väčším počtom zväzkov je vidieť, že dáta sú nerovnomerne rozložené v zväzkoch. Zatiaľ čo pri využití klúču z rovnomerného rozloženie je rozdelenie viditeľne lepšie a výpočty neprebiehajú len v zopár úlohách.

Po vykonaní optimalizácií nasleduje proces samotnej extrakcie v Spark úlohách. Počiatok extrakcie spočíva vo vytvorení potrebných objektov, ktoré sa podielajú na extrakcii. Požadovaný extraktor sa získa z Plaso komponenty `ParsersManager`. Tento manažér spravuje všetky registrované extraktory, ktoré sa v Plaso nachádzajú a na základe názvu extraktoru vytvorí požadovný extraktor. Následne sa vytvorí upravená verzia mediátoru extrakcie `ParserMediator`, ktorá je vytvorená na základe triedy `MediatorHolder`. Mediátoru sa nastaví vstupný súbor a proces extrakcie preberá samotný extraktor. Celý proces extrakcie je zaobalený v `try-except` bloku. Tento `try-except` blok je použitý z dovôdu využitia všeobecných extraktorov. V prípade, že takýto extraktor nepozná spracovávaný súbor ukončí extrakciu s výnimkou. Vyextrahované udalosti sú na konci extrakcie získané z fronty `ParserMediator` objektu a sú zaslané ako výsledok úlohy.

Formátovanie

Po skončení procesu extrakcie je výsledné dáta potrebné sformátovať. Prvou možnosťou, ktorú je možné v systéme využiť je Plaso formátovanie. Toto formátovanie zabezpečuje spätnú kompatibilitu s existujúcimi Plaso nástrojmi ako napríklad `pinfo` alebo `psort`. Formátovanie do tohto formátu zabezpečuje trieda `PlasoWrapper`. Popis formátovania touto triedou je popísaný v 6.3 a taktiež využité Plaso komponenty, ktoré sú využité. Proces Plaso formátovania neprebieha v Spark prostredí, kedže Plaso komponenty vyžadujú zápis do jedného súboru vytvoreným na začiatku extrakcie.

Druhou možnosťou, ktorú systém ponúka je využitie vlastného formátovania. Aktuálne je v systéme implementované formátovanie výstupných dát do JSON formátu. Formátovanie je zabezpečené `JsonFormatter` triedou, ktorá implementuje metódy rozhrania `Formatter`. Dané rozhranie obsahuje metódu `format`, ktorá je využitá v poslednom kroku Spark spracovania. Trieda `FormatterManager` spravuje triedy pre formátovanie a taktiež zabezpečuje jednoduchý spôsob pridania nových formátovaní do systému. Pre pridanie nového formátovania stačí vytvoriť novú triedu a zaregistrovať ju pomocou `FormatterManager` a jeho metódy `register`. Pre vybranie formátovania v extrakcii je možné nastaviť API paramater `formatter`, ktorý odpovedá názvu konkrétneho formáteru. Názov formátovacej triedy je definovaný v atribúte `NAME` danej triedy formáteru.

Kapitola 7

Vyhodnotenie a testovanie

Nasledovná kapitola sa zaoberá testovaním a vyhodnotením implementovaného systému **Plasospark**. Vyhodnotenie prebieha na dátovej sade, ktorá je poskytnutá autormi Plaso nástroja a je dostupná so zdrojovými kódmi tohto nástroja. Táto dátová sada obsahuje 431 súborov. V sade sa nachádzajú súbory, archívy a obrazy uložísk. Po rozbalení všetkých archív a obrazov pozostáva dátová sada zo 488 súborov. Testovacia sada pokrýva všetky dostupné Plaso extraktory a všetky formáty, ktoré je Plaso schopné spracovať. Z celej testovacej sady bolo vytvorených 5 testovacích skupín aby bolo možné testy jednoducho pomenovať a výsledky boli prehľadnejšie.

Vyhodnotenie prebiehalo na školských serveroch `grace1.fit.vutbr.cz` a `grace2.fit.vutbr.cz`. Konfigurácia strojov je v oboch prípadoch rovnaká a predstavuje:

- 64x CPU Intel (R) Xeon (R) Silver 4314 CPU @ 2.40GHz
- 251GiB RAM

Rozdelenie testovacej sady predstavuje 5 testovacích balíkov, z ktorých prvé tri majú každá iné súbory a posledné dve vznikli zlúčením všetkých troch predchádzajúcich. Jednotlivé testovacie balíky sú:

- **test_multi1** – Predstavuje dátovú sadu pozostávajúcu zo 156 vstupných súborov. Táto sada pozostáva zo súborov z kategórie docker súborov, systemd súborov, chrome/firefox cache súborov, Windows NT registry súborov.
- **test_multi2** – Dátová sada predstavuje 189 súborov, ktoré pozostávajú z obrazov disku, archívov, programové súbory vo formáte .DAT a súbory databáz.
- **test_multi3** – Posledna dátová sada, ktorá obsahuje 143 súborov. Tieto súboru sú napríklad súbory záznamov (z anglického logfile), plist súboru, JSON-L súbory a mix rôznych súborov.
- **test_multi4** – Kombinácia súborov **test_multi1** a **test_multi2**. Obsahuje 345 súborov.
- **test_multi5** – Všetky súbory testovacej sady v jednom teste. Obsahuje 488 súborov.

Vyhodnotenie je rozdelené do štyroch častí, ktoré boli podstatné pre vytvorený systém. V prvej časti je vyhodnotená optimalizácia, ktorá bola do systému pridaná po prvotnom

testovaní systému a objavení nedostatku rozloženia súborov do Spark zväzkov. Dalšia časť skúma efekt počtu zväzkov na rýchlosť extrakcie a optimálny počet zväzkov, do ktorých je extrakcia rozdelená. Tretia časť sleduje akým spôsobom sa mení čas extrakcie pri postupnom pridávaní prostriedkov pre Spark procesy. Posledná časť porovnáva rýchlosť vytvoreného systému oproti originálnej verzii Spark. Okrem rýchlosti je sledovaný aj počet udalostí, ktoré **Plasospark** a výsledok je porovnaný s originálnym nástrojom.

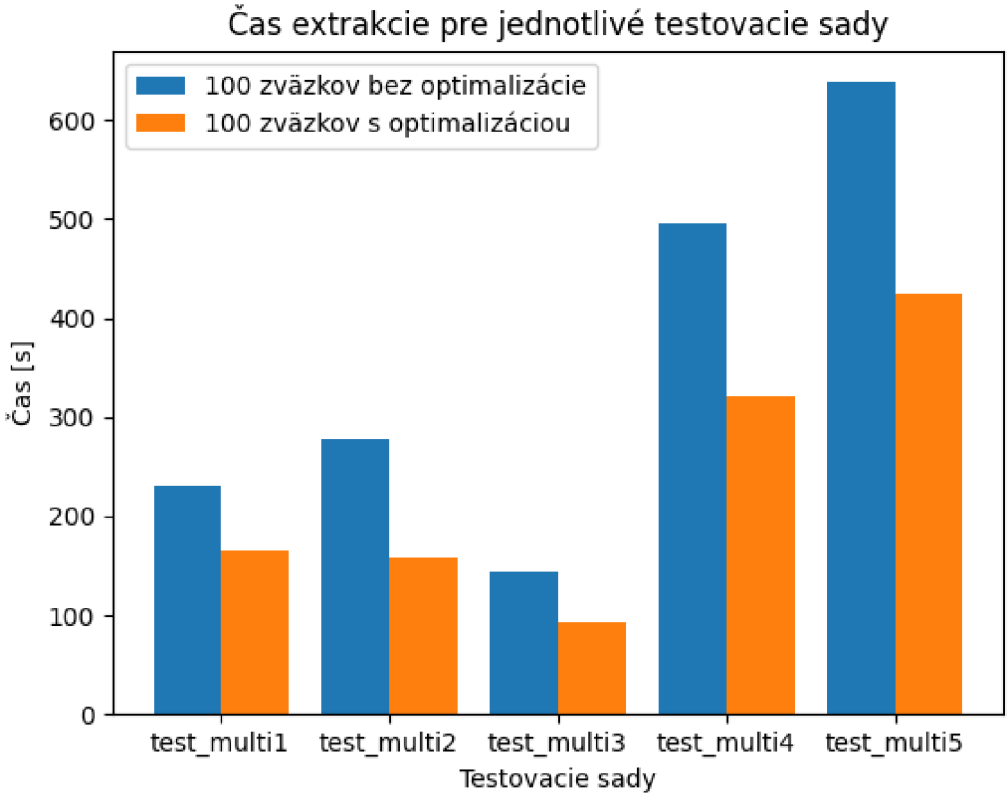
7.1 Testovanie optimalizácie zväzkov

Prvé vyhodnotenie je zamerané na vytvorenú optimalizáciu počtu vstupných súborov v Spark zväzkoch. Ako je ukázané v kapitole 6.6, počet súborov v jednotlivých zväzkoch nie je optimálny pri využití natívnej rozdelovacej metódy, ktorú poskytuje Spark. Overenie, že je optimalizácia funkčná prebieha spustením extrakcie nad testovacími sadami bez voliteľného parametru `partitions` a následne je extrakcia spustená s týmto parametrom. Bez parametra `partitions` systém prenechá réžiu rozdelenia súborov Spark logike. Spark vytvorí dvojnásobný počet zväzkov ako je v systéme dostupných CPU. Pri využití optimalizácie je pred samotnou extrakciou použitá Spark metóda `repartitionBy`, ktorá rozdelí súbory do požadovaného množstva zväzkov s novo vytvoreným kľúčom popísanej v 6.3.

Konfigurácia Spark uzlov a HDFS uzlov pre tento testovací scenár je nasledovný:

- **Spark** – 10x Spark pracovný uzlov (z anglického worker node), každý uzol má k dispozícií 5xCPU a 5GB RAM
- **HDFS** – 4x Dátový uzol, 1x menný uzol

Na grafe 7.2 sú zobrazené výsledky časov extrakcie pre jednotlivé testovacie sady. Počet zväzkov je v tomto teste ponechaný na dvojnásobok dostupných CPU v Spark clusteri, tak ako Spark automaticky nastaví pri spúštaní extrakcie s dostupnými zdrojmi. S tymto počtom zväzkov je následne spustená extrakcia, ktorá využíva optimalizáciu rozdelenia súborov.



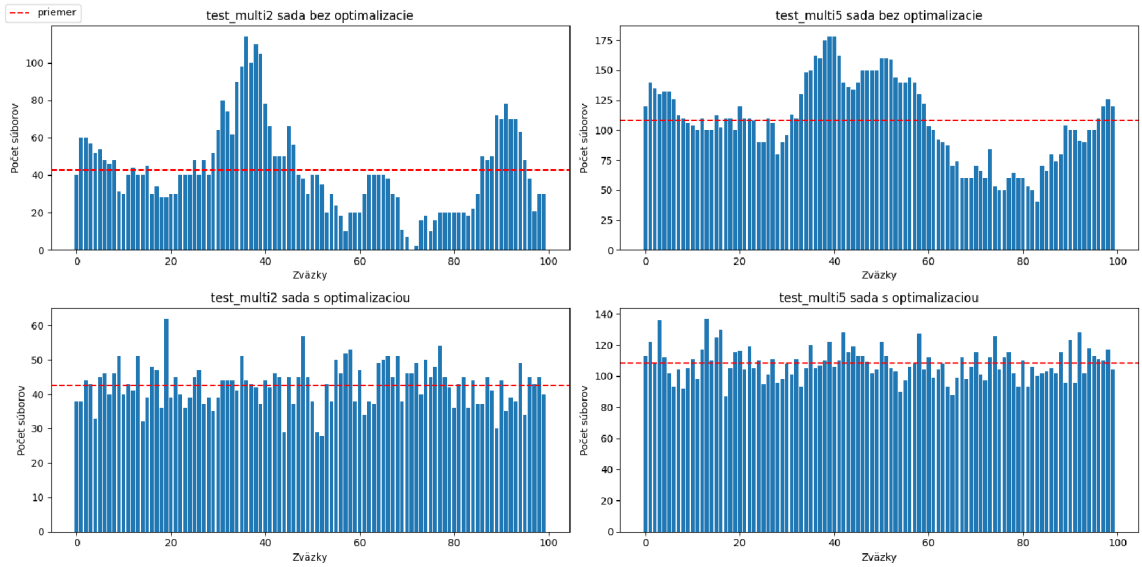
Obr. 7.1: Graf zobrazuje čas extrakcie pri využití optimalizácie rozloženia vstupných súborov do zväzkov a čas extrakcie pri nepoužití tejto optimalizácie.

Z grafu je vidieť, že rýchlosť extrakcie sa aj bez zmeny počtu zväzkov zväčšila. V tabuľke 7.1 je zobrazené o koľko sa extrakcia jednotlivých testovacích sád zrýchlila oproti pôvodnej rýchlosti. Priemerne sa rýchlosť zvýšila o $34,2\%$, čomu zodpovedá aj hodnota zrýchlenia v poslednej testovacej sade `test_multi5`, ktorá obsahuje všetky súbory z dátovej sady.

Testovacia sada	**multi1**	**multi2**	**multi3**	**multi4**	**multi5**
zrýchlenie [%]	27	42	34	35	33

Tabuľka 7.1: Zrychlenie

Dôvodom zrýchlenia pri využití optimalizácie je rovnomernejšie rozloženie vstupných súborov do zväzkov. Na grafe 7.2 je zobrazené rozloženie súborov v testovacej sade `test_multi2`, v ktorej bolo zrýchlenie až 42% a testovacej sady `test_multi5`, ktorá predstavuje celú dátovú sadu.



Obr. 7.2: Rozloženie súborov do Spark zväzkov pri spustení extrakcie. Priemer predstavuje optimálnu hodnotu súborov, ktoré sa majú nachádzať v každom zväzku.

V testovacej sade `test_multi2` je až 53% všetkých vstupných dát v 32 zväzkoch (zväzky 25-50 a 82-95). V ostatných 68 zväzkoch je zvyšných 47% dát. Po aplikovaní optimalizácie je vidieť, že obsah zväzkov je rozdelený rovnomerne. Pri extrakcii celej dátovej sady `test_multi5` je rozloženie dát bez použitia optimalizácie taktiež nerovnomerné. V tomto prípade je až 40% vstupných dát v 30 zväzkoch (zväzky 30-60). Opäť po aplikovaní optimalizácie je rozloženie rovnomerné medzi všetky zväzky.

Z výsledkov je vidieť, že optimalizácia funguje a efekt skreslenia dát (z anglického data skew) je optimalizáciou odstránený. V následujúcich vyhodnoteniach je táto optimalizácia vždy použitá pre dosiahnutie čo najlepších výsledkov.

7.2 Vyhodnotenie optimálneho počtu zväzkov

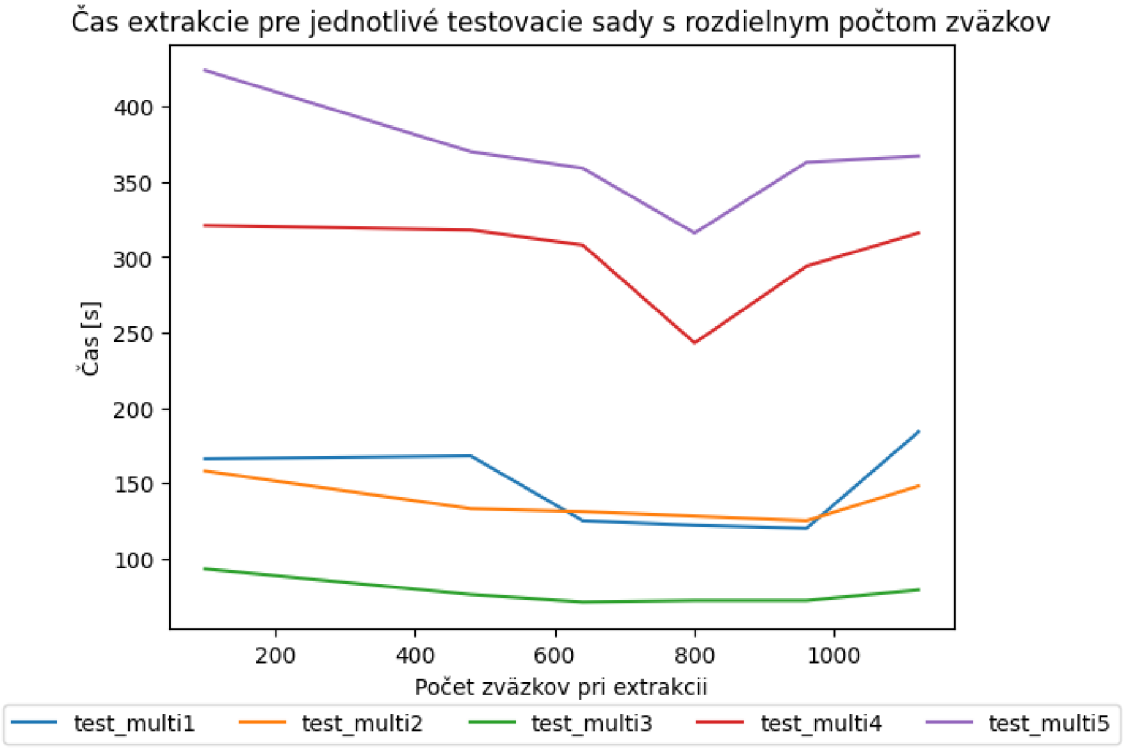
V Spark aplikáciach je počet zväzkov, do ktorých je rozdelená vstupná dátová sada dôležitým faktorom. V prípade, že zväzkov je príliš málo môže dôjsť k zníženiu efektivity paralelizmu alebo nevyužitiu všetkých zdrojov. Naopak pri využití veľkého počtu zväzkov môže nastať moment, že plánovanie úloh môže trvať dlhšie ako samotná úloha. Počet zväzkov, do ktorých sú dáta rozložené môže nastaviť automaticky Spark alebo je možné túto hodnotu zmenit. Nasledujúca časť sa zameriava na nájdenie optimálneho počtu zväzkov pre testovacie sady. Taktiež sa v tejto časti sleduje aký efekt má zmena zväzkov na celkový čas extrakcie. Pre zistenie efektu a optimálneho počtu zväzkov boli spustené extrakcie s postupným zväčšovaním počtu zväzkov.

Pre tieto experimenty bola konfigurácia Spark uzlov a HDFS uzlov následovná:

- **Spark** – 10x Spark pracovný uzlov (z anglického worker node), každý uzol má k dispozícií 5xCPU a 5GB RAM
- **HDFS** – 4x Dátový uzol, 1x menný uzol

Graf 7.3 znázorňuje výsledky efektu zvyšovania počtu zväzkov pre jednotlivé testovacie sady. Spark má dostupných 50 CPU a preto počiatočný počet zväzkov bol stanovený na 100.

Následné je počet zväzkov nastavený na väčšie hodnoty a to konkrétne: 480, 640, 800, 960, 1120.



Obr. 7.3: Efekt zmeny zväzkov na celkový čas extrakcie. Jednotlivé testovacie sady sú ovplyvnené počtom zväzkov a čas extrakcie je odlišný pri každej zmene. Najväčší rozdiel je vidieť pri najväčších testovacích sadách `test_multi4` a `test_multi5`.

Z výsledkov je vidieť, že postupným zväčšovaním počtu zväzkov sa čas extrakcie zmenšuje ale v niektorých prípadoch je vidieť, že väčší počet zväzkov má negatívny vplyv na čas extrakcie. Pri testovacie sade `test_multi3` je efekt narastajúceho počtu zväzkov zanedbateľný taktiež ako pri sade `test_multi2`. Efekt pri testovacej sade `test_multi1` je pri zvyšovaní v rádovo sekundách až do zlomoveho bodu po 480 zväzkoch. Následne je možné vidieť zrýchlenie o približne 25%. Testovacie sady `test_multi4` a `test_multi5` postupným zvyšovaním zväzkov získavajú na rýchlosti. V týchto dvoch sadách je vidieť, že väčší počet zväzkov pri väčšom počte súborov zlepšuje čas extrakcie.

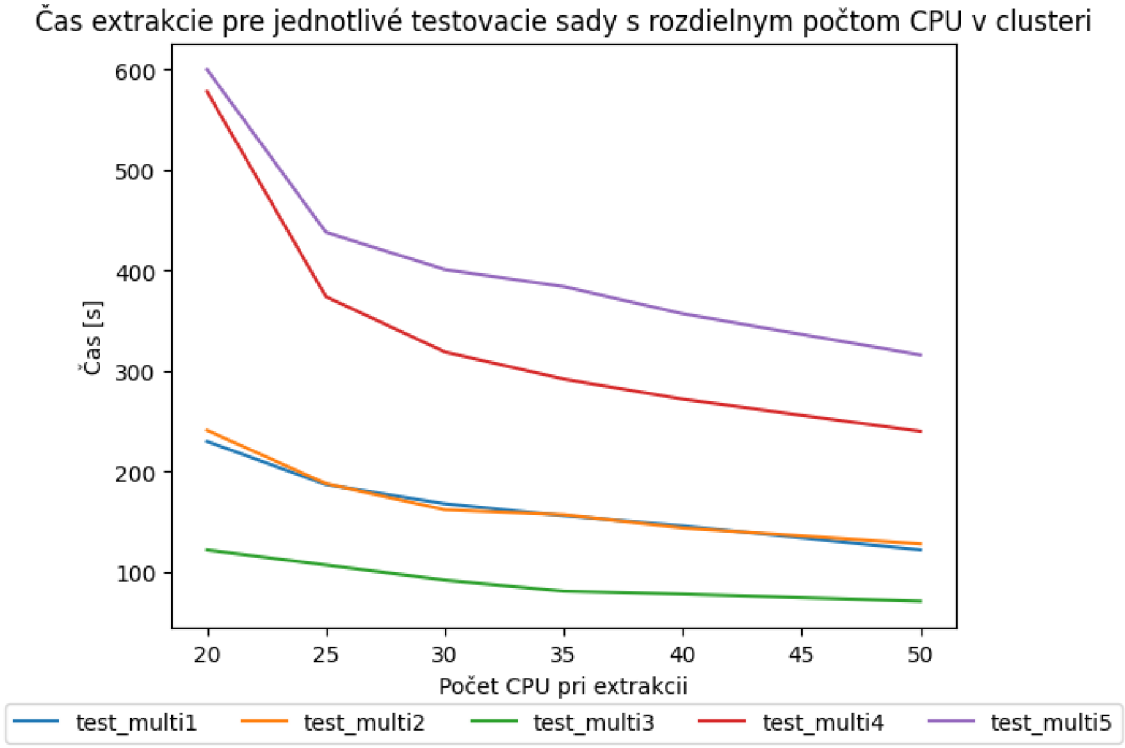
Na základe výsledkov je možné skonštatovať, že optimálny počet zväzkov pre všetky testovacie sady sa pohybuje okolo počtu 800. Po prekročení tejto hranice je vidieť, že čas extrakcie sa nezmenšuje, ale naopak zvyšuje. Spark úlohy obsahujú malý počet dát a ich plánovanie v Sparku zaberá väčší čas ako pri menšom počte zväzkov.

7.3 Vyhodnotenie škálovateľnosti

Poslednou skúmanou vlasnosťou v systéme je schopnosť škálovateľnosti. Kedže systém využíva distribuovaný výpočetný model Spark, je potrebné overiť, či implementovaný systém je schopný využiť nové výpočetné uzly pridané do Spark clusteru. Každý pridaným výpočetným uzlom by sa mal čas extrakcie zmenšovať, kedže Spark úlohy môže paralelene riešiť via-

cero výpočetných uzlov. Na dostupnom serveri `grace1.fit.vutbr.cz` a `grace2.fit.vutbr.cz` je maximálne dostupných 64 CPU. Systém bol spustený s maximálne 50 CPU pre Spark a ostatné zdroje boli ponechané aplikácií a HDFS uzly, ktoré boli na serveri taktiež spustené.

Na základe zistenia z kapitoly 7.2 boli všetky extrakcie spúšťané nad dátovými sadami s počtom zväzkov 800 pre dosiahnutie optimálneho výsledku. Každý nový Spark pracovný uzol, ktorý je do systému pridaný počas testovanie má konfiguráciu: 5xCPU a 5GB RAM.



Obr. 7.4: Škálovanie systéme pri postupom pridávaní nových pracovných Spark uzlov do systému. Z grafu je vidieť, že čas extrakcie sa s pridávanými uzlami zmenšuje, čo dokazuje správnosť škálovania systému.

Z výsledného grafu 7.4 je možné konsťatovať, že systém spĺňa vlasnosť škálovateľnosti. Každým pridaným výpočetným uzlom je rýchlosť extrakcie rýchlejšia. Ako je možné z grafu ďalej vidieť, najväčší nárasť výkonu je dosiahnutý pri počiatočnej zmene z 20 CPU na 25 CPU. Pri tejto zmene je zrýchlenie dosiahnuté priemerne až o 19%. V následujúcich zmenách počtu CPU je zrýchlenie každý pridám výpočetným uzlom priemerne o 8%.

7.4 Porovnanie s Plaso nástrojom

Posledným vyhodnotením je porovnanie rýchlosti vytvoreného systému s nástrojom Plaso. Nástroj Plaso bol spustený nad predchádzajúcimi piatimi dátovými sadami. Nasledujúca tabuľka 7.2 zobrazuje výsledné časy extrakcie pre nástroj Plaso a vytvorený systém Plaso-spark.

testovacia sada	Plaso	Plasospark
test_multi1	60s	230s
test_multi2	110s	188s
test_multi3	40s	107s
test_multi4	120s	374s
test_multi5	135s	438s

Tabuľka 7.2: Porovnanie originálneho nástroja a implementovaného systému. Tabuľka zobrazuje výsledné časy extrakcie pre oba systémy.

Ako je z tabuľky vidieť, implementovaný systém neprekoná s podobnou konfiguráciou rýchlosť originálneho nástroja Plaso. V niektorých testoch je Plaso rýchlejšie až trojnásobne. Dôvodom rýchlejšej extrakcie Plaso nástroja je spôsob, akým sú využíva rozširujúce moduly (z anglického plugin) extraktorov. V originálnom Plaso nástroji sú rozširujúce moduly počas extrakcie v jednotlivých extraktoroch postupne pridávané do fronty, ktorú spravuje `ParserMediator`. Ak extraktor na začiatku extrakcie zistí, že môže využiť rozširujúce moduly, využije mediátor k tomu, aby ich zaradil do fronty spracovania. Mediátor komunikuje s hlavným riadicim programom, ktorý z tejto fronty vyberá dostupné rozširujúce moduly a vytvára nové procesy, ktoré paralelne spúštajú extrakciu s jednotlivými modulami. Týmto je zabezpečená distribúcia výpočtu rozširujúcich modulov a extrakcia nečaká na dokončenie výpočtu modulu.

V implementovanom systéme je využitie tohto mediátoru nemožné, kedže výpočet jednotlivých extraktorov prebieha oddelene od hlavného riadiaceho programu. Mediator extrakcie je v každej Spark úlohe vytvorený priamo pre danú úlohu a nie je možné využiť tento mediátor na komunikáciu s riadiacim programom počas extrakcie. Mediátor v Spark úlohe ukladá do fronty jednotlivé rozširujúce moduly, ktoré by extraktor poslal do fronty pre dalšie spracovanie. Po skončení modulu extraktoru je z fronty vybraný následujúci rozširujúci modul a extrakcia pokračuje v danej Spark úlohe. Práve toto spôsobuje spomalenie celej extrakcie, kedže moduly majú hlavne všeobecné extraktory, ktoré sú používané najčastejšie.

Okrem času extrakcie je porovnaný aj počet udalostí, ktoré Plasospark systém vyextrahuje. V nasledujúcich tabuľkách sú vybrané niektoré extraktory a je porovnaný počet vyextrahovaných udalostí v Plaso nástroji a Plasospark systéme. Kedže počet udalosti vyextrahovaných oboma systémami je rovnaký sú vybraných určité záznamy a následne sú rozobrané extraktory, ktorých počet udalosti nie je rovnaký. Tieto informácie sú získané z `pinfo` nástroja, s ktorým je implementovaný systém kompatibilný.

Extraktor	Plaso	Plasospark
systemd_journal	4312	3212
winreg_default	201	201
chrome_cache	217	217
Rozdiely		
filestat	531	156

Tabuľka 7.3: Počet udalostí vyextrahovaný oboma nástroji pri extrakcii udalostí nad dátovou sadou `test_multi1`.

V tabuĺke 7.3 je porovnaný výstup nástrojov pre testovaciu sadu `test_multi1`. Táto sada obsahuje napríklad systemd súbory, cache súbory a Windows NT registry súbory. Z ta-

buľky je vidieť, že počet udalostí vyextrahovaných jednotlivými extraktormi je rovnaký pre oba systémy. Rozdiel medzi vyextrahovanými udalosťami je v extraktore `filestat`. Tento extraktor vytvára udalosti z metadát jednotlivých súborov. Rozdiel v počte vyextrahovaných udalostí je spôsobený predspracovaním vstupných súborov. Kedže v HDFS uložisku nie je možné vhodne pracovať s obrazmi diskov alebo archivov ako bolo spomenuté v 4.3. Pri vyextrahovaní obsahu týchto súborov sú metadáta o týchto súboroch stratené. Taktiež rozdielu prispieva aj využitie `pyarrow` knižnice pri implementácií `DFVFS` virtuálneho súborového systému. Ako je popísane v 6.3 táto knižnica neposkytuje niektoré metadáta o súboroch. Práve `filestat` extraktor tieto udaje používa pre vytvorenie udalostí.

Extraktor	Plaso	Plasospark
chrome_17_cookies	1680	1680
msie_webcache	5572	5572
srum	18543	18543
winreg_default	152905	152905
Rozdiely		
filestat	378	215

Tabuľka 7.4: Počet udalostí vyextrahovaný oboma nástroji pri extrakcii udalostí nad dátovou sadou `test_multi2`.

Tabuľka 7.4 obsahuje vybrané extraktory a počet udalostí pre oba nástroje pre dátovú sadu `test_multi2`. Vybrané extraktory vyextrahovali najviac udalostí a ich počet sa nelíši ani v jednom nástroji. Rozdiel je opäť v extraktore `filestat` ako v predchádzajúcom prípade.

Extraktor	Plaso	Plasospark
file_history	2713	2713
amcache	3519	3519
winevt	13002	13002
winreg_default	2310	2310
Rozdiely		
filestat	444	143

Tabuľka 7.5: Počet udalostí vyextrahovaný oboma nástroji pri extrakcii udalostí nad dátovou sadou `test_multi3`.

Posledná tabuľka 7.5 popisuje vybrané extraktory z testovacej sady `test_multi3`. Opäť sú rozdiely len pri extraktore metadát zo súborov. Ostatné extraktory extrahujú rovnaký počet udalostí bez rozdielov na systéme v ktorom boli spustené.

Kapitola 8

Záver

Hlavným cieľom diplomovej práce bolo transformovať existujúci nástroj pre forenznú analýzu Plaso do distribuovaného výpočtového modelu. Na základe získaných poznatkov z teoretickej časti o aktuálnych možnostiach využitia distribuovaných výpočtových modelov a distribuovaných súborových systémov bol navrhnutý Plasospark nástroj. Hlavný výstup tejto práce. Pre vytvorenie tohto nástroja bolo potrebné pochopiť a analyzovať ako funguje už existujúci nástroj Plaso. Analyzovať jeho vnútorné komponenty a triedy, ktoré vykonávajú operácie spojené s extrakciou a analýzou forenzných dát a transformovať jednolivé komponenty do distribuovaného výpočtového modelu. Ďalej bolo potrebné analyzovať forenzné dáta, ktoré Plaso spracováva a uložiť tieto dáta do distribuovaného súborového systému.

Ako najlepšie voľba sa v práci javila implementovať nástroj Plasospark s využitím Spark distribuovaného výpočtového modelu a ukladať forenzné dáta do Hadoop HDFS distribuovaného súborového systému. Pre nástroj bolo v práci taktiež implementovanie API rozhranie pre ovládanie tohto nástroja pomocou Flask python knižnice. Pri vývoji bol kladený dôraz na životnosť programu, znovupoužiteľnosť a korektné fungovanie extraktorov z Plaso nástroja. Cieľom bolo vytvoriť nástroj, ktorý je schopný bez zásahu využiť nové Plaso extraktory, ktoré môžu byť pridané v novej verzii nástroj a taktiež aby vyextrahované dáta boli čo najviac zhodné s Plaso nástrojom.

Implementovaný nástroj Plasospark je vďaka využitiu interných komponent originálneho Plaso nástroja schopný využiť novo pridané extraktory a taktiež schopný vyextrahovať rovnaký objem udalostí zo systému ako originálny nástroj. Nástroj je taktiež schopný vytvárať výstupný formát, ktorý je možné využiť s dalšími nástrojmi z Plaso balíčka alebo je možné nástroj využiť s vlastným formátovaním a napojiť tak nástroj do existujúceho procesu spracovania. Takiež vytvorený nástroj spĺňa vlastnosti škálovateľnosti vďaka využítemu Spark výpočetnému modelu.

Napriek tomu, že implementovaný nástroj spĺňa stanovené ciele znovupoužitelnosti, životnosti a koretnosti fungovania extraktorov je nástroj vo výslednej implementácií pomalší oproti originálnej Plaso verzii. V budúcich verziach by mohol byť upravený spôsob, akým sú využívané rozširujúce moduly extraktorov v Spark úlohách, čo by mohlo zlepšiť čas extrakcie. V budúcnosti by mohli byť taktiež pridané viaceré formátovače výstupných dát, pre ktoré je nástroj pripravený.

Literatúra

[1] A Review Study of Apache Spark in Big Data Processing. *International Journal of Computer Science Trends and Technology (IJCST)* [online]. Eighth Sense Research Group. Jún 2016, zv. 4, č. 3, s. 93–98, [cit. 2022-12-10]. ISSN 2347-8578. Dostupné z: http://www.ijcstjournal.org/volume-4/issue-3/IJCST-V4I3P16.pdf.

[2] *Digital Forensics Artifacts Repository* [online]. Apríl 2023 [cit. 2023-05-01]. Dostupné z: https://artifacts.readthedocs.io/en/latest/.

[3] *Plaso (log2timeline)* [online]. 2023 [cit. 2022-12-10]. Dostupné z: https://plaso.readthedocs.io/en/latest/index.html.

[4] Braam, P. a Schwan, P. Lustre: The intergalactic file system. In: *Proceedings of the Ottawa Linux Symposium* [online]. Január 2002, sv. 1, s. 50–54 [cit. 2023-01-10]. Dostupné z: https://www.researchgate.net/publication/241229733_Lustre_The_intergalactic_file_system.

[5] Carrier, B. *Sleuth Kit Help Documents* [online]. 2003, reevidované 5.6.2012 [cit. 2023-12-28]. Dostupné z: http://wiki.sleuthkit.org/index.php?title=Help_Documents.

[6] Daniel, L. *Digital Forensics for Legal Professionals: Understanding Digital Evidence from the Warrant to the Courtroom*. 1. vyd. Syngress, júl 2011. ISBN 978-1-59749-643-8.

[7] DataFlair. *Apache Spark Ecosystem – Complete Spark Components Guide* [online]. 2017 [cit. 2023-01-17]. Dostupné z: https://data-flair.training/blogs/apache-spark-ecosystem-components.

[8] Garg, A. *Apache Spark Architecture* [online]. Intellipaat Software Solutions Pvt. Ltd., 2021. reevidované 10.3.2023 [cit. 2023-04-05]. Dostupné z: https://intellipaat.com. Path: Home / Tutorial / Apache Spark Architecture.

[9] Ghemawat, S., Gobioff, H. a Leung, S.-T. The Google File System. In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles* [online]. Bolton Landing, NY: [b.n.], 2003, s. 20–43 [cit. 2023-01-10]. ISBN 978-1-58113-757-6. Dostupné z: https://static.googleusercontent.com/media/research.google.com/cs//archive/gfs-sosp2003.pdf.

[10] Guðjónsson, K. *Mastering the super timeline with log2timeline* [online]. White Paper, 1. vyd. August 2010 [cit. 2022-12-05]. Dostupné z: https://www.semanticscholar.org/paper/Mastering-the-Super-Timeline-With-log2timeline-Guethjoacutensson/8cd9a89206bf972caffebd20a2fd65a9d8d25565.

[11] Jameel, K. a Sabry, N. A Comprehensive Survey for Hadoop Distributed File System. *Asian Journal of Computer Science and Information Technology* [online]. August 2021, zv. 11, s. 46–57, [cit. 2023-01-11]. ISSN 2581-8260. Dostupné z: https://doi.org/10.9734/AJRCOS/2021/v11i230260.

[12] Morais, T. Survey on Frameworks for Distributed Computing: Hadoop, Spark and Storm. In: SousaEugénio Oliveira, A. A. de, ed. *Proceedings of the 10th Doctoral Symposium in Informatics Engineering* [online]. 1. vyd. Január 2015, s. 95–105 [cit. 2022-12-18]. ISBN 978-972-752-173-9. Dostupné z: https://doi.org/10.24840/978-972-752-173-9.

[13] N. Deshai, D. G. P. S. V. Research Paper on Big Data and Hadoop-Map Reduce Real Time Scheduling. *International Journal of Advanced Research in Science and Engineering* [online]. Február 2018, zv. 07, s. 750–761, [cit. 2022-12-10]. ISSN 2319-8354. Dostupné z: http://ijarse.com/images/fullpdf/1519302484_SVCET2097ijarse.pdf.

[14] Nicolaou, N. *Three Anti-Forensics Techniques that pose the Greatest Risks to Digital Forensic Investigations* [online]. Jún 2020 [cit. 2023-01-10]. Dostupné z: https://www.researchgate.net/publication/342420751_Three_Anti-Forensics_Techniques_that_pose_the_Greatest_Risks_to_Digital_Forensic_Investigations.

[15] Rabl, T., Traub, J., Katsifodimos, A. a Markl, V. Apache Flink in current research. *It - Information Technology* [online]. De Gruyter Oldenbourg. Január 2016, zv. 58, č. 4, s. 157–165, [cit. 2023-01-02]. ISSN 2196-7032. Dostupné z: https://doi.org/10.1515/itit-2016-0005.

[16] Reith, M., Carr, C. a Gunsch, G. H. An Examination of Digital Forensic Models. *International Journal of Digital Evidence* [online]. 3. vyd. Ohio: Department of Electrical and Computer Engineering Graduate School of Engineering and Management Air Force Institute of Technology. Máj 2002, zv. 1, [cit. 2023-01-04]. ISSN 1938-0917. Dostupné z: https://www.researchgate.net/publication/2589967_An_Examination_of_Digital_Forensic_Models.

[17] Rosenthal, R., Ushe, N. a Magaya, I. A framework for enterprise security and forensics in Zimbabwe- A case study of the Sony Hack. [online]. Február 2016, [cit. 2023-01-08]. Dostupné z: https://doi.org/10.13140/RG.2.1.2994.2803.

[18] Samadi, Y., Zbakh, M. a Tadonki, C. Performance comparison between Hadoop and Spark frameworks using HiBench benchmarks. *Concurrency and Computation: Practice and Experience* [online]. November 2017, zv. 30, č. 12, [cit. 2022-12-10]. ISSN 1532-0634. Dostupné z: https://doi.org/10.1002/cpe.4367.

[19] Shaikh, E., Mohiuddin, I., Alufaisan, Y. a Nahvi, I. Apache Spark: A Big Data Processing Engine. In: *2019 2nd IEEE Middle East and North Africa COMMunications Conference (MENACOMM)* [online]. IEEE, November 2019, s. 1–6 [cit. 2022-12-04]. ISBN 978-1-7281-3688-2. Dostupné z: https://ieeexplore.ieee.org/xpl/conhome/8967516/proceeding.

[20] Shvachko, K., Kuang, H., Radia, S. a Chansler, R. The Hadoop Distributed File System. In: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*

(MSST) [online]. Incline Village: [b.n.], Máj 2010, s. 1–10 [cit. 2023-01-29]. ISBN 9781424471522. Dostupné z: https://doi.org/10.1109/MSST.2010.5496972.

[21] Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M. et al. Apache Hadoop YARN: Yet Another Resource Negotiator. In: *Proceedings of the 4th Annual Symposium on Cloud Computing* [online]. New York, NY, USA: Association for Computing Machinery, 2013, č. 5, s. 1–16 [cit. 2022-12-15]. SOCC '13. ISBN 9781450324281. Dostupné z: https://doi.org/10.1145/2523616.2523633.

[22] Weil, S., Pollack, K., Brandt, S. a Miller, E. Dynamic Metadata Management for Petabyte-Scale File Systems. In: IEEE, ed. *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing* [online]. Pittsburgh: [b.n.], November 2004, s. 4–4 [cit. 2023-02-05]. ISBN 0769521533. Dostupné z: https://doi.org/10.1109/SC.2004.22.

[23] Weil, S., Brandt, S., Miller, E., Long, D. a Maltzahn, C. Ceph: A Scalable, High-Performance Distributed File System. In: [online]. Seattle: USENIX Association, November 2006, s. 307–320 [cit. 2023-01-25]. ISBN 1931971471. Dostupné z: https://www.ssrc.ucsc.edu/media/pubs/6ebbf2736ae06c66f1293b5e431082410f41f83f.pdf.

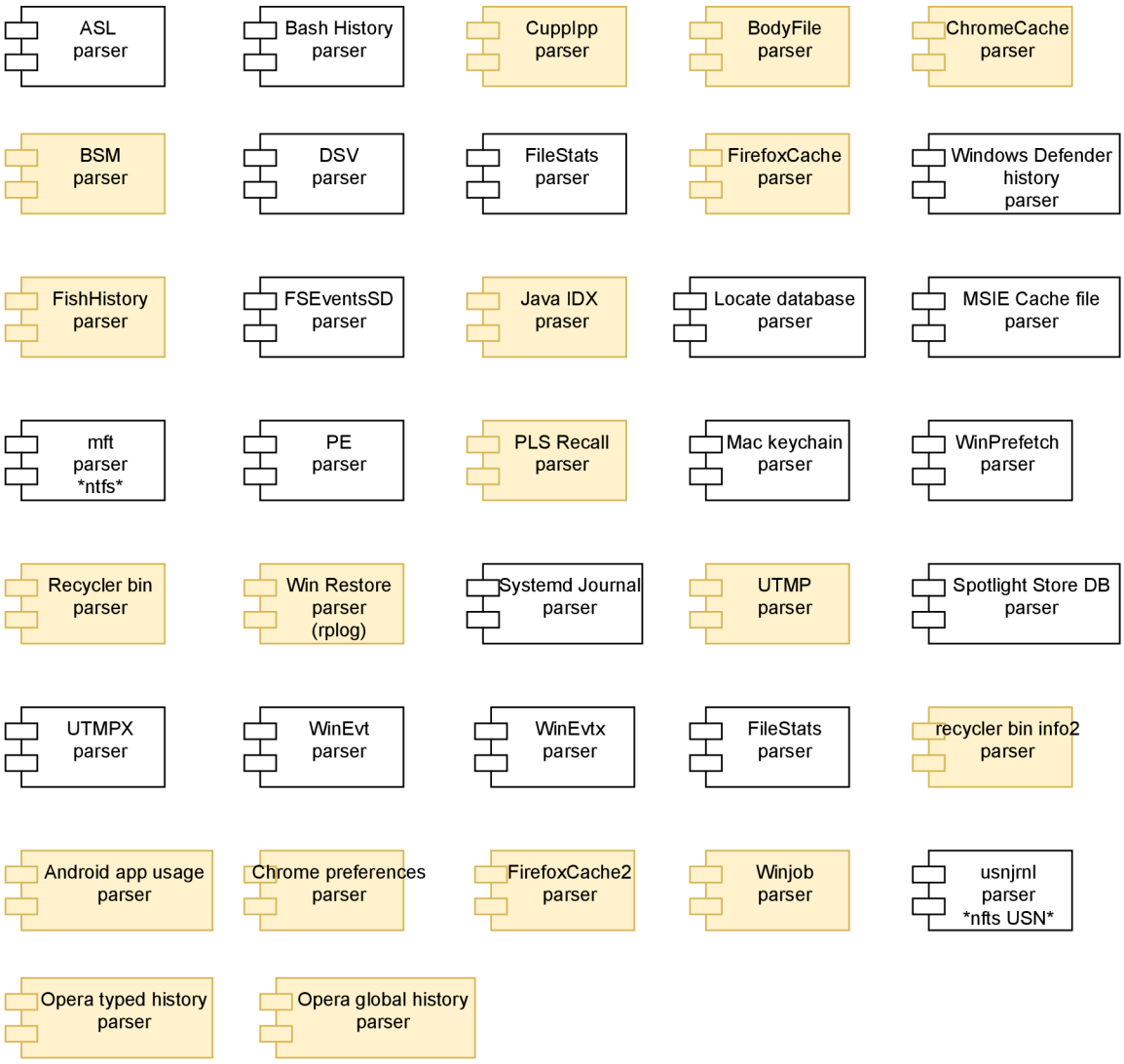
[24] Weil, S. A., Brandt, S. A., Miller, E. L. a Maltzahn, C. CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data. In: *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* [online]. Tampa: [b.n.], November 2006, s. 31–31 [cit. 2023-02-01]. Dostupné z: https://doi.org/10.1109/SC.2006.19.

[25] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J. et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for in-Memory Cluster Computing. In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* [online]. USA: USENIX Association, Apríl 2012, s. 2 [cit. 2022-12-10]. Dostupné z: https://dl.acm.org/doi/proceedings/10.5555/2228298.

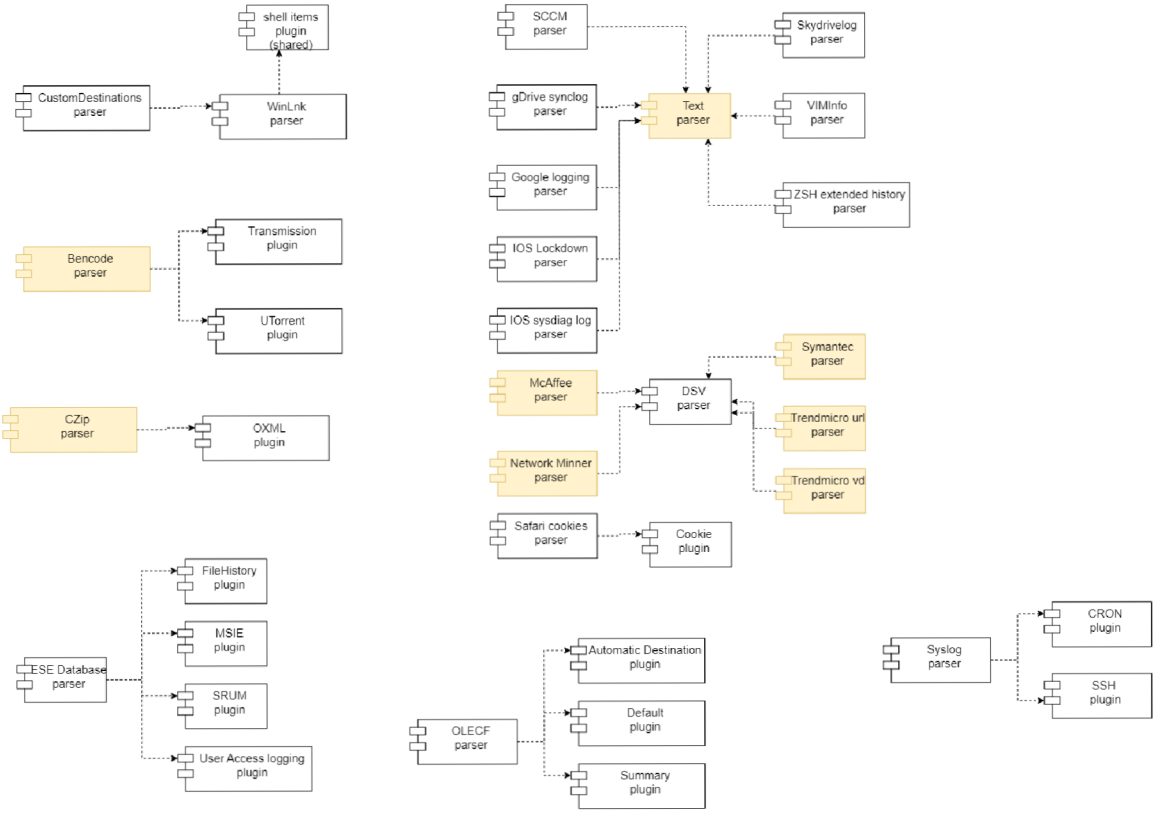
[26] Zhou, H., Sun, G., Fu, S., Liu, J., Zhou, X. et al. A Big Data Mining Approach of PSO based BP Neural Network for Financial Risk Management with IoT. *IEEE Access* [online]. Október 2019, zv. 7, s. 1–1, [cit. 2022-12-10]. ISSN 2169-3536. Dostupné z: https://doi.org/10.1109/ACCESS.2019.2948949.

Príloha A

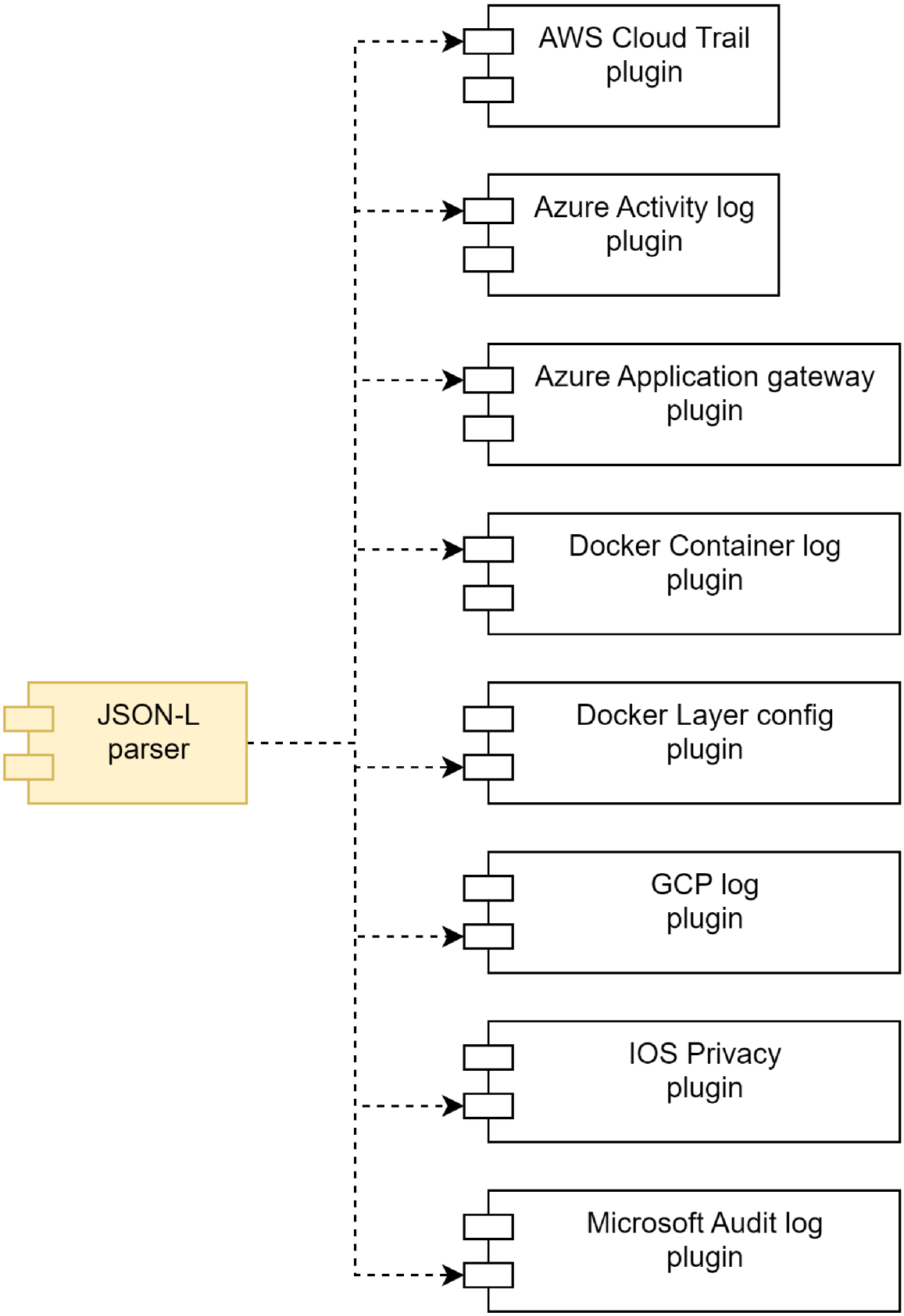
Extraktory Plaso nástroja



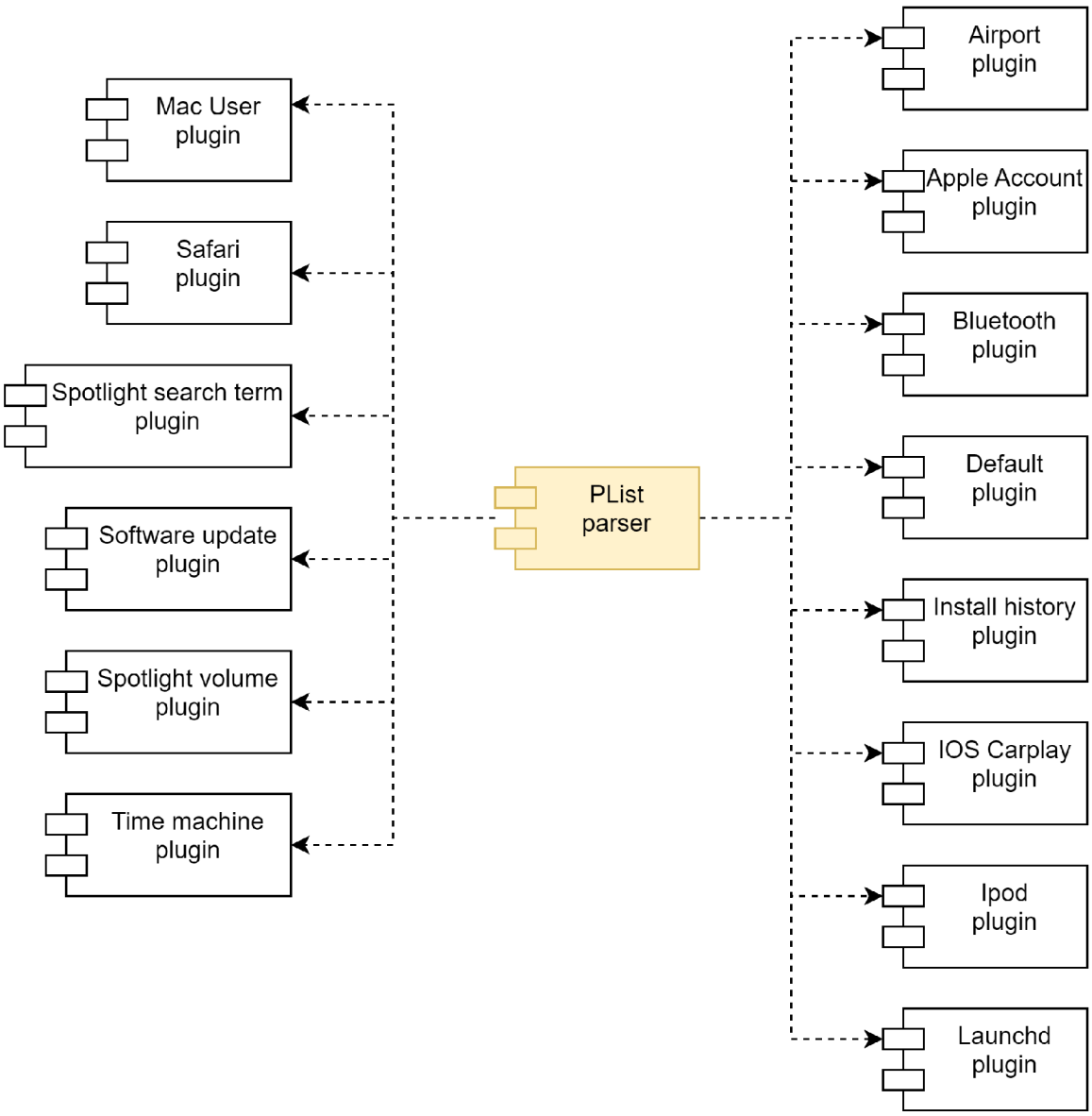
Obr. A.1: Množina extraktorov, ktoré nemajú v Plaso nástroji žiadne závislosti na iných extraktoroch.



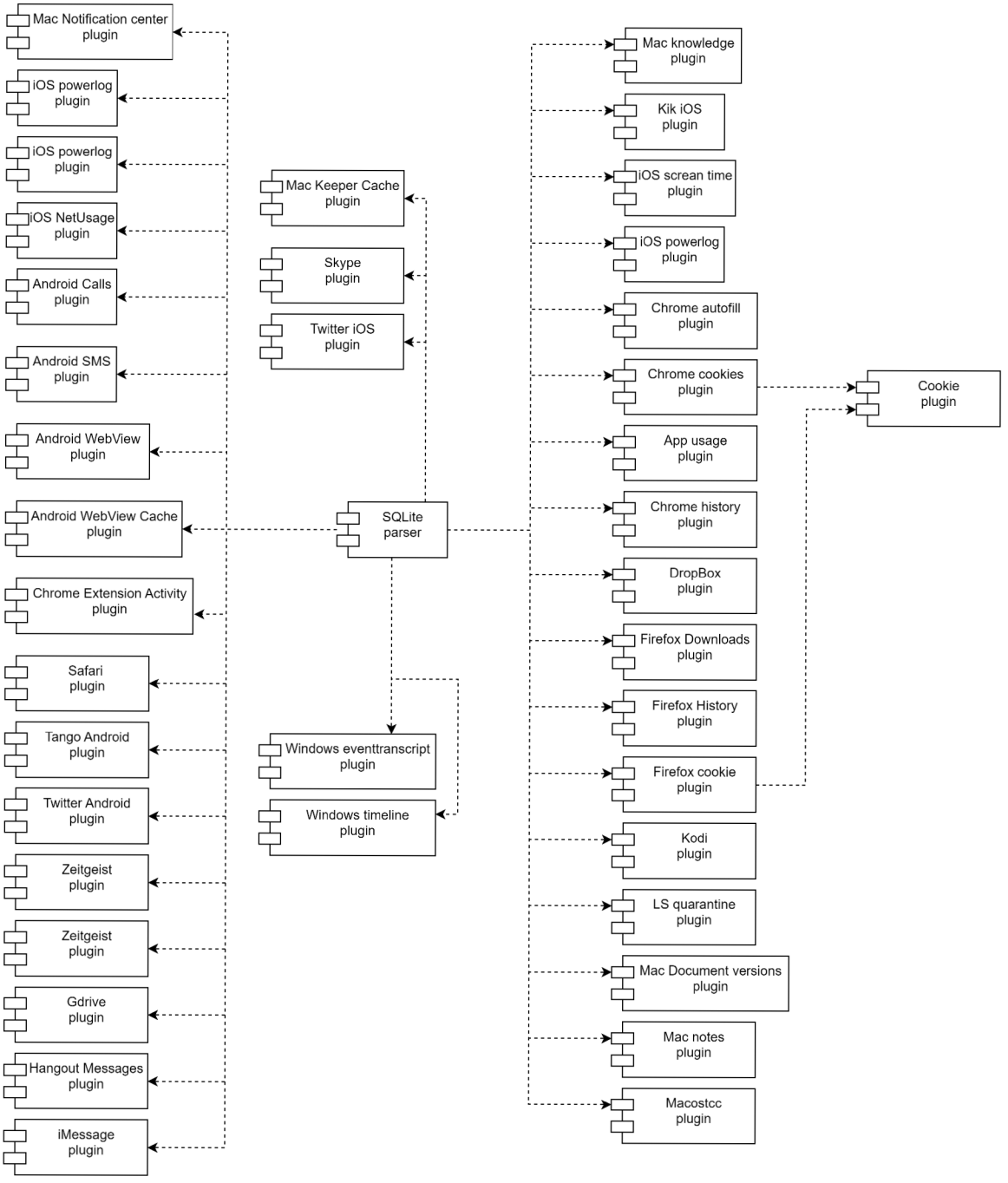
Obr. A.2: Množina extraktorov, ktoré majú závislosti na iné extraktory.



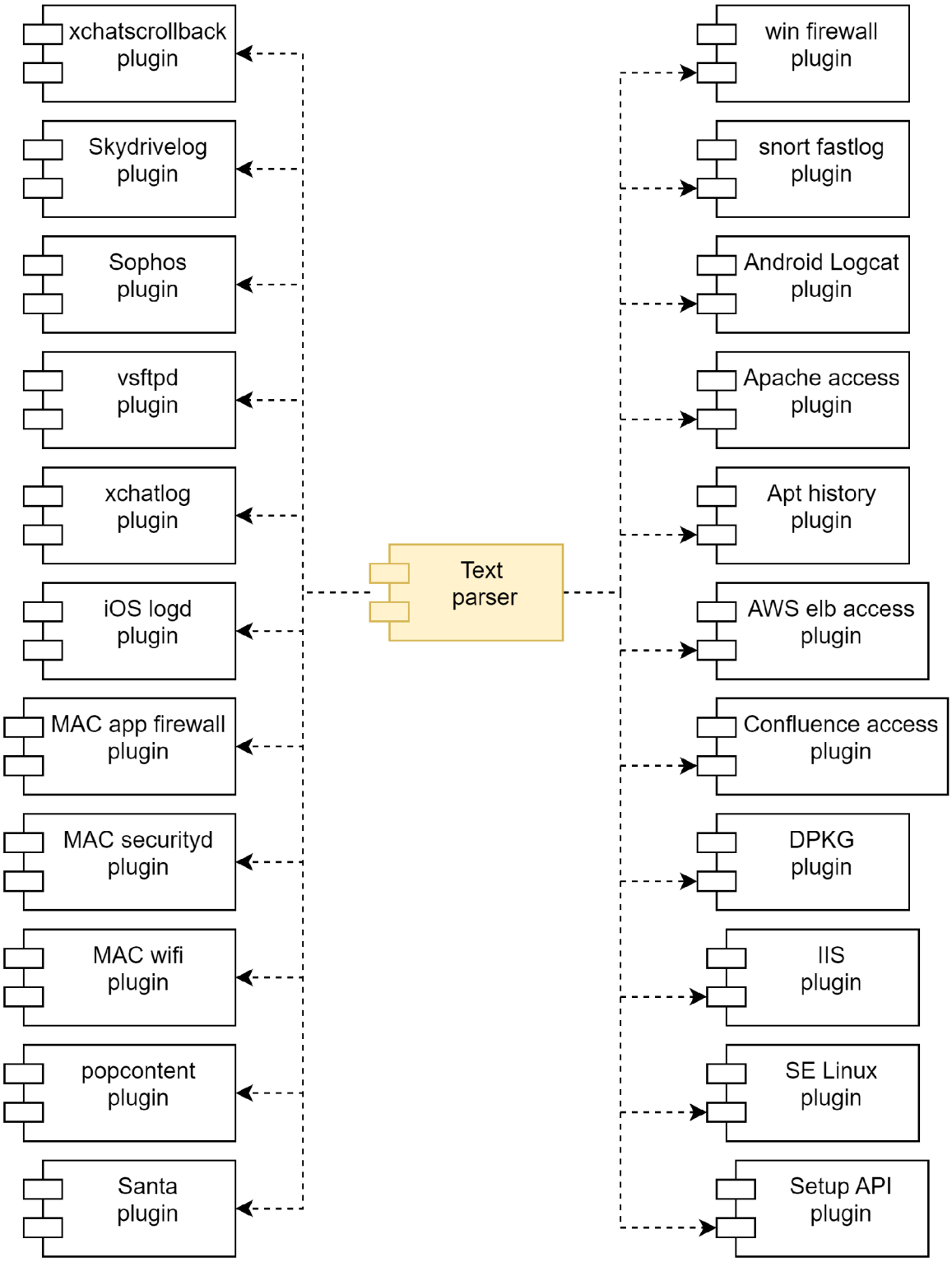
Obr. A.3: Extraktor pre JSONL formát spolu s rožširujúcimi modulmi, ktoré využíva.



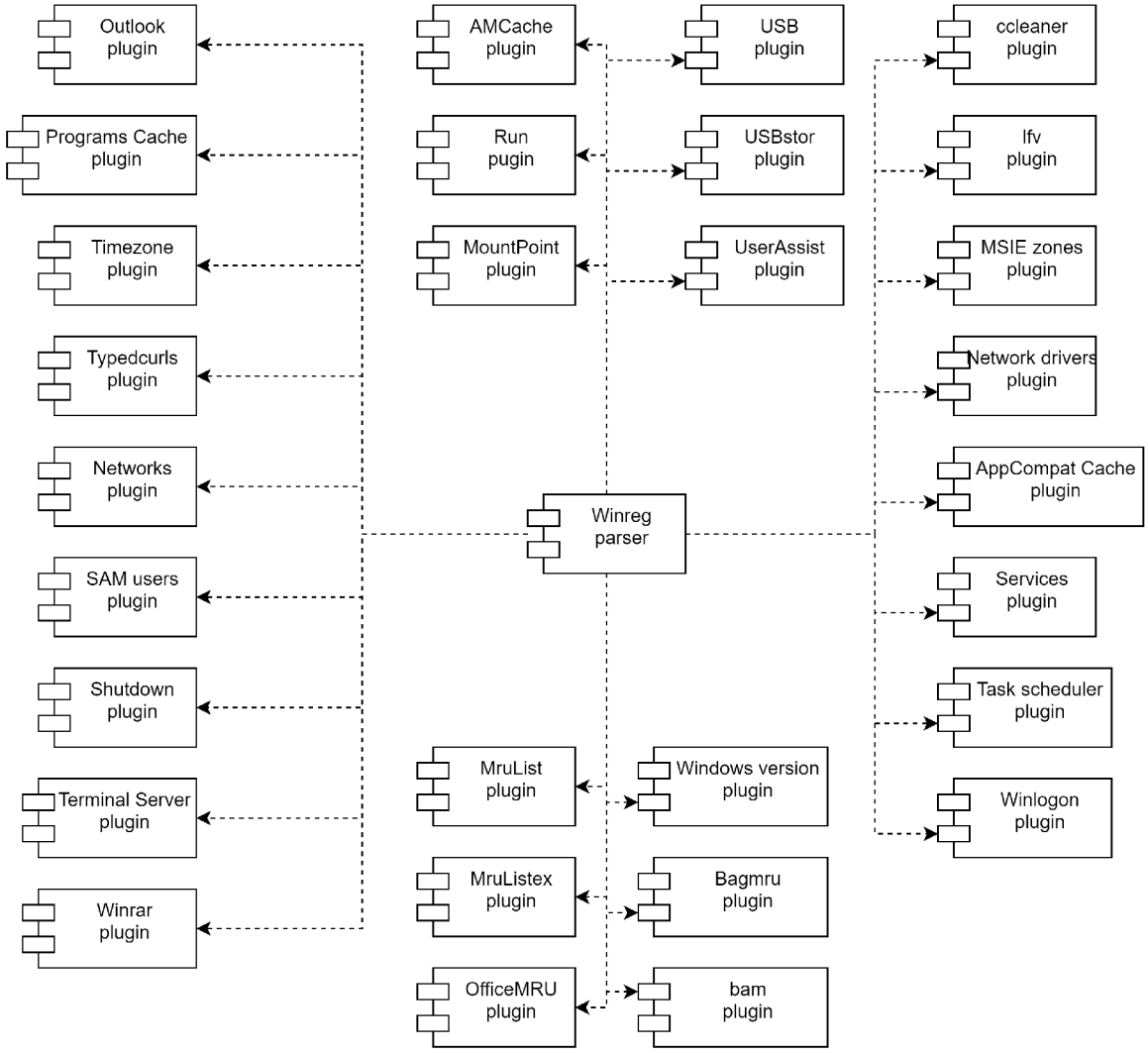
Obr. A.4: Extraktor pre PList formát spolu s rožširujúcimi modulmi, ktoré využíva.



Obr. A.5: Extraktor pre SQLite formát spolu s rožširujúcimi modulmii, ktoré využíva.



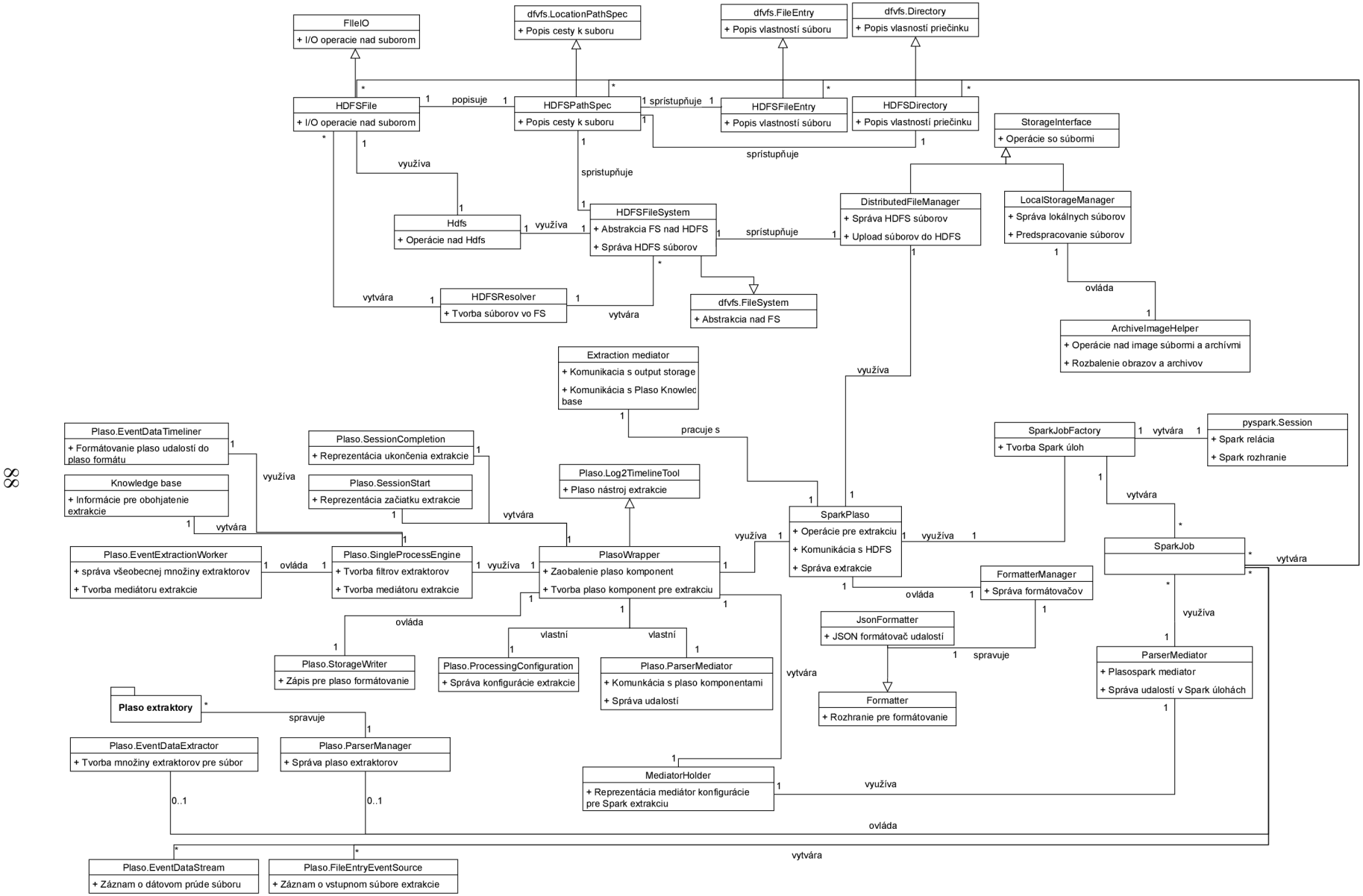
Obr. A.6: Extraktor pre Text formát spolu s rožširujúcimi modulmi, ktoré využíva.



Obr. A.7: Extraktor pre Winreg formát spolu s rožširujúcimi modulmi, ktoré využíva.

Príloha B

Diagram tried navrhnutého nástroja



Obr. B.1: Diagram tried pre navrhnuté riešenie Plasospark nástroja.