**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**
**BRNO UNIVERSITY OF TECHNOLOGY**

**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**
**FACULTY OF INFORMATION TECHNOLOGY**

**ÚSTAV INFORMAČNÍCH SYSTÉMŮ**
**DEPARTMENT OF INFORMATION SYSTEMS**


# ZPRACOVÁNÍ SÍŤOVÉ KOMUNIKACE V PROSTŘEDÍ APACHE SPARK

**THESIS TITLE**

**DIPLOMOVÁ PRÁCE**
**MASTER'S THESIS**


**AUTOR PRÁCE** — **MICHAL BÉDER**
**AUTHOR**

**VEDOUCÍ PRÁCE** — **Doc. Ing. ONDŘEJ RYŠAVÝ, Ph.D.**
**SUPERVISOR**



**BRNO 2018**

*Zadání diplomové práce/20651/2017/xbeder00*

**Vysoké učení technické v Brně - Fakulta informačních technologií**

Ústav informačních systémů Akademický rok 2017/2018

# Zadání diplomové práce

Řešitel: **Béder Michal, Ing.**
Obor: Bezpečnost informačních technologií
Téma: **Zpracování síťové komunikace v prostředí Apache Spark**
**Network Traces Analysis Using Apache Spark**
Kategorie: Paralelní a distribuované výpočty

Pokyny:

1. Seznamte se s problematikou zpracování síťové komunikace v distribuovaném prostředí
2. Nastudujte základní principy prostředí Apache Spark
3. Navrhněte řešení pro analýzu síťových toků v prostředí Apache Spark
4. Realizujte navržené řešení v jazyce Java/Scala pro prostředí Apache Spark
5. Vytvořte testovací data a proveďte experimenty pro vyhodnocení navrženého řešení
6. Vyhodnoťte dosažené řešení a diskutujte možná vylepšení a další rozšíření

Literatura:

- Omar Santos: Network Security with NetFlow and IPFIX: Big Data Analytics for Information Security, Cisco Press, 2015.
- Mike Frampton: Mastering Apache Spark. Packt Publishing, September 2015.
- Holden Karau, Rachel Warren: High Performance Spark - Best Practices for Scaling and Optimizing Apache Spark. O'Reilly Media, 2017.

Při obhajobě semestrální části projektu je požadováno:

- Dokončeny body 1 až 3 a rozpracování bodu 4.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Ryšavý Ondřej, doc. Ing., Ph.D.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 23. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
*vedoucí ústavu*

## Abstrakt

Táto práca rieši spôsob návrhu aplikácie na analýzu dát sieťovej komunikácie v prostredí distribuovaného systému Apache Spark. Implementáciu je možné rozdeliť do troch častí. Prvou je načítanie dát z distribuovaného úložiska HDFS, druhou analýza podporovaných sieťových protokolov a treťou distribuované vyhodnotenie výsledkov. Po vyhodnotení sú výstupy zobrazené v prostredí Apache Zeppelin. Výsledná aplikácia je schopná analyzovať jednotlivé pakety ako aj celé sieťové toky. Podporovanými formátmi vstupných dát sú `pcap` a `JSON`. Hlavným prínosom aplikácie je možnosť spracovania veľkých objemov dát. Jej výkonnosť je ovplyvnená hlavne formátom vstupných dát a využitím dostupných výpočetných jadier.

## Abstract

The aim of this thesis is to show how to design and implement an application for network traces analysis using Apache Spark distributed system. Implementation can be divided into three parts – loading data from a distributed HDFS storage, supported network protocols analysis and distributed data processing. As a data visualization tool is used web-based notebook Apache Zeppelin. The resulting application is able to analyze individual packets as well as the entire flows. It supports `JSON` and `pcap` as input data formats. The goal of the application is to allow Big Data processing. The greatest impact on its performance has the input data format and allocation of the available cores.

## Kľúčové slová

Apache Spark, distribuovaný, Big Data, HDFS, sieťový tok, pcap, Wireshark, Scala, Apache Zeppelin

## Keywords

Apache Spark, distributed, Big Data, HDFS, flow, pcap, Wireshark, Scala, Apache Zeppelin

## Citácia

BÉDER, Michal. *Zpracování síťové komunikace v prostředí Apache Spark*. Brno, 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Doc. Ing. Ondřej Ryšavý, Ph.D.

# Zpracování síťové komunikace v prostředí Apache Spark

## Prehlásenie

Prehlasujem, že som tento semestrálny projekt vypracoval samostatne pod vedením Doc. Ing. Ondřeja Ryšavého. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

. . . . . . . . . . . . . . . . . . . . . . .
Michal Béder
21. mája 2018


## Poďakovanie

Týmto by som chcel poďakovať pánovi Doc. Ing. Ondřejovi Ryšavému, Ph.D. za vedenie tejto diplomovej práce.

# Obsah


**1 Úvod** **2**

**2 Distribuované prostredie Apache Spark** **3**
- 2.1 Ekosystém Apache Spark . . . 3
- 2.2 Architektúra Spark klastra . . . 7
- 2.3 Distribuované dátové kolekcie . . . 8
- 2.4 Vizualizácia dát . . . 13

**3 Analýza požiadaviek na aplikáciu** **17**
- 3.1 Analýza obsahu paketov . . . 17
- 3.2 Analýza sieťových tokov . . . 18

**4 Návrh riešenia** **19**
- 4.1 Spark shell . . . 19
- 4.2 Architektúra aplikácie . . . 20

**5 Implementácia** **22**
- 5.1 Príprava vstupných dát . . . 23
- 5.2 Načítanie vstupov . . . 24
- 5.3 Spracovanie dát . . . 28
- 5.4 Analýza a vizualizácia dát . . . 31

**6 Experimenty** **41**
- 6.1 Konfigurácia objemu operačnej pamäte a počtu jadier . . . 41
- 6.2 Veľkosť vstupných pcap súborov . . . 43
- 6.3 Formáty vstupov a funkcie rozhrania aplikácie . . . 44

**7 Zhodnotenie výsledkov a možnosti ďalšieho vývoja** **47**

**8 Záver** **49**

**Literatúra** **50**

**Prílohy** **52**

**A Obsah priloženého pamäťového média** **53**

# Kapitola 1

# Úvod

Celosvetové objemy sieťovej komunikácie sa pohybujú v rádoch exabajtov až zettabajtov. Zachytené dáta sieťovej komunikácie je výpočetne náročné analyzovať už v rádoch gigabajtov. Ako riešenie sa ponúka využitie systému určeného na distribuované spracovanie dát. Existuje množstvo rôznych distribuovaných systémov, jedným z nich je aj Apache Spark. Výhodami distribuovaného systému Spark sú veľká výpočetná rýchlosť vďaka tzv. „in-memory computing“, jeho dostupnosť ako open-source projektu a schopnosť fungovať aj na nízkonákladovom hardvéri. Spark klastre sú tým pádom výhodné ako pre veľké nadnárodné firmy[1], tak aj pre bežných používateľov bez špecializovaného hardvéru. Spark klaster je možné vytvoriť už na štyroch počítačoch Raspberry Pi 2, aj keď v tomto prípade bude určený skôr na testovacie účely než na reálne použitie.

V tejto práci bude popísaný spôsob vývoja aplikácie, ktorá je schopná analyzovať vybrané parametre sieťovej komunikácie v prostredí Apache Spark. Kapitola 2 vysvetľuje činnosť systému Spark od najnižšej vrstvy distribuovaného úložiska dát až po jeho aplikačné rozhranie. Kapitola 3 definuje požiadavky na aplikáciu. Aplikácia musí byť schopná analyzovať vybrané parametre jednotlivých paketov, ako aj celých sieťových tokov. Kapitola 4 sa zaoberá návrhom aplikácie. Je tu popísané aj prostredie Spark shell ako jedno z možných hostiteľských prostredí pre navrhnutú aplikáciu. Prvé tri kapitoly boli prevzaté zo semestrálneho projektu. Kapitoly 3 a 4 boli v rámci diplomovej práce len mierne upravené. Kapitola 2 bola doplnená o popis distribuovaných dátových kolekcií a nástrojov vhodných na vizualizáciu dát. Popis implementácie od načítania dát po ich vizualizáciu sa nachádza v kapitole 5. Je tu podrobne popísaná aj implementácia jednotlivých úloh definovaných v kapitole 3, vrátane ukážok ich výstupov v prostredí Apache Zeppelin. Nasadenie výslednej aplikácie do reálneho Spark klastra je popísané v kapitole 6. V rámci tejto kapitoly sa testuje výkonnosť aplikácie v rôznych scenároch. Úlohou je nájsť optimálnu kombináciu vstupov a nastavení parametrov klastra s cieľom dosiahnuť čo najvyššiu rýchlosť spracovania dát. Posledná kapitola obsahuje súhrn dosiahnutých výsledkov a naznačuje, akým smerom sa môže uberať ďalší vývoj aplikácie.

[1] Spark klastre využívajú v produkcii firmy ako Amazon, Autodesk, eBay Inc. a ďalšie, pozri http://spark.apache.org/powered-by.html

# Kapitola 2

# Distribuované prostredie Apache Spark

Apache Spark je open-source distribuovaný systém napísaný v jazyku Scala, ktorý je primárne určený na spracovanie Big Data. Celý framework a jeho ekosystém poskytujú veľké množstvo nástrojov a nastavení, pomocou ktorých je možné systém rozširovať a prispôsobiť aktuálnym potrebám. Najdôležitejšie z nich z pohľadu tejto práce sú popísané ďalej v tejto kapitole. Apache Spark je považovaný za nástupcu známeho systému, schopného spracovávať Big Data, Apache Hadoop. Systém Spark je možné používať aj nezávisle na Apache Hadoop pomocou alternatívnych správcov klastra a dátových úložísk. Spark ale využíva množstvo knižníc a princípov prevzatých z prostredia Hadoop. Oproti systému Hadoop prichádza Spark s viacerými výhodami v oblasti spracovania dát. Najdôležitejšími sú rýchlosť, rozšírený súbor funkcií a API (Application Programming Interface) pre jazyky Scala, Python, R a Java.

## 2.1 Ekosystém Apache Spark

Predstavu o základnej štruktúre ekosystému vybudovaného okolo jadra Apache Spark (angl. Spark Core) si môžeme vytvoriť z obrázku 2.2. Existuje množstvo interpretácií popisujúcich tento ekosystém. To je dané množstvom existujúcich komponent, z ktorých je možné poskladať funkčný celok. Dva Spark systémy tak môžu byť zostavené z úplne odlišných komponent a spájať ich bude len jadro. Ako je na obrázku 2.2 vidieť, architektúra celého ekosystému sa skladá z jednotlivých vrstiev obsahujúcich alternatívne alebo dopĺňajúce sa komponenty.

### HDFS

HDFS (Hadoop Distributed File System) je distribuovaný súborový systém určený na uchovávanie veľkých objemov dát. Je napísaný v jazyku Java a pre svoj beh potrebuje JVM (Java Virtual Machine). Od iných distribuovaných súborových systémov sa líši vysokou odolnosťou proti poruchám a schopnosťou fungovať na nízkonákladovom hardvéri. Návrh systému HDFS počíta s tým, že niektorá časť hardvéru je vždy nefunkčná. Jeho cieľom je rýchle automatické zotavovanie z chýb. Vďaka tomuto návrhu je schopný fungovať aj na nespoľahlivom hardvéri.

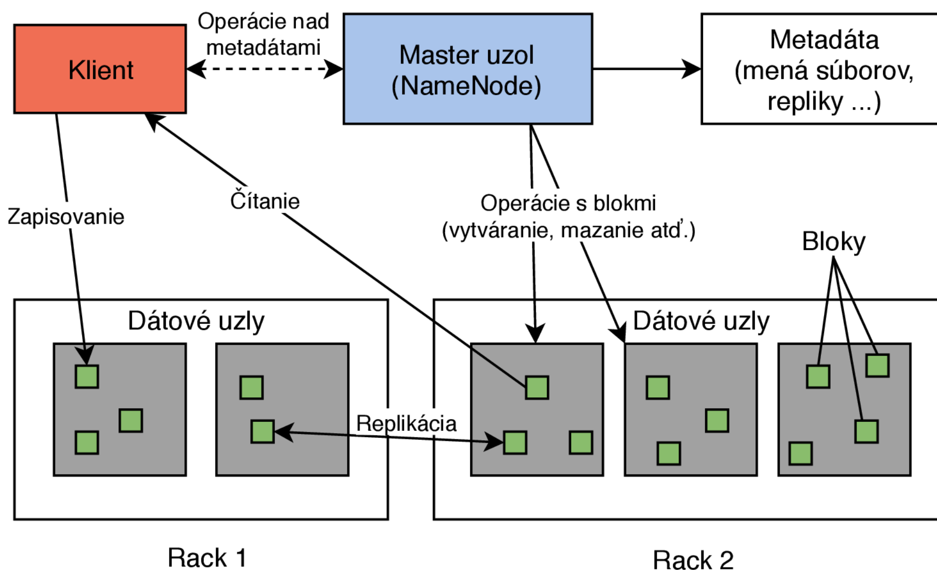
Architektúra HDFS je typu *master/slave*. HDFS klaster obsahuje jeden uzol typu *master* nazývaný NameNode. Cez tento uzol môžu klienti pristupovať do klastra. Ďalej klaster

obsahuje dátové uzly (tzv. DataNodes) typu slave. V jednom uzle v klastri typicky funguje jeden HDFS dátový uzol. Každý z týchto dátových uzlov spravuje svoju časť dátového úložiska. HDFS klaster sprístupňuje používateľom menný priestor systému, prostredníctvom ktorého je možné adresovať uložené súbory. Reálne sú všetky súbory rozdelené do blokov a uložené v rôznych dátových uzloch. *Master* uzol posiela dátovým uzlom príkazy na vytváranie, mazanie a replikáciu blokov, tým pádom má k dispozícii všetky HDFS metadáta v klastri. Reálne dáta cez tento uzol ale nikdy neprechádzajú, spôsobilo by to úzke miesto v architektúre. Menný priestor klastra má tradičnú hierarchickú organizáciu a pristupovať k súborom je možné pomocou URL (Uniform Resource Locator) v tvare:

```
hdfs://<master>:<port>/<path>,
```

kde `<master>` je meno alebo IP (Internet Protocol) adresa *master* uzla, `<port>` je číslo portu *master* uzla a `<path>` je cesta k súboru v systéme.

Z hľadiska dostupnosti dát a rýchlosti HDFS je kritická replikácia dát v klastri. Táto činnosť je veľmi časovo náročná a v súčasnej dobe sa na jej postupnom vylepšovaní stále pracuje. Najbežnejšie sa v úložisku vyskytujú 3 repliky dátových blokov. Z tohto čísla vychádza súčasná politika replikácie. Prvá replika sa uloží na ľubovolný dátový uzol, druhá na dátový uzol v inom racku a tretia na iný dátový uzol v racku, v ktorom sa nachádza druhá replika. Výhodou tejto politiky je zníženie objemu komunikácie medzi rackmi pri zapisovaní, nevýhodou je zníženie agregovanej rýchlosti čítania dátových blokov (rýchlosť čítania z 2 rackov je nižšia než z 3). V prípade vyžadovaného vyššieho počtu replík sa štvrtá a ďalšie repliky umiestňujú náhodne. Maximálny počet replík je obmedzený počtom dátových uzlov v klastri. Popis HDFS v tejto podkapitole vychádza z informácií uvedených v dokumentácii Apache Hadoop [14].




Obr. 2.1: Architektúra HDFS (inšpirované z [14])

### Správcovia klastra

Správcovia klastra (angl. cluster managers), v literatúre nazývaní aj plánovačmi úloh (angl. task schedulers), primárne zabezpečujú alokáciu zdrojov pre vykonávanie úloh, tj. rozdeľujú

prácu medzi jednotlivé uzly daného klastra. Správcovia rozhodujú o tom, v ktorých uzloch a kedy sa spustia exekútori. Takto alokovaným exekútorom môže potom *driver* prideľovať jednotlivé úlohy, z ktorých je zložená Spark aplikácia. Najpoužívanejší správcovia spomínaní v publikácii [3] sú:

- Standalone – je najjednoduchším správcom klastra a je natívne podporovaný prostredím Spark. Jeho hlavnou výhodou je jednoduché nasadenie vďaka tomu, že je súčasťou Spark distribúcie. V klaster móde (pozri podkapitolu 2.2) podporuje len jednoduché FIFO (First In First Out) plánovanie prideľovania zdrojov aplikáciám. Štandardne pridelí aplikácii všetky dostupné zdroje (procesorové jadrá a operačnú pamäť). Pre dosiahnutie paralelného prístupu aplikácií do klastra je potrebné nakonfigurovať maximálny počet zdrojov dostupných pre jednu aplikáciu.
- YARN (Yet Another Resource Negotiator) – je správca, ktorý prišiel už so systémom Apache Hadoop 2.0. Jeho výhodami sú podpora dátovej lokality v HDFS, podpora protokolu Kerberos a lepšie prideľovanie zdrojov než poskytuje Standalone správca. Architektúra správcu YARN pozostáva z troch hlavných komponent:
  - Resource Manager – je globálna (jedna pre všetky aplikácie) *master* komponenta, ktorá oddeľuje funkcionalitu plánovania úloh od spravovania zdrojov. Plánovač (angl. scheduler) je zodpovedný za prideľovanie zdrojov aplikáciám. Správca aplikácií (angl. application manager) sa stará o beh Application Master procesov.
  - Node Manager – slave komponenta, ktorá beží v každom uzle klastra. Jej úlohou je poskytovať informácie o tomto uzle.
  - Application Master – je komponenta vytvorená zvlášť pre každú aplikáciu a spolupracujúca s obidvomi vyššie uvedenými komponentami. Jej úlohou je starať sa o priebeh aplikácie.

  Ďalšie podrobnosti je možné nájsť spolu s vyššie uvedenými informáciami v dokumentácii Apache Hadoop [14].
- Mesos – je správca poskytujúci ilúziu jedného virtuálneho zdroja, ktorý môžu paralelne využívať rôzne frameworky. Je vhodný na správu rozsiahlych klastrov.
- Kubernetes – v súčasnosti experimentálny správca spomínaný v Spark dokumentácii [15].

### Jadro

Ako už bolo spomenuté jadro je časť, ktorá spája všetky rôzne poskladané Spark systémy. Jeho úlohami sú spravovanie pamäte a RDD (pozri podkapitolu 2.3), spravovanie a distribúcia úloh v spolupráci so správcom klastra (pozri podkapitolu 2.2), odolnosť proti poruchám a poskytovanie základných I/O operácií umožňujúcich spoluprácu s dátovým úložiskom.

### Moduly

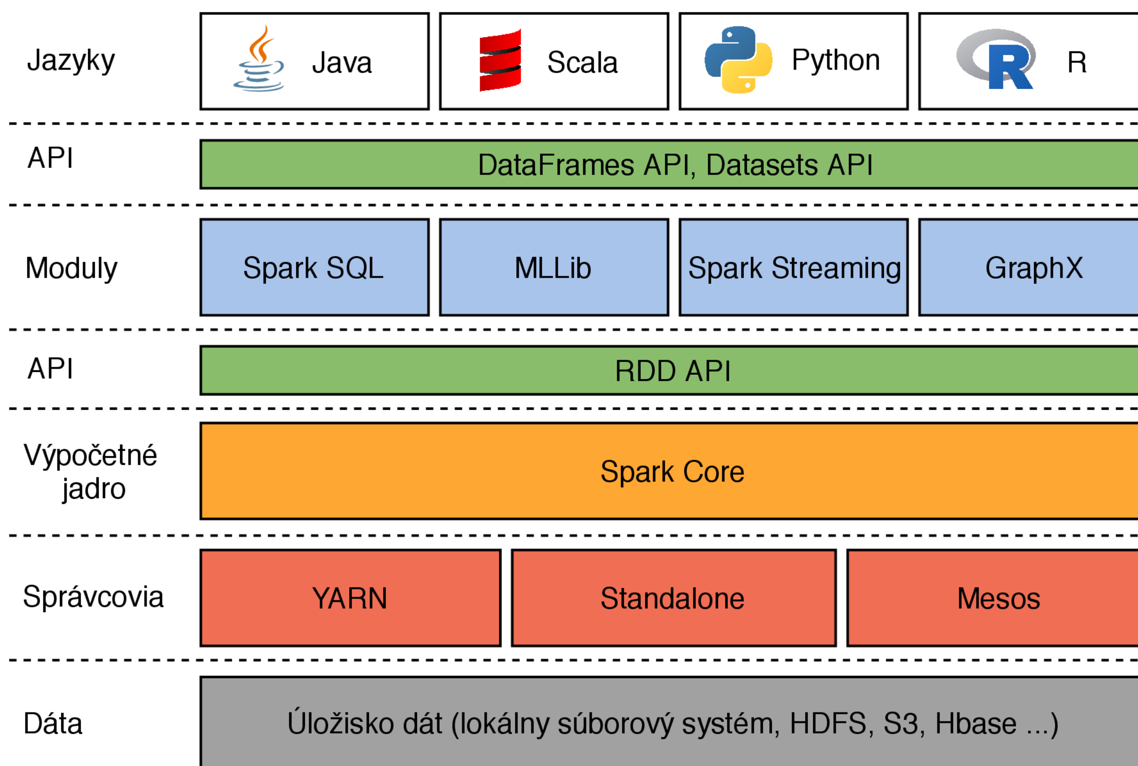
Moduly sú vysokoúrovňové komponenty rozširujúce funkcionalitu, ktorú poskytuje jadro Apache Spark. Moduly sú navrhnuté tak, aby mohli spolupracovať a dali sa medzi sebou kombinovať. Štyri hlavné moduly popisované v dokumentácii Apache Spark [15] a publikácii Learning Spark [8] sú:

- Spark SQL – je modul určený na spracovanie štruktúrovaných dát. Pracuje sa s ním cez Dataset a DataFrame rozhrania, ktoré modulu poskytujú viac informácií o štruktúre dát než je schopné poskytovať RDD rozhranie. Tieto informácie využíva Spark SQL na optimalizovanie výpočtov nad dátami. Pri výpočtoch sú dáta spracovávané rovnakým výpočtovým jadrom nezávisle na jazyku a použitom API. Podporované formáty vstupných dát sú napríklad `JSON` (JavaScript Object Notation), `parquet`, `csv`, `txt` a ďalšie.

- MLLib – je modul, ktorý prichádza s funkciami strojového učenia. Poskytuje implementácie rôznych algoritmov ako sú klasifikácia, regresia, zhlukovanie a ďalšie. Od verzie Apache Spark 2.0 je primárnym API pre túto knižnicu DataFrame API namiesto RDD API.

- Spark Streaming – je modul určený na spracovávanie tokov dát v reálnom čase. Vstupné toky môžu pochádzať zo zdrojov ako Apache Kafka, Apache Flume, TCP (Transmission Control Protocol) soketov atď. Prijímané toky sú rozdeľované na tzv. dávky, ktoré sú ďalej spracovávané systémom Spark a výstupom sú výsledné dávky. Na ich spracovanie je možné použiť aj ostatné knižnice ako MLLib alebo GraphX.

- GraphX – je modul určený na prácu s grafmi. Poskytuje rozšírené RDD rozhranie, ktoré dovoľuje priraďovať vlastnosti hranám a vrcholom v grafe. Tento modul ďalej obsahuje knižnice s grafovými algoritmami a knižnice pre tvorbu a manipuláciu grafov.

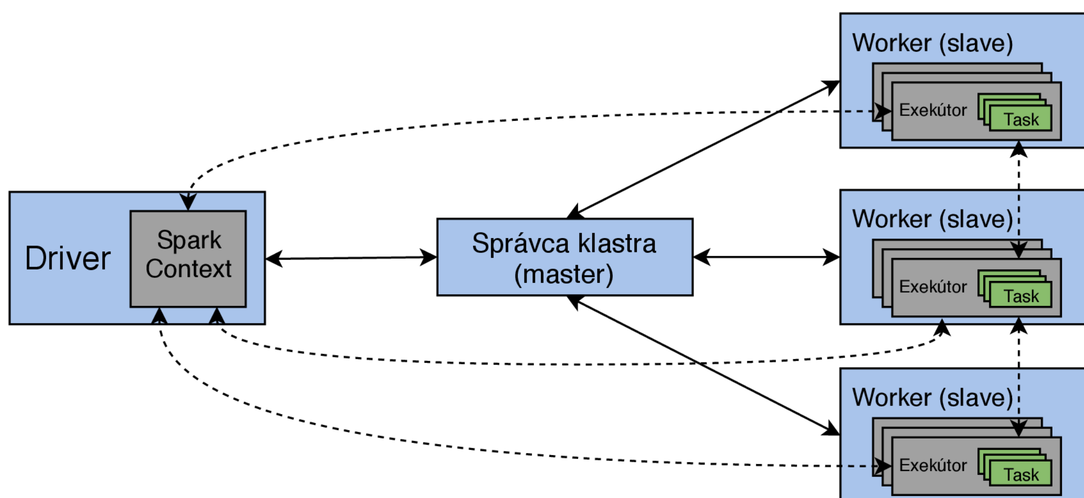



Obr. 2.2: Ekosystém Apache Spark (inšpirované z [8])

## 2.2 Architektúra Spark klastra

Spark využíva architektúru *master/worker (slave)*. Komponenty tvoriace Spark klaster sú (ilustrácia spolupráce týchto komponent je uvedená na obrázku 2.3):

- *driver* – je to proces (program), ktorý vytvára `SparkContext` a obsahuje metódu `Main()`. Z pohľadu ekosystému je `SparkContext` prístupový bod k funkcionalite jadra prostredia Spark. *Driver* ako proces beží vo svojom vlastnom JVM. Jeho úlohou je vytvárať úlohy (angl. tasks), ktoré sa následne zadávajú na vykonanie *worker* uzlom v klastri. *Driver* je nezávislý na *master/slave* uzloch a môže bežať na akomkoľvek stroji adresovateľnom *worker* uzlami zo siete, v ktorej sa nachádza klaster. *Driver* tak môže bežať vo veľkej fyzickej vzdialenosti od klastra. V závislosti od toho kde sa *driver* nachádza sa rozlišujú 2 módy:
  - klientský (angl. client) – *driver* beží na stroji, na ktorom bol spustený. Tento mód je, okrem iného, vhodný pre potreby ladenia. Výhodou je možnosť zobraziť výsledky výpočtov bez nutnosti ukladania dát na disk a možnosť mať na lokálnom stroji priamo k dispozícii webové rozhranie poskytované *driverom*. Toto rozhranie zobrazuje množstvo informácií o prebiehajúcich/hotových výpočtoch v klastri. V reálnom nasadení je dobré *driver* v klientskom móde spúšťať v rovnakej sieti ako je klaster alebo inej blízkej sieti. Minimalizuje sa tak oneskorenie spôsobené sieťovou komunikáciou.
  - klaster (angl. cluster) – *driver* beží priamo v klastri na jednom z *worker* uzlov. O tom, na ktorom z *worker* uzlov bude *driver* bežať rozhoduje *master* uzol podľa dostupných zdrojov týchto uzlov. Tento režim je výhodný hlavne na minimalizovanie oneskorení spôsobených prenosom dát po sieti. Používa sa pri spúšťaní Spark programu zo vzdialenej siete. V čase písania tejto práce uvádza dokumentácia Apache Spark [15], že ho nie je možné použiť pri Python aplikáciách so Standalone správcom.




Obr. 2.3: Architektúra Apache Spark (inšpirované z [15])

- správca klastra (angl. cluster manager) – tento uzol vystupuje v úlohe *master*. Jeho miesto v Spark ekosystéme je vyznačené červenou farbou na obrázku 2.2. Jeho konkrétne implementácie sú popísané v podkapitole 2.1.
- *worker* – je jeden z výpočetných uzlov v Spark klastri. *Worker* prijíma serializované úlohy, ktoré následne vykonáva. V reálnom klastri je väčšinou na jednom výpočetnom stroji spustený jeden *worker*. Iný počet má zmysel prevažne len na testovacie účely. *Worker* uzly spúšťajú samostatný JVM proces, ktorý sa nazýva exekútor (angl. executor) a vykonáva pridelené úlohy. Systém spúšťania týchto procesov závisí na použitom správcovi klastra. Každý samostatný program spustený v klastri má pre svoje úlohy vlastný oddelený JVM proces.

Na testovacie účely môže Spark fungovať v lokálnom režime. V tomto prípade nie je potrebné vytvárať celý funkčný klaster. Pre spustenie lokálneho režimu stačí mať štandardne nainštalovaný balík Spark a pomocou priblalených nástrojov (spark-shell, spark-submit) špecifikovať master URL ako `--master = local[n]`. Číslo „n“ určuje počet vlákien, ktoré môže Spark využiť na paralelizáciu. Často sa tento parameter zadáva ako „*“, čo znamená využitie maximálneho počtu vlákien dostupných na lokálnom stroji. V tomto režime sa celá infraštruktúra (*driver*, exekútori atď.) spustí v rámci jedinej inštancie JVM. Ako ďalej uvádza Laskowski v publikácii [10], lokálny režim sa zvykne nazývať aj „Spark in-process“.

## 2.3 Distribuované dátové kolekcie

Primárnou abstrakciou pre reprezentáciu dát v Sparku sú tzv. RDD kolekcie. Ďalšími abstrakciami reprezentujúcimi dáta na vyššej úrovni sú DataFrame a Dataset. Obidve tieto pokročilejšie abstrakcie sú vybudované na základoch RDD. V tejto podkapitole budú bližšie popísané všetky tieto abstrakcie spolu s operáciami nad nimi a ich spracovaním v Spark klastri.

### RDD (Resilient Distributed Datasets)

RDD sú základnou reprezentáciou dát spracovateľných v prostredí Apache Spark. Sú to kolekcie odolné proti poruchám umožňujúce paralelné spracovanie. Dáta sú v rámci RDD rozdelené do celkov, ktoré sa nazývajú oddiely (angl. partitions). Každý z týchto celkov je spracovávaný v niektorom z uzlov klastra, čo umožňuje paralelizmus. Ako uvádza dokumentácia [15], RDD sa dajú vytvoriť dvomi spôsobmi:

- paralelizáciou už existujúcej kolekcie – volaním funkcie `parallelize`, ktorú poskytuje objekt `SparkContext` v *driver* programe. Pomocou parametra tejto metódy je možné nastaviť aj požadovaný počet oddielov, na ktorý Spark túto kolekciu rozdelí. Štandardne sa Spark snaží kolekciu rozdeliť medzi uzly automaticky tak, aby spracovanie bolo čo najefektívnejše v danom klastri.
- Načítaním súboru z externého úložiska – pomocou funkcií Hadoop API je Spark schopný načítať dáta z externých dátových úložísk (napr. HDFS) a transformovať ich do RDD. Hadoop API poskytuje nasledujúce funkcie pre spracovanie rôznych typov vstupných dát:
  - `textFile` – načíta vstupné textové súbory tak, že jeden záznam v kolekcii zodpovedá jednému riadku textového súboru. Funkcia podporuje aj komprimované súbory.

- `wholeTextFiles` – načíta súbor ako celok a vytvorí kolekciu dvojíc, ktoré sa skladajú z mena súboru a jeho obsahu. Táto funkcia je vhodná napríklad pre väčšie množstvo súborov malej veľkosti. Pri použití na veľké súbory by degradovala výkon systému.
- `sequenceFile` – spracováva súbory, ktoré obsahujú dáta vo formáte kľúč-hodnota.
- `hadoopFile`/`newAPIHadoopFile`, `hadoopRDD`/`newAPIHadoopRDD` – funkcie vhodné pre ostatné formáty, ktoré Spark natívne nepodporuje. Spracovanie týchto formátov je potrebné implementovať pomocou triedy zdedenej od triedy Hadoop frameworku `inputFormat`. Tento princíp je popísaný ďalej v implementácii aplikácie, kde bol použitý na spracovanie `pcap` súborov (pozri podkapitolu 5.2).

Všetky funkcie Hadoop API sú dostupné prostredníctvom objektu `SparkContext`. Načítanie dát z lokálneho úložiska je možné rovnako ako z externého, s tým rozdielom, že dáta musia byť dostupné každému uzlu v klastri. Lokálne úložisko je vhodné na testovacie účely pre Spark v lokálnom móde.

RDD kolekcie sú podobné kolekciám v jazyku Scala a sú poskytované jadrom systému Spark (pozri podkapitolu 2.1). Rozdiel medzi týmito kolekciami je, že kým kolekcie jazyka Scala sú spracovávané v rámci jedného JVM, RDD môžu byť spracovávané vo viacerých JVM naprieč celým klastrom. Paralelizmus je dosiahnutý rozdelením kolekcie RDD na oddiely a ich distribúciou medzi výpočetné uzly. Tento proces zostáva vďaka RDD abstrakcii pre používateľa (programátora) skrytý. Ovplyvňovať ho môže používateľ napríklad konfiguráciou počtu oddielov, na ktoré sa RDD rozdelí. Ďalšími dôležitými vlastnosťami RDD vybranými podľa publikácie [10] sú:

- ukladanie v operačnej pamäti – hlavným prínosom tohto princípu je výrazné zvýšenie rýchlosti spracovávania dát (napr. proti systému Hadoop),
- nemennosť (angl. immutability) – vytvorené RDD nie je možné meniť, je ho možné pomocou operácií transformovať na iné RDD,
- *lazy evaluation* – dáta v RDD sú dostupné až pri ich vyžiadaní niektorou akciou. Do tej doby sú reprezentované len odkazom na zdroj dát a sériou operácií, ktoré sa nad zdrojovými dátami majú vykonať. Tento princíp dovoľuje zadávať väčšie množstvo menších operácií bez toho, aby degradovali výkon systému. Redukovaná je takto tiež komunikácia medzi *driverom* a klastrom. Z hľadiska práce s kolekciami je výhodou možnosť abstrakcie nekonečných kolekcií, keďže reálne sú vyhodnocované len vyžiadané dáta.
- Typovosť – RDD môžu obsahovať všetky triedy podporovaných jazykov, vrátane tried definovaných používateľom.

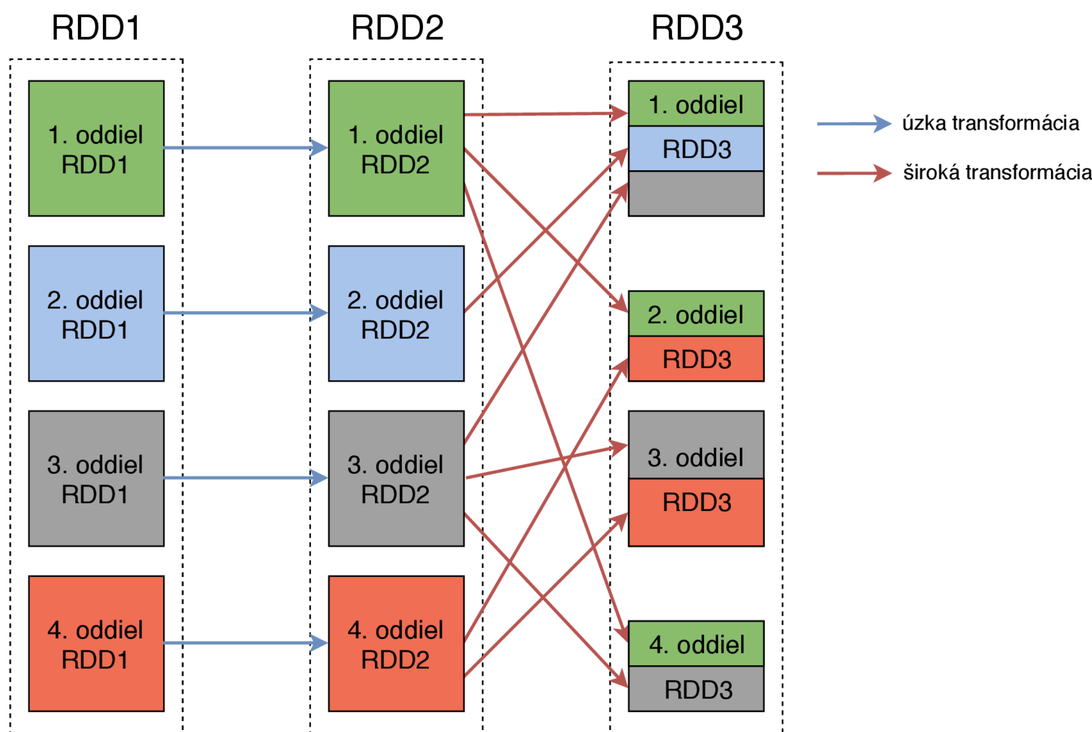
RDD podporujú dva typy základných operácií:

- akcie – sú operácie nad RDD, ktoré nevracajú ďalšie RDD, ale hodnoty iného typu. Tieto operácie spôsobujú spustenie vyhodnocovania dát v RDD. Ich výsledky sú posielané naspäť programu *driver* alebo sú ukladané do úložiska. Príkladom funkcií posielajúcich výsledky na *driver* sú `count`, `collect`, `take`, `reduce`, `foreach` a ďalšie. Funkcie ukladajúce výsledky do úložiska sú napríklad `saveAsTextFile` alebo `saveAsObjectFile`.

- Transformácie – sú operácie, ktoré z existujúcich RDD vytvárajú nové RDD. Sú vyhodnocované pomocou *lazy evaluation* a delia sa na dva základné typy.

  Prvým typom sú tzv. úzke transformácie. Sú to transformácie pri ktorých platí, že oddiely RDD potomka môžu závisieť buď na jednom konkrétnom oddieli rodičovského RDD alebo na jedinečnej podmnožine oddielov rodičovského RDD. Príklad tohto typu transformácie je uvedený v ľavej časti obrázku 2.4, kde RDD1 je rodičovské RDD a RDD2 je potomok. Úzka transformácia môže byť vykonaná na určitom oddieli RDD bez potreby znalosti iných oddielov, všetky závislosti sú tak známe už v čase prekladu a nezávisia na konkrétnych dátach. Transformácia tak môže byť vykonaná v každom uzle klastra nezávisle a zlepšiť tým výkonnosť systému. Príkladom sú funkcie `map`, `flatMap`, `filter` alebo `union`.

  Druhým typom sú tzv. široké transformácie, ktoré závisia na konkrétnych dátach. Dáta musia byť usporiadané v oddieloch tak, ako to vyžaduje konkrétna funkcia. Napríklad pri funkcii `groupByKey` musia byť všetky dáta s rovnakým kľúčom v rovnakom oddieli. Pokiaľ nie sú dáta takto usporiadané v rodičovskom RDD, musia sa tieto dáta premiešať medzi oddielmi, aby bolo možné vykonať operáciu. Premiešanie a zoskupenie potrebných dát môže viesť k zmene počtu oddielov. Na rozdiel od úzkych transformácií bývajú tieto operácie, kvôli potrebe presunu vstupných dát medzi oddielmi, časovo náročné. Tento typ transformácií ilustruje obrázok 2.4 v pravej časti pomocou červených šípok. Veľkosť a počet RDD oddielov potomka závisí od konkrétnej použitej operácie. Príkladmi ďalších operácií tejto kategórie sú `sort`, `reduceByKey` atď.




Obr. 2.4: Transformácie RDD (inšpirované z [9])

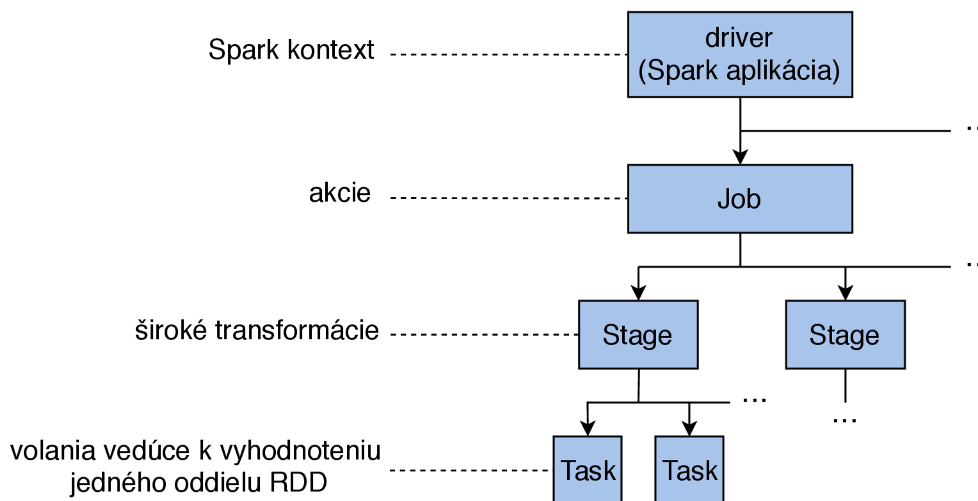
Podrobný popis všetkých uvedených funkcií sa nachádza v dokumentácii Apache Spark [15]. Popis operácií nad RDD vychádza z publikácie [9].

### Spracovanie RDD

Ako už bolo spomenuté v časti 2.3, transformácie RDD sa vyhodnocujú pomocou *lazy evaluation* a ich vyhodnocovanie je spustené až keď sú dáta vyžiadané nejakou akciou. Toto dovoľuje vytváranie tzv. graf závislostí (angl. lineage graph alebo dependency graph) RDD v rámci jedného vyhodnotenia. Graf závislostí obsahuje RDD, z ktorého sú vyžadované dáta a všetky RDD, ktoré ho predchádzajú (tj. od rodičov až po zdroj dát). Každé RDD v tomto grafe drží odkaz na svojho rodiča/rodičov.

Graf závislostí slúži ako vstup pre DAG plánovač, ktorý je súčasťou prostredia Apache Spark. Ten z grafu závislostí vytvára orientovaný acyklický graf (ďalej DAG, podľa angl. Directed Acyclic Graph), ktorého uzlami sú RDD a orientované hrany reprezentujú operácie (transformácie) vykonávané nad RDD. Orientovaný je kvôli smeru hrán od rodičovského uzla k potomkovi a acyklický kvôli nemožnosti vzniku cyklov z dôvodu, že každá hrana ukazuje na ďalšiu generáciu uzlov a nikdy nie na predchádzajúcu. Príklad vzťahov medzi RDD, z ktorých sa vytvára DAG je možné vidieť na obrázku 2.4. DAG vytvorený v DAG plánovači sa delí na komponenty, ktorých hierarchia je naznačená na obrázku 2.5. Týmito komponentami sú (aby nedošlo k zámene pojmov uvádzam pomenovania komponent v originálnom anglickom znení):

- *job* – je to najvyšší element hierarchie a zodpovedá jednému DAG, tj. jednej akcii. Hrany DAG zodpovedajú vzťahom medzi RDD oddielmi, ale akcie vracajú objekty, ktoré nie sú typu RDD. To je dôvod prečo musí byť každý *job* ohraničený akciami. Každý *job* sa skladá z určitého počtu úzkych a širokých transformácií.

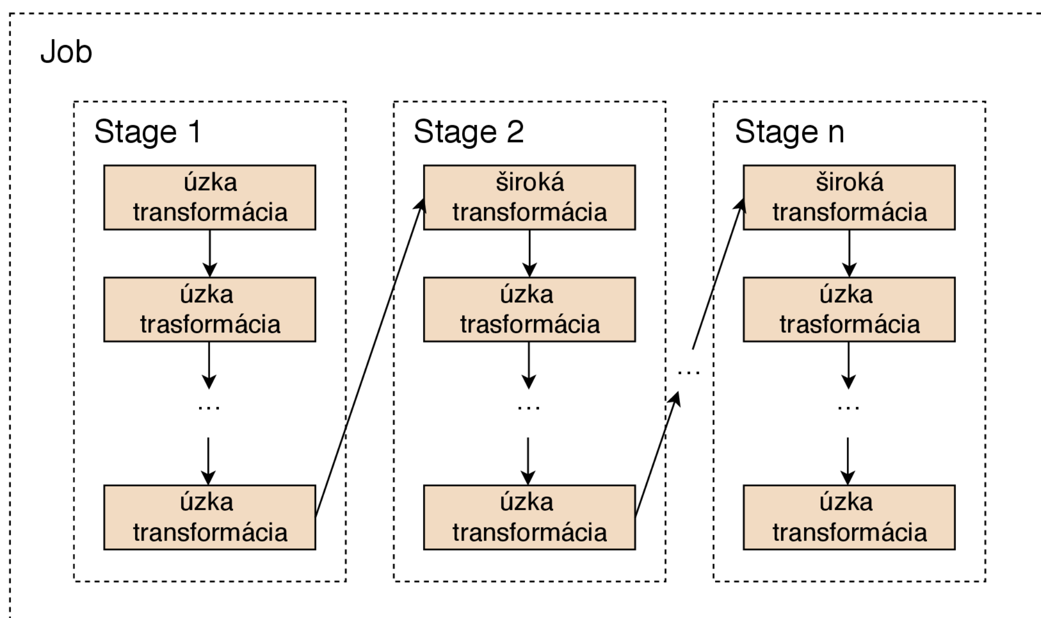



Obr. 2.5: Hierarchia komponent spracovania dát v Spark aplikácii. V ľavej časti sú uvedené volania Spark API vedúce k vytvoreniu jednotlivých komponent (pravzaté z [9])

- *Stage* – je komponentou, na ktoré sa delí jeden *job*. K deleniu vedú široké transformácie, ako je vidieť na obrázku 2.6. Za jeden *stage* sú považované výpočty, ktoré môže vykonať jeden exekútor (pozri obrázok 2.3) bez toho, aby musel komunikovať s ostatnými. Kvôli potrebe komunikácie s *driverom* musia byť za sebou idúce *stage* v rámci jedného *jobu* spracovávané sekvenčne a nie paralelne. Výnimku tvoria transformácie, ktorých vstupom nie je len jedno RDD (napr. operácia `join`). V tomto prípade môžu byť *stage* každého vstupného RDD spracované paralelne. Sekvenčné spracovávanie je dôvodom časovej náročnosti širkoých transformácií.

- *Task* – je komponentou, na ktoré sa delí jeden *stage*. Každý *task* reprezentuje vykonanie rovnakých operácií na rôznych dátach. Počet *taskov* v rámci jedného *stage* zodpovedá počtu oddielov výstupného RDD z daného *stage*. To je aj dôvodom, prečo jeden *stage* môže obsahovať maximálne jednu širokú transformáciu. Pred začiatkom ďalšieho *stage* musia byť dokončené všetky *tasky* aktuálneho. O tom koľko *taskov* môže byť paralelne spustených jedným exekútorom rozhoduje počet jadier, ktoré má daný exekútor alokované. Maximálny počet jadier na jedného exekútora sa dá nastaviť v konfigurácii.

Popis grafu závislostí RDD a DAG vychádza z publikácie [10], popis komponent spracovania RDD z publikácie [9].




Obr. 2.6: Detail delenia úloh v Spark aplikácii (Inšpirované podľa obrázku z Data Flair tutorialu https://data-flair.training/blogs/dag-in-apache-spark/)

### DataFrame a Dataset

Primárnou dátovou štruktúrou bolo v Spraku až do verzie 1.3 RDD. V tejto verzii vznikol nový typ distribuovanej kolekcie DataFrame. Vo verzii 1.6 bol uvedený Dataset ako ďalší typ distribuovanej kolekcie, ktorý v sebe spojil výhody kolekcií DafaFrame a RDD. Vo verzii Spark 2.0 boli kolekcie DataFrame a Dataset zjednotené.

Ako uvádza dokumentácia Apache Spark [15], DataFrame je distribuovaná kolekcia dát organizovaná do pomenovaných stĺpcov, ktorá je konceptuálne ekvivalentná tabuľke v relačnej databáze. DataFrame môže byť vytvorený napríklad z existujúcich RDD, štruktúrovaných dátových súborov, tabuliek atď. DataFrame API je dostupné vo všetkých Sparkom podporovaných jazykoch. Výhody kolekcie DataFrame, uvádzané aj v dokumentácii [15] a publikácii [9] sú:

- Catalyst – je vstavaný optimalizátor. Vďaka optimalizáciám má DataFrame väčšiu výkonnosť oproti RDD, ktorého operácie nie sú optimalizované žiadnym vstavaným optimalizátorom.

- Tungsten – je nový správca pamäte, ktorý bol uvedený ako komponenta Spark SQL vo verzii Spark 1.4. Tungsten reprezentuje dáta v binárnej forme, ktorá zaberá menej miesta v pamäti než dáta serializované Kryo serializátorom. Spolu s ďalšími optimalizáciami umožňuje, okrem šetrenia miesta, aj rýchlejšie operácie nad dátami.
- Schéma – popisuje štruktúru dát v DataFrame. Konkrétne dáta v rámci jedného riadku kolekcie DataFrame. RDD podobnú schému neobsahujú. Vďaka nej môže Tungsten efektívnejšie ukladať dáta.

Nevýhodou DataFrame kolekcie je priebeh typovej kontroly až za behu, na rozdiel od RDD a Datasetu, kde prebieha typová kontrola v už v čase prekladu (pozri tabuľku 2.1). Táto vlastnosť je v kontraste so silnou statickou typovosťou jazyka Scala. Od verzie Spark 2.0 je DataFrame v jazyku Scala len alias pre `Dataset[Row]`. V jazyku Java sa DataFrame používa ako `Dataset<Row>`. Vo verzii Spark 2.2.0 nie je Dataset API pre jazyky Python a R dostupné. Ako uvádza Laskowski v publikácii [10], objekt `Row` je generický objekt reprezentujúci jeden riadok kolekcie DataFrame, ktorý je dostupný pomocou indexu, mena alebo zhodou vzorov v jazyku Scala.

Poslednou a najpokročilejšou kolekciou dostupnou v Apache Spark 2.2.0 je Dataset. Dataset má všetky spomínané výhody kolekcie DataFrame, ale na rozdiel nej prichádza s typovou kontrolou v čase prekladu a možnosťou funkcionálnych operácií nad kolekciou. Ako je vidieť aj z tabuľky 2.1 spája týmto výhody kolekcií RDD a DataFrame.

| | **RDD** | **DataFrame** | **Dataset** |
|---|---|---|---|
| schéma | × | ✓ | ✓ |
| funkcionálne operácie | ✓ | × | ✓ |
| optimalizácia výkonu | × | ✓ | ✓ |
| optimalizácia pamäte | × | ✓ | ✓ |
| podporované jazyky vo verzii Spark 2.2.0 | Scala, Java, Python, R | Scala, Java, Python, R | Scala, Java |
| syntaktická analýza | v čase prekladu | v čase prekladu | v čase prekladu |
| sémantická analýza | v čase prekladu | za behu | v čase prekladu |

Tabuľka 2.1: Porovnanie kolekcií RDD, DataFrame a Dataset

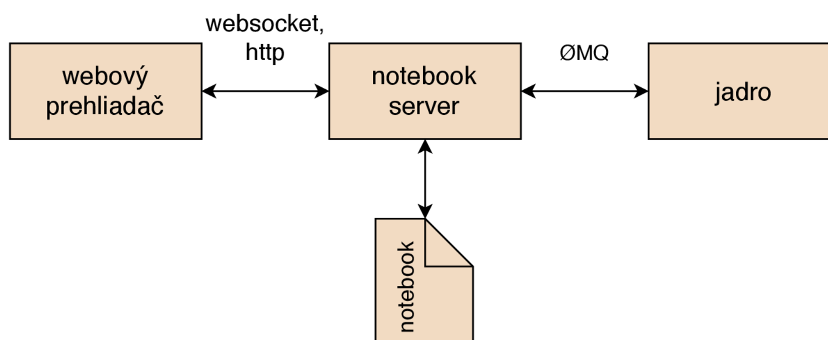
## 2.4 Vizualizácia dát

Dáta spracované v Apache Spark klasrti prostredníctvom konzolového Spark shellu je možné zobraziť priamo v konzole alebo uložiť do výstupného súboru. Tieto možnosti sú vhodné pri zobrazovaní výrazne agregovaných dát alebo pri menších objemoch vstupných dát. Distribuované spracovanie je však v prvom rade určené na spracovanie veľkých objemov dát. Zvýšeniu prehľadnosti výsledkov spracovania dát pomôže využitie niektorého z vizualizačných nástrojov a ich zobrazenie napríklad v podobe grafov. Po spracovaní a uložení dát (do výstupného súboru alebo vhodnejšie do databázy) je možné použiť prakticky akýkoľvek vizualizačný nástroj nezávislý na ostatných použitých technológiách. Existujú však riešenia, ktoré sú schopné priamo využívať infraštruktúru Spark klastra. Takýmito riešeniami sú:

- Jupyter Notebook – je interaktívna webová aplikácia na vytváranie dokumentov, ktoré kombinujú kód, popisný text, výpočty a vizualizáciu (notebookov). Aplikácia v minulosti známa pod názvom IPython Notebook vychádza z IPython projektu. IPython projekt zahŕňa:
  - interaktívny shell a
  - výpočetné jadro, ktoré poskytuje možnosť výpočtov s používateľským rozhraním. Používateľské rozhranie môže mať napríklad podobu spomínaných notebookov.

  Jupyter podporuje rôzne druhy jazykov práve implementáciou rôznych výpočetných jadier. Jadrá vhodné pre Spark a Scalu sú napríklad Apache Toree alebo sparkmagic[1]. Na obrázku 2.7 je vidno, že do kontaktu s notebookmi neprichádza jadro, ale len notebook server. To dovoľuje vytvárať a upravovať notebooky aj bez jadra potrebného pre preklad. Jadro je možné doinštalovať dodatočne. Výhodami Jupyteru sú:
  - podpora Apache Spark,
  - vytváranie notebookov, úprava notebookov a spúšťanie kódu v prostredí webového prehliadača,
  - open source projekt s aktívnou komunitou,
  - možnosť vizualizácie výsledkov v rôznych formátoch ako sú HTML, LaTeX, PNG, atď.

  Podrobnejšie informácie sú k dispozícii v dokumentácii projektu Jupyter [7].




Obr. 2.7: Princíp fungovania Jupyter notebookov. ØMQ alebo tiež ZeroMQ je knižnica podporujúca posielanie asynchrónnych správ, určená pre distribuované prostredie. (Inšpirované podľa schémy architektúry z dokumentácie Jupyteru http://jupyter.readthedocs.io/en/latest/architecture/how_jupyter_ipython_work.html)

- Beaker[2] – je notebookovo orientované, multiplatformné prostredie podporujúce, okrem iných, aj jazyk Scala a Apache Spark. Princíp fungovania aj výhody sú podobné projektu Jupyter. Zaujímavou vlastnosťou je automatický preklad medzi jazykmi. Ten umožňuje pracovať s dátami v každej bunke notebooku v inom jazyku, bez nutnosti vytvárať prechodné súbory s dátami.

[1] Prehľad jadier pre Jupyter je dostupný na https://github.com/jupyter/jupyter/wiki/Jupyter-kernels.

[2] Informácie o projekte Beaker nad rámec súhrnu sú dostupné na http://beakernotebook.com/.

- Spark Notebook[3] – je notebookovo orientovaný projekt zameraný priamo na jazyk Scala a distribuované prostredie Apache Spark. Vo svojom webovom rozhraní dokáže kombinovať Scala kód, SQL požiadavky, Markup a JavaScript.

- Apache Zeppelin – je takisto ako predchádzajúce vizualizačné nástroje open-source webová interaktívna aplikácia umožňujúca vytváranie záznamov. Záznam (angl. note) je v terminológii Apache Zeppelin ekvivalentný názvu notebook v prostredí Jupyter Notebook. Jedná sa teda o dokument, v ktorom sa kombinuje kód, textový popis a vizualizácia. Príklad záznamu vytvoreného v Apache Zeppelin je uvedený na obrázku 2.8. Ako je na ňom vidieť, je potrebné vždy označiť interpret, ktorým sa má následný kód interpretovať (prekladať). Zeppelin v aktuálnej verzii 0.7.3 podporuje celú radu interpretov[4] vrátane Sparku.



Obr. 2.8: Ukážka záznamu v notebooku Apache Zeppelin

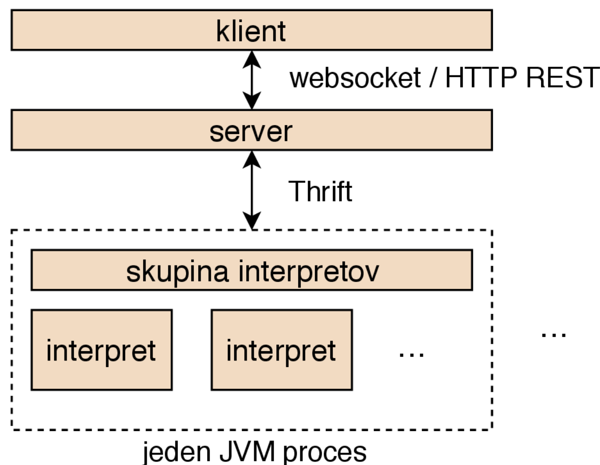
Každý interpret je zaradený do jednej zo skupín. Interprety z jednej skupiny sú spustené v rámci jedného JVM a môžu sa vzájomne referencovať. Týmto spôsobom je napríklad možné zdieľať `SparkContext` naprieč interpretmi zo skupiny Spark. V tejto skupine sú:

  – %spark – interpret pre jazyk Scala,

[3]Informácie o projekte Spark Notebook nad rámec súhrnu sú dostupné na https://github.com/spark-notebook/spark-notebook.

[4]Úplný zoznam podporovaných interpretov je možné nájsť na https://zeppelin.apache.org/supported_interpreters.html.

– %spark.pyspark – interpret pre jazyk Python,
– %spark.r – interpret pre jazyk R,
– %spark.sql – interpret pre Spark SQL,
– %spark.dep – dynamické načítanie závislostí pre %spark a %spark.pyspark.




Obr. 2.9: Zjednodušená architektúra Apache Zeppelin (inšpirované z [16])

Na obrázku 2.9 je naznačená zjednodušená architektúra Apache Zeppelin. Server komunikuje s rôznymi skupinami interpretov pomocou rozhrania Thrift, ktoré slúži na RPC komunikáciu medzi rôznymi platformami a jazykmi. Klient potom vo forme záznamov v notebooku zobrazuje dáta poskytované serverom. Vďaka jeho architektúre je možné vytvárať a pridávať aj ďalšie intreprety.

Podporou spomínaných interpretov je Apache Zeppelin vhodným nástrojom na vizualizáciu dát spracovaných Sparkom. Tieto a ďalšie informácie ohľadne tohto projektu je možné nájsť v dokumentácii [16].

# Kapitola 3

# Analýza požiadaviek na aplikáciu

V rámci tejto práce sa predpokladá vytvorenie aplikácie, ktorá bude schopná spracovávať a vyhodnocovať vybrané prvky sieťovej komunikácie. Odlišovať sa bude od iných aplikácií určených sa spracovanie sieťovej komunikácie[1] tým, že bude orientovaná na Big Data. K svojmu behu bude využívať voľne dostupnú, open-source infraštruktúru distribuovaného prostredia Apache Spark. Analýza bude prebiehať na zaznamenaných statických dátach, analýza v reálnom čase je uvažovaná len ako možné rozšírenie v budúcnosti. Požadovanú funkcionalitu aplikácie je možné rozdeliť na dve časti:

- analýzu obsahu paketov a
- analýzu sieťových tokov.

Aplikácia bude schopná spracovávať vstupné dáta vo formátoch `pcap` a `JSON`.

## 3.1 Analýza obsahu paketov

Pri veľkých objemoch dát je analýza obsahu paketov veľmi časovo náročná. Použitie distribuovaného prostredia by malo priniesť významné urýchlenie tejto činnosti. Plánovaná funkcionalita aplikácie z hľadiska analýzy obsahu paketov je táto:

- Získanie URL z hlavičky HTTP (Hypertext Transfer Protocol) – túto informáciu bude možné ďalej spracovať, napríklad vyhľadávať URL regulárnym výrazom, hľadať URL používané malvérom a pod.

- Extrakcia a opätovné zloženie TCP toku – obsah TCP komunikácie bude poskladaný do podoby dátového toku, ktorý bude možné ďalej analyzovať alebo uložiť do výstupného súboru.

- Extrakcia súborov pre protokoly HTTP a SMTP (Simple Mail Transfer Protocol) – pomocou extrakcie TCP toku z predchádzajúceho bodu bude možné v ďalšej analýze toku (tzv. postprocesingu) extrahovať súbory prenášané HTTP a SMTP protokolmi.

- Vyhľadávanie kľúčových slov – v spracovávaných vstupných paketoch bude možné vyhľadávať predom definovaný zoznam kľúčových slov.

[1]Napríklad programy Wireshark (`https://www.wireshark.org/`) a jeho konzolová verzia TShark alebo tcpdump (`https://www.tcpdump.org/`).

- Extrakcia informácií z DNS (Domain Name System) komunikácie – získané informácie poskytnú možnosť ďalšej analýzy ako je identifikácia kompromitovaných DNS serverov a ďalších bezpečnostných rizík spojených s DNS.
- Najfrekventovanejšie DNS domény – aplikácia ukáže, ktoré DNS domény boli najviac zastúpené v DNS požiadavkách.
- Oneskorenie DNS servera – pre každú DNS komunikáciu sa vypočíta čas potrebný pre odpoveď servera. Tieto údaje predstavujú dôležité informácie o dostupnosti DNS služieb.
- Extrakcia informácií z SSL (Secure Sockets Layer)/TLS (Transport Layer Security) v `JSON` formáte – informácie o certifikátoch, použitých šifrovacích algoritmoch, metainformáciách o SSL/TLS a pod. bude možné použiť na ďalšiu analýzu šifrovanej komunikácie. Analýza šifrovanej komunikácie môže pomôcť s odhalením škodlivých šifrovaných sieťových tokov.

## 3.2 Analýza sieťových tokov

Existujú rôzne definície sieťových tokov. Podľa dokumentu RFC (Request for Comments) [12] je sieťový tok sada paketov prechádzajúca určitým bodom v určitom časovom intervale. Všetky pakety patriace do rovnakého toku majú niektoré spoločné vlastnosti. Týmito vlastnosťami najčastejšie bývajú zdrojová IP adresa, cieľová IP adresa, zdrojový port, cieľový port, L3 protokol a vstupné/výstupné rozhranie. Publikácia [13] definuje toky ako jednosmerné rady paketov medzi zdrojom a cieľom s rovnakou zdrojovou IP adresou, cieľovou IP adresou, zdrojovým portom, cieľovým portom a IP protokolom. V tejto práci sa budú pakety klasifikovať do tokov podľa pätice parametrov, ktorými sú zdrojová IP adresa, cieľová IP adresa, zdrojový port, cieľový port a L4 protokol. Aplikácia by mala byť schopná analyzovať tieto údaje o sieťových tokoch:

- Časová os – aplikácia ukáže množstvo tokov, prenesených dát a paketov v čase.
- HTTP vs. HTTPS (Hypertext Transfer Protocol Secure) – zobrazí sa množstvo HTTP a HTTPS komunikácie v čase a ich vzájomné porovnanie.
- LAN (Local Area Network) vs. WAN (Wide Area Network) – zobrazí a porovná sa množstvo komunikácie v rámci lokálnej siete a smerom von z lokálnej siete.
- Zariadenia prenášajúce najviac dát – zobrazia sa zariadenia, ktoré vytvorili najviac sieťových tokov resp. preniesli najviac dát.
- Štruktúra sieťovej komunikácie – zobrazí sa členenie komunikácie súvisiacej s elektronickou poštou v závislosti od použitého protokolu. Anomálie v tejto komunikácii môžu pomôcť pri detekcii šírenia SPAMu.
- Najčastejšie meno hostiteľa (angl. hostname) – zobrazí sa prvých N webových serverov zostupne podľa množstva komunikácie.
- Najčastejšia adresa koncového bodu – zobrazí sa prvých N adries zostupne podľa množstva komunikácie.
- Najčastejšie SMTP servery a odosielatelia e-mailov – zobrazí sa prvých N klientov a serverov zostupne podľa množstva vygenerovanej komunikácie.

# Kapitola 4

# Návrh riešenia

Ako už bolo spomenuté v predchádzajúcej kapitole, inšpiráciou pre návrh funkčnosti aplikácie sú programy na analýzu sieťovej komunikácie akým je napríklad TShark. Voľba implementačného jazyka Java korešponduje s implementačným jazykom existujúcej demo-aplikácie *ndx*, z ktorej v tejto práci vychádzam. Táto demo-aplikácia je schopná spracovávať vstupy vo formáte `pcap`. Používa sa prostredníctvom konštrukcií jazyka Scala v prostredí Spark shell. Súčasťou návrhu je aj jej prispôsobenie a začlenenie do výslednej aplikácie. Výsledná aplikácia bude schopná v podobe jediného `jar` súboru fungovať takisto aj v prostredí Spark shell. Popis jej architektúry a princíp fungovania v tomto prostredí je popísaný ďalej v tejto kapitole. Vďaka takémuto návrhu je možné aplikáciu v nezmenenej podobe použiť aj v ďalších hostiteľských prostrediach, akým je napríklad nástroj na vizualizáciu dát Apache Zeppelin. Tento vizualizačný nástroj bol zvolený z dôvodu jeho prítomnosti na serveroch referenčného Spark klastra. V prípade potreby je možné obdobným spôsobom použiť jeden z ďalších vizualizačných nástrojov spomenutých v podkapitole 2.4.

## 4.1 Spark shell

Súčasťou distribúcie Apache Spark sú interaktívne shelly. Z používateľského hľadiska fungujú tieto shelly veľmi podobne ako dobre známy unixový Bash alebo shell jazyka Python. Na rozdiel od týchto shellov, Spark shell nepracuje len na lokálnom stroji, ale dovoľuje prístup k celému Spark klastru. Z hľadiska architektúry Apache Spark predstavuje Spark shell komponentu *driver*. Jeho fungovanie a spôsob akým pristupuje ku klastru sú popísané v podkapitole 2.2. Spark shelly sú dostupné len v jazykoch Python (PySpark) a Scala (Spark shell), pričom API je pre obidva jazyky rovnaké. Spark shelly pri spustení vytvárajú a poskytujú používateľovi dva objekty:

- `SparkContext` – ako prístupový bod k funkciám jadra systému Spark,
- `SparkSession` – ako prístupový bod k Dataset a DataFrame API.

S pomocou týchto objektov je možné v programe Spark shell vytvárať alebo spúšťať plnohodnotné programy pre Spark. Napriek tomu, že pre jazyk Java nie je k dispozícii shell, dajú sa programy napísané v tomto jazyku spúšťať zo shellu jazyka Scala. V návrhu aplikácie sa ráta so Spark shellom ako vstupným bodom ku klastru. Vďaka interoperabilite jazykov Java a Scala je možné pracovať v programe Spark shell pre jazyk Scala aj s triedami napísanými v jazyku Java. To umožňuje používateľovi jednoducho vytvárať rozšírenia aplikácie

v jazyku Scala a prispôsobiť ju aktuálnym potrebám. Parametre Spark shellu potrebné pre spustenie navrhovanej aplikácie sú:

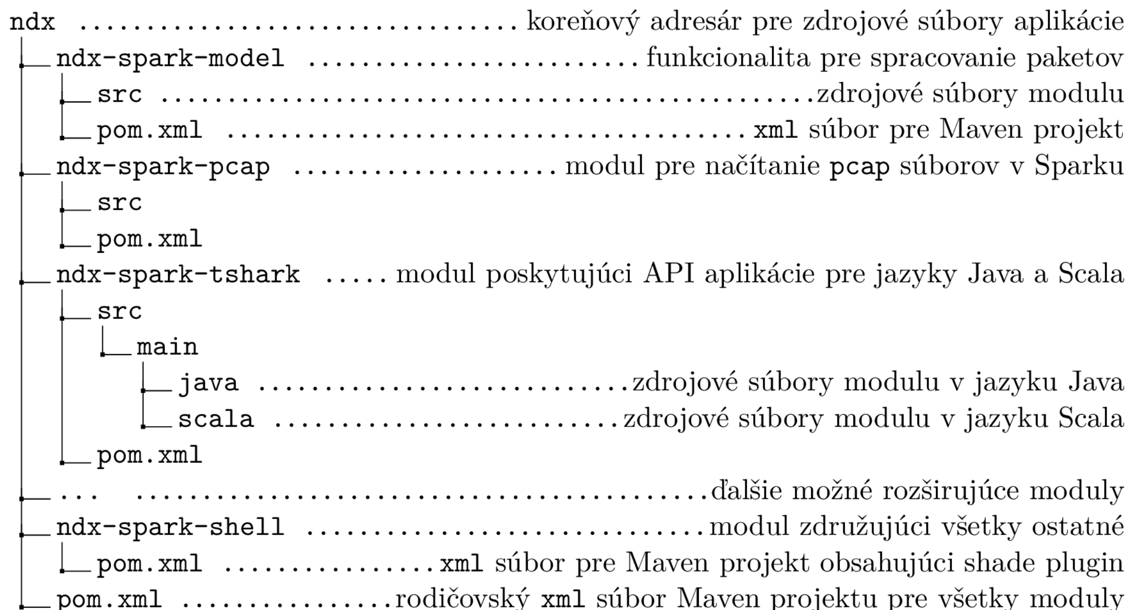
- *--master MASTER_URL* – adresa master uzla v klastri, v ktorom sa majú príkazy Spark shellu spúšťať. Pre Standalone správcu je táto adresa v tvare `spark://host:port`, kde `host` je názov alebo IP adresa *master* uzla a `port` je port *master* uzla.
- *--deploy-mode DEPLOY_MODE* – spustí klientský (*DEPLOY_MODE* = client) alebo klaster (*DEPLOY_MODE* = cluster) mód, ktoré sú popísané v podkapitole 2.2.
- *--jars JARS* – umožní pri spustení Spark shellu pridať do *drivera* uvedené `jar` súbory. *JARS* sú čiarkami oddelené `jar` súbory, ktoré sa dajú následne používať v *driveri*. Pomocou tohto parametra je možné v prostredí Spark shell spúšťať a používať aj programy napísané v jazyku Java.

## 4.2 Architektúra aplikácie

Nástrojom pre preklad aplikácie je Apache Maven, ktorý umožňuje rozdelenie aplikácie na moduly, do ktorých bude logicky rozdelená funkcionalita aplikácie. To uľahčí začlenenie existujúcej demo-aplikácie, ktorá má implementovanú funkcionalitu v dvoch moduloch:

- `ndx-spark-pcap` – slúži na načítanie `pcap` súborov v prostredí Apache Spark,
- `ndx-spark-model` – slúži na spracovanie `pcap` súborov a získanie jednotlivých paketov. Tento modul sa rozšíri a bude v ňom implementovaná spoločná funkcionalita pre všetky ostatné moduly.

Architektúra aplikácie bude modulárna a bude rešpektovať nasledujúcu šablónu, ktorá vychádza z konvencií nástroja Maven:




Modul `ndx-spark-shell` obsahuje Maven shade plugin, ktorý slúži na zabalenie všetkých modulov do jedného `jar` súboru tzv. „fat `jar`“. Jeden `jar` súbor pre všetky moduly je

výhodný pri nasadení aplikácie do nástrojov ako Apache Zeppelin alebo Apache Spark, ktorým potom nie je potrebné predávať každý modul zvlášť. Tento spoločný `jar` súbor ostane relatívne malý (jednotky až nízke desiatky MB), pretože nebude potrebné pribaliť Spark a Hadoop knižnice, ktoré sú už prítomné v hostiteľskom prostredí. Pre budúce rozšírenia predstavuje táto architektúra výhodu možnosti jednoduchého pridávania ďalších nezávislých modulov, ktoré rozšíria funkcionalitu aplikácie.

Ako API aplikácie budú slúžiť triedy so statickými metódami v jazyku Java a tzv. „companion objekt" s metódami pre jazyk Scala. Kvôli prístupu k Spark API bude potrebné metódam dodať inštancie tried `SparkContext` a `SparkSession`. Týmto spôsobom bude implementovaná požadovaná funkcionalita uvedená v kapitole 3. Pomocou jazyka Scala a Spark shellu bude možné kombinovať implementované funkcie aj inými spôsobmi.

# Kapitola 5

# Implementácia

Implementácia aplikácie vychádza zo šablóny uvedenej v podkapitole 4.2. Podľa analýzy požiadaviek uvedených v kapitole 3 musí byť aplikácia schopná spracovávať vstupy vo formátoch `pcap` a `JSON`. Táto kapitola sa postupne venuje popisu implementácie po krokoch, v ktorých aplikácia pracuje a to:

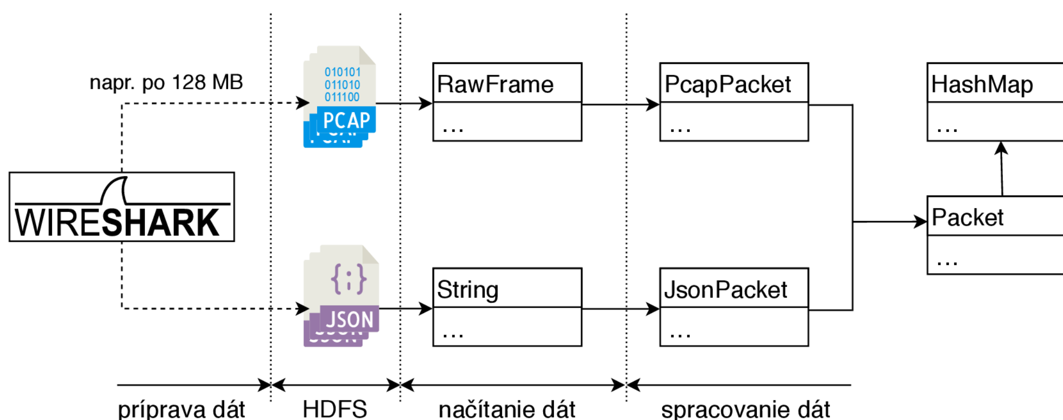
- príprava vstupných dát,
- ich načítanie do aplikácie,
- spracovanie týchto načítaných dát,
- analýza obsahu paketov a sieťových tokov
- a vizualizácia výsledkov získaných analýzou.

Pri implementácii boli použité tieto jazyky a nástroje:

- Java SE 8 – hlavný implementačný jazyk aplikácie. Sú v ňom implementované moduly `ndx-spark-model`, `ndx-spark-pcap` a časť `ndx-spark-tshark`. Demo-aplikácia bola napísaná pôvodne v jazyku Java 6, pri začleňovaní do výslednej aplikácie bola upravená na verziu 8.
- Scala 2.11 – bola použitá na implementáciu paralelne spracovateľných operácií a aplikačného rozhrania pre prostredia Spark shell a Apache Zeppelin. Pôvodne bola aj táto funkcionalita implementovaná v jazyku Java, ale v priebehu vývoja sa ukázalo ako vhodnejšie použitie jazyka Scala. Jedným z dôvodov bola náročnejšia práca s triedami jazyka Java v prostredí Spark shell, pri ktorej bolo potrebné vykonávať konverzie medzi jazykmi a strácala sa efektivita jazyka Scala v tomto prostredí. Ďalším dôvodom bola prehľadnosť a lepšia podpora funkcionálnych operácií v jazyku Scala pri práci s distribuovanými kolekciami RDD. Verzia jazyka bola vybraná kvôli podpore v prostredí Apache Spark 2.2.0 (pozri dokumentáciu [15]). Hlavné verzie jazyka Scala nie sú vzájomne kompatibilné, preto je potrebné dodržať kompiláciu v jednej z podporovaných verzií. Naopak všetky podverzie jazyka Scala 2.11.x sú binárne kompatibilné, takže aplikáciu je možné preložiť pomocou ktorejkoľvek z týchto podverzií bez dopadu na funkčnosť.
- Apache Maven – preklad pomocou tohto nástroja vyhovuje všetkým požiadavkám aplikácie. Natívne umožňuje rozdelenie aplikácie na moduly a automatické sťahovanie

závislostí. Pomocou pluginov je zase možné zabaliť celú aplikáciu do jedného `jar` súboru a simultánne kompilovať Java kód a Scala kód do jedného výsledného bytekódu.

- Apache Spark 2.2.0 – pri vývoji bola použitá lokálna inštalácia Sparku v rovnakej verzii ako inštalácia v referenčnom klastri (podrobnejší popis inštalácie v klastri je k dispozícii v kapitole 6). Lokálna inštalácia bola spúšťaná vždy len v lokálnom móde v rámci jednej inštancie JVM. Beh celej aplikácie v rámci jedného JVM umožnil efektívne ladenie aplikácie pomocou debuggeru. Toto nie je pri paralelnom behu aplikácie vo viacerých JVM možné. Pri vývoji nevznikli žiadne problémy spôsobené odlišnosťami lokálneho a klaster módu. Vo všeobecnosti by mal kód funkčný v klastri fungovať aj v lokálnom móde, ale naopak to nemusí vždy platiť.
- Apache Zeppelin 0.7.3 – bol použitý na zobrazenie požadovaných výstupov z kapitoly 3. Pre tieto účely bol vytvorený záznam pozostávajúci zo Scala a Spark SQL kódov.




Obr. 5.1: Schéma implementácie načítania a spracovania vstupných dát

## 5.1 Príprava vstupných dát

V priebehu implementácie spočívala príprava vstupných dát vo vytvorení `pcap`, `cap` a `JSON` súborov, ktoré obsahovali vzorky dát vhodné na testovanie funkcionality uvedenej v kapitole 3. Veľkosť testovacích súborov určených pre spracovanie v lokálnej inštalácii a lokálnom režime Spraku sa pohybovala v rozmedzí od jednotiek kilobajtov po maximálne 150 megabajtov. Tieto súbory boli načítavané z lokálneho súborového systému. Väčšie vstupy boli testované už priamo v referenčnom klastri. V tomto prípade bolo potrebné uložiť dáta do distribuovaného úložiska HDFS. Vzdialené nahrávanie súborov do HDFS je možné napríklad pomocou konzolovej aplikácie hdfs príkazom

```
HADOOP\_USER\_NAME=<username> hdfs dfs -put <src\_path> <dst\_uri>,
```

kde `<username>` je meno používateľa s prístupom do HDFS, `<src_path>` je cesta k nahrávanému súboru na lokálnom stroji a `<dst_uri>` je URI (Uniform Resource Identifier) HDFS klastra. Meno používateľa je potrebné zadať z dôvodu, že program hdfs použije meno používateľa, pod ktorým je lokálne spustený.

Súbory `pcap` a `cap` je možné zachytávať pomocou Wiresharku alebo hociktorého iného nástroja podporujúceho tento formát. Podporované `JSON` súbory je potrebné generovať

konzolovou aplikáciou TShark. Konvertovať `pcap` súbor na `JSON` je možné aplikáciou TShark príkazom

```
tshark -T ek -r input.pcap > output.json.
```

Dôležitý je parameter „-T ek“, ktorý určuje typ vygenerovaného `JSON` súboru. Programy Wireshark/TShark môžu generovať buď typicky formátované `JSON` súbory alebo pomocou spomínaného parametra naformátované tak, že na každom riadku je samostatný `JSON` s údajmi o pakete. Takýto formát vznikol kvôli distribuovanému spracovaniu v prostredí Elasticsearch[1], tým pádom je vhodnejší aj pre distribuované spracovanie v prostredí Spark. Tento formát sa môže naprieč verziami programu TShark mierne líšiť, pretože stále podlieha vývoju. Formát podporovaný implementovanou aplikáciou bol generovaný programom TShark vo verzii 2.4.4. Táto verzia `JSON` formátu má tieto vlastnosti:

- každý paket je reprezentovaný údajmi práve na dvoch riadkoch. Prvý riadok obsahuje atribúty index, typ a skóre paketu a druhý riadok je samotný paket.
- Obsahuje duplicitné kľúče, čo nie je pre formát `JSON` typické.
- Neobsahuje žiadne polia, iba `JSON` objekty. Polia sú nahradené objektami, v ktorých záleží na poradí elementov, čo odporuje definícii `JSON` formátu. Táto vlastnosť spôsobuje aj prítomnosť duplicitných kľúčov.

## 5.2 Načítanie vstupov

Načítavanie vstupov prebieha na najnižšej úrovni aplikácie pomocou Hadoop API, ktoré je súčasťou distribúcie Sparku. Zjednodušená schéma implementácie načítania dát je uvedená na obrázku 5.1. V tejto podkapitole sú popísané najskôr princípy implementácie delenia vstupných súborov Sparkom, následne potom implementácie načítania vstupov vo formátoch `pcap/cap` a `JSON`, ktoré z týchto princípov vychádzajú.

### Delenie vstupných súborov

Najvýhodnejším spôsobom uloženia súborov v klastri je použitie jedného z distribuovaných úložísk akým je napríklad HDFS. V prípade použitia nedistribuovaného úložiska sú princípy delenia súborov rovnaké, rozdielom je len absencia výhod distribuovaného úložiska (napr. rýchlosť, odolnosť proti poruchám atď.) a nutnosť zaistenia prístupu všetkých uzlov klastra k rovnakým dátam. Súbory sú v klastri uložené v podobe fyzických blokov. Tieto bloky sú časti dát umiestnené v rôznych uzloch klastra (pozri podkapitolu 2.1). Abstrakciu medzi fyzickými blokmi a dátami spracovávanými v rôznych *worker* uzloch v klastri tvoria logické bloky. Z hľadiska implementácie neobsahujú logické bloky žiadne dáta, sú to len ukazovatele na fyzické bloky súboru. V podkapitole o spracovaní distribuovaných kolekcií 2.3 sú tieto logické bloky uvedené pod pojmom oddiely. Z hľadiska teórie spracovania dát v Sparku chápeme oddiely ako bloky dát, na ktoré sú rozdelené vstupné súbory a neprihliadame k spôsobu ich implementácie. V tejto kapitole, zaoberajúcej sa implementáciou, budú oddiely označované pod pojmom logické bloky, aby bol jasný rozdiel medzi blokmi fyzických dát a logickou abstrakciou.

Súbory rozdelené na príliš veľké logické bloky degradujú výkon celého distribuovaného systému, pretože výpočet potom nie je možné dostatočne paralelizovať. Postup delenia súborov v prostredí Spark je nasledujúci:

[1] Elasticsearch je distribuovaný RESTful vyhľadávač, pozri https://www.elastic.co/.

1. súbor je v klastri uložený v distribuovanom úložisku HDFS a je rozdelený na fyzické bloky podľa konfigurácie úložiska.

2. Spark načíta súbor pomocou triedy `InputFormat`, ktorá je zodpovedná za jeho rozdelenie na logické bloky. Z hľadiska efektivity často zodpovedá veľkosť týchto logických blokov veľkosti fyzických. Veľkosť logických blokov sa konfiguruje týmito tromi parametrami:

   (a) `minSize` – konfiguračný parameter *mapreduce.input.fileinputformat.split.minsize* (predvolená hodnota je 1),

   (b) `goalSize` – sa vypočíta podľa vzťahov

   $$splits = \begin{cases} 1 & \text{pre } numSplits = 0 \\ numSplits & \text{v ostatných prípadoch} \end{cases} \tag{5.1}$$

   $$goalSize = \frac{totalSize}{splits} \tag{5.2}$$

   kde `totalSize` je celková veľkosť vstupného súboru, a `numSplits` je konfigurovateľný požadovaný počet logických blokov s predvolenou hodnotou rovnou 0,

   (c) `blockSize` – je to predvolená hodnota veľkosti bloku úložiska (pre HDFS klaster je to v súčasnej verzii 128 MB).

   Tieto parametre je možné konfigurovať na rôznych miestach, viac v dokumentácii Apache Spark [15]. Výsledná veľkosť logického bloku je vypočítaná podľa vzťahu:

   $$splitSize = max(minSize, min(goalSize, blockSize)) \tag{5.3}$$

3. Pomocou triedy implementujúcej rozhranie `RecordReader` je Spark schopný čítať záznamy (riadky alebo iné celky v závislosti od formátu) z logických blokov. Hranice posledného záznamu v logickom bloku nemusia korešpondovať s hranicami logického bloku a väčšinou ani nekorešpondujú. Spark musí vedieť korektne načítať aj takýto záznam, ktorý je rozdelený hranicou logického bloku. To sa deje použitím dvoch princípov:

   (a) z logického bloku v uzle sa prečíta maximum dostupných záznamov. V prípade, že na konci logického bloku ostane časť záznamu, prečíta sa zvyšok zo začiatku ďalšieho logického bloku, ktorý sa ale nachádza na inom vzdialenom uzle.

   (b) Vždy keď sa číta začiatok logického bloku, tak sa preskočí úvodný záznam v prípade, že nie je celý. Záznam sa preskočí nájdením jeho konca v logickom bloku a čítať sa potom začína až za ním. Ukončenie záznamu je dané vstupným formátom. V prípade textového vstupu je to napríklad ukončenie riadku.

   Týmto spôsobom je možné implementovať načítanie vstupných dát bez toho, aby sa nejaké záznamy stratili a korektne tak načítať celý vstupný súbor.

Znalosť delenia súborov na fyzické a logické celky je dôležitá nielen z hľadiska implementácie, ale bude potrebná aj kvôli optimalizácii výkonu v experimentálnej časti. Základy načítania súborov do Apache Spark popisuje publikácia [17]. Všetky podrobnejšie informácie v tejto časti pochádzajú zo zdrojových kódov Apache Hadoop vo verzii 2.6.5, ktoré sú využité na načítanie súborov do systému Spark v tejto práci.

### Formát `pcap`

Funkcionalita načítavania dát z `pcap` súborov bola prevzatá úpravou a začlenením demo-aplikácie do projektu. Spark natívne nepodporuje načítanie `pcap` formátu, preto je nutné vytvoriť vlastnú implementáciu. Demo-aplikácia na to využíva tieto triedy a funkcie:

- `hadoopFile` – je funkcia pre načítavanie dát, ktorá je súčasťou Hadoop API Sparku. Na načítavanie vstupov využíva triedy odvodené od triedy `InputFormat`. Kedže Spark nepodporuje formát `pcap`, je potrebné takúto triedu implementovať. V demo-aplikácii plní túto úlohu trieda `PcapInputFormat`.

- `PcapInputFormat` – trieda umožňujúca načítavať dáta z `pcap` súborov. Táto trieda dedí zo štandardnej triedy na načítanie vstupov `FileInputFormat`, ktorá je odvodená z triedy `InputFormat`. Trieda `InputFormat` je práve tá, ktorú používa Spark na delenie súborov do logických blokov (pozri predchádzajúcu podkapitolu o delení súborov 5.2).

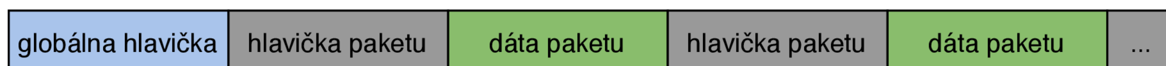
- `PcapRecordReader` – trieda implementujúca rozhranie `RecordReader`, ktorú aplikácia používa na načítavanie údajov z logických blokov (v tomto prípade dát z `pcap` súborov).

Takýmto spôsobom je možné implementovať aj načítanie akéhokoľvek iného vstupného formátu, ktorý Spark natívne nepodporuje.

Štruktúra súborov `pcap` je naznačená na obrázku 5.2. Ako je vidno, každý súbor má práve jednu globálnu hlavičku. Na obrázku 5.3a je možné vidieť všetky údaje, ktoré táto globálna hlavička obsahuje. Z hľadiska implementácie sú dôležité tieto:

- magické číslo – slúži na detekciu formátu `pcap` a detekciu poradia bajtov. V podporovanej verzii formátu má hodnotu buď 0xa1b2c3d4 alebo 0xd4c3b2a1 pre opačné poradie bajtov. Všetky ďalšie polia všetkých hlavičiek v celom súbore je potrebné čítať v poradí bajtov zistenom z magického čísla.

- typ hlavičky na linkovej vrstve – aplikácia podporuje tieto typy protokolov linkovej vrstvy[2]:
  - LINKTYPE_NULL – BSD loopback zapuzdrenie,
  - LINKTYPE_ETHERNET – ethernet,
  - LINKTYPE_RAW – paket začína ipv4 alebo ipv6 hlavičkou,
  - LINKTYPE_LOOP – OpenBSD loopback zapuzdrenie,
  - LINKTYPE_LINUX_SLL – linuxové zapuzdrenie.




Obr. 5.2: Štruktúra `pcap` súboru (prevzaté z [5])

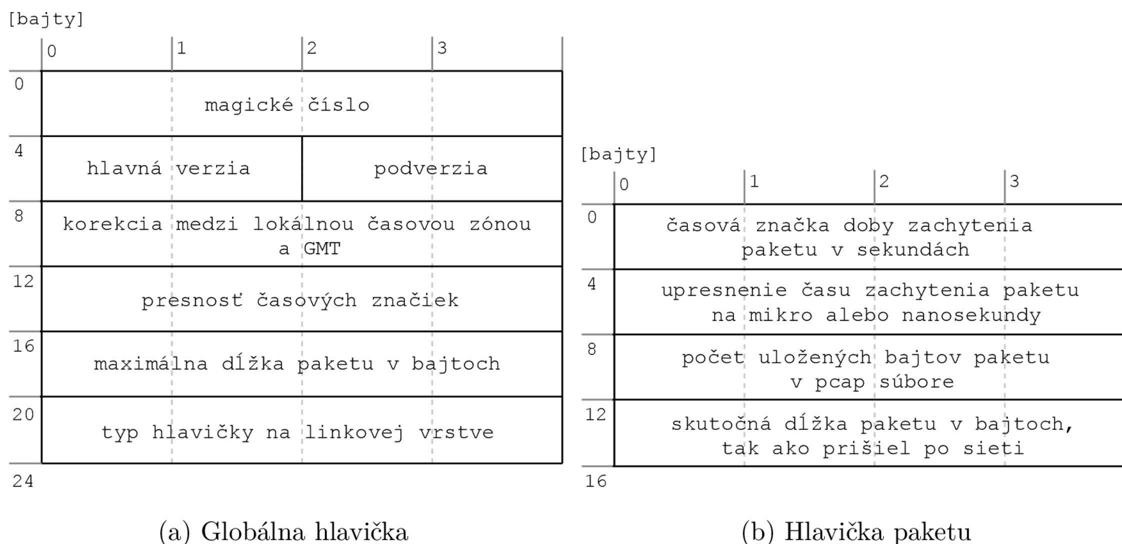
Existencia práve jednej globálnej hlavičky v každom `pcap` súbore je problémom pri distribuovanom spracovaní. Kvôli globálnej hlavičke musí byť spracovávaný jeden `pcap` súbor

[2]Celý zoznam protokolov linkovej vrstvy aj s podrobnejším popisom sa nachádza na http://www.tcpdump.org/linktypes.html.

vždy len jedným uzolom v klastri, v opačnom prípade by ostatné uzly nemali prístup k informáciám z tejto hlavičky. Aby sa týmto nedegradoval výkon systému, musia byť `pcap` súbory rozdelené na menšie s vlastnými globálnymi hlavičkami. Takto je možné dosiahnuť paralelné spracovanie na úrovni súborov. Rozdelenie `pcap` súborov na menšie je možné vykonať napríklad programom tcpdump alebo editcap. Veľkosti, na ktoré je súbory vhodné rozdeliť sú diskutované v kapitole 6. Implementačne je zakázané delenie `pcap` súborov na logické bloky, každý súbor je vždy len jedným logickým blokom. Po prečítaní globálnej hlavičky číta trieda `PcapRecordReader` postupne hlavičky paketov (obr. 5.3b) a ich dáta. Pre každú hlavičku s príslušnými dátami vytvorí inštanciu triedy `RawFrame`, ktorá reprezentuje jeden paket. Údaje, ktoré predáva týmto inštanciám sú:

- časová značka zachytenia paketu (z hlavičky paketu),
- dĺžka paketu (z hlavičky paketu),
- typ hlavičky na linkovej vrstve (z globálnej hlavičky),
- poradové číslo paketu – je možné počítať vďaka nedeliteľnosti `pcap` súborov. Keby sa súbory mohli v klastri deliť, každý uzol by poradové číslo počítal zvlášť pre svoju časť dát. Pri paralelnom spracovaní by tak nebolo možné pakety rozumne číslovať.
- Obsah paketu – dáta paketu v binárnej forme.

Aplikácia nepodporuje formáty ako `pcapng` alebo `pcap` s rozlíšením v nanosekundách. Podporované sú len najbežnejšie používané formáty s vyššie spomenutým magickým číslom rovným 0xa1b2c3d4 alebo 0xd4c3b2a1. Celá popísaná funkcionalita je implementovaná v module `ndx-spark-pcap`. Informácie týkajúce sa `pcap` formátu prezentované v tejto podkapitole vychádzajú z údajov uvedených v dokumentácii [5].




(a) Globálna hlavička

(b) Hlavička paketu

Obr. 5.3: Hlavičky `pcap` súborov

### Formát `JSON`

Formát `JSON` je v Sparku priamo podporovaný prostredníctvom rozhrania Spark SQL. V tomto prípade jeho využitie nie je možné z dôvodov už spomínaných duplicitných kľúčov

a faktu, že údaje o každom pakete nie sú len na jednom riadku, ale na dvoch. Dáta z týchto JSON súborov sú preto načítavané na nižšej úrovni, podobne ako tomu bolo pri pcap súboroch, pomocou Hadoop API. Na rozdiel od pcap súborov nie je v tomto prípade potrebné implementovať vlastné triedy na načítavanie dát, ale je možné použiť priamo štandardnú triedu Sparku na načítavanie textu TextInputFormat. Týmto spôsobom sa načíta každý vstupný JSON súbor do RDD kolekcie, v ktorej je každý riadok reprezentovaný textovou položkou. Túto RDD kolekciu je možné deliť na ľubovoľný počet logických blokov a spracovávať ju naprieč celým klastrom. To je možné vďaka tomu, že JSON súbory neobsahujú žiadne globálne údaje ako obsahovali súbory pcap. Nakoniec sa odfiltrujú všetky nepotrebné riadky (tj. každý druhý) a ponechajú sa len riadky s dátami paketov. Táto filtrácia, ako aj všetky ostatné operácie nad JSON dátami prebiehajú v klastri paralelne. Ako je vidieť aj na obrázku 5.1, výstupom tejto fázy je RDD kolekcia reťazcov, z ktorých každý reprezentuje jeden paket.

## 5.3 Spracovanie dát

Implementácia spracovania dát sa nachádza v module ndx-spark-model. Spracovaním dát sa v tejto práci rozumie transformácia načítaných vstupov do jednotnej internej reprezentácie, ktorú tvoria triedy naznačené na obrázku 5.1:

- Packet – je abstraktná trieda, ktorá združuje spoločnú funkcionalitu potrebnú pre prácu s paketmi, ako je napríklad generovanie flow reťazcov, prístup k spracovaným dátam atď. Pomocou tejto triedy je možná jednotná práca so spracovanými paketmi bez ohľadu na formát zdroja dát. Drobné rozdiely v spracovaných dátach napriek tomu existujú a sú popísané ďalej v tejto podkapitole. Táto trieda je potomkom štandardnej triedy jazyka Java HashMap. Všetky potrebné dáta jej potomkov sú tak dostupné pomocou rozhrania triedy HashMap, ktoré môže byť v prípade potreby aj reimplementované. Všetky kľúče nezbytné k prístupu k spracovaným dátam sú dostupné v podobe statických hodnôt práve v triede Packet.

- JsonPacket – je potomkom triedy Packet, ktorý sa stará o spracovanie dát zo vstupov vo formáte JSON. Inštancie tejto triedy reprezentujú jednotlivé pakety a spracovávajú výstupy z fázy načítania dát, ktorými sú reťazce vo formáte JSON. Problémom pri implementácii sú duplicitné JSON kľúče prítomné v spracovávaných vstupoch. Množstvo implementácií knižníc pre spracovanie formátu JSON duplicitné kľúče nepodporuje a dáta buď vôbec nespracuje alebo hodnoty viacnásobných kľúčov zahodí a použije iba prvú z nich. Obidva typy takéhoto správania sú nežiaduce, pretože hodnoty prislúchajúce k duplicitným kľúčom sú z hľadiska informácií o pakete dôležité a nie je možné ich zahodiť. Ďalším problémom sú objekty, v ktorých záleží na poradí kľúčov. Riešením je ukladanie hodnôt duplicitných kľúčov do polí, v ktorých ostáva ich poradie zachované. Toto riešenie postačuje potrebám tejto práce, vyskytnúť sa však môže problém s väzbami medzi hodnotami asociovanými s duplicitnými a bežnými kľúčmi. Takéto väzby by sa v JSON formáte nemali vyskytovať, v tomto prípade sa bohužiaľ vyskytnúť môžu. Nedá sa predvídať, ako sa bude tento formát vyvíjať v budúcnosti. Žiaduce by bolo, keby zodpovedal špecifikácii JSON, čo by výrazne uľahčilo jeho spracovanie. Knižnica pre spracovanie formátu JSON je od triedy JsonPacket oddelená rozhraním a implementáciou tohto rozhrania je možné ju v prípade potreby nahradiť inou. Požadované hodnoty dát sú získavané zo vstupu volaním funkcií tohto rozhrania.

- `PcapPacket` – je takisto potomkom triedy `Packet` a stará sa o spracovanie vstupov vo formáte `pcap`. Na rozdiel od textových `JSON` vstupov sú v tejto triede primárne spracovávané binárne dáta. V prevzatej demo-aplikácii plnila táto trieda úlohu hlavnej triedy pre spracovanie dát. Do výsledného projektu boli od nej prebraté niektoré princípy, ako napríklad rozhranie `HashMap` pre prístup k spracovaným dátam.

S využitím triedy `Packet` a polymorfizmu je jednoduché doimplementovať aj podporu spracovania iných vstupných formátov. S ohľadom na požadovanú funkčnosť sú v oboch prípadoch spracovávané tieto vrstvy ISO/OSI modelu (ku každej vrstve sú uvedené podporované protokoly):

- linková vrstva – Ethernet,
- sieťová vrstva – IPv4, IPv6,
- transportná vrstva – TCP, UDP (User Datagram Protocol), ICMP (Internet Control Message Protocol),
- aplikačná vrstva – DNS, SMTP, POP3 (Post Office Protocol), IMAP (Internet Message Access Protocol), HTTP.

Ďalej sú čiastočne podporované protokoly SSL/TLS, ktoré sa nedajú jednoznačne zaradiť v rámci ISO/OSI modelu. Tieto šifrovacie protokoly sú podporované len vo formáte `JSON`. Aplikácia spracováva paket postupne vrstvu po vrstve. V prípade, že na niektorej vrstve narazí na nepodporovaný protokol skončí so spracovávaním, ale dáta z nižších vrstiev od podporovaných protokolov budú dostupné. Inými slovami, aplikácia sa snaží z paketov extrahovať maximum dát, ktorým rozumie. Na aplikačnej vrstve je potrebné pri `pcap` vstupoch detekovať protokol. V prípade `JSON` formátu je aplikačný protokol už detekovaný nástrojom, ktorý ho vytvoril (Wireshark/TShark). Z hľadiska snahy o jednotnosť dát z oboch zdrojov je detekcia aplikačného protokolu vykonávaná tak, aby čo najviac kopírovala detekciu vykonávanú programami Wireshark/TShark. Tieto programy používajú tzv. „dissection“ funkcie, ktoré slúžia na spracovanie konkrétnych aplikačných protokolov. Vhodná funkcia sa vyberá primárne na základe portov transportnej vrstvy. Ďalšími možnosťami sú napríklad označenie protokolu používateľom alebo metóda pokus-omyl. V aplikácii sú taktiež primárne na túto detekciu použité čísla portov. Metóda pokus-omyl je aplikovaná napríklad pri DNS alebo HTTP protokole, kde sú pevne definované časti protokolu a podľa nich je možné určiť, či sa jedná o daný protokol. Poslednou možnosťou je ponechanie určenia protokolu na používateľovi. To je implementované pomocou parametra odkazu na funkciu, ktorá sa má použiť pri spracovaní protokolu aplikačnej vrstvy. Takúto funkciu si môže v prípade potreby implementovať aj sám používateľ (používateľom sa rozumie programátor). Štandardne je použitá funkcia, ktorá sa pokúsi určiť tento protokol sama predošlými dvoma uvedenými spôsobmi. Výhodou tohto prístupu je, že v prípade keď používateľ potrebuje spracovať nejaký špecifický nepodporovaný protokol môže dodať implementáciu jeho spracovania úplne bez zásahu do existujúcej implementácie.

### Rozdiely v spracovaných dátach podľa vstupného formátu

Táto podkapitola v krátkosti pojednáva o rozdieloch v spracovaných dátach v závislosti na vstupnom formáte. Napriek snahe o jednotnú podobu dát po fáze spracovania, nie je toto možné dosiahnuť vo všetkých prípadoch. Nemožnosť dosiahnuť úplnú zhodu vyplýva z rozdielnosti informácií obsiahnutých v zodpovedajúcich si paketoch v odlišných formátoch.

Informácie obsiahnuté v súboroch vo formáte `JSON` plne závisia od toho, ako ich spracoval program Wireshark alebo TShark, ktorým boli vytvorené. Spomínanými odlišnosťami sú:

- TCP sekvenčné čísla – v `pcap` súboroch sa nachádzajú skutočné sekvenčné čísla, zatiaľ čo v `JSON` dátach sú dostupné len relatívne sekvenčné čísla tj. začínajúce od 0 pre každé TCP spojenie.
- Znovu zostavené pakety (angl. reassembled) – v `pcap` súboroch sú prítomné vždy dáta z paketov, tak ako prišli po sieti. V `JSON` formáte tomu tak nie je a nachádzajú sa tu pakety, ktorých obsah bol poskladaný programom Wireshark. Príklad uvediem na dvojici HTTP paketov, nech sú označené poradovými číslami 1 a 2. Po sieti prídu v takejto podobe:
  - paket 1 – obsahuje HTTP hlavičku a časť dát,
  - paket 2 – obsahuje zvyšnú časť dát, ktorá sa nezmestila do paketu č. 1.

  V `pcap` súboroch sú tieto informácie obsiahnuté presne v tejto podobe. Pri spracovaní je paket 1 označený ako HTTP a paket 2 ako TCP. Pri paralelnom spracovaní jednotlivých paketov nie je možné spracovať aplikačnú vrstvu paketu 2, lebo ten môže byť umiestnený v hociktorom uzle v klastri a nie je tak známy kontext (TCP tok), v ktorom paket prišiel. V prípade `JSON` formátu budú dáta vyzerať takto:
  - paket 1 – označený ako bežný TCP paket bez dostupných dát vo forme kľúč-hodnota,
  - paket 2 – znovu zostavený paket obsahujúci HTTP hlavičku a všetky príslušné HTTP dáta z paketov 1 aj 2.

  V tomto prípade bude pri spracovaní paket 1 označený ako bežný TCP a paket 2 ako HTTP s príslušnými dátami. Jednotlivé pakety sa síce líšia, ale vo výsledku sú všetky potrebné dáta dostupné (aj keď v iných paketoch). Po agregácii bude výsledok v obidvoch prípadoch rovnaký, pretože nie je podstatné v ktorom pakete sa dáta nachádzajú, ale len ich samotná prítomnosť po spracovaní. Tento rozdiel je potrebné brať do úvahy hlavne pri extrakcii dát z TCP tokov, ktorá sa bude líšiť v závislosti na vstupnom formáte.
- Tvar IPv6 adresy – v `JSON` formáte sa používa skrátený tvar, zatiaľ čo vo formáte `pcap` je to plný tvar. Tento rozdiel je vo väčšine prípadov nepodstatný, pretože oba zápisy sú ekvivalentné.
- Hodnota RDATA pri DNS paketoch – v JSON formáte pridáva Wireshark/TShark do reťazcov RDATA ďalšie svoje reťazce. Napríklad pri zázname typu A je pridaný reťazec „address“ a pod. V dátach pochádzajúcich z `pcap` súborov sú hodnoty položky RDATA v pôvodnej podobe, v akej boli odoslané DNS servermi.

Odlišností medzi formátmi je samozrejme viac, uvedené sú tie, ktoré majú dopad na výsledky prezentované v tejto práci. Tieto odlišnosti je potrebné brať do úvahy pri spracovávaní dát. Výsledky sa vzhľadom na vstupný formát môžu líšiť hlavne pri analýze individuálnych paketov.

## 5.4 Analýza a vizualizácia dát

Analýza a vizualizácia dát sú dve najvyššie vrstvy aplikácie. Pre analýzu dát je vyhradený modul `ndx-spark-tshark`, ktorý slúži zároveň ako aplikačné rozhranie. Rozhranie je k dispozícii v dvoch jazykoch, Java a Scala. Ako je uvedené aj v úvode tejto kapitoly, pôvodné aplikačné rozhranie bolo napísané v jazyku Java. Z dôvodov náročnejšej, menej efektívnej práce s triedami jazyka Java v prostredí Sprak shell a horšej podpory funkcionálneho programovania, bolo vytvorené rozhranie v jazyku Scala. Pôvodné Java rozhranie bolo rozdelené na dve časti:

- hlavné rozhranie – toto rozhranie poskytuje základné funkcie potrebné pre prácu s aplikáciou v prostredí Spark shell. Týmito funkciami sú načítanie dát, spracovanie dát a poskytnutie výslednej RDD kolekcie paketov. S touto kolekciou je potom možné ďalej pracovať.

- Testovacie rozhranie – slúži na rýchle testovacie výpisy rôznych typov spracovaných dát. Pre svoju prácu využíva hlavné rozhranie. Zároveň poskytuje návod ako s dátami poskytnutými hlavným rozhraním ďalej pracovať v jazyku Java.

Rozhranie v jazyku Scala poskytuje všetky operácie potrebné pre získanie a vizualizáciu dát uvedených v analýze požiadaviek v kapitole 3. Z tohto rozhrania je volaná funkcionalita načítania a spracovania vstupných dát, implementovaná v jazyku Java, ktorá bola popísaná v tejto kapitole. Výstupom fázy spracovania dát je tu RDD kolekcia reprezentujúca jednotlivé pakety. Nad touto kolekciou sú potom vykonávané paralelné transformácie potrebné pre získanie požadovaných dát. Princíp transformácií je podrobne vysvetlený v podkapitole 2.3.

V poslednej fáze prebieha vizualizácia získaných dát. V rámci tejto práce sú dáta vizualizované pomocou nástroja Apache Zeppelin (pozri podkapitolu 2.4). V tomto nástroji bol vytvorený záznam, v ktorom sa získavajú dáta pomocou interpretu Spark shell. Získané dáta musia byť vo forme tabuliek kolekcie DataFrame. Podpora tejto funkcionality je implementovaná v rámci aplikačného rozhrania v jazyku Scala. K týmto dátam sa potom pristupuje pomocou rozhrania Spark SQL, formou SQL požiadaviek zapísaných v zázname. Vytvorený záznam slúži zároveň ako návod na vytváranie obdobných záznamov. Vhodný typ zobrazenia sa môže pre rôzne vstupné sady dát líšiť. Záznam je preto vhodné mierne upraviť vzhľadom na povahu dát vstupnej sady. Úpravy môžu spočívať buď len v zmene typu zobrazenia (napr. tabuľka/grafy) alebo úprave SQL požiadavky (napr. zmena rozlíšenia časovej osi na dni, hodiny, minúty a pod.).

Ďalej v tejto podkapitole sú jednotlivo popísané všetky implementované úlohy, ktoré boli v požiadavkách rozdelené do týchto celkov:

- analýza obsahu paketov – v tejto časti sa pakety spracovávajú individuálne bez ohľadu na kontext. Rozhodujúce sú informácie, ktoré sa podarilo extrahovať behom fázy spracovania.

- Analýza sieťových tokov – ku každému paketu sa generuje tzv. reťazec toku (angl. flow string). Reťazec má takýto tvar:

  ```
  [<protocol>@<src_adr>#<src_port>→ <dst_adr>#<dst_port>],
  ```

  kde <protocol> je protokol transportnej vrstvy, <src_adr> je zdrojová IP adresa, <src_port> je zdrojový port, <dst_adr> je cieľová IP adresa a <dst_port> je cieľový

port. Reťazec slúži ako kľúč pri širokých transformáciách ako sú napríklad `groupByKey` alebo `reduceByKey`. Pomocou týchto transformácií sa pakety združujú alebo agregujú v skupinách s rovnakým kľúčom, tj. tokoch.

### Získanie URL z HTTP hlavičky

Protokol HTTP (Hypertext Transfer Protocol) je postavený na princípe požiadavka-odpoveď. Štruktúru protokolu uvádzajú autori v publikácii [4], jej stručný popis je nasledujúci:

```
<first_line>
<headers>
<entity_body>
```

Uvedené časti protokolu znamenajú:

- `<headers>` – sú hlavičky protokolu HTTP. Paket môže obsahovať nula alebo viac hlavičiek. Typy hlavičiek sú definované vo viacerých RFC dokumentoch, ich prehľad sa nachádza na stránkach organizácie IANA (Internet Assigned Numbers Authority) [6]. Každá hlavička má takýto tvar:

  ```
  <name>: <value><crlf>,
  ```

  kde `name` je názov hlavičky, `value` je príslušná hodnota a `crlf` ukončovacia sekvencia oddeľujúca hlavičky. Sekcia s hlavičkami je oddelená dvomi za sebou idúcimi sekvenciami `crlf`.

- `<entity_body>` – sú HTTP dáta. Paket túto položku nemusí obsahovať.

- `<first_line>` – reprezentuje prvý riadok v HTTP protokole. Tento riadok má pevne stanovený formát pre požiadavky

  ```
  <method> <url> <version>
  ```

  a pre odpovede

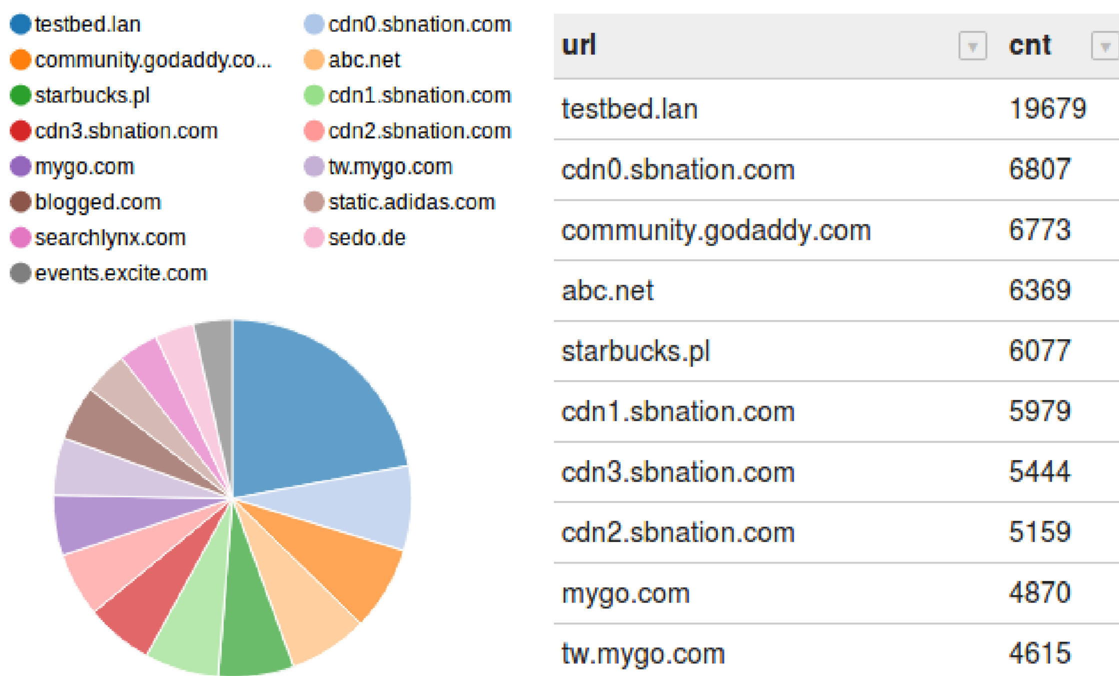
  ```
  <version> <status> <reason-phrase>.
  ```

  kde `<method>` je jedna z HTTP metód (ako sú GET, POST atď.), `<url>` je lokátor špecifikujúci zdroj, na ktorý je požiadavka adresovaná, `<version>` je verzia HTTP protokolu, `<status>` sú tri číslice popisujúce stav akým skončilo spracovanie príslušnej požiadavky a `<reason-phrase>` je textovo popísaný stav požiadavky. V obidvoch prípadoch sú tieto riadky oddelené, takisto ako HTTP hlavičky, sekvenciou `crlf`.

Behom spracovania paketov z `pcap` súborov sa zisťuje či je paket valídnym HTTP paketom metódou pokus-omyl. Vezme sa neznámy paket a otestovaním validity prvého riadku sa zistí, či sa jedná o HTTP paket.

V tejto úlohe sa berie do úvahy hlavička s názvom Host. RFC 7230 [2] uvádza, že hlavička poskytuje informácie o názve hostiteľa a nepovinne o porte. Podpora hlavičky je implementovaná od verzie protokolu HTTP 1.1. Na jednom fyzickom serveri (jednej IP adrese) môžu byť v prevádzke viaceré virtuálne servery obsluhujúce rôzne domény. Informácia v Host hlavičke dáva takémuto serveru informáciu, pre ktorý virtuálny server je požiadavka určená. Rôzne klientské stanice udávajú názov hostiteľa v rôznych formách ako napríklad `http://doména`, `www.doména` a pod. V implementácii aplikácie je z tejto hlavičky extrahovaný reťazec obsahujúci len názov domény. Takto sa aj rôzne zapísané názvy hostiteľov

môžu správne agregovať podľa tohto reťazca ako kľúča. Výstupom sú potom doménové mená a počet ich zastúpení vo vstupných dátach. Na obrázku 5.4 je ukázané zobrazenie takého výstupu z testovacích dát v nástroji Apache Zeppelin. Z tabuľky 5.4b a grafu 5.4a je zrejmé, že v danej testovacej sade bolo najfrekventovanejším doménovým menom v Host hlavičkách HTTP paketov `testbed.lan`.




| url | cnt |
|---|---|
| testbed.lan | 19679 |
| cdn0.sbnation.com | 6807 |
| community.godaddy.com | 6773 |
| abc.net | 6369 |
| starbucks.pl | 6077 |
| cdn1.sbnation.com | 5979 |
| cdn3.sbnation.com | 5444 |
| cdn2.sbnation.com | 5159 |
| mygo.com | 4870 |
| tw.mygo.com | 4615 |

(a) Koláčový graf zastúpenia domén

(b) Tabuľka zastúpenia domén

Obr. 5.4: Zobrazenie informácií z HTTP hlavičky Host v testovacej sade dát (z Apache Zeppelin)

### Extrakcia TCP tokov a HTTP/SMTP súborov

Extrakcia TCP tokov je špeciálny prípad extrakcie tokov. Extrakcia tokov bola popísaná v rámci analýzy sieťových tokov na začiatku tejto podkapitoly. Jednotlivé toky sú separované pomocou reťazca toku v tvare:

```
[TCP@<src_adr>#<src_port>→ <dst_adr>#<dst_port>].
```

Pakety v takto získaných TCP tokoch sa nakoniec zoradia vzostupne podľa časových značiek.

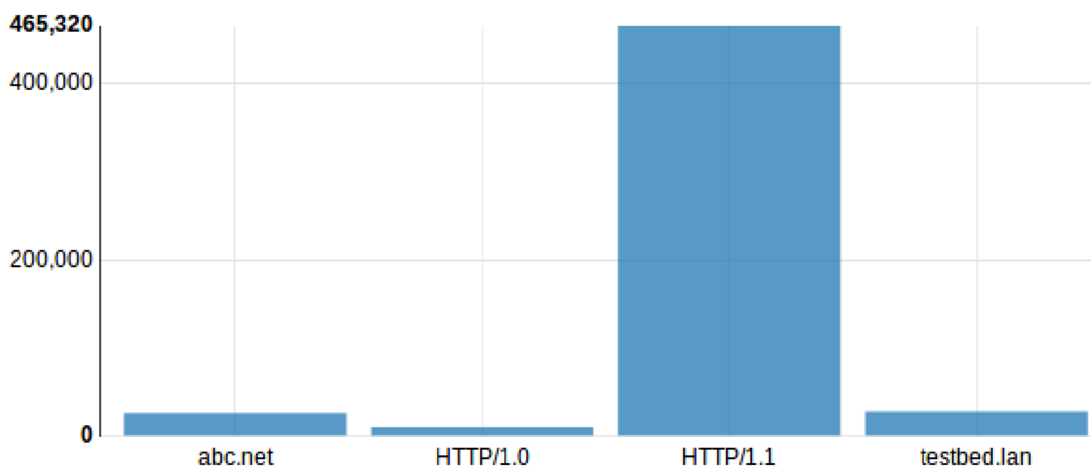
HTTP a SMTP súbory je možné extrahovať v postprocesingu. Implementácia postprocesingu nie je predmetom tejto práce, ani súčasťou aplikácie. Vzhľadom na to, že aplikácia musí poskytovať všetky dáta potrebné v postprocesingu, uvádzam návod ako ho implementovať. Prvým krokom postprocesingu je extrakcia jednotlivých TCP spojení v rámci tokov. Na získanie TCP spojení je potrebné spracovávať každý tok sekvenčne a zoradiť pakety podľa sekvenčných čísiel. Hlavne pri dátach z `JSON` vstupov hrozia kolízie sekvenčných čísiel v rámci jedného toku, pretože sú použité relatívne čísla začínajúce od 0 pre každé TCP spojenie. Tým, že sú pakety v tokoch usporiadané podľa času je pri takej kolízii najpravdepodobnejším nasledujúcim paketom TCP spojenia najbližší paket v kolekcii s požadovaným

sekvenčným číslom. Ďalej je potrebné odstrániť duplicitne poslané pakety. Z takto zoradených paketov TCP spojení je možné extrahovať posielané HTTP a SMTP súbory. Dáta pripravené na takéto spracovanie poskytuje aplikácia v podobe hexadecimálnych reťazcov reprezentujúcich dáta aplikačnej vrstvy. Potom stačí tieto reťazce pospájať do celkov podľa poradia paketov v TCP spojení a pomocou HTTP/SMTP analyzátorov jednoducho získať posielané súbory.

### Vyhľadávanie kľúčových slov

Vyhľadávanie kľúčových slov je aplikáciou podporované zadaním zoznamu hľadaných slov. Slová sa vyhľadávajú v rámci dát aplikačnej vrstvy v paketoch s transportným TCP protokolom. Formát `JSON` neobsahuje pre pakety s UDP protokolom na transportnej vrstve informáciu o dátach aplikačnej vrstvy v nespracovanej podobe. Kvôli tejto chýbajúcej informácii nie je možné implementovať vyhľadávanie kľúčových slov s rovnakým výsledkom bez ohľadu na formát vstupných dát. Aby bol obsiahnutý čo najväčší objem aplikačných protokolov je potrebné vyhľadávať v nespracovaných dátach, v opačnom prípade by si vyhľadávanie vyžadovalo implementáciu analyzátoru pre každý aplikačný protokol zvlášť.

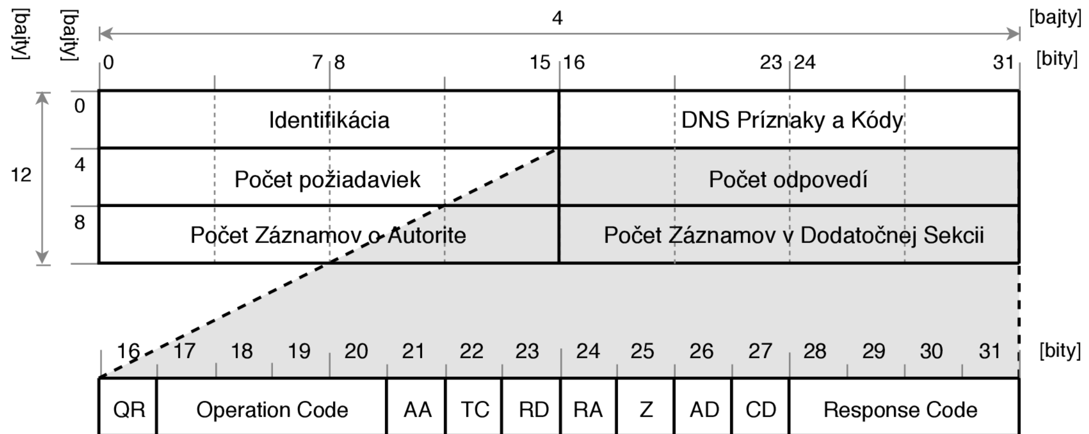
Požadované vyhľadávané slová by mali byť zadávané v takej podobe, ako a vyskytujú v nespracovaných paketoch. Tento prístup umožňuje vyhľadávanie nie len bežných textových reťazcov, ale v prípade potreby aj binárnych sekvencií zadaných v textovej podobe. Príklad vyhľadania slov je uvedený na obrázku 5.5. Vyhľadávané boli dve doménové mená, ktoré sú zastúpené v textovom HTTP protokole v rámci testovacej sady dát. Vhodné doménové mená na demonštráciu vyhľadávania kľúčových slov boli vybrané na základe analýzy hlavičky Host HTTP protokolu (pozri obrázok 5.4). Ďalšími vyhľadávanými kľúčovými slovami v tejto demonštrácii sú verzie protokolu HTTP, ktoré bývajú v HTTP paketoch uvedené v textovej forme. Zo stĺpcového grafu na obrázku je na prvý pohľad vidieť, že v testovacej sade je protokol HTTP vo verzii 1.1 výrazne viac zastúpený. Voľbou vhodných kľúčových slov sa týmto spôsobom dajú vykonávať rôzne analýzy vstupných dát.




Obr. 5.5: Zobrazenie počtu nájdených kľúčových slov (z Apache Zeppelin)

### Extrakcia informácií z DNS komunikácie

Hlavička DNS protokolu má pevne definovanú štruktúru ako je vidieť na obrázku 5.6. Vďaka tejto pevnej štruktúre je možné spoľahlivo identifikovať DNS pakety z `pcap` súborov behom fázy spracovania, podobne ako boli identifikované HTTP pakety.




Obr. 5.6: Hlavička DNS protokolu (prevzaté z [1])

Z informácií uvedených v hlavičkách DNS paketov sú dôležité tieto:

- identifikácia – potrebná na párovanie DNS požiadaviek s odpoveďami,
- QR príznak – potrebný na rozpoznávanie požiadaviek a odpovedí.

Ďalšie informácie potrebné na implementáciu úloh vyplývajúcich z analýzy v kapitole 3 sa nachádzajú v tele paketu. Týmito informáciami sú:

- NAME – meno uzla, ktorému DNS záznam patrí,
- TYPE – 16 bitová hodnota určujúca typ záznamu (A, AAAA, PTR atď.),
- CLASS – 16 bitová hodnota určujúca triedu záznamu.
- RDATA – textový reťazec premenlivej dĺžky nachádzajúci sa v DNS odpovediach. Jeho formát závisí od typu a triedy záznamu.

Položky NAME, TYPE a CLASS sa nachádzajú ako v DNS požiadavkách, tak aj v odpovediach. V implementácii aplikácie sú tieto hodnoty reprezentované v podobe jedného reťazca vo formáte CSV[3]. V prípade požiadaviek je namiesto hodnoty RDATA v reťazci prázdne miesto. Práve položka RDATA sa líši v závislosti od formátu vstupných dát, pretože Wireshark/TShark pridáva do formátu `JSON` svoje dodatočné reťazce (pozri rozdiely formátov v kapitole 5.3). Všetky uvedené informácie o DNS paketoch pochádzajú z RFC 1035 [11]. Podrobnejšie je DNS protokol popísaný v mojej bakalárskej práci [1].

Na obrázku 5.7 je demonštrované zobrazenie DNS dát z testovacej sady. V ľavej časti na obrázku 5.7a je uvedená tabuľka zastúpenia najfrekventovanejších domén v DNS požiadavkách. Najviac zastúpená bola doména reverzného mapovania DNS z požiadaviek typu

[3]CSV (Comma-separated values) je jednoduchý formát, v ktorom sú jednotlivé položky od seba oddelené čiarkami, viac napríklad na https://tools.ietf.org/html/rfc4180

PTR, ako prezrádza doména najvyššieho rádu *in-addr.arpa*. V pravej časti na obrázku 5.7b je vidieť zastúpenie paketov podľa typu (zahrnuté sú požiadavky aj odpovede). Najviac zastúpené sú tu záznamy typu A, ktoré sa týkajú mapovania IPv4 adries na doménové mená. Tieto záznamy mali výraznú prevahu vo väčšine testovacích sád.

### Extrakcia informácií z SSL/TLS

Extrakcia dát z SSL/TLS protokolov je implementovaná len v `JSON` formáte. Implementácia pre `pcap` vstupy je uvažovaná v rámci možného rozšírenia. Štruktúra protokolov sa v rôznych verziách protokolu mierne odlišuje, preto by bolo potrebné implementovať `pcap` analyzátory v závislosti od verzie. Údaje, ktoré aplikácia získava z protokolu sú:

- verzia protokolu – SSL3.0, TLS1.0, TLS1.1 alebo TLS1.2,
- typ protokolu SSL/TLS záznamu,
- zvolené šifrovanie serverom,
- interval platnosti X.509 certifikátu.

V `JSON` formáte sú jednotlivé správy protokolu v rámci jedného paketu reprezentované buď pomocou rôznych objektov alebo v rámci jedného objektu, v ktorom záleží na poradí záznamov. Všetky informácie sú extrahované nezávisle na kontexte a nie sú spolu nijako previazané. Môžu byť použité napríklad na tieto analýzy:

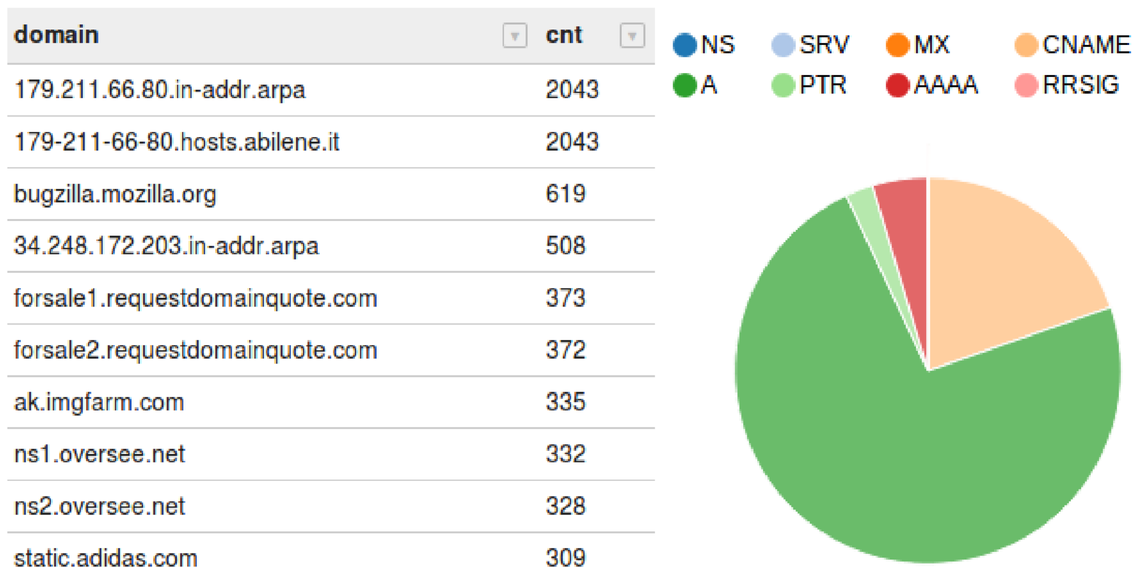
- najviac zastúpená verzia protokolu,
- najčastejší spôsob šifrovania komunikácie,
- priemerný interval platnosti certifikátov, atď.

V zázname notebooku je ako demonštrácia tejto funkcionality implementované zistenie najčastejšieho spôsobu šifrovania voleného serverom.

### Štatistiky dát sieťových tokov

V tomto type úloh sa pakety agregujú podľa kľúča toku. Po agregácii vznikne RDD kolekcia obsahujúca tieto dáta pre každý tok:

- časové značky – časy prvého resp. posledného paketu v toku,
- adresy L3 rstvy – zdrojovú a cieľovú IPv4 alebo IPv6 adresu,
- L4 porty – zdrojový a cieľový port TCP alebo UDP protokolu,
- smer toku – smer z koncového bodu k serveru je označený ako „up“, naopak ako „down“,
- servis – číslo portu služby v zachytenom toku,
- počet paketov a bajtov prenesených v rámci toku,
- údaj o lokalite komunikácie – komunikácia v rámci lokálnej siete alebo mimo nej,



| domain | cnt |
|---|---|
| 179.211.66.80.in-addr.arpa | 2043 |
| 179-211-66-80.hosts.abilene.it | 2043 |
| bugzilla.mozilla.org | 619 |
| 34.248.172.203.in-addr.arpa | 508 |
| forsale1.requestdomainquote.com | 373 |
| forsale2.requestdomainquote.com | 372 |
| ak.imgfarm.com | 335 |
| ns1.oversee.net | 332 |
| ns2.oversee.net | 328 |
| static.adidas.com | 309 |



(a) Zastúpenie domén v DNS požiadavkách (b) Zastúpenie DNS paketov podľa typu

Obr. 5.7: Zobrazenie informácií z DNS paketov v testovacej sade dát (z Apache Zeppelin)

- údaj o e-mail protokole – slúži na zjednodušenie práce pri úlohách zaoberajúcich sa elektronickou poštou.

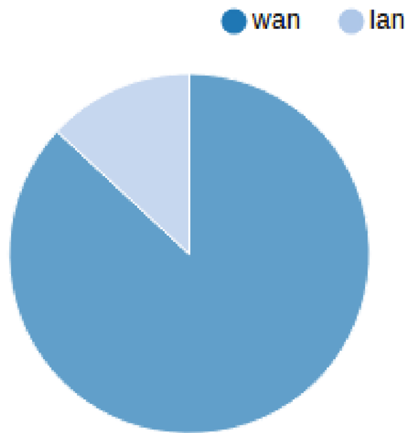
V prípade potreby je v implementácii možné jednoducho pridať ďalšie položky so štatistikami. Napríklad minimálnu a maximálnu veľkosť paketu v toku a pod. Nasleduje popis implementovaných úloh založených na štatistikách sieťových tokov:

- HTTP vs. HTTPS – porovnanie komunikácie týchto dvoch protokolov je založené na informácii o čísle portu služby v zachytenom toku. Za HTTP sú považované toky s číslom portu 80, za HTTPS zase s číslom portu 443. Množstvo komunikácie je porovnávané na základe počtu prenesených paketov. Na obrázku 5.9 je uvedený príklad takého porovnania na testovacej sade s prevahou HTTP komunikácie. Alternatívne sa môže množstvo komunikácie porovnávať aj na základe množstva prenesených dát alebo množstva tokov.

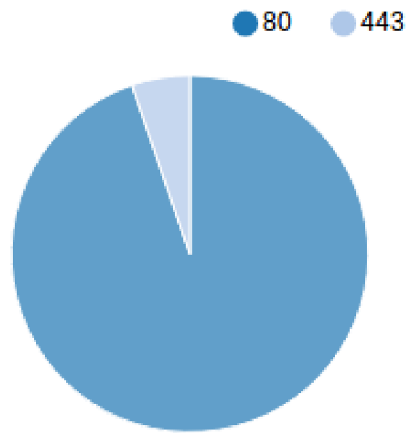
- LAN vs. WAN – za komunikáciu v lokálnej sieti (LAN) sú považované toky, ktoré majú privátnu zdrojovú aj cieľovú IP adresu[4]. Ostatné toky sú považované za komunikáciu mimo lokálnej siete (WAN). IPv4 adresy považované za lokálne sú:

  - nesmerovateľná IPv4 adresa `0.0.0.0`,
  - link-local unicast adresy v rozsahu `169.254.0.0/16`,

[4]Zoznam týchto adries sa nachádza na stránkach organizácie IANA a obsahuje odkazy na RFC, v ktorých sú definované. Pre IPv4 unicast `https://www.iana.org/assignments/iana-ipv4-special-registry/iana-ipv4-special-registry.xhtml`,
IPv4 multicast `https://www.iana.org/assignments/multicast-addresses/multicast-addresses.xhtml`,
IPv6 unicast `https://www.iana.org/assignments/iana-ipv6-special-registry/iana-ipv6-special-registry.xhtml` a
IPv6 multicast `https://www.iana.org/assignments/ipv6-multicast-addresses/ipv6-multicast-addresses.xhtml`.




Obr. 5.8: Porovnanie počtu tokov vyskytujúcich sa v lokálnych sieťach (lan) a mimo nich (wan) (z Apache Zeppelin)




Obr. 5.9: Porovnanie HTTP (80) a HTTPS (443) komunikácie na základe počtu paketov (z Apache Zeppelin)

- loopback adresy v rozsahu `127.0.0.0/8`,
- privátne adresy tried A, B a C v rozsahoch `10.0.0.0/8`, `172.16.0.0/12` a `192.168.0.0/16`
- a lokálne multicast adresy v rozsahoch `224.0.0.0/24`, `239.255.0.0/16` a `239.192.0.0/14`.

V prípade IPv6 adries sú za lokálne považované nasledujúce:

- adresa `::/0`,
- link-local unicast adresy v rozsahu `fe80::/10`,
- loopback adresa `::1`
- a lokálne multicast adresy v rozsahoch `ffx1::/16`, `ffx2::/16`, `ffx5::/16` a `ffx8::/16`.

Príklad klasifikácie LAN a WAN komunikácie je uvedený na obrázku 5.8. V testovacej sade sú vo výraznej prevahe toky smerujúce mimo lokálnu sieť. V minulosti mala v bežných podmienkach prevahu lokálna komunikácia, ale jej podiel sa postupne zmenšoval na úkor komunikácie mimo lokálnu sieť.

- Zariadenia prenášajúce najviac dát – zariadenia sú identifikované podľa zdrojovej adresy toku. Pre každú zdrojovú adresu sa následne agregujú počty prenesených dát v toku resp. počty tokov. Zostupným zoradením týchto agregovaných hodnôt sa určia adresy zariadení, ktoré preniesli najviac dát resp. vytvorili najviac tokov.
- Štruktúra sieťovej komunikácie súvisiacej s elektronickou poštou – vďaka údaju o e-mailovom protokole v štatistikách toku je pomocou Spark SQL jednoduché určiť štruktúru e-mailovej komunikácie. Rozpoznávané sú protokoly POP3, SMTP a IMAP, ktoré sú v rámci tokov určené číslami portov používaných týmito protokolmi. Štruktúru komunikácie je možné určiť podľa sumy prenesených dát, podľa počtu tokov alebo podľa počtu prenesených paketov. Vo vytvorenom zázname v notebooku je štruktúra komunikácie určená objemom prenesených dát.

- Najčastejšie meno hostiteľa – webové servery sú identifikované pomocou IP adries. Za webové servery sú považované zariadenia komunikujúce na portoch 80 (HTTP) a 443 (HTTPS). V zázname v notebooku sú radené podľa objemu komunikácie, ktorú vyprodukovali v jednotkách bajtov. Do úvahy pripadajú ešte klasifikácie na základe prijatých dát alebo prijatých aj odoslaných dát.
- Najčastejšia adresa koncového bodu – koncové body sú takisto identifikované pomocou IP adries. Pre každú cieľovú adresu zo štatistík tokov sú agregované hodnoty prenesených objemov dát v bajtoch. Po zostupnom zoradení cieľových IP adries podľa objemov dát sa určí N adries s najväčším objemom obdržaných dát. Inou metrikou by mohol byť objem odoslaných dát v bajtoch alebo suma objemov odoslaných a prijatých bajtov dát. Podobným spôsobom môžu byť vyhodnotené aj počty prenesených paketov alebo vytvorených tokov.
- Najčastejšie SMTP servery a SMTP klienti – v tomto prípade môžu byť použité takisto rôzne typy metrík, ako boli popísané v predchádzajúcom bode. V zázname v notebooku je použitá metrika na základe objemu odoslaných/obdržaných dát v bajtoch. SMTP toky sú rozpoznané podľa údaja o e-mailovom protokole v štatistikách o tokoch. Klienti a servery sú rozpoznaní podľa smeru toku, kde smer „up“ určuje klientskú stanicu a smer „down“ server.

V úlohách porovnávajúcich množstvo komunikácie som kvôli zachovaniu prehľadnosti implementoval vždy jednu, maximálne dve metriky. Ostatné spomínané metriky je možné implementovať pomocou jednoduchých úprav Spark SQL požiadaviek implementovaných v referenčnom zázname.

### Časová os

Zobrazenia na časovej osi vychádzajú zo štatistických dát. Agregujú sa hodnoty počtu tokov, počtu paketov a počtu prenesených bajtov. Tieto hodnoty sú agregované podľa časovej jednotky zadaného časového rozlíšenia. Rozlíšenie sa môže pohybovať od jednotiek mikrosekúnd až po roky. Voľba rozlíšenia je závislá na obsahu vstupných dát. Na obrázku 5.10 je príklad zobrazenia počtu paketov v čase. Časová os je v rozlíšení hodín a je vidieť, že testovacia sada paketov je z 12.6.2010 a časové značky paketov sú v rozmedzí 5 a 20 hodiny.

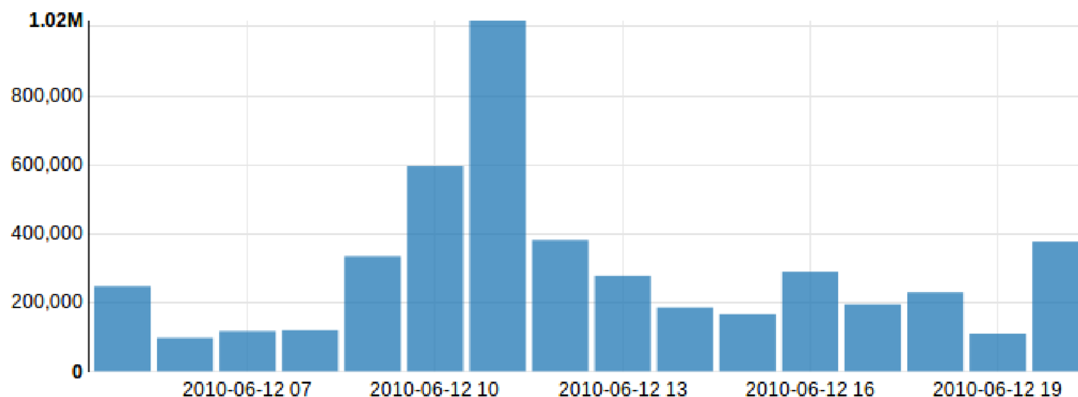
Časový priebehu HTTP a HTTPS komunikácie je v referenčnom zázname zobrazený podľa počtu prenesených paketov. Táto komunikácia je filtrovaná podľa príslušných čísiel portov jednotlivých tokov.

### Oneskorenie DNS servera

Výpočet oneskorení pre komunikáciu server klient je v distribuovanom prostredí pomerne náročnou úlohou. Definícia oneskorenia sa môže líšiť v závislosti na konkrétnom protokole. Napríklad čas nadviazania TCP spojenia, čas prijatia konkrétnej správy atď. Pri DNS komunikácii cez protokol UDP sa bude oneskorením rozumieť rozdiel časov odoslanej DNS požiadavky a prvej odpovede s parametrami zodpovedajúcimi požiadavke. Požiadavky a odpovede sú roztriedené podľa nasledujúceho kľúča:

`[<protocol>@<lo_adr>#<lo_port>`↔ `<hi_adr>#<hi_port>]<dns_id>`,

kde `<protocol>` je L4 protokol, `<lo_adr>` je adresa spodná adresa, `<lo_port>` je spodný port, `<hi_adr>` je vrchná adresa, `<hi_port>` je vrchný port a `<dns_id>` je identifikačné číslo




Obr. 5.10: Počet paketov v čase (z Apache Zeppelin)

DNS správy. IP adresy sa porovnávajú ako reťazce pričom spodná adresa je tá, ktorá je lexikograficky nižšia. Vrchný a spodný port sa určia podľa toho ku ktorej IP adrese patria.

Po roztriedení DNS paketov vzniknú kolekcie s rovnakým kľúčom. Do úvahy sa ďalej berú kolekcie, ktoré obsahujú práve jednu požiadavku a práve jednu odpoveď. Prípady kedy kolekcie obsahujú iné údaje môžu byť spôsobené:

- viacnásobným odoslaním odpovede,
- kolíziou rovnakého čísla portu a DNS identifikácie pri komunikácii rovnakých zariadení,
- ich kombináciou, atď.

V takýchto prípadoch nie je možné jednoznačne priradiť všetky požiadavky k odpovediam. Tieto údaje sa môžu zo štatistického hľadiska zanedbať. Klientské stanice generujú pri každej požiadavke nové číslo portu a novú DNS identifikáciu ako ochranu proti podvrhnutiu odpovede. V súčasnosti by takýmto spôsobom mala fungovať každá klientská stanica. V prípade potreby by bolo možné pridať do kľúča ďalšie údaje ako napríklad typ DNS správy. Párovanie na základe uvedeného kľúča však poskytuje dostatočné množstvo informácií, na základe ktorých sa vypočíta oneskorenie servera. Oneskorenie je počítané v jednotkách milisekúnd.

# Kapitola 6

# Experimenty

Cieľom distribuovaného spracovania dát je minimalizácia výpočetného času. V tejto kapitole bude diskutovaná dĺžka výpočetného času v referenčnom klastri vzhľadom na rôzne scenáre spracovania dát. Parametre referenčného klastra sú takéto:

- 5 *worker* uzlov,
- z prvých 4 *worker* uzlov má každý k dispozícii 46 GB operačnej pamäte, posledný *worker* uzol má k dispozícii 22 GB operačnej pamäte,
- každý z 5 *worker* uzlov má k dispozícii 16 výpočetných jadier (procesorov),
- správca klastra je Standalone a dáta sú načítavané z distribuovaného úložiska HDFS.

V každej z nasledujúcich podkapitol je cieľom určiť optimálnu konfiguráciu parametrov vstupujúcich do testovacieho scenára s ohľadom na najkratší čas spracovania a efektivitu využitia zdrojov. Ako *driver* program je vo všetkých scenároch využitý Spark shell.

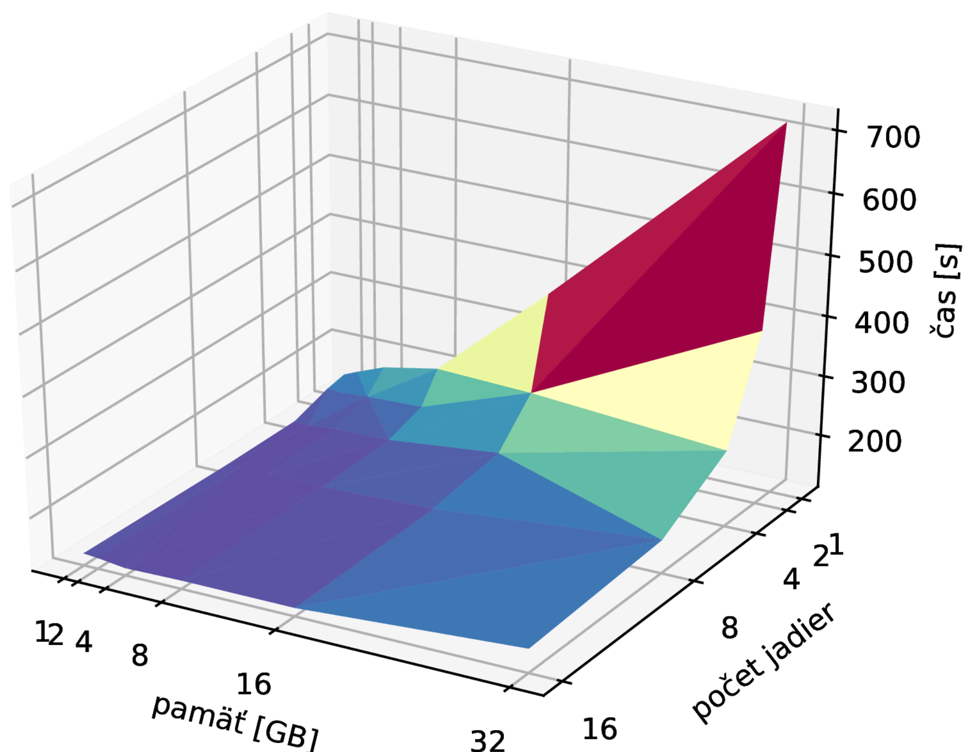
Pojmy používané v tejto kapitole boli podrobne vysvetlené už v kapitole 2. Preto uvediem len ich krátku rekapituláciu. Exekútori sú spúšťaní na *worker* uzloch v klastri. Každý z týchto procesov beží vo vlastnom JVM a je určený na spracovanie jemu pridelených *taskov*. Na *tasky* sa delí každá úloha podľa princípov popísaných v podkapitole 2.3. Spark shell je možné konfigurovať tak, aby bol každému exekútorovi v celom klastri pridelený zadaný objem zdrojov. Zdrojmi tu rozumieme počet výpočetných jadier a objem operačnej pamäte.

## 6.1 Konfigurácia objemu operačnej pamäte a počtu jadier

Tento testovací scenár skúma vplyv konfigurácie zdrojov pridelených jednotlivým exekútorom na rýchlosť spracovania dát. Vstupné dáta boli vo forme `JSON` súboru s veľkosťou 58, 93 GB. Tento súbor bol Sparkom rozdelený na logické bloky podľa vzťahov 5.1, 5.2 a 5.3. Nastavenie delenia súborov na logické bloky bolo ponechané v réžii Sparku. Výpočet veľkosti logických blokov teda prebehol následne:

$$goalSize = \frac{58,93}{1}$$
$$splitSize = max(1\text{ MB}, min(58,93\text{ GB}, 128\text{ MB})) = 128\text{ MB}.$$

Vo výsledku je vstupný súbor rozdelený na 472 logických blokov po 128 MB. Testovacou funkcia v tomto scenári je funkcia `getHttpHostnames`.




Obr. 6.1: Čas vyhodnotenia funkcie `getHttpHostnames` vzhľadom na počet jadier a veľkosti pamäte, ktoré sú pridelené jednému exekútorovi. Merané na vstupnom `JSON` súbore s veľkosťou $58,93$ GB

Graf s výsledkami merania je uvedený na obrázku 6.1. Z tohto grafu a tabuľky 6.1 je vidieť, že hlavným faktorom ovplyvňujúcim dobu spracovania je celkový dostupný počet jadier, ktoré je možné pri výpočtoch využiť. Čím vyšší počet jadier sa môže zapojiť do výpočtu, tým kratšie spracovanie trvá. Počet jadier využiteľných pri výpočte závisí na celkovom počte zdrojov *worker* uzla a konfigurácii zdrojov dostupných jednotlivým exekútorom. Napríklad pri pridelení 2 jadier a 8 GB operačnej pamäte jednému exekútorovi sa počet využiteľných jadier vypočíta takto:

1. podľa počtu pridelených jadier je v každom uzle možné vygenerovať $\lfloor \frac{16}{2} \rfloor = 8$ exekútorov, pričom každý môže využiť 2 jadrá.

2. Štyri uzly majú k dispozícii po 46 GB operačnej pamäte. V týchto uzloch je možné vytvoriť maximálne $\lfloor \frac{46}{8} \rfloor = 5$ exekútorov. V uzle s dostupnými 22 GB operačnej pamäte je možné vytvoriť maximálne $\lfloor \frac{22}{8} \rfloor = 2$ exekútorov.

3. V tomto prípade je maximálny počet exekútorov obmedzený požiadavkami na operačnú pamäť. Spolu v celom klastri je tak možné vytvoriť maximálne $4 \times 5 + 2 = 22$ exekútorov.

4. V celom klastri je možné vytvoriť 22 exekútorov, z ktorých má každý k dispozícii 2 jadrá. To dáva dokopy 44 využiteľných jadier, tak ako je to uvedené aj v tabuľke 6.1.

Rovnakým spôsobom sa dá spočítať počet jadier využiteľných pri výpočte aj pre ostatné kombinácie dostupných a nakonfigurovaných zdrojov.

Pri zapojení všetkých 80 jadier sa časy výpočtov, vzhľadom na konfiguráciu zdrojov pridelených exekútorovi, príliš nelíšia. Nie je však dobré prideľovať príliš veľký objem operačnej pamäte jednému exekútorovi. Správa veľkého objemu operačnej pamäte v rámci jedného JVM môže viesť k spomaleniu (napríklad spúšťaním Garbage collectoru). Naopak príliš malý objem pridelenej pamäte nemusí exekútorom stačiť na vykonanie danej úlohy, čo vedie k chybe a pádu výpočtu. Napríklad široké transformácie sú náročnejšie na objem operačnej pamäte pri výmene dát medzi procesmi. Takisto nie je dobré prideľovať príliš málo jadier, pretože to vedie k vytvoreniu viacerých JVM v jednom uzle a to spôsobuje ďalšiu zbytočnú réžiu.

| pamäť [GB] \ počet jadier | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| 1 | 80 | 80 | 80 | 80 | 80 |
| 2 | 75 | 80 | 80 | 80 | 80 |
| 4 | 49 | 74 | 80 | 80 | 80 |
| 8 | 22 | 44 | 72 | 80 | 80 |
| 16 | 9 | 18 | 36 | 72 | 80 |
| 32 | 4 | 8 | 16 | 32 | 64 |

Tabuľka 6.1: Počet využitých jadier vzhľadom na zdroje pridelené exekútorom

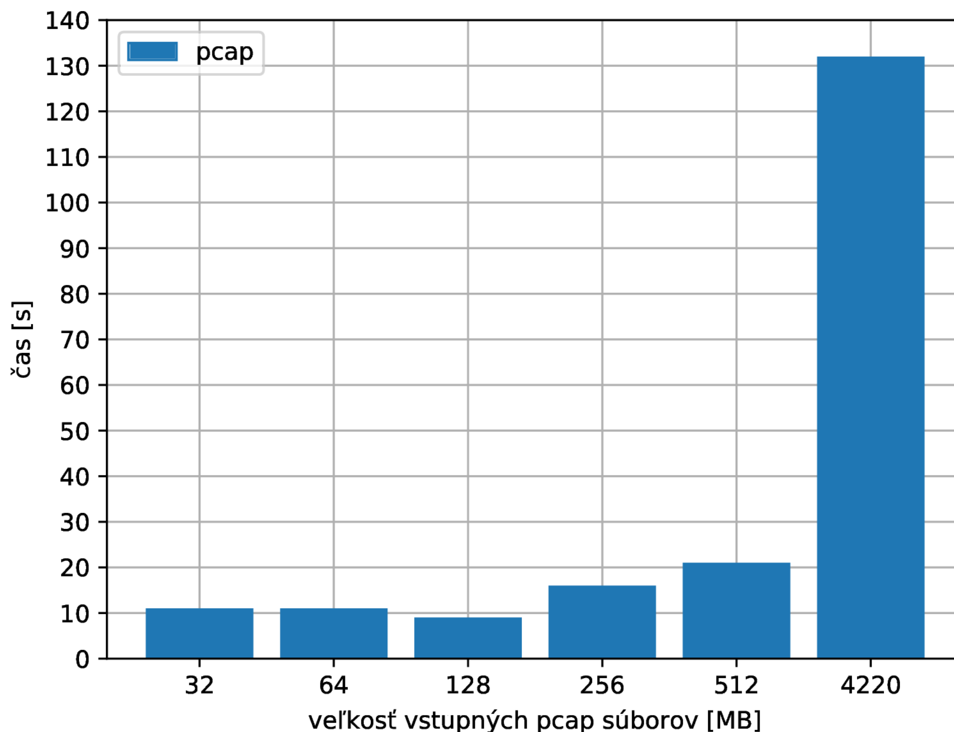
Zo získaných výsledkov je možné pre implementovanú aplikáciu určiť optimálnu voľbu prideľovania zdrojov v danom klastri. Počet výpočetných jadier pridelených jednému exekútorovi by nemal byť menší než 4 a objem operačnej pamäte by sa mal pohybovať medzi 4 až 16 GB.

## 6.2 Veľkosť vstupných pcap súborov

Zatiaľ čo `JSON` súbory môže Spark deliť ľubovolne do logických blokov a čítať po riadkoch, `pcap` súbory sú kvôli globálnej hlavičke v Sparku už ďalej nedeliteľné. Veľkosť jedného `pcap` súboru je totožná s veľkosťou logického bloku. Preto sa tieto súbory musia rozdeliť na menšie ešte pred ich spracovaním.

Tento testovací scenár sa zaoberá optimálnou voľbou veľkosti vstupných `pcap` súborov. Testovacie dáta v pôvodnej podobe reprezentuje `pcap` súbor s veľkosťou $4,22$ GB. Ten bol rozdelený na 5 sád menších súborov. Veľkosť súborov v prvej sade je 32 MB, v ďalších postupne 64 MB, 128 MB, 256 MB a v poslednej 512 MB. Voľba pridelenia zdrojov na jedného exekútora vyplýva z analýzy ich optimálneho nastavenia v predchádzajúcom scenári (pozri podkapitolu 6.1). Každému exekútorovi je pridelených 16 GB operačnej pamäte a 16 výpočetných jadier.

Na obrázku 6.2 je uvedený graf dĺžky spracovania dát pre všetky uvedené veľkosti vstupov. Dáta boli v tomto prípade spracovávané funkciou `getHttpHostnames`, ktorá obsahuje len úzke transformácie. Preto je čas spracovania dát výrazne nižší než pri spracovaní funkciou `getTcpFlows`, ktorá využíva širokú transformáciu `groupByKey`. Časy spracovania dát touto funkciou môžeme vidieť v grafe na obrázku 6.3. Relatívne časy spracovania vzhľa-




Obr. 6.2: Čas vyhodnotenia funkcie `getHttpHostnames` vzhľadom na rôzne veľkosti vstupných `pcap` súborov. Funkcia obsahuje len úzke transformácie

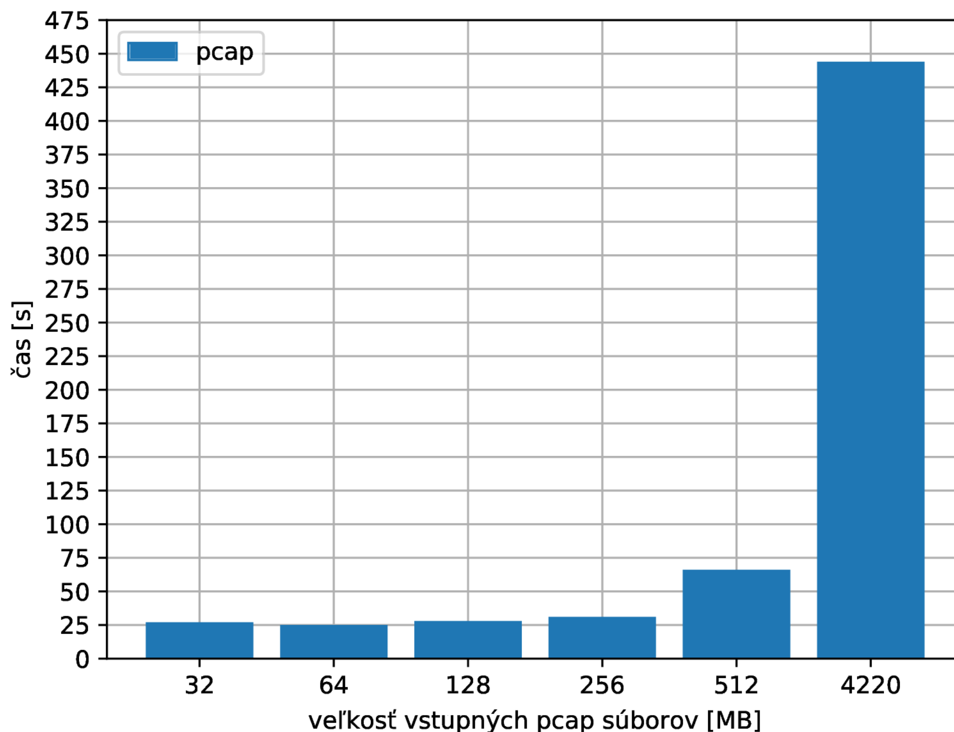
dom na veľkosť vstupných súborov sú však v obidvoch grafoch skoro rovnaké. Najrýchlejšie sú spracované vstupy s veľkosťami 32 MB, 64 MB a 128 MB. Pri väčších veľkostiach začína spracovanie výraznejšie spomaľovať. Pri pôvodnej veľkosti súboru je výkon systému degradovaný viac než 10 násobne oproti optimálnemu riešeniu.

Z uvedených výsledkov vyplýva, že optimálna veľkosť vstupných `pcap` súborov je 128 MB. Delenie na menšie časti nemá zmysel, lebo neprináša už žiadne zrýchlenie. 128 MB je veľkosť fyzických blokov, na ktoré sa súbory delia v HDFS a aj veľkosť logických blokov v Sparku so štandardnou konfiguráciou. Týmto testom sa potvrdila výhodnosť štandardného nastavenia veľkosti logického bloku v Sparku z hľadiska výkonu systému.

## 6.3 Formáty vstupov a funkcie rozhrania aplikácie

Posledný testovací scenár je zameraný na rýchlosť spracovania `JSON` a `pcap` vstupov pomocou nasledujúcich funkcií rozhrania aplikácie:

- funkcie obsahujúce len úzke transformácie:
  - `getHttpHostnames` – slúži na získanie URL z HTTP hlavičky.
  - `getDnsData` – poskytuje RDD s dátami z DNS komunikácie.
  - `getCipherSuites` – slúži na získanie spôsobu šifrovania komunikácie v SSL/TLS protokoloch.



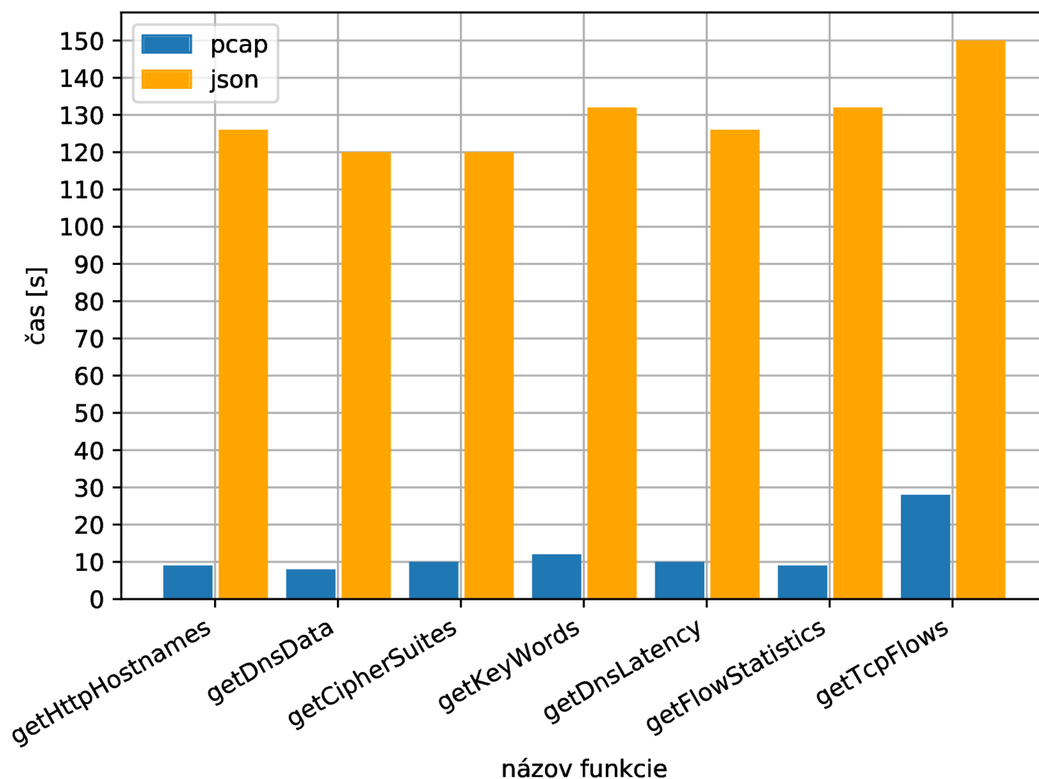

Obr. 6.3: Čas vyhodnotenia funkcie `getTcpFlows` vzhľadom na rôzne veľkosti vstupných `pcap` súborov. Funkcia obsahuje širokú transformáciu `groupByKey`

- funkcie obsahujúce širokú transformáciu:
  - `getDnsLatency` – slúži na získanie informácií o oneskorení DNS serverov.
  - `getFlowStatistics` – poskytuje štatistické informácie o sieťových tokoch.
  - `getTcpFlows` – poskytuje dáta TCP tokov vrátane dát aplikačnej vrstvy vo forme hexadecimálneho reťazca.
- funkcia obsahujúca akciu – `getKeyWords` obsahuje akciu `reduce` a následne transformuje získanú kolekciu na RDD.

Takisto ako v minulom scenári bolo každému exekútorovi pridelených 16 GB operačnej pamäte a 16 výpočetných jadier. Vstupnými súbormi v tomto meraní boli 58, 93 GB veľký `JSON` súbor a sada `pcap` súborov s veľkosťou súboru 128 MB. Vstupný `JSON` súbor bol vytvorený z pôvodného nerozdeleného `pcap` súboru. Obsahom informácií si tak vstupy v obidvoch formátoch zodpovedajú.

Výsledky tohto merania sú zhrnuté v grafe na obrázku 6.4. `JSON` je textový formát, preto majú `JSON` súbory rádovo väčšiu veľkosť než im zodpovedajúce binárne `pcap` súbory. Vďaka tomu vidíme v grafe, že čas potrebný na spracovanie `JSON` vstupu je výrazne vyšší ako zodpovedajúceho vstupu vo formáte `pcap`. V prípade rovnakých absolútnych veľkostí vstupov v obidvoch formátoch by doba ich spracovania bola podobná.

Čas spracovania je pri funkciách obsahujúcich širokú transformáciu mierne vyšší. Výraznejšie vyšší je len u funkcie `getTcpFlows`. To je dané väčším objemom dát výslednej kolekcie, ktorá obsahuje dáta aplikačnej vrstvy vo forme hexadecimálnych reťazcov.




Obr. 6.4: Čas vyhodnotenia jednotlivých funkcií. Oranžovou farbou je označené vyhodnotenie funkcie so vstupným `JSON` súborom s veľkosťou 58,93 GB, modrou vyhodnotenie funkcie s `pcap` vstupom rozdeleným na 33 `pcap` súborov, každý s veľkosťou 128 MB

Spracovanie rovnakého objemu informácií je výrazne rýchlejšie pri vstupoch vo formáte `pcap`, pokiaľ majú tieto vstupy vhodnú maximálnu veľkosť (napr. 128 MB ako v tomto prípade). Z hľadiska rýchlosti spracovania je dobré tiež používať čo najmenej širokých transformácií. V implementovanej aplikácii je v rámci jednej funkcie použitá vždy maximálne jedna široká transformácia. Tieto transformácie tak nemajú výrazný vplyv na výkon.

# Kapitola 7

# Zhodnotenie výsledkov a možnosti ďalšieho vývoja

Aplikácia je implementovaná vo forme knižnice a je spustiteľná prostredníctvom rôznych nástrojov, ktoré sú schopné pristupovať k Spark klastru. V tejto práci boli využité nástroje Spark shell a Apache Zeppelin. Prostredie programu Apache Zeppelin slúžilo na vizualizáciu spracovanej sieťovej komunikácie a pomocou Spark shellu bola testovaná rýchlosť spracovania dát. Čas potrebný na spracovanie vstupných dát je do značnej miery ovplyvnený dostupným hardvérom v klastri. V referenčnom klastri, popísanom v minulej kapitole, dokáže aplikácia spracovávať desiatky až nízke stovky gigabajtov dát z `JSON` vstupov v rádoch minút. Formát `pcap` dokáže reprezentovať rovnaký objem informácií ako `JSON` v súboroch s viac než desaťnásobne menšou veľkosťou. Napríklad $4,22$ GB `pcap` súboru zodpovedá $58,93$ GB súbor `JSON`. Súbory vo formáte `pcap` s veľkosťou v jednotkách GB dokáže aplikácia spracovať v jednotkách až nízkych desiatkach sekúnd. Rýchlosť spracovania vzhľadom k objemu informácií je tak výrazne vyššia pri vstupoch vo formáte `pcap`. Uvedené časy spracovania dát platia pri optimálnom nastavení parametrov a použití jednej z funkcií rozhrania aplikácie. Optimálne nastavenie parametrov bolo určené v experimentálnej časti v kapitole 6 a zahŕňa:

- pridelenie pamäte a výpočetných jadier na jedného exekútora tak, aby sa do výpočtu mohlo zapojiť čo najviac dostupných jadier,
- rozdelenie veľkých vstupných `pcap` súborov na menšie s veľkosťou 128 MB.

V tabuľke 7.1 je uvedené porovnanie rýchlosti spracovania dát na jednom stroji programom TShark a distribuovaného spracovania implementovanou aplikáciou ndx-spark. Rýchlosť spracovania je v týchto dvoch prípadoch primárne závislá od hardvéru, na ktorom sú aplikácie spustené. Preto je potrebné brať namerané hodnoty len orientačne. Ide tak o voľné porovnanie, pri ktorom bola aplikácia ndx-spark spustená v referenčnom klastri a program TShark na jednom zo strojov, ktoré tento klaster tvoria. Čas distribuovaného výpočtu je meraný za rovnakých podmienok ako v podkapitole 6.3. Z výsledkov je vidieť, že distribuované spracovanie je pri väčšom objeme dát výrazne rýchlejšie. Časy distribuovaného spracovania 128 MB a $4,22$ GB dát sa príliš nelíšia, pretože počet vytvorených *taskov* je v obidvoch prípadoch menší ako počet dostupných výpočetných jadier. Naopak rozdiel časov spracovania týchto objemov dát je pri výpočte na jednom stroji až 10 minút. Distribuovaný výpočet v Sparku vždy spotrebuje nejaký čas na vlastnú réžiu. Z toho dôvodu môže byť spracovanie objemov dát menších než 128 MB pri použití programu TShark

na jednom stroji rýchlejšie. V absolútnych číslach je však z používateľského hľadiska tento rozdiel zanedbateľný.

| | Analýza HTTP komunikácie | | Analýza DNS komunikácie | |
|---|---|---|---|---|
| celková veľkosť `pcap` vstupov | 128 MB | 4,22 GB | 4,22 GB | 6,86 GB |
| TShark | 19 s | 10 min 44 s | 11 min 48 s | 19 min 39 s |
| ndx-spark | 7 s | 10 s | 9 s | 20 s |

Tabuľka 7.1: Porovnanie časov distribuovaného spracovania dát sieťovej komunikácie pomocou implementovanej aplikácie ndx-spark a spracovania na jednom stroji programom TShark.

Priestor na rozšírenie aplikácie je veľký. Uvediem hlavné smery, ktorými sa ďalší vývoj môže uberať:

- vstupné formáty – ďalšími podporovanými formátmi by mohli byť napríklad `pcapng` alebo odlišne formátovaný `JSON`. Takéto vstupy nie sú Sparkom podporované, vyžadujú vlastnú implementáciu načítania, podobne ako formát `pcap`. Okrem načítania by bolo potrebné implementovať aj podporu spracovania dát. V prípade formátu `pcapng` by stačilo rozšíriť súčasnú implementáciu spracovania.

- Podporované protokoly – iným možným rozšírením je podpora ďalších sieťových protokolov naprieč všetkými vrstvami. Podporu nových protokolov je možné implementovať bez väčších zásahov do súčasnej implementácie. V prípade `JSON` formátu je potrebné, aby bol daný protokol podporovaný aj programom, ktorý `JSON` generuje. V opačnom prípade je možné takýto protokol spracovávať len z `pcap` vstupov.

- Analýza v reálnom čase – by bola možná priamo v prostredí Spark využitím modulu Spark Streaming a nástrojov ako sú Apache Flume alebo Apache Kafka. Inšpiráciou môže byť projekt OpenSOC (The Open Security Operations Center) od spoločnosti Cisco, slúžiaci na bezpečnostnú analýzu Big Data. Informácie o tomto projekte a analýze Big Data spoločnosťou Cisco boli publikované v knihe od Omara Santosa [13].

- Vizualizácia výsledkov – bola implementovaná v nástroji Apache Zeppelin. V podkapitole 2.4 sú spomínané ďalšie nástroje, v ktorých by zobrazenie dát mohlo byť implementované. Pre ďalší vývoj aplikácie by mohli byť nové možnosti zobrazenia dát zaujímavé.

# Kapitola 8

# Záver

Cieľom tejto práce bolo ukázať možnosti spracovania veľkého objemu dát sieťovej komunikácie, ktoré poskytuje distribuované prostredie Apache Spark. Práca prináša pohľad na činnosť celého distribuovaného systému, ktorý je zložený z množstva rôznych komponent.

Výsledná aplikácia má podobu knižnice, ktorá funguje v rôznych hostiteľských prostrediach schopných pristupovať k Sparku klastru (tzv. *driver*). Načítanie a spracovanie vstupných dát je implementované v jazyku Java. Jazyk Scala je využitý na implementáciu aplikačného rozhrania a manipuláciu s distribuovanými kolekciami. Implementáciou týchto častí v jazyk Scala je dosiahnuté väčšej prehľadnosti kódu a jednoduchšej práce s aplikáciou v hostiteľskom prostredí.

Vstupné dáta v reálnom Spark klastri je potrebné mať uložené v distribuovanom úložisku. V tejto práci bolo použité úložisko HDFS. Aplikácia je schopná spracovávať dáta sieťovej komunikácie vo formátoch `JSON` a `pcap`. Súbory vo formáte `pcap` obsahujú globálnu hlavičku a nie je ich preto možné deliť priamo v prostredí Apache Spark. Tieto vstupné súbory je potrebné rozdeliť na súbory s požadovanou veľkosťou ešte pred vstupom do aplikácie. To je možné napríklad pomocou programu tcpdump. Najlepšie výsledky rýchlosti spracovania dosahuje aplikácia pri veľkosti vstupných `pcap` súborov do 128 MB. Vstupy vo formáte `JSON` sú vytvárané programom Wireshark. Informácie, ktoré obsahujú sú tak plne závislé na spracovaní týmto programom. To vedie k odlišnostiam medzi výsledkami spracovania dát v závislosti na vstupnom formáte. Spracovanie vstupov vo formáte `JSON` je vo všeobecnosti pomalšie, pretože tento formát reprezentuje rovnaký objem informácií v súboroch s viac než desaťnásobne väčšou veľkosťou oproti formátu `pcap`.

Výsledky sú zobrazované v hostiteľskom prostredí Apache Zeppelin. Tento aj všetky ostatné nástroje použité v tejto práci sú voľne dostupné open-source projekty. Aplikácia prináša možnosť spracovania aj takých objemov dát, ktoré by buď nebolo vôbec možné spracovať na jednom stroji alebo by to bolo časovo príliš náročné. Spracovanie `pcap` súboru s veľkosťou v jednotkách GB trvá programu Wireshark na jednom stroji desiatky minút. Za predpokladu optimálneho nastavenia dokáže aplikácia spracovať takéto vstupy v rádoch desiatok sekúnd.

# Literatúra


[1] Béder, M. M*obilní aplikace pro testování zranitelností DNS*. Brno: Vysoké učení technické v Brně, Fakulta informačních technologií, 2016. Vedúci práce Ing. Michal Kováčik.

[2] Fielding, R.; Reschke, J. : Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230, IETF, Jún 2014 [cit. 2018-04-27]. Dostupné z: <https://tools.ietf.org/html/rfc7230>

[3] Frampton, M. *Mastering Apache Spark*. Packt Publishing, 2015. ISBN 978-1-78398-714-6.

[4] Gourley, D.; Totty, B.; Sayer, M.; aj. *HTTP: The Definitive Guide: The Definitive Guide*. Definitive Guides, O'Reilly Media, 2002. ISBN 978-1-56592-509-0.

[5] *Development/LibpcapFileFormat – The Wireshark Wiki* [online]. Guy Harris, 2018 [cit. 2018-04-20]. Dostupné z: <https://wiki.wireshark.org/Development/LibpcapFileFormat>.

[6] *Message Headers* [online]. IANA, 2018 [cit. 2018-04-27]. Dostupné z: <https://www.iana.org/assignments/message-headers/message-headers.xhtml>.

[7] *The Jupyter notebook – Jupyter Notebook 5.5.0.dev0 documentation* [online]. Jupyter Steering Council, 2018 [cit. 2018-04-10]. Dostupné z: <https://jupyter-notebook.readthedocs.io/en/latest/>.

[8] Karau, H.; Konwinski, A.; Wendell, P.; aj. *Learning Spark: Lightning-Fast Big Data Analysis*. O'Reilly Media, 2015. ISBN 978-1-449-35862-4.

[9] Karau, H.; Warren, R. *High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark*. O'Reilly Media, 2017. ISBN 978-1-491-94320-5.

[10] Laskowski, J. : *M*astering Apache Spark 2 [online]. GitBook, [cit. 2017-12-20]. Dostupné z: <https://www.gitbook.com/book/jaceklaskowski/mastering-apache-spark/details>

[11] Mockapetris, P. : Domain names - implementation and specification. STD 13, IETF, November 1987 [cit. 2018-05-01]. Dostupné z: <https://tools.ietf.org/html/rfc1035>

[12] Quittek, J.; Zseby, T.; Claise, B.; aj. : Requirements for IP Flow Information Export (IPFIX). RFC 3917, IETF, Október 2004 [cit. 2017-12-22]. Dostupné z: <https://tools.ietf.org/html/rfc3917>

[13] Santos, O. *Network Security with NetFlow and IPFIX: Big Data Analytics for Information Security*. Cisco Press, 2015. ISBN 978-1-58714-438-7.

[14] *Apache Hadoop 3.0.0* [online]. The Apache Software Foundation, 2017 [cit. 2017-12-20]. Dostupné z: <http://hadoop.apache.org/docs/r3.0.0/>.

[15] *Apache Spark Documentation* [online]. The Apache Software Foundation, 2017 [cit. 2017-12-15]. Dostupné z: <https://spark.apache.org/docs/2.2.0/>.

[16] *Apache Zeppelin 0.7.3 Documentation* [online]. The Apache Software Foundation, 2018 [cit. 2018-04-15]. Dostupné z: <https://zeppelin.apache.org/docs/0.7.3/>.

[17] Yadav, R. *Spark Cookbook*. Packt Publishing, 2015. ISBN 978-1-78398-706-1.

# Prílohy

# Príloha A

# Obsah priloženého pamäťového média

Priložené CD obsahuje:

- `DIP.pdf` – text tejto práce vo formáte `pdf`,
- `README.txt` – návod na preklad a spustenie aplikácie,
- `/src` – adresár obsahujúci zdrojové súbory aplikácie,
- `/tex` – adresár obsahujúci zdrojové LaTeX súbory tejto práce,
- `/ndx-spark` – adresár obsahujúci archív s preloženou aplikáciou.