VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

DISTRIBUOVANÝ REPOSITÁŘ DIGITÁLNÍCH FORENZNÍCH DAT

DISTRIBUTED FORENSIC DIGITAL DATA REPOSITORY

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. MARTIN JOSEFÍK**
AUTHOR

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

BRNO 2018

Zadání diplomové práce/20639/2017/xjosef00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2017/2018

Zadání diplomové práce

Řešitel: **Josefík Martin, Bc.**
Obor: Informační systémy
Téma: **Distribuovaný repositář digitálních forenzních dat**
Distributed Forensic Digital Data Repository
Kategorie: Databáze

Pokyny:

1. Seznamte se s formáty digitálních forenzních dat a způsoby jejich uložení. Prozkoumejte existující systémy pro uložení digitálních forenzních dat (např. AFF4). Seznamte se s distribuovanými databázemi a úložišti pro rozsáhlá strukturovaná i nestrukturovaná data.
2. Navrhněte distribuované úložiště rozsáhlých digitálních forenzních dat (Big data) vč. aplikačního rozhraní pro optimální přístup k různým datům (sekvenční a náhodní čtení, dotazování, zpracování Big data přístupy). Zvolte vhodné technologie pro implementaci úložiště.
3. Po konzultaci s vedoucím navržené úložiště implementujte. Úložiště musí umožňovat přidávání podpory pro nové druhy forenzních dat za běhu. Otestujte použití a výkon úložiště pro vybrané druhy digitálních forenzních dat.
4. Výsledky zdokumentujte, vyhodnoťte a projekt zveřejněte jako open-source.

Literatura:

- Advanced Forensic Framework 4 (AFF4). [http://forensicswiki.org/wiki/AFF4]
- Quick, Darren, Kim-Kwang Raymond Choo. "Big forensic data reduction: digital forensic images and electronic evidence". *Cluster Computing* 19.2 (2016): 723-740.

Při obhajobě semestrální části projektu je požadováno:

- Body 1 a 2.

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: **Rychlý Marek, RNDr., 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

Tato práce se zabývá návrhem distribuovaného repositáře zaměřeného jako úložiště rozsáhlých digitálních forenzních dat. Teoretická část práce pojednává o forenzní analýze digitálních dat a co je jejím cílem. Současně také vysvětluje Big data, vhodné úložiště, jejich vlastnosti, výhody a nevýhody. Hlavní část práce se pak zabývá návrhem a implementací distribuovaného úložiště pro digitální forenzní data. Návrh se rovněž zaměřuje na vhodnou indexaci uložených dat a rozšiřitelnost pro podporu nových druhů digitálních forenzních dat do budoucna.

Abstract

This work deals with the design of distributed repository aimed at storing digital forensic data. The theoretical part of the thesis describes digital forensics and what is its purpose. There are also explained Big data, suitable storages, their properties, advantages and disadvantages, in this part. The main part of the thesis deals with the design and implementation of distributed storage for digital forensic data. The design is also focused in suitable indexing of stored data, and supporting new types of digital forensic data.

Klíčová slova

Big data, forenzní analýza digitálních dat, formáty digitálních forenzních dat, AFF4, distribuované databáze, NoSQL databáze, Kafka, Big data přístupy a technologie, asynchronní komunikace

Keywords

Big data, digital forensics, forensics file formats, AFF4, distributed databases, NoSQL databases, Kafka, Big data approaches and technologies, asynchronous communication

Citace

JOSEFÍK, Martin. *Distribuovaný repositář digitálních forenzních dat.* Brno, 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce RNDr. Marek Rychlý, Ph.D.

Distribuovaný repositář digitálních forenzních dat

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana RNDr. Marka Rychlého, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Martin Josefík
19. dubna 2018

Poděkování

Rád bych touto cestou poděkoval vedoucímu mé diplomové práce panu RNDr. Markovi Rychlému, Ph.D. za odborné vedení, poskytované rady a připomínky v průběhu řešení práce.

Obsah

Kapitola 1

Úvod

Během začátku 21. století došlo k ohromnému růstu v oblastech internetu, sociálních sítí, multimédií, chytrých telefonů a dalších digitálních zařízení, či mobilních plateb a on-line transakcí. V podstatě ve všech oblastech jako je například věda, politika, strojírenství, medicína, doprava, energetika, se nevyhneme použití digitálních zařízení. S tím velmi úzce souvisí nárůst objemu digitálních dat, která je potřeba ukládat, analyzovat, zpracovávat, a také vyšetřovat.

Digitální zařízení se mohou stát terčem útoků, může se jednat o krádež citlivých dat, podvržení dat, sledování a další. Digitální zařízení mohou být také nástrojem zločinu.

Rozdílnost mezi formáty digitálních dat, jejich strukturované a nestrukturované vlastnosti, a také prudký nárůst vyžadují mnoho rozličných přístupů a technologií pro jejich zpracování.

Cílem této práce je navrhnout a vytvořit distribuované úložiště pro digitální forenzní data. V rámci teoretické části práce, se kapitola 2 zaměřuje na forenzní analýzu digitálních dat, vysvětluje, jak probíhá a co je jejím cílem. Zabývá se také formáty digitálních forenzních dat a existujícími systémy, které slouží k uchování těchto dat a důkazního materiálu. Následně je představen systém AFF4.

Kapitola 3 pojednává o úložištích pro rozsáhlá strukturovaná i nestrukturovaná data. V jejich kontextu vysvětluje také termín Big data. Následují odstavce zaměřeny na distribuované databáze včetně jejich výhod a nevýhod. Kapitola je završena uvedením principů NoSQL databází, rozdělení podle typů, a srovnání s relačními databázemi. Na NoSQL databázích bude založeno úložiště distribuovaného repositáře.

Cílem praktické části práce je navrhnout a implementovat distribuovaný repositář. V kapitole návrhu 4 je vysvětlena architektura systému, aplikační rozhraní, vlastnosti úložišť, princip ovládání repositáře a také rozšiřitelnost pro nové druhy digitálních forenzních dat. Systém využívá Big data technologie. Pro komunikaci s klientem jsou zprávy přenášeny tzv. `Message brokerem` Kafka. Jako úložiště slouží, v závislosti na tom, o jaký typ forenzních digitálních dat se jedná, NoSQL databáze a distribuovaný souborový systém HDFS.

Kapitola 5 se zabývá implementací distribuovaného úložiště, technickými detaily, způsobem asynchronní komunikace s databázemi, ukládáním a dotazováním nad daty (sekvenčním i náhodným). Systém pracuje nad frameworkem Spring, který usnadňuje konfiguraci a použití Big data komponent. Jako běhové prostředí byl zvolen projekt `Docker`, který poskytuje jednotné rozhraní pro izolaci aplikací do kontejnerů.

Poslední kapitola 6 se věnuje výkonnosti systému, hardwarovým požadavkům a konfiguraci, a použití systému pro vybrané druhy digitálních forenzních dat.

Kapitola 2

Forenzní analýza digitálních dat

Forenzní analýza digitálních dat je věda identifikující, zachovávající, obnovující, analyzující a předávající fakta ohledně digitálních důkazů nalezených v počítačích nebo digitálních úložištích mediálních zařízení. Nezabývá se tedy pouze počítači, ale také ostatními digitálními technologiemi včetně mobilních telefonů a tabletů, mobilních sítí, internetového bankovnictví, datových médií apod.

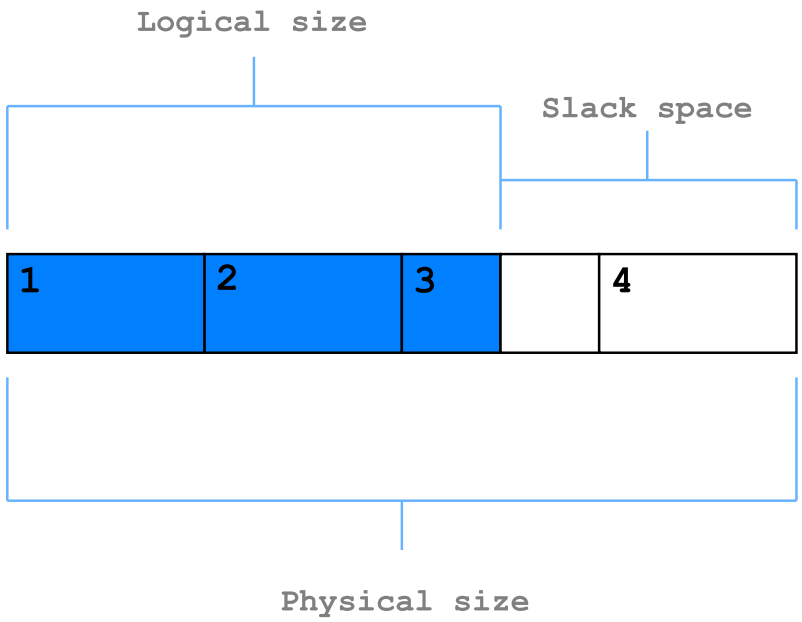
Forenzní analýza digitálních dat slouží k získání digitálního důkazního materiálu, který může být použit v soudní síni proti obviněnému. Nalezené výstupy nejsou omezeny k použití pouze u soudu. Častokrát může nějaká firma řešit interní záležitosti jako například porušení firemní politiky, kdy se zase musí najít (digitální) důkaz, který potvrzuje nebo vyvrací obvinění.

Pod výše uvedenými aktivitami se skrývá [28]:

- Identifikace – Jedná se o první část celého procesu. Předtím, než je cokoliv zkoumáno a analyzováno, je důležité identifikovat, kde jsou data uložena. Typicky jsou uložena na diskových jednotkách, serverech, flash klíčenkách, síťových zařízeních.
- Zachování – Důležitá je ochrana důkazů, tzn. pro sběr a analýzu informací je potřeba zachovat původní data, musí se zabránit jejich změně a ztrátě. Bez integrity je důkazní materiál nepoužitelný. Identifikovaná původní data mohou být zajištěna chronologickým řetězcem dokumentů (anglicky `chain of custody`), který zaznamenává veškeré aktivity s digitálním důkazem jako je přenos, úschova, kontrola a stav.
- Obnovení – Součástí procesu je i obnova dat, která může zahrnovat obnovu smazaných dat procesy operačního systému, úmyslně smazané soubory, soubory chráněné heslem a také poškozené soubory. I po obnovení smazaného souboru musí být stále zachována integrita.
- Analýza – Jedná se o hlavní část vyšetřování. Cílem je shromáždit co nejvíce relevantních artefaktů. Prohledávána je paměť, registry, výpisy z logů aplikací, historie internetového prohlížeče apod. Podstatná je i dokumentace všech kroků, které byly provedeny.
- Předání – Po analýze jsou artefakty důkladně zdokumentovány a odevzdány například ve formě protokolu. Po shromáždění všech nalezených informací může ale nemusí dojít k definitivnímu rozhodnutí. Co se s informacemi stane už není v rukou vyšetřovatele. Tady proces forenzní analýzy končí.

Vyšetřování digitálních forenzních dat obvykle zahrnuje vytvoření forenzní duplikace zkoumaného média. Provádí se proto, aby se neznehodnotil původní zdroj. Například dojde k vytvoření obrazu disku, který je kopií celého disku nebo jeho části bit po bitu. Neduplikuje se celý systém, pokud je prostor příliš velký. Obraz je statický snímek, který může být analyzován za účelem odhalení nebo stanovení událostí ohledně incidentů, a může tak být použitý jako důkaz v soudní síni. Analýza je prováděna na kopii pro zachování integrity originálu.

Vyšetřovatel zanalyzuje obraz pomocí snímacích technik, aby získal relevantní data z disku. Forenzní obraz obsahuje soubory z disku, nealokovaný prostor a tzv. `slack space`. Slack space je pozůstatek diskového prostoru, který byl alokován pro nějaký počítačový soubor a ten všechen prostor nepotřebuje. Právě v těchto prostorech mohou být nalezeny relevantní artefakty a informace jako např. smazané soubory, či jejich fragmenty [34].



Obrázek 2.1: Příklad znázorňuje slack space pro parametry – sektory velikosti 512 bytů jsou shluknuty v operačním systému do skupin po čtyřech, tzn. velikost svazku je 2048 bytů; velikost souboru je 1280 bytů (vyznačen modrou barvou). Pro tento soubor byl alokován celý svazek 2048 bytů. Nevyužité místo (označeno bílou barvou) je slack space, v tomto případě se jedná o 768 bytů [23].

2.1 Formáty digitálních forenzních dat

Typů digitálních forenzních dat existuje spousta. Každý typ takových dat může být reprezentován jiným formátem. Tato sekce čerpá informace převážně z [7].

Mnoho forenzních počítačových programů používá své vlastní formáty pro uložení informace. Můžeme je rozdělit na nezávislé (anglicky `Independent File Formats`) a programově specifické formáty (anglicky `Program-Specific File Formats`):

- Nezávislé – Tyto formáty byly vyvinuty nezávisle na konkrétním forenzním programu. Patří mezi ně `AFF`, `AFF4`, `gfzip`, `Raw Image Format`.

- Programově specifické – Byly vyvinuty pro použití specifickými forenzními programy. Většinou každý takový formát je unikátní, a proto je pro přečtení potřeba unikátního nástroje. Zástupci jsou například `Encase image file format`, `ProDiscover image file format`, `IXimager file formats`.

Identifikace formátu

Identifikace formátu souboru je proces určení formátu nějaké sekvence bytů. Operační systém toto typicky dělá rozpoznáním přípony souboru nebo podle zabudované `MIME` informace. Forenzní aplikace musí identifikovat typy souborů podle obsahu. Mezi existující systémy patří například tyto projekty – `libmagic`, `PRONOM`, `Apache Tika` [1] [6].

2.2 Existující systémy

Vyznamným zástupcem pro uložení digitálních forenzních dat je systém AFF4, který bude popsán detailněji.

2.2.1 AFF4

Jedná se o open source formát pro ukládání digitálních důkazů a dat. Jeho výhodami jsou správa metadat a možnost komprese. Tato sekce čerpá převážně z [1]. Je založen na objektově orientované architektuře. Veškerá množina známých objektů je označována jako `AFF4 universe`. Takový prostor je definovaný jako nekonečný, protože AFF4 je navržen pro škálování obrovského množství důkazního materiálu. Všechny objekty jsou adresovatelné jejich jménem, které je v rámci AFF4 universe unikátní.

Příkladem jména nějakého AFF4 objektu může být:
`urn:aff4:f3eba626-505a-4730-8216-1987853bc4d2`
Jedná se o standardní URN notaci, URN je unikátní.

AFF4 universe používá RDF notaci pro specifikaci atributů objektů. V nejjednodušší podobě je RDF množina tvrzení o objektu ve formátu:
`Subject Attribute Value`

Příklad:
```
******** Object urn:aff4:f3eba626-505a-4730-8216-1987853bc4d2 ***********
aff4:stored = urn:aff4:4bdbf8bc-d8a5-40cb-9af0-fd7e4d0e2c9e
aff4:type = image
aff4:interface = stream
aff4:timestamp = 0x49E9DEC3
aff4:chunk_size = 32k
aff4:compression = 8
aff4:chunks_in_segment = 2048
aff4:size = 10485760
```

Příklad ukazuje, že objekt má tyto atributy a hodnoty. Nazýváme je relace nebo fakta. Celý AFF4 universe je sestavený z takových faktů.

[1] http://tika.apache.org/

AFF4 objekty existují, protože dělají něco užitečného, což závisí na rozhraní, které představují. Aktuálně existuje několik rozhraní, nejvýznamnější jsou `Volume` a `Stream`. Rozhraní objektu je fakt o objektu, který nalezneme v atributu `aff4:interface`.

Rozhraní Volume

Rozhraní Volume definujeme jako mechanismus ukládání, který dokáže uložit segment (bit binárních dat) pod nějaké jméno, a získat jej podle tohoto jména. Aktuálně existují dvě implementace: `Directory` a `ZipFile`.

- `Directory Volume` – Tato implementace ukládá segmenty jako soubory uvnitř běžného adresáře v souborovém systému. Hodí se zejména, pokud potřebujeme uložit obraz na souborový systém FAT, přičemž velikost segmentu je malá a nenarazíme tak na omezení velikosti souboru. Je také možné založit adresář na nějaké HTTP adrese, což nám umožní používat obraz přímo z webu.
- `ZipFile Volume` – Jak napovídá název, tato implementace ukládá segmenty uvnitř zip archivu. Malé soubory lze bez problémů otevřít obyčejným průzkumníkem (v OS Windows např. `Windows Explorer`) a data extrahovat. Zase je možné zapsat zip archiv přímo na HTTP server a používat obraz přímo ze serveru.

Je možné převádět mezi oběma formáty z jednoho na druhý, extrahovat zip archiv do adresáře a nebo z adresáře vytvořit zip archiv.

Rozhraní Stream

Streamy jsou základním rozhraním pro ukládání dat obrazu. Stream obsahuje metody typu `read`, `seek`, `tell` a `close`. Podporuje ještě `write`, ale ne k modifikaci obrazu, nýbrž k jeho vytvoření (kvůli výše uvedené integritě zdroje). Pokud nějaký AFF4 objekt podporuje rozhraní stream, lze provést náhodné čtení jeho dat. Existuje několik specifických implementací rozhraní stream, některými z nich jsou:

- `FileBackedObjects` – Stream, který ukládá data v souboru v souborovém systému, jehož pozice je určena URN souboru.
- `HTTPObject` – Pozice souboru je udána pomocí URL. Objekt lze ukládat a číst z HTTP serveru. Implementace umožňuje přečíst určité rozmezí bytů. Režie síťového provozu mezi klientem a serverem je minimální. Je možné vyšetřit vzdálený obraz přes HTTP bez potřeby celé kopie obrazu. Z důvodu bezpečnosti by měl být server pro zápis nějak omezen, například hesly, SSL certifikáty apod. Podpora čtení může být poskytnuta bez omezení, pokud je zdroj dat zašifrován. Zabezpečení serveru je ale mimo rozsah systému AFF4.
- `Segments` – Segmenty jsou komponenty uloženy přímo ve `Volume`. Volume je zjednodušeně řečeno objekt uchovávající segmenty. Segmenty by měly být použity pro malé streamy, protože prohledávat v komprimovaných segmentech může být drahá operace. Segmenty jsou užitečné, pokud potřebujeme vytvořit logický obraz nějaké podmnožiny souborového systému (pouze některé soubory) a ne forenzní obraz celkového systému.
- `Image streams` – Tyto streamy jsou opakem segmentů. Pro velké obrazy nemůžeme použít segmenty, protože by nebyly zkomprimovány efektivně. Image stream ukládá obraz v tzv. `chunks`.

Kapitola 3

Úložiště pro rozsáhlá strukturovaná i nestrukturovaná data

V této kapitole bude představen a vysvětlen termín Big data, následován systémy a mechanismy pro uložení, jako například distribuované databáze a NoSQL databáze, včetně jejich vlastností, výhod a nevýhod.

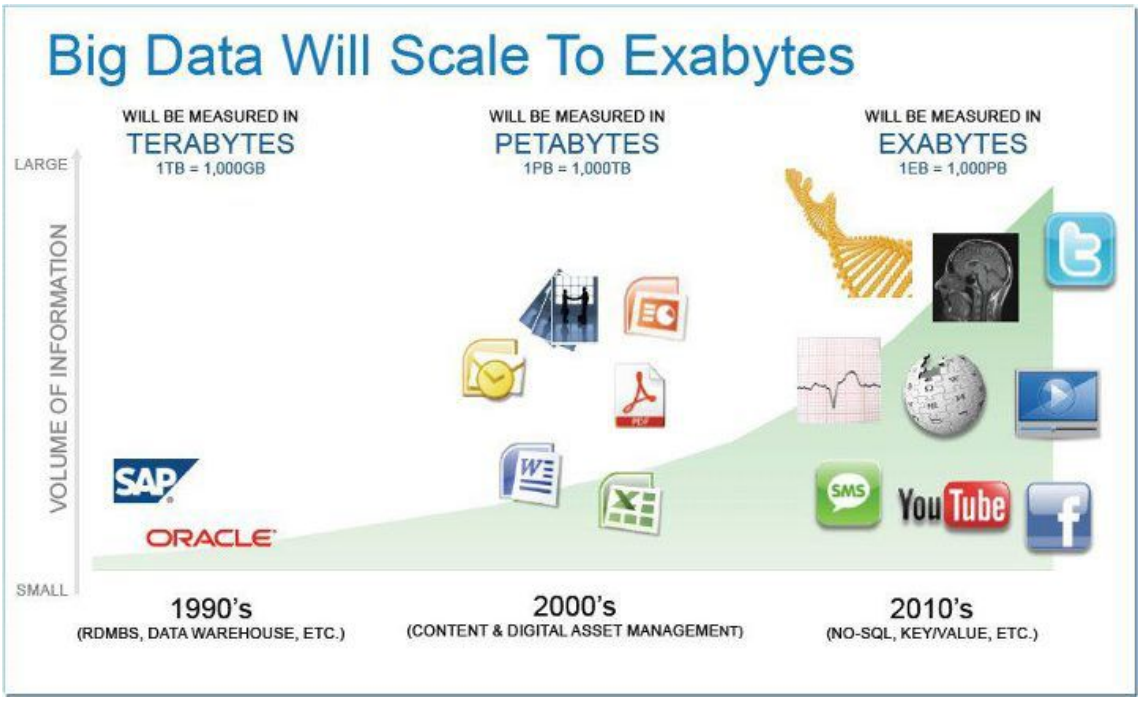
3.1 Big data

Definicí pro frázi Big data existuje několik. Jedná se o termín použitý na soubory dat, které jsou příliš komplexní z hlediska velikosti a různorodosti, a které je nemožné zpracovávat běžně používanými přístupy a softwarovými nástroji v rozumném čase.

Objem takových dat rychle roste. Vyskytují se v mnoha odvětvích, například sběr informací o počasí, sociální sítě, energetické a telekomunikační společnosti, ekonomie a finančnictví, či data z kamer, měření z různých senzorů apod. Z toho plyne, že se jedná o data různorodých typů, mohou být strukturovaná i nestrukturovaná. Proto je potřeba existence různých technologií pro jejich uložení, zpracování, analýzu i zobrazení.

Pojem Big data je často definován jako 4V z anglických slov Volume, Velocity, Variety a Value [17]:

- Volume – Značí množství nebo velikost dat. Je vyžadováno zpracování vysokých objemů dat neznámých hodnot, například síťový provoz, data sesbírána ze senzorů apod.
- Velocity – Vyjadřuje rychlost z hlediska vzniku dat a potřeby jejich analýzy, některá vyžadují zpracování v reálném čase. Nejdůležitější data se zapisují přímo do paměti, a ne na disk, z důvodu co nejrychlejšího zpracování.
- Variety – Znamená různorodost typů. Jedná se především o nestrukturovaná data, například text, audio, video, data o geografické poloze a další. Jsou na ně kladeny velmi podobné požadavky jako na data strukturovaná – sumarizace, monitorování, důvěrnost [17].
- Value – Data mají vlastní hodnotu, která musí být analyzována a zjištěna. Nejedná se o jednoduchý proces, je stále potřeba nových metod a technik zpracování.



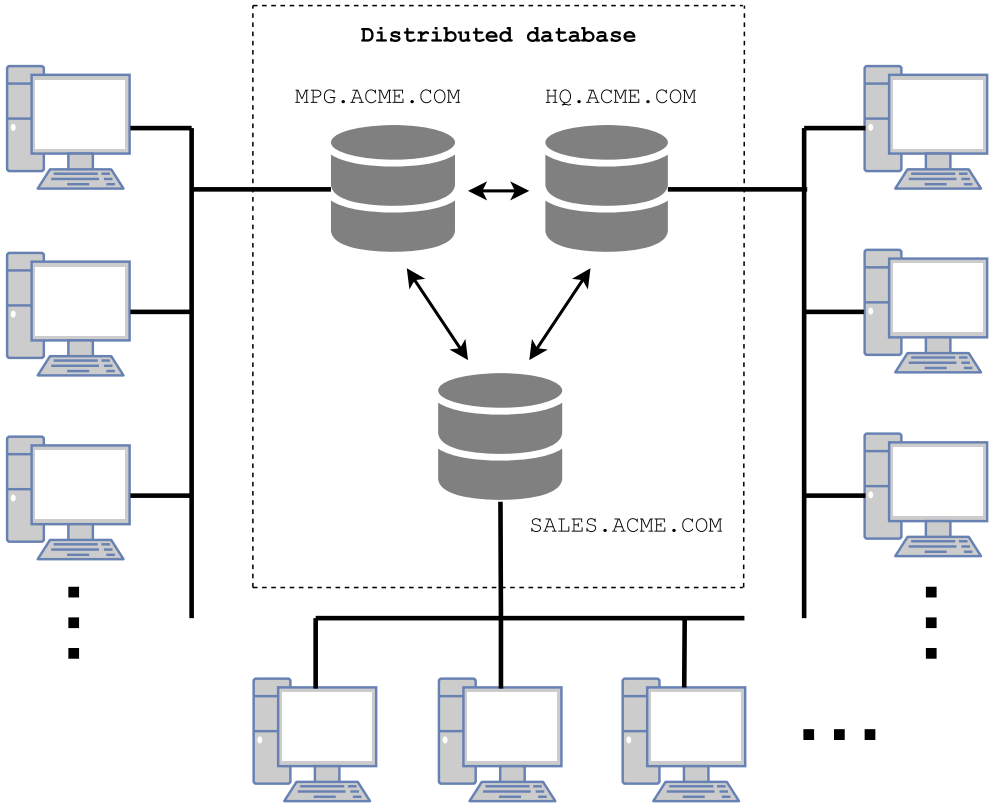
Obrázek 3.1: S přibývajícími novými technologiemi se masivně zvyšuje růst dat a přibývají nové typy [21].

3.2 Distribuované databáze

Distribuovaná databáze se skládá z většího počtu samostatných databází, které mohou být geograficky rozmístěny na jiných pozicích. Jednotlivé uzly spolu komunikují přes počítačovou síť. Každý uzel je sám o sobě databázový systém. DSŘBD neboli systém řízení distribuované báze dat (anglicky `Distributed Database Database Management System`) zajišťuje, že se distribuovaná databáze uživatelům jeví jako jedna jediná databáze. Data jsou fyzicky uložena na různých pozicích. Mohou být spravována rozdílnými SŘBD nezávisle na ostatních pozicích.

Systém řízení distribuované báze dat je centralizovaný systém s těmito vlastnostmi [31]:

- Umí vytvářet, získávat, upravovat a mazat distribuované databáze.
- Zajišťuje důvěrnost a integritu databází.
- Periodicky synchronizuje databázi a poskytuje mechanismy přístupu tak, aby se databáze uživatelům jevila transparentní.
- Zajišťuje, že změna dat v kterémkoliv uzlu se promítne i v ostatních uzlech.
- Je využíván v aplikacích, kde se předpokládá zpracování velkých objemů dat, ke kterým přistupuje současně mnoho uživatelů.
- Je navržen pro heterogenní databázové platformy.



Obrázek 3.2: Schéma distribuované databáze a současný přístup více zařízení k ní [22].

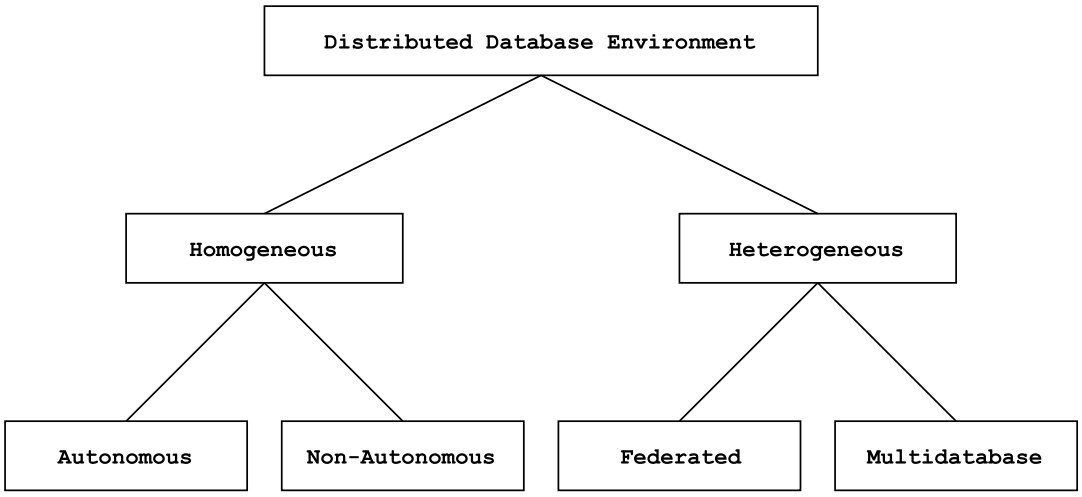
Výhody

- Rozšiřitelnost – Pokud je potřeba databázový systém rozšířit do nových míst nebo přidat další uzly, stačí přidat nový(é) počítač(e) a lokální data v nové pozici, a nakonec je připojit k distribuovanému systému, bez jakéhokoliv přerušení funkcionality. Analogický postup je při odebrání uzlu.
- Spolehlivost – Když nějaký z připojených uzlů selže, nepřestane distribuovaná databáze fungovat, sníží se maximálně výkon.
- Ochrana (záloha) dat – Při zničení jednoho uzlu a smazání dat z něj, mohou být stejná data zálohována i na jiných uzlech.
- Výkonnost – Pokud jsou data efektivně distribuována, může být uživatelův požadavek uspokojen rychleji. Transakce mohou být také distribuované a provedeny rychleji.

Nevýhody

- CAP teorém – Pojednává o tzv. eventuální konzistenci (na rozdíl od absolutní konzistence u ACID). Definuje, že distribuovaný systém může zajistit maximálně 2 z těchto 3 vlastností – konzistence (angl. Consistency), dostupnost (angl. Availability) a odolnost k přerušení (angl. Partition tolerance). Je nutné si zvolit mezi konzistencí a dostupností v případě výpadku části sítě.
- Integrita dat – Data musí být průběžně synchronizována na více uzlech, aby na stejné dotazy nebyly z různých uzlů vraceny rozdílné odpovědi.

- Komunikační režie – I zdánlivě jednoduchá operace může vyžadovat spoustu zbytečné komunikace.
- Cena – DSŘDB vyžaduje drahý a složitý software ke koordinaci uzlů a zajištění transparentnosti [31].
- Mezi další patří – složitost, zabezpečení a řízení souběžného přístupu k datům.



Obrázek 3.3: Distribuované databáze můžeme podle jejich vlastností dělit [31].

3.2.1 Rozdělení

Homogenní

Všechny uzly používají identické SŘBD a operační systémy. Uzly mají informace o ostatních uzlech a spolupracují při zpracování uživatelských požadavků. Homogenní distribuovaná databáze se navenek jeví uživateli jako jeden systém. Je jednodušší jej navrhnout a spravovat.

Autonomní – Každá databáze je nezávislá, neexistuje žádný centrální uzel. Databáze jsou spravovány aplikací, a pro předávání dat používají zasílání zpráv.

Neautonomní – Data jsou distribuována napříč homogenními uzly. O aktualizaci a správu dat se stará systém řízení distribuované báze dat, který běží na centrálním nebo také master uzlu.

Heterogenní

Uzly mohou mít rozdílné operační systémy a SŘBD, které nejsou kompatibilní. Mohou také využívat rozdílná schémata (relační, objektově orientované, hierarchické, ...). Rozdílnost schématu je hlavním problémem při zpracování dotazu a transakcí. Kvůli tomu je také složité dotazování [5]. Heterogenní distribuované databáze můžeme rozdělit na federované (angl. `federated`) nebo multidatabázové.

Architekturami distribuovaných databází jsou centrální architektura, klient-server, peer-to-peer, a multi-databázová architektura.

3.3 Úložiště

V této sekci budou popsána strukturovaná a nestrukturovaná data, jaký je mezi nimi rozdíl a jaká jsou jejich úložiště.

3.3.1 Strukturovaná data

Strukturovanými daty jsou libovolná data, pro která lze sestavit datový model. Datový model přesně organizuje jednotlivé elementy dat a specifikuje, jak spolu elementy souvisí, jaké mají vazby, jak budou uloženy, přístup k nim, a jak souvisí s jednotlivými elementy reálného světa [3]. Typickým úložištěm strukturovaných dat je relační databáze. Pro čtení a dotazování byl vytvořen dotazovací jazyk SQL.

3.3.2 Nestrukturovaná data

Jak už bylo zmíněno v sekci 3.1, Big data patří převážně mezi nestrukturovaná data. Data nemají přesně definovanou strukturu, neexistuje datový model, který by definoval jednotlivé dílčí elementy a jejich vztahy. Mezi typické zástupce patří: videa, obrázky, audio, webové stránky, text, obsah e-mailové komunikace apod. Mezi vhodná úložiště mohou patřit disková úložiště, NAS (celým názvem `Network Attached Storage`), HDFS (`Hadoop Distributed File System`) a NoSQL databáze.

NoSQL databáze

Pod zkratkou se nachází `non SQL` nebo také `Not only SQL`. Jedná se o databázový koncept, ve kterém datové úložiště i zpracování dat používají jiné prostředky než tabulková schémata tradiční relační databáze [10]. NoSQL databáze jsou navrženy pro distribuovaná ukládání a dotazování dat, souběžný přístup, a manipulaci s obrovským objemem dat. Mezi výhody patří: horizontální i vertikální škálovatelnost, distribuovaný přístup, flexibilita schématu, dynamičnost. Nevýhodami však mohou být: absence standardizace, omezené schopnosti dotazování, konzistence [11].

NoSQL databáze mohou být rozděleny do čtyř kategorií [27]:

- Klíč-hodnota (angl. Key-Value databases) – Jsou nejjednoduššími NoSQL úložišti. Každá položka v databázi je uložena jako atribut (klíč) společně se svou hodnotou. Hodnota je typu `blob`, takže pouze aplikace dokáže hodnotu správně interpretovat. Databáze pouze ukládá binární data, kterým nerozumí. Klíč může být složený, např. z několika částí, které lze použít jako ID do struktury a nebo ID jejich položek. Klíče mohou být seřazené, což umožňuje efektivní procházení, nebo také organizovány do hierarchií. Zástupci jsou databáze `Riak` a `Redis`.
- Dokumentové (angl. Document databases) – Párují každý klíč se složitou datovou strukturou, nazývanou dokument. Představuje v podstatě princip klíč-hodnota, ale hodnota je strukturovaná. Dokument může obsahovat mnoho rozdílných párů klíč-hodnota, párů klíč-pole, nebo dokonce vnořené dokumenty. Jsou vhodné pro dokumenty formátu XML a JSON (BSON). Databáze dokáže interpretovat strukturovanou hodnotu (dokument), toho lze efektivně využít hlavně při dotazování. Dotazy mohou být i složitější než přes klíče, např. pomocí `XPath`. Mezi zástupce patří `MongoDB` a `CouchDB`.

- Grafové (angl. Graph databases) – Jsou navrženy pro ukládání entit (uzly) a jejich vztahů (hrany). Uzly i hrany mohou mít svoje atributy. Jsou vhodné pro reprezentaci sítí a jejich topologií, např. sociální či dopravní sítě, topologie počítačových sítí a další [25]. Zástupcem je `Neo4J`.
- Sloupcové (angl. Column-oriented databases / Column family stores) – Data jsou organizována v tabulkách. Tabulka má řádky jako v relační databázi, ale u řádku pak lze definovat sloupce s hodnotami. Sloupce mohou být pro každý řádek různé (flexibilní schéma), i různý počet (řídké pole). Sloupcové databáze jsou optimalizovány pro dotazování nad velkými objemy dat. Sloupce mohou být zobecněny na adresáře (anglicky `supercolumn`), kde potom řádek obsahuje kolekci supersloupců, a z nich každý obsahuje kolekci sloupců [25]. Zástupci sloupcových databází jsou například `Cassandra` a `HBase`.

3.3.3 Srovnání relačních a NoSQL databází

Podle výše uvedených vlastností úložišť pro strukturovaná a nestrukturovaná data můžeme vidět tyto rozdíly:

Relační databáze

- Datový model je formalizovaný, databáze umožňuje definovat integritní omezení, kontroly, model reflektuje elementy reálného světa.
- S daty lze provádět transformace v podobě spojení, agregací, řazení apod.
- Kvůli podpoře transakcí a ACID vlastností není úplně snadná škálovatelnost.
- Mají pevné schéma databáze. Vzniklé problémy po úpravách se musí řešit např. migračními skripty.

NoSQL databáze

- Jedná se o nízkoúrovňové úložiště, kde ve většině případů databáze není schopna data interpretovat. Zajištění konzistence je ponecháno aplikaci.
- Data lze dostat pouze v podobě, v jaké byla uložena.
- Jsou navrženy pro jednoduchou škálovatelnost.
- Mají dynamické schéma, které neomezuje definovat libovolnou strukturu a provádět flexibilní úpravy.

Kapitola 4

Návrh distribuovaného úložiště

Tato kapitola se zabývá návrhem distribuovaného úložiště rozsáhlých digitálních forenzních dat. Bude popsána komunikace se systémem včetně aplikačního rozhraní, použité technologie, zvolené druhy úložišť a také zpracování požadavků.

4.1 Požadavky na systém

Nejdůležitějším požadavkem je, aby systém byl distribuovaný, tzn. škálovatelný na více výpočetních uzlech. S tím souvisí také výběr vhodných technologií.

Systém musí umožnit optimální přístup k různým datům, jiný přístup bude pro strukturovaná data a jiný pro nestrukturovaná. Z forenzních digitálních dat proběhne zaměření převážně na `PCAP` soubory.

Posledním klíčovým požadavkem je umožnit přidávání podpory pro nové druhy digitálních forenzních dat přímo za běhu systému.

Ze sekce 3.3 vyplývá, že je vhodné využít NoSQL distribuovaných databází a HDFS jako úložiště. Pro komunikaci s klientem bude sloužit Kafka (společně se ZooKeeper). Všechny tyto technologie jsou distribuované a ověřené pro použití v Big data prostředích. Jak jednotlivé technologie fungují bude vysvětleno v této kapitole.

4.2 Aplikační rozhraní

Aplikační rozhraní umožňuje klientovi komunikovat s distribuovaným repositářem. Komunikace je založena na asynchronním zasílání zpráv. Systém repositáře nenabízí klientovi přímo žádné metody či funkce, které by klient mohl volat. Ovládání repositáře probíhá pomocí zaslání zprávy obsahující tzv. příkaz. Příkaz specifikuje typ operace a typ dat. Podle typu příkazu je určeno, jak se daný příkaz zpracuje.

V následující sekci bude vysvětleno, jak ovládání repositáře probíhá, jak vypadá zpráva, a bude také uvedeno schéma celé komunikace jako příklad.

4.2.1 Komunikace

Komunikace s klientem, který chce do distribuovaného repositáře data uložit, nebo naopak z něj nějaká data získat, probíhá pomocí zaslání zprávy tzv. `MQ brokeru` Kafka. Systém bude pracovat na principu požadavek – odpověď, ale asynchronním způsobem. Zpráva požadavku obsahuje parametry – atributy příkazu a případně řetězec dat.

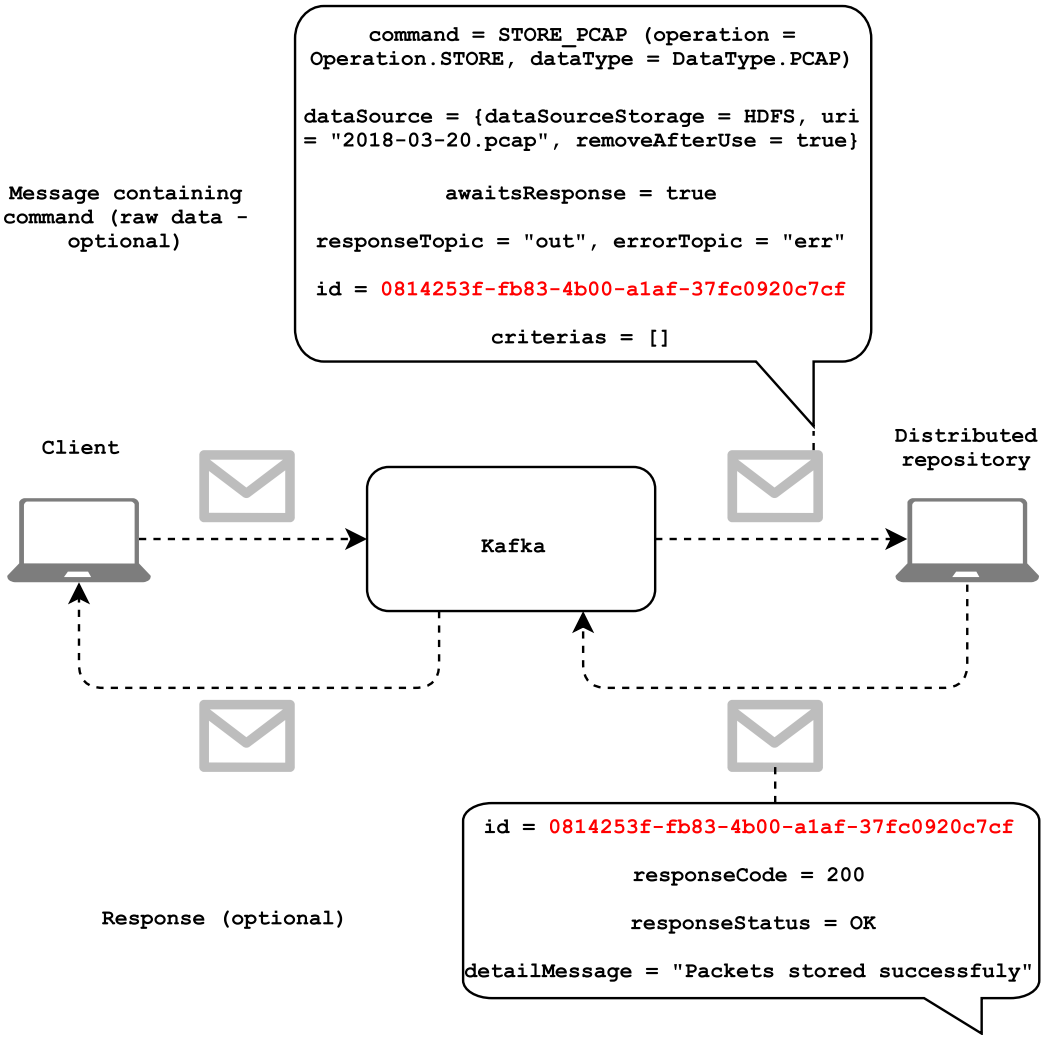
Název a význam parametrů:

- `command` – Určuje typ příkazu, příklady příkazů mohou být `STORE_PCAP` a `LOAD_PCAP`. Příkaz v sobě ještě zapouzdřuje typ operace, které mohou být `SAVE` a `LOAD`, a také typ dat jako například `PCAP`, `Packet`, `BINARY`, `LOG` apod.
- `id` – Povinný parametr, udává unikátní ID zprávy, aby klient potom dokázal identifikovat už zpracované příkazy.
- `awaitsResponse` – Dvoustavová hodnota, jestli klient/odesilatel zprávy očekává od repositáře odpověď. Pokud je tento parametr zadán jako `True`, je nutné zadat také další parametr `responseTopic`.
- `responseTopic` – Ve které frontě (angl. topic) je odpověď na straně klienta očekávána.
- `errorTopic` – Na straně distribuovaného úložiště se může vyskytnout chyba, například nedostupnost databáze, chyba při zpracování příkazu a další. Tento parametr slouží pro zadání fronty, kam se budou zasílat chybová hlášení. Jedná se o povinný parametr.
- `dataSource` – Zdrojová data k uložení lze zaslat dvojím způsobem. První z nich je poslat data v binární podobě přímo přes systém Kafka společně s příkazem. Tento způsob lze využít pro data, jejichž velikost není příliš velká, protože se data načítají do paměti RAM. Pro velké objemy dat takový přístup není vhodný. Proto existuje ještě druhý způsob, a to předat data přes distribuovaný souborový systém HDFS. Parametr dataSource obsahuje název úložiště v podobě výčtové konstanty `Kafka` nebo `HDFS`, cestu k souboru, pokud se jedná o HDFS, a také atribut `removeAfterUse` pro odstranění souboru po použití. Lze využít i pro příkazy čtení dat ke způsobu předání výsledku – přes HDFS nebo Kafku.
- `criterias` – Představuje seznam kritérií pro dotazování. Tento parametr je vhodné vyplnit pouze při zasílání příkazu čtení.

Řetězec dat obsahuje vlastní data, která mají být na straně repositáře zpracována. Typy příkazů, a v podstatě typy operací s typy dat lze libovolně přidávat.

Zpráva odpovědi nese tyto parametry:

- `id` – Jedná se o unikátní ID zkopírované ze zprávy požadavku. Klient po přijetí odpovědi bude vědět, ke kterému požadavku odpověď patří.
- `responseTopic` – Název výstupní fronty, do které je odpověď zaslána.
- `responseCode` – Návratový kód odpovědi symbolizující jak operace dopadla. Analogie s HTTP návratovými kódy, možnými hodnotami jsou: `OK(200)`, `BAD_REQUEST(400)`, `UNSUPPORTED_MEDIA_TYPE(415)`, `INTERNAL_SERVER_ERROR(500)`.
- `status` – Rezervovaný parametr pro status odpovědi.
- `detailMessage` – Pokud se objeví při zpracování požadavku chyba, může být v odpovědi odesláno, proč chyba nastala, nebo její přičina.

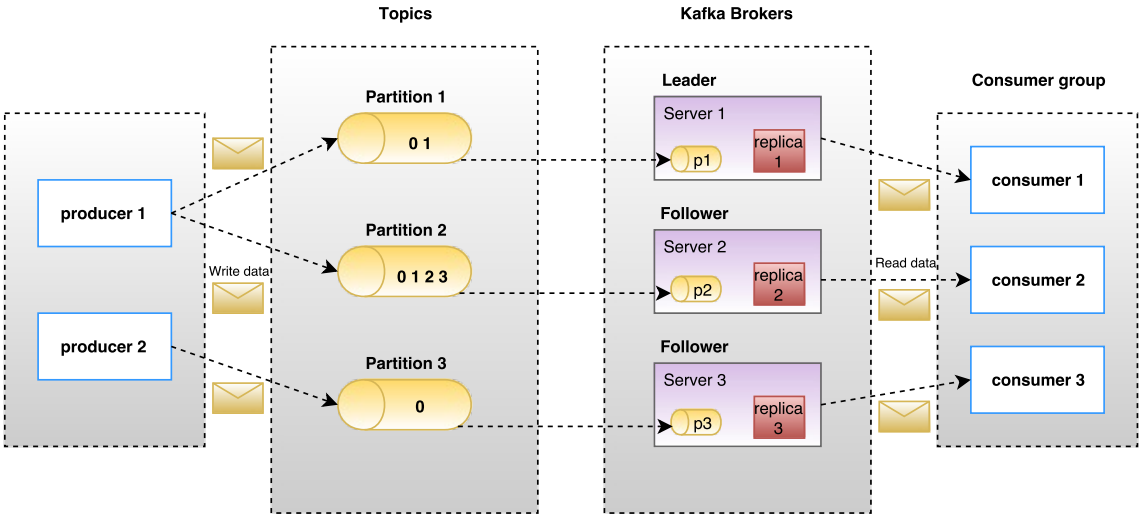


Obrázek 4.1: Demonstrace komunikace – zaslání zprávy požadavku pro uložení souboru PCAP. Klient zašle příkaz přes Kafku, příkaz je zpracován na straně distr. repositáře, a případně je odeslána odpověď klientovi. Můžeme si všimnout uvedených ID, jsou stejné – klient zjistí, ke kterému příkazu odpověď patří.

4.2.2 Kafka

Apache Kafka je distribuovaný systém zpráv s robustními frontami založený na mechanismu `publish-subscribe` umožňující přenášet vysoké objemy dat. Kafka zprávy jsou perzistentně uloženy na disku a replikovány v rámci clusteru kvůli prevenci ztráty dat. Systém Kafka je postaven na synchronizační službě ZooKeeper [29]. Výhody jsou:

- Spolehlivost – Kafka je distribuovaný systém, segmentovaný, replikovaný a odolný proti chybám.
- Škálovatelnost – Je možné připojit nové uzly do clusteru.
- Odolnost – Zprávy jsou uloženy na disku.
- Výkon – Vysoká propustnost pro obě akce `publish` a `subscribe`, je zachován stabilní výkon i při objemu zpráv v terabytech.



Obrázek 4.2: Diagram zobrazující klíčové komponenty systému Kafka [29].

V diagramu 4.2 je fronta (angl. `topic`) konfigurována do tří oddílů (angl. `partition`). Pokud je faktor replikace fronty nastaven na hodnotu 3, Kafka vytvoří tři identické repliky každého oddílu a umístí je do clusteru. Každý prostředník (angl. `broker`) ukládá jeden nebo více těchto oddílů za účelem vyvažování zátěže. Více producentů, respektive konzumentů, může zároveň vydávat (angl. `publish`), resp. odebírat, (angl. `subscribe`) zprávy [29].

Hlavní komponenty systému Kafka z diagramu 4.2:

- Fronta – Jedná se o proud zpráv patřící k příslušné kategorii. Data jsou uložena ve frontách. Fronty jsou rozděleny do oddílů.
- Oddíl – Fronty mohou mít mnoho oddílů, tak aby zvládaly zpracování libovolného množství dat.
- Offset oddílu (angl. `Partition offset`) – Každá zpráva v oddílu má unikátní sekvenci ID nazývanou offset.
- Replika oddílu – Je pouhou zálohou oddílu. Repliky nejsou využívány ke čtení nebo zápisu dat, slouží pouze jako prevence před ztrátou dat.
- Prostředník – Jednoduchý systém zodpovědný za správu dat v oddílech, resp. frontách. Prostředníci slouží k balancování zátěže.
- Kafka Cluster – Pokud existuje víc než jeden prostředník, pak se systém nazývá cluster. Cluster může být rozšířen bez prodlev o další prostředníky.
- Producent – Je odesílatel zprávy do jedné nebo více Kafka front. Producenti posílají data prostředníkům. Pokaždé když producent pošle zprávu prostředníkovi, prostředník přidá zprávu na konec oddílu. Producent nečeká na žádná potvrzení, posílá zprávy tak rychle, jak jen prostředník dokáže přijímat.
- Konzument – Čte data od prostředníka(ů). Odebírá z jedné nebo více front a konzumuje zprávy vytažením dat od prostředníků.

4.3 Úložiště

Tato sekce se zaměřuje na úložiště, kde budou digitální forenzní data uchována. Můžeme je rozdělit na strukturovaná a nestrukturovaná.

4.3.1 Strukturovaná data

V souvislosti se systémem repositáře se obecně jedná o data, která mohou být rozdělena na menší části při zachování možnosti interpretace, a mohou být efektivně serializována pro uložení. Mezi strukturovaná data můžeme zařadit například soubory formátu PCAP obsahující síťovou komunikaci. Tyto soubory lze rozparsovat na jednotlivé segmenty pakety. Pakety lze potom serializovat na pole bytů a uložit do NoSQL databáze. Pro strukturovaná data byla vybrána databáze Cassandra.

Cassandra

Jedná se o zástupce sloupcových NoSQL databází. Výhodami jsou předně: škálovatelnost, odolnost proti chybám, rychlost, distribuovanost, konzistence a podpora transakcí. Cassandru využívají i světoznámé společnosti Facebook, Twitter, Cisco, ebay, Twitter, Netflix a další. Cílem Cassandry je zvládat vysoké objemy dat napříč mnoha uzly. Data jsou pak distribuována na všech uzlech clusteru. Uzly lze libovolně přidávat. Každý uzel je nezávislý a zároveň propojený s ostatními. Každý uzel v clusteru se může podílet na požadavcích čtení či zápisu nezávisle na tom, kde jsou data skutečně uložena. Když uzel selže, požadavky čtení a zápisu mohou být obslouženy jinými uzly sítě [30].

4.3.2 Nestrukturovaná data

Pro uchování nestrukturovaných dat typu například audio, video, logů, a obecně binárních dat bude sloužit distribuovaný souborový systém HDFS. Taková data mohou mít velký objem, a bylo by plýtvání výkonem provádět jejich serializaci nebo rozdělování na části při ukládání do NoSQL databází.

Hadoop a HDFS

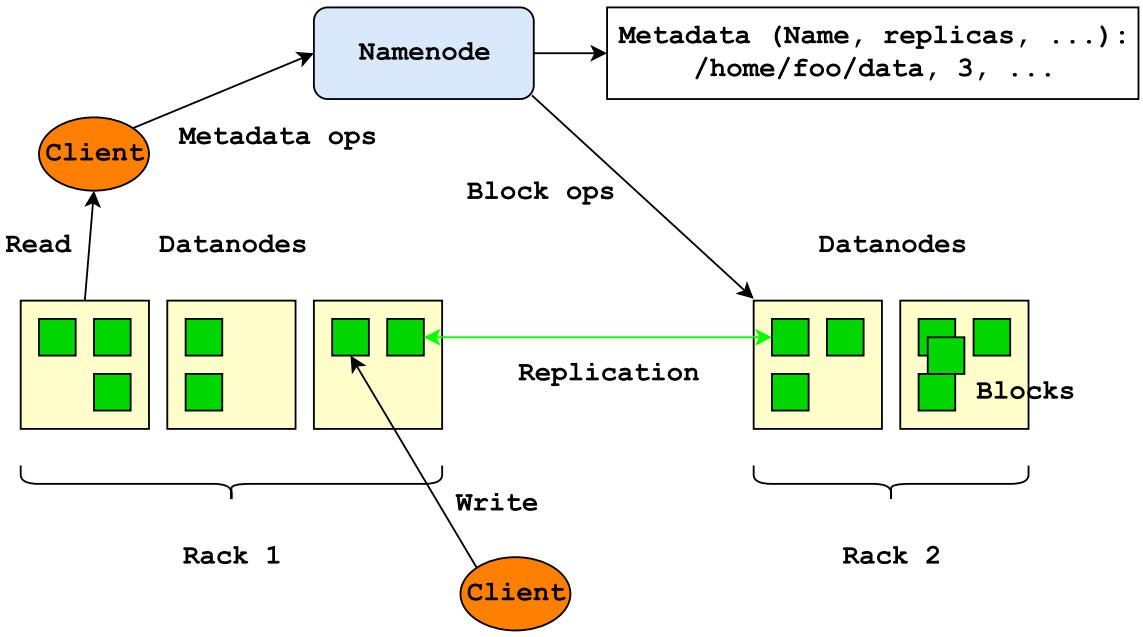
Distribuovaný souborový systém HDFS patří pod projekt `Apache Hadoop`, který je navržen pro ukládání a zpracování obrovského množství dat v distribuovaném prostřední napříč mnoha počítači. Počítá se škálovatelností od jednoho serveru po tisíce strojů, kde každý z nich nabízí lokální úložiště a výpočetní zdroje [32].

Platforma Hadoop zahrnuje tyto moduly:

- `Hadoop Common` – Knihovny a pomocné nástroje požadované ostatními Hadoop moduly. Poskytují abstrakce souborového a operačního systému potřebné pro jazyk Java, a skripty nutné pro spuštění Hadoop-u.
- `Hadoop YARN` – Jedná se o framework pro plánování distribuovaných úloh a správu zdrojů.
- `HDFS` – Distribuovaný souborový systém poskytující vysokou propustnost přístupu k aplikačním datům, škálovatelnost, odolnost proti chybám a efektivní správu zdrojů.

- `Hadoop MapReduce` – Systém pro paralelní zpracování obrovského množství vstupních dat. Tento modul je pro systém distribuovaného úložiště zbytečný.

Kromě výše uvedených modulů existují i další pomocné nástroje: `Apache Spark`, `Apache HBase` – distribuovaná databáze, `Apache Hive` – nástroj pro dolování dat nad platformou Hadoop, `Apache Pig` atd.



Obrázek 4.3: Architektura HDFS [9] – datové uzly jsou uspořádány do `rack`-ů, které se nachází v jedné lokalitě, což zajišťuje rychlejší propojení uzlů. Bloky jsou replikovány mezi odlišné uzly.

HDFS je virtuální souborový systém vybudovaný nad běžnými souborovými systémy jednotlivých uzlů. Řeší problém nalezení úložiště a přístupu k datům, nikoliv fyzické uložení v uzlu. Byl navržen pro sekvenční přístup k souborům, nikoliv náhodný [26].
HDFS používá architekturu typu `master/slave`. Master se skládá z jednoho uzlu nazývaného `NameNode`, který se stará o správu metadat a jmenného prostoru souborového systému. Dále reguluje přístup k souborům a provádí operace typu otevření, zavření, přejmenování souborů a adresářů. Protože je NameNode pouze jeden, měl by být spolehlivý a výkonný, při jeho výpadku způsobí poruchu systému (jde o tzv. `single point of failure`). Uzlů typu slave může být více, jsou nazývány datové uzly (angl. `DataNodes`), a ukládají data. Soubor je rozdělen do několika bloků (typicky po 64 nebo 128 MB), a každý blok souboru je nezávisle replikován v několika datových uzlech slave. Bloky jsou uloženy v lokálním souborovém systému na datových uzlech. Datové uzly jsou také zodpovědné za zpracování požadavků čtení a zápisu od klientů. NameNode se stará o mapování bloků datovým uzlům, sleduje počet replik jednotlivých bloků. Pokud se nějaká replika ztratí vinou selhání datového uzlu, je vytvořena nová replika bloku [2].

HDFS je implementován v jazyce Java, na každém stroji s nainstalovanou Javou může běžet NameNode nebo DataNode software. Typicky při nasazení existuje jeden stroj s běžícím NameNode, na dalších strojích potom běží datové uzly (lze ovšem najít i scénáře, kde na jednom stroji běží více datových uzlů). Důležité je, že uživatelská data nikdy neprochází přes NameNode [9].

4.3.3 Metadata

V rámci zpracování příchozí zprávy do repositáře informující o operaci uložení, by bylo vhodné provést předzpracování dat za účelem vhodnější indexace a rychlejšího nalezení pro budoucí operace čtení. Jednalo by se tak o mechanismus podobný například pamětem cache. Uvažme následující scénář pro podrobnější vysvětlení.

Do repositáře bylo v průběhu několika dnů uloženo mnoho milionů paketů, které obsahovaly síťovou komunikaci mezi několika komunikujícími stranami. Data paketů jsou uložena v serializované podobě v NoSQL databázi. Na úrovni databáze nedávají taková data žádný význam, pouze aplikace je dokáže správně interpretovat jako pakety. Po několika dnech by uživatel rád zjistil podrobnější informace o komunikaci mezi uzly A a B. Nezbývalo by mu nic jiného, než všechna data z NoSQL databáze načíst na aplikační úrovni a analyzovat paket po paketu, aby zjistil minimálně IP adresy zdroje a cíle. Takový způsob by byl velmi pomalý a neefektivní.

Následně uvažme, jak by vypadal postup pro zjištění informací o komunikaci mezi uzly A a B, kdyby byla k dispozici cache, či paměť metadat. Cache by mohla uchovávat základní informace o dříve uložených paketech v jednotně definované struktuře. Mezi tyto informace by patřily – zdrojová a cílová IP adresa, zdrojová a cílová MAC adresa, typ protokolu, časové razítko a další. Co je však důležité, u těchto informací by bylo uvedeno ID záznamu z NoSQL databáze, kde je uchován celý paket. Pro zjištění podrobnějších informací by uživatel mohl načíst pouze ty pakety, které jsou pro něj relevantní.

Předzpracování dat je výhodné před jejich uložením, protože v kontextu provádění aplikace jsou data interpretovatelná. Výtah základních informací z nich nezpůsobí propad výkonu. Naopak se výkon zvýší pro případné operace čtení a hledání.

Výše uvedený princip by se dal zobecnit pro libovolný typ forenzních digitálních dat. Tak by došlo k vytvoření tzv. registru, který by i mimo jiné informoval, co je jak a kde uloženo. Vhodnou strukturou pro uložení těchto metadat může být například formát XML nebo JSON. Z toho vyplývá použití dokumentové NoSQL databáze jako například MongoDB.

MongoDB

MongoDB je multiplatformní, dokumentově orientovaná databáze poskytující vysoký výkon a dostupnost a jednoduchou škálovatelnost. Pracuje na principu kolekcí, které nevyžadují schéma. Kolekce je skupina MongoDB dokumentů, a je ekvivalentem SŘBD tabulek. Dokumenty v kolekcích mohou mít dynamická schémata a rozdílné atributy. Dynamické schéma znamená, že dokumenty ve stejné kolekci nemusí mít stejnou strukturu atributů. Počet atributů, obsah a velikost dokumentu se může mezi dokumenty lišit. Dokument je množina párů klíč-hodnota [33].

Relační databáze má určité schéma skládající se z tabulek a vztahů mezi nimi. Koncept vztahu je v MongoDB řešen jinak. Existují dva mechanismy umožňující aplikaci reprezentovat vztahy – reference a vnořené dokumenty. Reference uchovávají vztahy mezi daty vložením odkazů (referencí) z jednoho dokumentu do druhého. Aplikace může díky odkazu přistoupit k příslušným datům. Vnořené dokumenty zachycují vztahy mezi daty ukládáním příslušných dat přímo do jednoho dokumentu. Namísto atributu dokumentu může být uložen i jiný dokument. Výhoda tohoto modelu je, že aplikace získá všechna data v jedné databázové operaci [4].

Dynamičnost MongoDB je vhodná pro správu metadat, zvlášť když bude do repositáře potřeba přidávat nové druhy dat.

4.4 Jádro distribuovaného úložiště

V předchozích sekcích byla představena komunikace s repositářem, jak vypadají příkazy a jakým způsobem jsou posílána data. Následně byly popsány možnosti ukládání. Tato sekce se zaměřuje na jádro distribuovaného repositáře, převážně na zpracování příkazů a manipulaci s databázemi.

Přijetí příkazu zajišťuje konzument zpráv. Konzument se v krátkých časových intervalech dotazuje na zprávy z Kafka fronty. Po přijetí zprávy dojde k přečtení příkazu určeného typem operace a typem dat společně s dodatečnými parametry uvedenými v 4.2.1. Na základě příkazu dojde k výběru konkrétní obsluhy (tzv. `handler`-u), která má za úkol zpracování příkazu. Může se jednat o uložení dat, o načtení dat podle zadaných parametrů, přečtení metadat apod. Handler ukrývá všechny potřebné operace, které mají proběhnout při zpracování příkazu. Mimo jiné se jedná i o správu metadat zmíněnou v sekci 4.3.3. Při implementaci budou objektu obsluhy nastaveny potřebné závislosti pro řešení databázových operací pro daný typ dat. Každý typ úložiště (souborový systém nebo NoSQL databáze) může mít jiné rozhraní přístupu.

Důležitým požadavkem je, aby úložiště umožňovalo přidávání podpory pro nové druhy forenzních dat za běhu. Tento požadavek je implicitně zajištěn systémem Kafka, který každou přijatou zprávu zapíše na disk a zpráva tam zůstává, dokud ji konzument úspěšně nezpracuje. V tom případě lze systém repositáře zastavit, ale Kafka zprávy bude pořád přijímat. Mezitím může proběhnout do-implementování nové funkcionality pro jiný druh forenzních dat, provést kompilaci, nasazení a spuštění systému. Po spuštění se konzument začne dotazovat Kafky na nové nepřijaté zprávy.

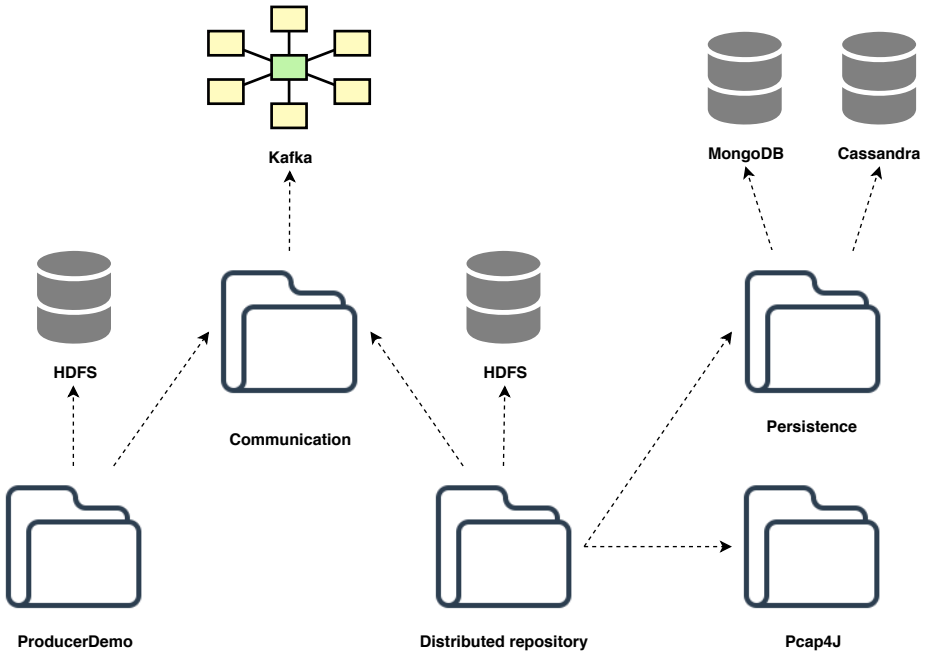
Kapitola 5

Implementace

5.1 Dekompozice do modulů

Repositář je komplexní distribuovaný systém, který je postaven na frameworku Spring, sestávající z několika modulů a knihoven. V této sekci je uvedeno, jak byl systém dekomponován do modulů a jaké knihovny byly zvoleny. Pro správu závislostí a sestavení aplikace byl zvolen nástroj `Maven`. Pro přehled byly využity tyto knihovny a frameworky – `Spring Boot`, `Spring Data`, `Spring Kafka`, `Spring Hadoop`, `Pcap4J`, a samozřejmě ovladače pro jednotlivé databáze.

Jedná se o dvě Spring Boot aplikace – `DistributedRepository` a `ProducerDemo`, které komunikují pomocí komunikačního rozhraní implementovaného v modulu `Communication`. Přenos zpráv zajišťuje projekt Spring Kafka. Aplikace DistributedRepository má přístup k oběma databázím Cassandra a MongoDB díky modulu `Persistence` obsahujícím mimo jiné projekt Spring Data, a také k HDFS pomocí projektu Spring Hadoop. K HDFS má přístup i klientská aplikace ProducerDemo.



Obrázek 5.1: Schéma klíčových modulů a závislostí systému.

5.1.1 Pcap4J

Jedná se o knihovnu pro jazyk Java pro zachytávání, konstruování a odesílání paketů. Knihovna je stále ve vývoji. Pcap4J pracuje nad nativní knihovnou (`libpcap`, `WinPcap`, nebo `Npcap` v závislosti na operačním systému) přes JNA (`Java Native Access`) [1] a poskytuje aplikační rozhraní pro jazyk Java. Mimo výše uvedené činnosti dokáže pracovat s PCAP soubory, vytvářet a parsovat je na jednotlivé pakety. Každý paket implementuje rozhraní `Packet`. Toto rozhraní nabízí mimo jiné dvě klíčové metody `contains` a `get`. Metoda `contains` slouží ke kontrole, jestli je paket konkrétního typu, který chceme získat. Metoda má jako parametr třídu, které je paket typem, hlavička metody potom vypadá:

```
boolean contains(Class<T> clazz)
```

Ke konkrétnímu typu paketu, např. IP paketu, se přistupuje pomocí reflexe voláním metody:

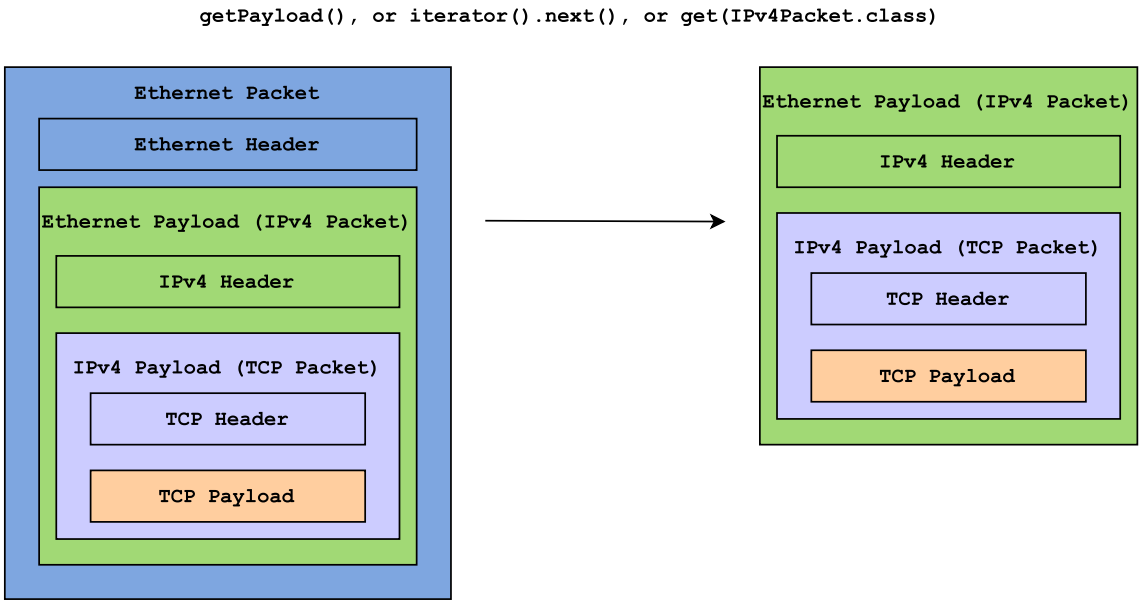
```
T get(Class<T> clazz)
```

Tedy například:

```
IPv4Packet ipv4Packet = ethernetPacket.get(IPv4Packet.class);
```

Návratová hodnota metody `get` je objekt třídy uvedené v parametru. Před voláním `get` je vhodné zkontrolovat typ pomocí metody `contains`. Z konkrétního paketu už lze získávat potřebné informace, v případě IP paketu údaje z hlavičky jako zdrojová a cílová IP adresa, verze protokolu apod. Analogicky lze postupovat i v případě ostatním typů paketů, Z TCP paketu lze získat údaje o portech, sekvenční čísla, kontrolní součet a další.

Knihovna reflektuje zapouzdření, které spočívá ve vložení protokolové datové jednotky (anglicky `Protocol Data Unit`) vyšší vrstvy do protokolové jednotky nižší vrstvy. Takže ethernetový paket může být současně i IP paketem a podobně.



Obrázek 5.2: Schéma znázorňující výše uvedený příklad pro manipulaci s pakety [35].

[1] https://github.com/java-native-access/jna

5.1.2 Spring, Spring Boot a Spring projekty

Spring je velmi populární framework pro jazyk Java umožňující vytvářet webové a enterprise aplikace. Věnuje se mnoha obecným principům a problémům jako jsou například `dependency injection`, konfigurace, aspektově orientované programování, ORM, validace, bezpečnost, testování, integrace s jinými frameworky atd. V současné době pod něj spadají desítky projektů, každý zaměřený na jiný aspekt [2].

Spring poskytuje tři způsoby konfigurace – pomocí XML, anotací a konfigurace přímo v Java. S rozšiřující se funkcionalitou se zvyšuje komplexita a i konfigurace se stává obtížná a náchylná k chybám. Z důvodu lepšího způsobu konfigurace byl vyvinut Spring Boot. Spring Boot přichází s principem auto-konfigurace, ponechává však možnost předefinovat výchozí nastavení. Klíčovými vlastnostmi jsou [18]:

- Jednoduchá správa závislostí – Pokud chceme použít Spring Boot k běhu aplikace, je potřeba importovat `spring-boot-starter-parent` jako modulovou závislost. Existují další `Spring Boot Starter` závislosti, které se hodí pro určitý typ nebo aspekt vyvíjené aplikace. Existují např. `Test Starter`, `Web Starter`, `Security Starter`, `Data JPA Starter`, `AOP Starter` atd. Pokud chceme vyvíjet např. webovou aplikaci, Web Starter poskytne všechny potřebné závislosti zahrnující MVC modul, validační API, prostředky k serializaci dat atp. Webová aplikace typicky potřebuje pracovat s databází – Data JPA Starter poskytne všechny potřebné závislosti zahrnující transakční API, `Hibernate` knihovny, ORM implementaci atd.

- Auto-konfigurace – Nejenže Starter Web poskytne potřebné závislosti, ale proběhne konfigurace běžně používaných objektů tříd `ResourceHandlers`, `MessageSource` atd. výchozími hodnotami. Analogicky proběhne konfigurace objektů nutných k používání JPA – `DataSource`, `TransactionManager` a `EntityManagerFactory`. Uživatel tyto objekty sám nevytváří, o jejich vytvoření se postará Spring Boot na základě poskytnutých údajů v konfiguračním souboru `application.properties` (příklad konfigurace aplikace uveden v příloze A). Spring Boot tedy podle zdrojů uvedených v `classpath` provádí konfiguraci celé aplikace.

- Podpora zabudovaného kontejneru – Pokud vyvíjíme webovou aplikaci, která bude běžet v nějakém kontejneru typu `Tomcat`, není potřeba provádět žádná nasazení do externího kontejneru. Kontejner je při kompilaci automaticky stažen spolu s ostatními závislostmi a je zabudovaný, takže aplikaci stačí pouze spustit a Spring Boot se postará o nasazení do zabudovaného kontejneru. Samozřejmě lze zvolit i jiný typ kontejneru, např. `Jetty` apod.

Spring Boot umožňuje vytvářet tzv. `beans` přímo v Java kódu, bez použití XML konfiguračních souborů. Každou bean lze vytvořit pomocí anotace `@Bean`, např.

```
@Configuration
public class ParserBeans {
    @Bean
    public PcapParser<PcapPacket> pcapParser() {
        return new ParserImpl();
    }
}
```

[2] https://spring.io/docs/reference

Použití vytvořené bean je velmi jednoduché pomocí anotace `@Autowired`:

```
@Autowired
private PcapParser<PcapPacket> pcapParser;
```

Uživatel tedy definoval bean v konfigurační třídě označené anotací `@Configuration`, a tato instance je pak dodána principem dependency injection.

Spring Kafka

Projekt Spring Kafka aplikuje klíčové koncepty Springu pro vývoj systémů založených na platformě Kafka. Poskytuje šablonu jako vysokoúrovňovou abstrakci pro zasílání zpráv [24]. Velkým přínosem je výše zmíněná auto-konfigurace ze zadaných parametrů, která umožňuje okamžité použití komponent bez vytváření instancí uživatelem.

Mezi nejdůležitější třídy patří `KafkaTemplate`, která zapouzdřuje mnoho metod pro odeslání zpráv do Kafka front. Podporuje asynchronní i synchronní odesílání. Použití této šablony lze vidět ve třídě `ResponseProducer` z diagramu tříd 5.5.

Zprávy mohou být přijímány pomocí zvoleného kontejneru `MessageListenerContainer`, který poskytuje implementaci tzv. `Message Listener` nebo je označen pomocí anotace `@KafkaListener`. Existují 2 implementace kontejneru – `KafkaMessageListenerContainer` (aktuálně použitá implementace v systému repositáře), kde jsou přijímány všechny zprávy ze všech front a oddílů v jednom vláknu; a `ConcurrentMessageListenerContainer`, který deleguje zpracování 1 nebo více kontejnerům, což poskytuje vícevláknový odběr zpráv a tudíž i vícevláknové zpracování [24], více v [3]. Lze také nastavit různé filtry a vzorce pro přijímané zprávy, a rozdělit tak příjem zpráv podle jejich struktury jiným objektům.

Spring Hadoop

Další z projektů, které patří pod Spring, je Spring Hadoop, zjednodušující použití Hadoop API pro Javu. Přináší zase výše zmíněný jednotný konfigurační model usnadňující vývoj systémů postavených na platformě Hadoop, využívající HDFS, paradigma MapReduce, Hive a Pig. Poskytuje také integraci s jinými projekty ze Spring ekosystému jako například `Spring Integration` a `Spring Batch`. Podporuje tvorbu Hadoop aplikací, které jsou konfigurovány principem dependency injection a spuštěny jako standardní Java aplikace nebo pomocí nástrojů Hadoop pro příkazovou řádku. Velmi jednoduše lze integrovat se Spring Boot pro zjednodušení připojení k HDFS za účelem manipulace s daty [15]. Jedná se o komplexní systém obsahující obrovské množství dalších nástrojů, podsystémů a nastavení, které ovšem překračují použití v distribuovaném úložišti, více lze nalézt na referenčních webových stránkách projektu Spring Hadoop [4].

Pro účely distribuovaného repositáře byla využita převážně možnost propojení s HDFS. Přistupovat k HDFS lze obecně několika způsoby – použít Java API [5] (nutno stále kontrolovat výjimky a ošetřovat chybové stavy), nebo souborový systém ovládat pomocí příkazové řádky nástrojem `hadoop` [6] mimo Java aplikaci. Oba způsoby nejsou uživatelsky přívětivé. Spring Hadoop propojuje tyto dva způsoby skrze intuitivní jednoduché Java API [19].

HDFS nabízí tři API, jejichž vlastnosti znázorňuje tabulka 5.1.

[3] https://docs.spring.io/spring-kafka/docs/2.1.6.BUILD-SNAPSHOT/reference/html/_reference.html
[4] https://docs.spring.io/spring-hadoop/docs/2.5.1.BUILD-SNAPSHOT/reference/html/
[5] https://hadoop.apache.org/docs/r2.7.3/api/org/apache/hadoop/fs/FileSystem.html
[6] https://hadoop.apache.org/docs/r2.7.3/hadoop-project-dist/hadoop-common/FileSystemShell.html

Soubor. systém	Metoda	Schéma/Prefix	Zápis/Čtení	Verze
HDFS	RPC	`hdfs://`	Zápis i čtení	Pouze stejná verze
HFTP	HTTP	`hftp://`	Pouze čtení	Nezávislé na verzi
WebHDFS	HTTP (REST)	`webhdfs://`	Zápis i čtení	Nezávislé na verzi

Tabulka 5.1: Druhy API, které HDFS poskytuje, převzato z referenční dokumentace [19].

Pro účely distribuovaného úložiště bylo zvoleno první API, tzv. HDFS, založené na RPC volání – tento způsob vyžaduje, aby klient i Hadoop cluster běželi na stejné verzi (rozdílné verze způsobí problémy v serializaci). Další alternativou by bylo API WebHDFS, které ovšem vyžaduje další konfiguraci, a přenos pomocí HTTP přináší režii navíc ve formě zpomalení ukládání a čtení dat. API HFTP v tomto případě využít nelze, protože podporuje pouze čtení.

Praktickým nástrojem poskytovaným distribucí Hadoop je klient příkazové řádky, přes kterého lze spouštět příkazy podobné těm linuxovým přímo nad HDFS – existenci, mazaní, kopírování, přesouvání souborů, nebo nastavení přístupových práv. Tento nástroj je dostupný pouze přes příkazovou řádku, což stěžuje použití přímo z Java aplikace. Spring Hadoop poskytuje plně zabudovanou funkcionalitu výše zmíněného nástroje uvnitř třídy `FsShell`, která zrcadlí většinu příkazů dostupných z příkazové řádky (metody typu `chmod`, `cp`, `copyFromLocal`, `get`, `ls`, `mkdir`, `put`, `rm`, atd) [19]. Pomocí této třídy jsou prováděny veškeré operace s HDFS.

5.2 Úložiště

Jak už bylo uvedeno v kapitole 4, úložiště je tvořeno NoSQL databázemi Cassandra a MongoDB, a distribuovaným souborovým systémem HDFS. Zatímco předešlá kapitola jen nastínila vlastnosti těchto úložišť, zde budou vysvětleny technické detaily a práce s nimi.

5.2.1 Cassandra

Schéma databáze Cassandra je aktuálně velmi jednoduché. Nejzevnější úroveň tvoří tzv. `Keyspace` představující kontejner tabulek. Keyspace svými atributy udává replikační faktor (angl. `replication factor`), strategii umísťování replik (angl. `replica placement strategy`), a třídy sloupců (angl. `column families`). Aktuální nastavení keyspace vypadá následovně:

```
CREATE KEYSPACE structured_data WITH replication =
{ 'class':'SimpleStrategy', 'replication_factor':1 }
```

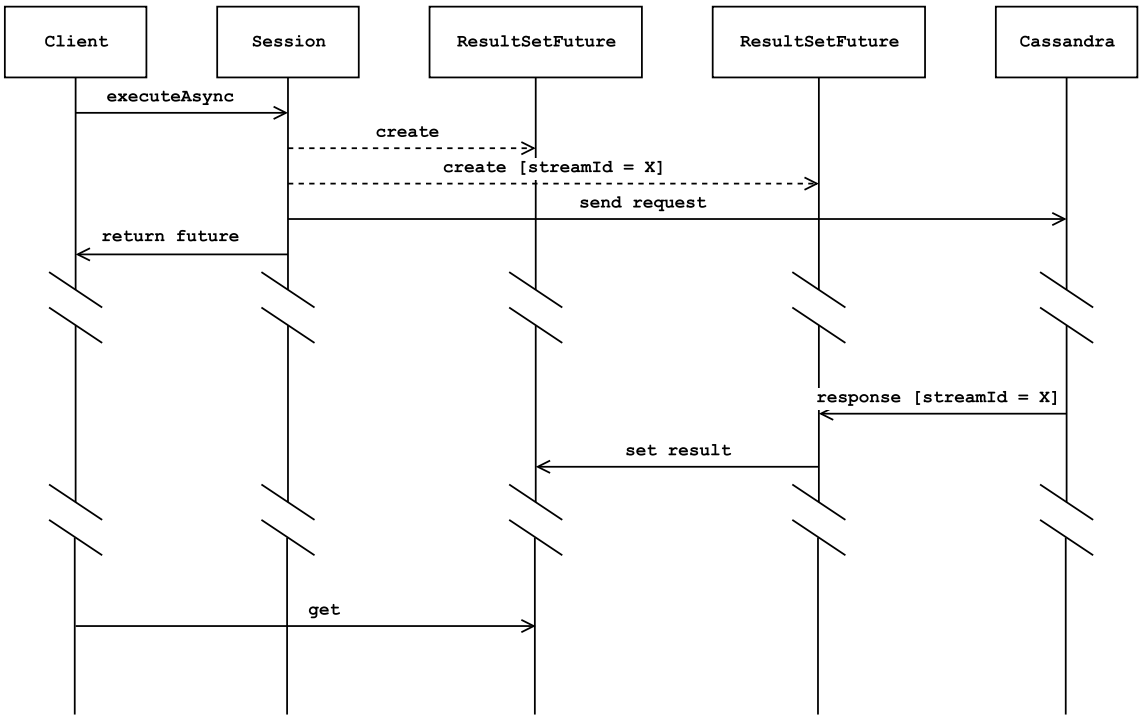
Je použita strategie `SimpleStrategy`, vhodná pouze pro jedno sdružení počítačových uzlů (`rack`). Není optimální pro současné použití mnoha datovými centry, k tomu slouží strategie `NetworkTopologyStrategy`. Replikační faktor vyjadřuje počet strojů v clusteru, které obdrží stejnou kopii dat. Zde nastaveno na hodnotu 1.

Schéma tabulky pro ukládání paketů je taktéž velmi jednoduché. Tabulka obsahuje pouze primární klíč ID a hodnotu paketu v binární podobě. Struktura tabulky:

```
packet ( id timeuuid PRIMARY KEY, packet blob )
```

Asynchronní dotazy

Distribuovaný repositář komunikuje s databází asynchronně z důvodu co největší propustnosti a rychlosti. Asynchronní komunikace je umožněna díky třídám a metodám ovladače pro jazyk Java od společnosti DataStax, který je stále ve vývoji [7]. Tento ovladač používá asynchronní architekturu. Takový způsob komunikace dovoluje klientské aplikaci ukládat data a provádět nad nimi dotazy neblokujícím způsobem, díky tzv. `Future` instancím [20].



Obrázek 5.3: Sekvenční diagram znázorňující jednotlivá volání metod při vykonávání asynchronního dotazu do databáze Cassandra [20].

Kromě zaslání dotazu do databáze, ovladač registruje interní obsluhu v podobě objektu `ResponseHandler`, který zpracuje odpověď dotazu, až bude k dispozici. Po zaregistrování obsluhy je předáno řízení vykonávání volajícímu programu společně s objektem třídy `ResultSetFuture`, pomocí kterého klient dokáže získat výsledek dotazu a dále s ním pracovat. Až databáze dotaz dokončí a vrátí odpověď, ovladač avizuje ResponseHandler (obecně může být zaregistrováno mnoho takových obsluh pro různé dotazy, párování je provedeno pomocí unikátního ID `streamId`, které bylo zasláno s dotazem). Obsluha třídy ResponseHandler dokončí volání upozorněním objektu třídy ResultSetFuture. Klientský kód získá výsledek dotazu provedením metody `get` nad objektem ResultSetFuture. Volání této metody je blokující, pokud ještě nebyl nastaven výsledek objektu ResultSetFuture [20].

Čekat na výsledek blokujícím způsobem není efektivní, proto existuje i způsob bez blokování. V takovém případě musí klient k objektu ResultSetFuture zaregistrovat svůj tzv. `callback`, který bude vykonán, až bude výsledek k dispozici. Lze zvolit i jiné vlákno pro jeho vykonání, aby aktuálně běžící kód nemusel být pozastaven. To lze například pomocí knihovny `Guava` od Google a jejich tříd `Futures` a `FutureCallback`.

[7] https://github.com/datastax/java-driver

Ovladač od DataStax je obecně celý asynchronní, uvnitř jeho synchronních metod je zavolána asynchronní verze, a pak okamžitě blokující metoda get. Ovladač umožňuje paralelně provádět pouze omezený počet dotazů. Tento počet je udán vnitřními parametry `Cluster` objektu (více detailních informací lze nalézt na referenčních www stránkách ovladače v sekci `Connection pooling` [8]):

- `maxRequestsPerConnection` – Udává maximální počet požadavků na jedno připojení.
- `maxConnectionsPerHost` – Udává maximální počet připojení na uzel.

Počet paralelních připojení do databáze Cassandra je v systému limitován semaforem. Jinak by vždy při vyčerpání dostupných připojení docházelo k selhání ve formě výjimky `NoHostAvailableException`. Při každém dotazu do databáze, čtení nebo zápis, tedy dojde k uzamčení semaforu, při dokončení operace (či při chybě) zase k odemknutí semaforu. Semafor byl nastaven na počet přístupů daných výrazem:

```
maxConnectionsPerHost * maxRequestsPerConnection
```

Tímto přístupem je tedy maximálně využito systémových prostředků a docíleno zrychlení provádění dotazů.

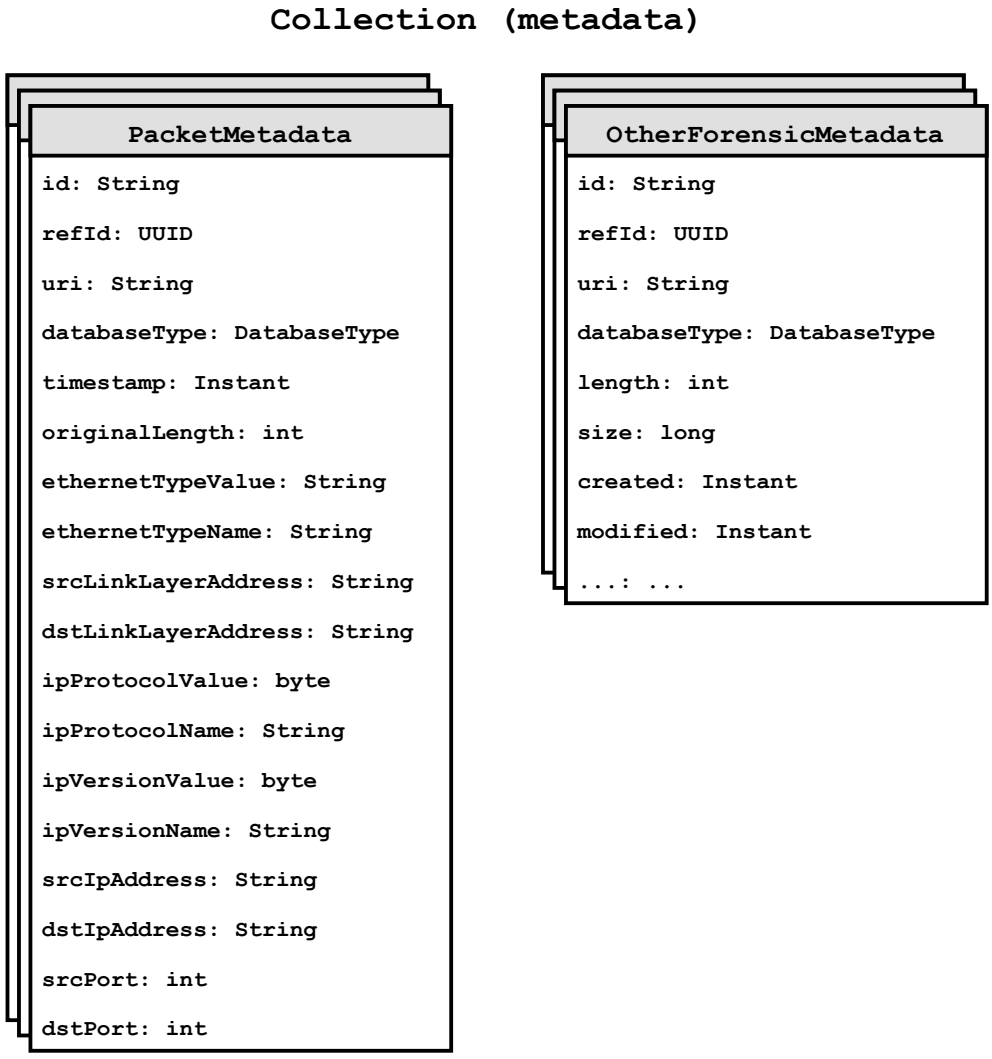
5.2.2 MongoDB

MongoDB slouží k uchování registru metadat. Důvody k zavedení podsystému metadat jsou uvedeny v 4.3.3. Druh metadat pro nějaký typ forenzních dat lze reprezentovat jako dokument ve formátu JSON. Dokumenty stejného typu jsou pak ukládány do kolekce. Pro každý typ ukládaných forenzních dat existuje takový dokument. Při ukládání dat do Cassandry nebo HDFS je dokument metadat sestaven a vyplněn klíčovými atributy, podle kterých bude možné později vyhledávat.

Pro pakety byl zvolen dokument typu `PacketMetadata` s těmito atributy:

- `id` – Unikátní ID v rámci MongoDB.
- `refId` – Unikátní ID pro záznamy v databázi Cassandra.
- `uri` – Pokud jsou korespondující forenzní data uložena v HDFS, je vyplněn tento atribut cestou k datům v HDFS.
- `databaseType` – Určuje typ úložiště, povolenými hodnotami jsou `Cassandra` a `HDFS`.
- `timestamp` – Časové razítko paketu.
- `originalLength` – Délka paketu v bytech.
- `ethernetTypeValue` – Hodnota z ethernetové hlavičky paketu, která udává typ protokolu v hexadecimálním formátu, např. 0x0800 pro IPv4, 0x86dd pro IPv6 atd.
- `ethernetTypeName` – Hodnota z ethernetové hlavičky paketu, která udává typ protokolu v řetězcovém formátu, např. IPv4, IPv6, ARP atd.

[8] https://docs.datastax.com/en/developer/java-driver/2.1/manual/pooling/



Obrázek 5.4: Schéma databáze pro metadata, obsahuje zatím pouze metadata pro pakety sdružované v kolekci dokumentů typu PacketMetadata. Kolekce dokumentů typu `OtherForensicMetadata` je zde uvedena jako příklad pro rozšíření do budoucna.

- `srcLinkLayerAddress` – Zdrojová fyzická adresa ve formátu `xx:xx:xx:xx:xx:xx`.
- `dstLinkLayerAddress` – Cílová fyzická adresa ve formátu `xx:xx:xx:xx:xx:xx`.
- `ipProtocolValue` – Hodnota z IP hlavičky paketu, která udává typ transportního protokolu v celočíselném formátu, např. 4 pro ICMPv4, 6 pro TCP, 17 pro UDP atd.
- `ipProtocolName` – Hodnota z IP hlavičky paketu, která udává typ transportního protokolu v řetězcovém formátu, např ICMPv4, IGMP, TCP, IGP atd.
- `ipVersionValue` – Hodnota z IP hlavičky paketu udávající číslo verze IP protokolu, 4 pro IPv4, 5 pro ST, 6 pro IPv6 apod.
- `ipVersionName` – Hodnota z IP hlavičky paketu udávající verzi IP protokolu v řetězcovém formátu, např. IPv4, ST, IPv6 atd.

- `srcIpAddress` – Zdrojová IP adresa v řetězcovém formátu, lze tedy uchovat IPv4 i IPv6 adresu. IPv6 adresa může mít více řetězcových interpretací, např. řetězce `ff02:0:0:0:0:0:0:c` a `ff02::c` vyjadřují tu samou IPv6 adresu. IP adresy jsou získány z paketu pomocí knihovny Pcap4J 5.1.1, která IP adresy vrací jako Java objekty typu `InetAddress`. Tyto objekty nelze serializovat do databáze, proto jsou ukládány jen řetězcové hodnoty. Nicméně třída InetAddress nabízí statickou metodu `InetAddress getByName(String host)`, kterou lze IP adresy normalizovat na stejný řetězcový tvar. Obě následující volání vrátí stejný řetězec:

  ```
  InetAddress.getByName("ff02:0:0:0:0:0:0:c").getHostAddress()
  ```

  ```
  InetAddress.getByName("ff02::c").getHostAddress()
  ```

- `dstIpAddress` – Cílová IP adresa ve stejném formátu jako zdrojová.
- `srcPort` – Zdrojový port.
- `dstPort` – Cílový port.

První čtyři atributy jsou společné pro všechny druhy metadat.

Reaktivní dotazy

Pro komunikaci s databází MongoDB existuje několik ovladačů, tzn. synchronní, asynchronní, a také ovladač založen na reaktivním paradigmatu. Poslední zmíněný byl využit pro distribuovaný repositář. Tento ovladač poskytuje asynchronní zpracování dotazů neblokujícím způsobem. Zcela implementuje aplikační rozhraní tzv. `Reactive Streams` [9]. Mezi silné stránky tohoto paradigmatu patří: funkcionální přístup, asynchronní zpracování chyb, jednoduchá multivláknovost [13]. Často je prezentováno jako rozšíření návrhových vzorů Pozorovatel (angl. `Observer`) a iterátor (angl. `Iterator`). Reaktivní paradigma se samozřejmě netýká jen databází, platí obecně a lze s ním vyvíjet celé aplikace.

Manipulaci s metadaty zajišťuje tzv. reaktivní JPA (`Java Persistence API`) patřící pod projekt Spring. Tato vrstva využívá výše zmíněného reaktivního ovladače pro MongoDB. Následuje detailnější pohled na tento programovací model a aplikační rozhraní [8].

Základem je vytvoření entitní třídy, která reprezentuje objekty ukládané do tabulky databáze nebo v tomto případě do kolekce. Pro entitní třídu lze definovat rozhraní, které představuje použití návrhového vzoru `Repository`. Výhodou je, že objekty nemají ponětí o tom, jakým způsobem jsou ukládány. O persistenci se postará Repository. Definované rozhraní představující Repository musí dědit rozhraní

```
interface ReactiveCrudRepository<T, ID>
```

se dvěma typovými parametry, kde typ `T` udává typ entitní třídy a typ `ID` udává typ unikátního ID pro záznamy dané entitní třídy. Toto rozhraní definuje doménově specifické `CRUD` metody s parametry reaktivních typů `Flux` a `Mono`, které budou vysvětleny dále. Příkladem pro manipulaci s metadaty je rozhraní:

```
interface PacketMetadataRepository extends
        ReactiveCrudRepository<PacketMetadata, String>
```

[9] http://www.reactive-streams.org/

Projekt Spring Data se postará o implementaci tohoto rozhraní pro všechny CRUD operace nabízené tímto rozhraním.

Reaktivní přístup byl zvolen hlavně z důvodu vrstvy JPA, kterou zajišťuje sám Spring. Přístup pomocí asynchronních dotazů by šel využít taktéž, ale manipulace s metadaty, ukládání a dotazování, by byla těžkopádná, potřebovala výrazně více režijního kódu, a byla by nepřehledná, protože asynchronní ovladač pracuje primárně na úrovně dokumentů vkládaných do kolekcí. S tím by souviselo manuální sestavování a parsování objektů reprezentujících dokument.

Reaktivní typy

Výchozími reaktivními typy jsou Flux a Mono pocházející z projektu `Project Reactor` (implementací reaktivních typů existuje více, další je např. `ReactiveX`). Flux slouží pro vztahy typu N, Mono pak pro vztahy typu 0 nebo 1 [12]. Reaktivní typy nejsou určeny k tomu, aby zpracovaly požadavky nebo data rychleji, ve skutečnosti představují malou režii ve srovnání s běžným blokujícím zpracováním. Jejich síla spočívá v obsluze více požadavků paralelně, a ve zpracování operací s latencemi, např. dotaz pro data z databáze nebo ze vzdáleného serveru, mnohem efektivněji. Poskytují lepší plánování zdrojů, zacházení s časem a latencemi. Na rozdíl od tradičního zpracování blokujícím způsobem, které pozastaví aktuální vlákno čekáním na výsledek operace, reaktivní aplikační rozhraní čekající na výsledek nestojí žádný čas, dotazuje se pouze na objem dat, který je schopné zpracovat. Zabývá se celými streamy dat, nikoliv pouze individuálními elementy jeden za druhým [16].

Reaktivní aplikační rozhraní poskytuje operátory podobné streamům z jazyka Java, ale tyto operátory pracují obecně s jakoukoliv sekvencí, nejsou omezeny pouze na kolekce, a umožňují definovat řetězec transformačních operací, které se aplikují na data procházející streamem. Streamy dokáží zpracovat synchronní i asynchronní operace, data lze řetězit, sloučit, nebo na ně aplikovat různé transformace [16].

Implementace reaktivního rozhraní je založena na výše zmíněné specifikaci Reactive Streams. Základním kamenem jsou čtyři rozhraní `Publisher`, `Subscriber`, `Subscription` a `Processor`. Jejich zodpovědnosti jsou:

- Publisher – Poskytovatel potenciálně neomezeného počtu elementů v sekvenci, odesílá je podle požadavků obdržených od svých příjemců. Může obsluhovat dynamicky v čase mnoho příjemců.
- Subscriber – Příjem elementů od poskytovatele, na elementy se dotáže sám. Typicky má tyto 3 metody: `onNext` – zpracování elementu, `onError` – zpracování chyby, a `onComplete` – signalizace dokončení operace `onNext`.
- Subscription – Může být použito pouze jedním příjemcem. Představuje jednotku přijetí elementu od poskytovatele k odběrateli.
- Processor – Procesory jsou speciálním případem poskytovatele, který je zároveň příjemcem [14]. Reprezentují jednotku vykonávání.

Reaktivní typy Flux a Mono implementují rozhraní poskytovatele. Současně umožňují přidávat transformační operace ke každému elementu sekvence. Jednoduché je i zpracování chyb asynchronním způsobem bez použití bloků `try` a `catch`. Klientský kód se chová jako odběratel.

5.3 Architektura systému

Tato sekce se zabývá architekturou systému. Následující dva diagramy tříd 5.5 a 5.6 zobrazují klíčové třídy, metody a závislosti. Z důvodu přehlednosti obsahují pouze důležitá rozhraní a třídy systému. Nejsou zde vyznačeny všechny implementační třídy a všechny vazby závislosti (vynechány jsou především závislosti na třídy pocházející z knihoven).

Diagram 5.5 představuje komunikační jádro systému z pohledu distribuovaného repositáře. Repositář obsahuje třídu `DistributedRepositoryConsumer` obsluhující komunikaci s Kafkou. Má také referenci na správce všech obslužných akcí pro příkazy – `handlerManager`. Všechny typy příkazů jsou určeny výčtovým typem `Command`. Metoda `listen` (je označena anotací `@KafkaListener` – více v 5.1.2), je zavolána při přečtení zprávy požadavku, která je reprezentována třídou `KafkaRequest`, z fronty. Konzument přijme zprávu z fronty, zjistí typ příkazu a vybere podle něj korespondující obsluhu ze správce handler-ů. Handler potom zpracuje příkaz. V rámci zpracování může být odeslána asynchronní odpověď klientovi, pokud klient nastavil při odeslání zprávy parametry `awaitsResponse` a `responseTopic`. K odesílání odpovědí do Kafka fronty slouží třída `ResponseProducer`. Odpověď bude obsahovat základní informace jako například kód a status, a také ID, aby si klient dokázal spárovat zpracované zprávy. Více informací ohledně sktruktury odpovědí je uvedeno v 4.2.1.

Všechny použité technologie, tzn. Kafka, Cassandra, MongoDB, i HDFS, jsou distribuované, je možné přidávat další výpočetní uzly pro navýšení výkonu, a všechny tak počítají s rozšiřitelností do budoucna.

Konfiguraci parametrů systému a technologií lze provést pomocí specifických souborů s příponou `.properties` obsahujících páry klíč-hodnota (konfigurace uvedena v příloze A).

5.3.1 Rozhraní pro dotazování dat

V této sekci bude uvedeno jakým způsobem lze zadávat kritéria pro dotazování. Kritéria slouží k filtrování záznamů, uplatní se na záznamy – dokumenty metadat. Formát jednotlivých kritérií výrazně odpovídá tvaru parametrů dokumentů v MongoDB.

Jeden objekt kritéria je reprezentován instancí třídy `KafkaCriteria` pocházející z modulu Communication. Obsahuje tyto parametry:

- `field` – Řetězcová hodnota udávající název atributu v dokumentu MongoDB.
- `operation` – Jedná se o hodnotu výčtu `MetadataOperation`, který obsahuje tyto názvy operací – `is`, `ne`, `lt`, `lte`, `gt`, `gte`, `in`, `nin`. Tyto názvy porovnávacích operací reflektují názvy metod ve třídě `Criteria` z modulu Spring Data, takže lze pomocí reflexe tento objekt sestavit. Způsob sestavení přes reflexi je zvolen kvůli chybějící závislosti modulu Persistence klientské aplikaci. Objekt `Criteria` lze potom předat do reaktivního dotazu do MongoDB.
- `value` – Představuje hodnotu, podle které se atribut `field` bude filtrovat. Lze zadat libovolný Java objekt.
- `values` – Operace `in` a `nin` vyžadují seznam hodnot k filtrování. K předání seznamu objektů slouží tento parametr.

Objektů typu `KafkaCriteria` lze sestavit libovolné množství, a předat je poté do zprávy požadavku `KafkaRequest` 4.2.1. Jednotlivé filtrovací operace se společně s uvedenými atributy a hodnotami zřetězí pomocí logické operace `and` do objektu třídy `Criteria`.

5.3.2 Scénáře

Tato sekce uvede podporované příkazy a jakým způsobem jsou zpracovány. Podporovány jsou dva scénáře – uložení PCAP souboru a čtení paketů podle zadaných kritérií. Jádro distribuovaného repositáře ilustruje diagram tříd 5.6.

Zpracování proběhne následovně – metoda `listen` třídy DistributedRepositoryConsumer zkonzumuje zprávu z fronty a zavolá metodu `consume`, která spustí obsluhu příkazu. Aktuálně jsou definovány dvě obsluhy:

- Příkaz `STORE_PCAP` - Zpracování řeší třída `StorePcapHandler`.
- Příkaz `LOAD_PCAP` - Obsluhu poskytuje třída `LoadPcapHandler`.

Zpracování a uložení PCAP souboru

Klient odešle příkaz `STORE_PCAP` s dodatečnými parametry udávajícími zdroj dat a výstupní frontu (více v 4.2.1). Po přijetí zprávy požadavku proběhne uložení PCAP souboru do pomocného souboru na lokálním disku (z HDFS a nebo z paměti RAM), protože knihovna Pcap4J neumí zpracovávat PCAP soubory z HDFS ani z paměti RAM. Následně proběhne parsování na jednotlivé pakety. K tomu slouží rozhraní `PcapParser<T>` s implementací dodanou ve třídě `PcapParserImpl`. Pakety jsou jeden po druhém ukládány do databáze Cassandra asynchronním způsobem 5.2.1. Současně jsou také vytvářeny záznamy metadat. Extrahování informací z paketů zajišťuje jednotné rozhraní `PacketExtractor<T, B>` s několika implementacemi podle vrstev TCP/IP modelu – `EthernetPacketExtractor`, `IpPacketExtractor` a `TransportPacketExtractor`. Záznamy metadat se ukládají reaktivním způsobem 5.2.2 po kolekcích obsahujících několik záznamů (počet záznamů, které se uloží najednou, lze konfigurovat pomocí položky `packet.metadata.max.list.size` A). Až jsou všechny pakety i záznamy metadat uloženy, lze odstranit dočasný PCAP soubor z lokálního disku. Poté je případně odeslána odpověď klientovi s návratovým kódem `OK(200)`.

Čtení paketů podle kritérií

Klient odešle příkaz `LOAD_PCAP` s dodatečnými parametry podobně jako v předchozím případě. Navíc zadá dotazovací kritéria jejichž sémantika je uvedena v 5.3.1. Po přijetí zprávy požadavku proběhne sestavení objektu `Criteria` z kritérií uvedených ve zprávě požadavku. Tento objekt je předán do `packetMetadataRepository`, kde je pomocí Spring Data API provedeno čtení metadat paketů z MongoDB. Čtení probíhá reaktivně se zabudovanými `callback`-y, které pro každý načtený záznam spustí další čtení, v tomto případě už hodnoty celého paketu z databáze Cassandra. Načítání paketů probíhá zase asynchronně, k metodě je přidán další `callback`, který obslouží načtený paket tím, že jej zapíše do PCAP souboru (klient musí zadat cestu k výslednému PCAP souboru pomocí atributu `dataSource` ve zprávě požadavku 4.2.1, předání souboru přes Kafku není možné z důvodu neznámé velikosti a počtu nalezených paketů). Přístup k výslednému PCAP souboru je možný skrze rozhraní `PcapDumper<T>`, implementovaného ve třídě `DumperImpl`. PCAP soubor je až do konce čtení paketů uložen jako dočasný soubor (kvůli limitaci Pcap4J knihovny), potom je přesunut na HDFS. Celý proces končí odesláním odpovědi klientovi.

5.3.3 Rozšíření pro nový typ forenzních dat

Rozšíření v podobě podpory nových druhů digitálních forenzních dat je velmi jednoduché. Spočívá v přídání implementace:

- Rozšíření výčtů `Command`, `Operation` a `DataType`, kde lze přidat nové typy příkazů.
- Zpracování příkazu – Je potřeba specializovat třídu `BaseHandler` novou implementací, která bude kompletně řídit zpracování příkazu (`BaseHandler` má přístup k HDFS a k šabloně obsluhující komunikaci s Kafkou). Tato implementace se musí nakonfigurovat jako bean ve třídě `HandlerBeans` a asociovat s typem příkazu přes správce handlerů v `HandlerManagerBeans`.

Po těchto krocích lze zasílat zprávy požadavků s novým typem příkazu z klientské aplikace.

5.4 Klientská aplikace

Pro účely testování byla implementována i klientská demo aplikace ProducerDemo 5.1. Jedná se také o aplikaci postavené na Spring Boot s možností auto-konfigurace. Systém podporuje zaslání zpráv požadavků pro dva výše zmíněné scénáře – požadavek pro uložení PCAP souboru, a požadavek pro čtení paketů podle kritérií. Data předává v obou případech přes HDFS z důvodu šetření paměti RAM. Odpovědi z distribuovaného úložiště přijímají dvě rozdílné komponenty:

- `AcknowledgementConsumerHandler` – Přijímá zprávy pouze z výstupní Kafka fronty `output_topic`.
- `ErrorConsumerHandler` – Přijímá zprávy z chybové fronty `error_topic`.

Čtení zpráv z těchto dvou front je rozděleno do různých tříd pro potenciálně lepší zpracování chyb. Aktuálně jsou všechny příchozí odpovědi (v obou třídách) zapsány do logů.

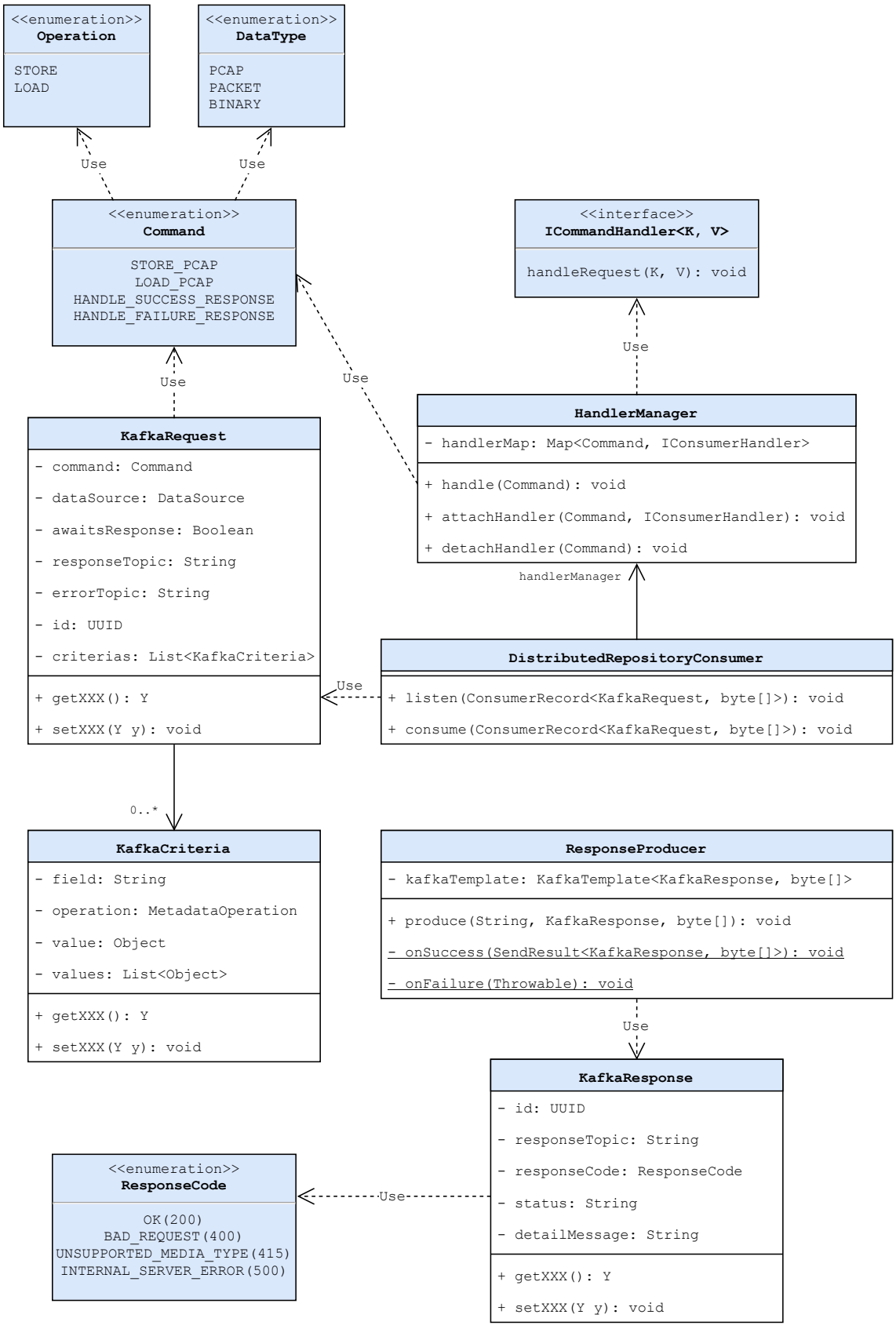
Příklady spuštění aplikací jsou uvedeny v příloze C.

5.5 Logování

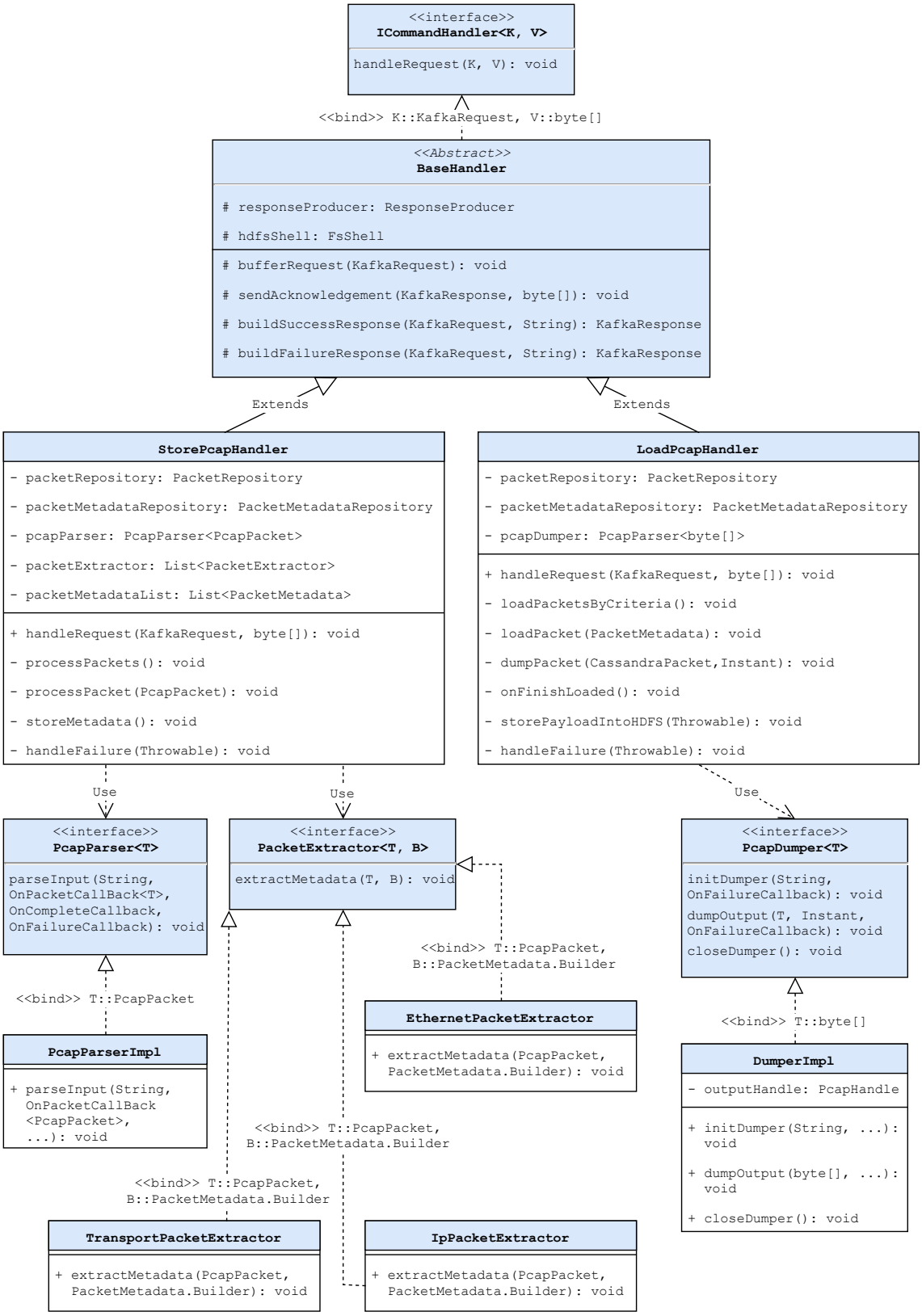
Logování (angl. `logging`) pomáhá trasování běžícího systému. Existuje mnoho implementací pro trasování v Java (`Log4j`, `Log4j 2`, `Logback` atd.). Konkrétní implementaci už v sobě jako závislost zahrnuje přímo Spring Boot. Logování lze konfigurovat mnoha parametry. Lze určit, do kterého souboru se bude zapisovat, jakou úroveň logování použít, formát zapisovaných zpráv, nastavení se může lišit pro jednotlivé balíčky apod. Každá implementace logovacího systému by měla podporovat následující úrovně logování – `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`, `OFF`. Jednotlivé úrovně potom vyjadřují význam zprávy v logu. Odpovídající nastavení v systému distribuovaného úložiště lze vidět v příloze A.

5.6 Zpracování chyb

Při běhu systému může nastat několik druhů chyb – nefungující připojení k databázi, pokud databáze neběží korektně; nedostupnost Kafka front; chyby při ukládání souborů na HDFS např. kvůli kolizi jmen nebo nedostatečnému oprávnění; chyby při parsování PCAP souboru či zápisu do PCAP souboru. Výjimky jsou odchyceny v blocích `try` a `catch`, v případě reaktivních či asynchronních dotazů jsou zaregistrovány odpovídající `callback`-y pro chybové stavy. Po odchycení výjimek jsou výjimky logovány s úrovní `ERROR`. Poté je sestavena odpověď pro klienta s návratovým kódem `INTERNAL_SERVER_ERROR(500)` a odpověď odeslána do chybové fronty zadané parametrem `errorTopic` uvedeného ve zprávě požadavku.



Obrázek 5.5: Diagram tříd komunikace systému.



Obrázek 5.6: Diagram tříd jádra systému.

Kapitola 6

Výkon

Tato kapitola uvádí zhodnocení výkonosti systému, počáteční a finální implementace. Zaměřuje se také na optimální hardwarové konfigurace, které jsou doporučeny pro provoz použitých distribuovaných technologií.

6.1 Prototyp

V rámci semestrálního projektu byl implementován jednoduchý prototyp distribuovaného repositáře reflektující body uvedené v kapitole Návrhu distribuovaného úložiště 4. Prototyp sloužil pouze pro ověření základních aspektů návrhu jako např. komunikace s klientem, práce s databází Cassandra a parsování PCAP souborů na pakety. Podporoval pouze jeden příkaz – `STORE_PCAP`, a to také bez použití metadat. Prototyp nebyl vhodný pro nasazení neboť obsahoval několik nedostatků, které byly v průběhu další implementace eliminovány. Mezi nejdůležitější nedostatky patřily tyto záležitosti:

- předávání všech dat libovolné velikosti v binární podobě přes systém Kafka,
- manuální a neprůhledná konfigurace technologií,
- neefektivní komunikace s databází,
- nebylo optimálně využito hardwarových prostředků,
- celé objekty paketů byly do databáze serializovány.

6.1.1 Eliminace nedostatků prototypu

Původně bylo navrženo a i v prototypu implementováno předávání dat pouze přes systém Kafka. Při velkých objemech přenášených dat bylo plýtváno pamětí RAM a současně také diskovým prostorem, protože Kafka ukládá přijaté zprávy i na disk. Tento problém vyřešilo předávání dat přes HDFS. Tímto způsobem jsou data uložena jen jednou a to na disku, zároveň je šetřena paměť RAM.

Po přidání HDFS a databáze MongoDB pro metadata značně narostl počet konfigurovatelných parametrů. Manuálním načítáním konfiguračních parametrů a vytvářením mnoha objektů realizujících připojení do databází nebo na disk se také ztížila orientace v kódu, který se stal nepřehledným. Tyto operace velmi usnadnila integrace stávajícího kódu s projekty Spring, a hlavně získání auto-konfigurace ze Spring Boot 5.1.2.

Ukládání paketů do databáze Cassandra probíhalo i v prototypu asynchronně, ale nebylo žádným způsobem řešeno vykonávání několika dotazů současně. Díky tomu nastávalo často čekání vlákna apod. Přístupy pro řízení dotazů a pro maximální využití dostupných prostředků jsou uvedeny v sekci 5.2.1.

V původní implementaci byly pakety také ukládány neefektivně z pohledu výkonu a velikosti dat. Každý objekt reprezentující paket byl serializován na pole bytů a toto pole pak uloženo do databáze. Konverze na pole bytů vyžadovala značnou režii, a hlavně velikost serializovaného paketu byla značně vyšší než jeho skutečná velikost. Ve finální implementaci je pole bytů, které obsahuje hodnotu paketu, získáno přímo z objektu paketu pomocí metody `byte[] getRawData()` knihovny Pcap4J.

6.1.2 Porovnání výkonu prototypu a finálního systému

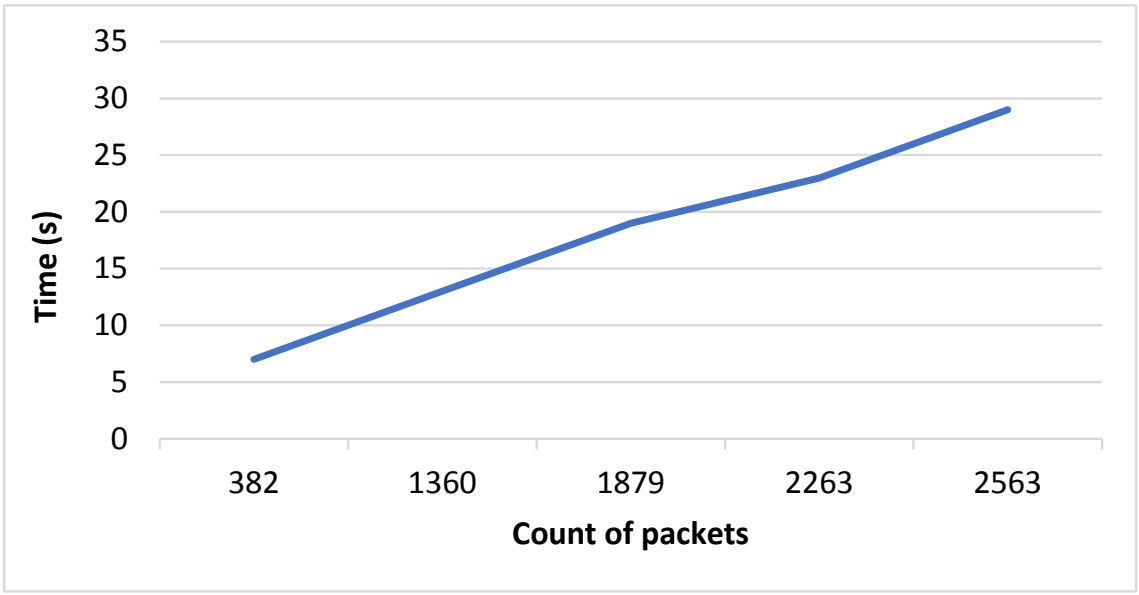
Test rychlosti byl proveden na této hardwarové konfiguraci:

```
CPU: Intel Core i5-4200U 1.6Ghz @ 2.6Ghz
RAM: 8 GB
OS: Windows 8.1
```

Big data technologie Kafka, ZooKeeper, HDFS, Cassandra a MongoDB běžely jako oddělené kontejnery v prostředí Docker, více informací o běhovém prostředí je uvedeno v příloze B. Virtuální stroj v rámci Docker měl toto nastavení:

```
CPU: 2 cores
RAM: 4 GB
OS: boot2docker
```

Prototyp

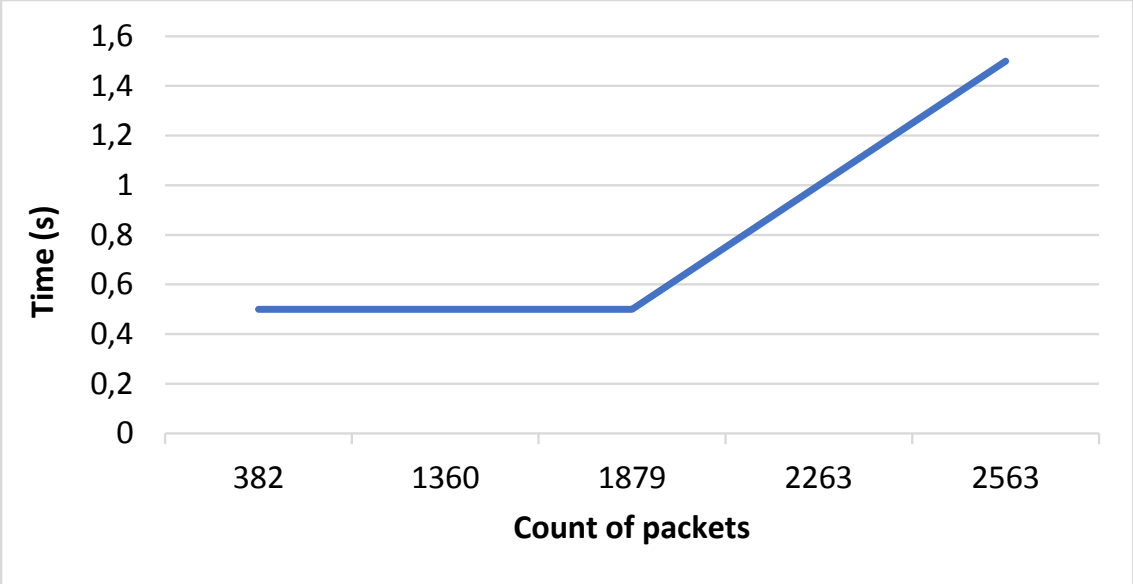


Obrázek 6.1: Graf zobrazující počet uložených paketů z jednoho PCAP souboru za čas.

Pro toto srovnávací testování byly použity pouze PCAP soubory obsahující od stovkek paketů po několik tisíc. Větší PCAP soubory už prototyp na uvedené HW konfiguraci nedokázal zpracovávat z důvodu vyčerpání paměti RAM.

Tabulka D.1 obsahuje informace z měření rychlosti prototypu. První sloupec udává velikost PCAP souboru v bytech, druhý sloupec obsahuje počet paketů na daný PCAP soubor a třetí sloupec zobrazuje průměrnou velikost paketu. Čas uvedený ve čtvrtém sloupci je měřen od odeslání příkazu od klienta po obdržení odpovědi z repositáře. Poslední pátý sloupec udává počet paketů zpracovaných za sekundu. Pro test PCAP souborů větších než 30 MB (obsahujících přes 100 tisíc paketů) už výše uvedená konfigurace nestačila. Po několika minutách aplikace zhavarovala pro nedostatek paměti RAM.

Finální systém



Obrázek 6.2: Graf zobrazující počet uložených paketů z jednoho PCAP souboru za čas.

Tabulka D.2 udává naměřená data pro výsledný systém. Z tabulky, i porovnáním grafů 6.1 a 6.2, lze vidět podstatné zvýšení výkonu. Pro takto malé PCAP soubory je už ve finálním systému většina času spotřebována komunikací klienta se systémem přes Kafku než parsováním paketů, získáváním metadat, a ukládáním metadat a hodnot paketů do databází. I přes přidaný podsystém metadat a předávání PCAP souborů přes HDFS bylo pomocí vylepšení uvedených v 6.1.1 docíleno zrychlení systému.

Z uvedeného grafu pro rychlost prototypu 6.1 vyplývá lineární složitost. Podle údajů v tabulce D.1 lze průměrně zpracovat 89 paketů za sekundu při průměrné velikosti jednoho paketu 450 bytů. V grafu výsledného systému 6.2 už se jedná o velmi malá čísla a tak lineární složitost není z grafu přímo patrná. Přesto pro úplnost, lze zpracovat přibližně 2242 paketů za sekundu, při průměrné velikosti jednoho paketu 450 bytů.

Pozn.: Z výše uvedené hardwarové konfigurace vyplývá, že nebylo využito maximálního potenciálu použitých technologií, a to zejména distribuovanosti systému, který by běžel místo jednoho počítače v clusteru. V takovém prostředí a s rychlým internetovým připojením by byla výkonnost systému znatelně vyšší.

6.2 Srovnání výkonu finálního systému na odlišných HW konfiguracích

6.3 Doporučené HW požadavky distribuovaných technologií

Kapitola 7

Závěr

7.1 Výsledky

V rámci teoretické části práce jsem se seznámil s forenzní analýzou digitálních dat. Prozkoumal jsem existující systémy pro uložení digitálních forenzních dat (včetně AFF4). Seznámil jsem se s distribuovanými databázemi a NoSQL databázemi.

Navrhl jsem distribuované úložiště rozsáhlých digitálních forenzních dat včetně aplikačního rozhraní, a vybral vhodné technologie pro realizaci.

Implementovaný systém řeší požadované aspekty, je založen na distribuovaných technologiích počítajících se škálovatelností. Architektura umožňuje přidat podporu pro nové druhy forenzních dat. S pomocí Spring Boot a ostatních Spring projektů je jednoduchá konfigurace celého systému. Detaily k běhovému prostředí implementovaných aplikací jsou uvedeny v příloze B.

TODO: Vyhodnocení výkonu

7.2 Navržená rozšíření

Literatura

[1] *Advanced Forensic Framework 4 (AFF4)*. [Online; navštíveno 28.10.2017].
URL http://forensicswiki.org/wiki/AFF4

[2] *Apache Hadoop HDFS*. [Online; navštíveno 05.04.2018].
URL https://hortonworks.com/apache/hdfs/

[3] *Data model*. [Online; navštíveno 22.12.2017].
URL https://en.wikipedia.org/wiki/Data_model

[4] *Data Modeling Introduction — MongoDB Manual 3.6*. [Online; navštíveno 27.12.2017].
URL https://docs.mongodb.com/manual/core/data-modeling-introduction/

[5] *Distributed database*. [Online; navštíveno 02.10.2017].
URL https://en.wikipedia.org/wiki/Distributed_database

[6] *File Format Identification*. [Online; navštíveno 01.04.2018].
URL http://www.forensicswiki.org/wiki/File_Format_Identification

[7] *Forensics File Formats*. [Online; navštíveno 28.10.2017].
URL http://www.forensicswiki.org/wiki/Category:Forensics_File_Formats

[8] *Going reactive with Spring Data*. [Online; navštíveno 04.04.2018].
URL https://spring.io/blog/2016/11/28/going-reactive-with-spring-data

[9] *HDFS Architecture Guide*. [Online; navštíveno 05.04.2018].
URL https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

[10] *NoSQL*. [Online; navštíveno 22.12.2017].
URL https://cs.wikipedia.org/wiki/NoSQL

[11] *NoSQL introduction*. [Online; navštíveno 22.12.2017].
URL https://www.w3resource.com/mongodb/nosql.php

[12] *Project Reactor*. [Online; navštíveno 05.04.2018].
URL https://projectreactor.io/

[13] *ReactiveX - An API for asynchronous programming with observable streams*. [Online; navštíveno 04.04.2018].
URL http://reactivex.io/

[14] *Reactore 3 Reference Guide*. [Online; navštíveno 05.04.2018].
URL https://projectreactor.io/docs/core/release/reference/

[15] *Spring for Apache Hadoop*. [Online; navštíveno 09.04.2018].
URL https://projects.spring.io/spring-hadoop/

[16] *Understanding Reactive types*. [Online; navštíveno 05.04.2018].
URL https://spring.io/blog/2016/04/19/understanding-reactive-types

[17] Heller, P.; Piziak, D.; Stackowiak, R.; aj.: *An Enterprise Architect's Guide to Big Data*. [Online; navštíveno 26.09.2017].
URL http://www.oracle.com/technetwork/topics/entarch/articles/oea-big-data-guide-1522052.pdf

[18] Katamreddy, S. P. R.: *Why Spring Boot?* [Online; navštíveno 08.04.2018].
URL https://dzone.com/articles/why-springboot

[19] Leau, C.; Risberg, T.; Valkealahti, J.: *Spring for Apache Hadoop - Reference Documentation*. [Online; navštíveno 08.04.2018].
URL https://docs.spring.io/spring-hadoop/docs/2.5.1.BUILD-SNAPSHOT/reference/html/

[20] Michallat, O.: *Asynchronous queries with the Java driver*. [Online; navštíveno 03.04.2018].
URL https://www.datastax.com/dev/blog/java-driver-async-queries

[21] Nambiar, R.: *What is Big Data ?* . [Online; navštíveno 27.09.2017].
URL http://rrnamb.blogspot.cz/2012/09/what-is-big-data.html

[22] Oracle Help Center: *Distributed Database Architecture*. [Online; navštíveno 29.09.2017].
URL
https://docs.oracle.com/cd/B28359_01/server.111/b28310/ds_concepts001.htm

[23] Rouse, M.: *What is slack space (file slack space)?* [Online; navštíveno 01.04.2018].
URL
http://whatis.techtarget.com/definition/slack-space-file-slack-space

[24] Russell, G.; Bilan, A.; Kunjummen, B.: *Spring for Apache Kafka*. [Online; navštíveno 08.04.2018].
URL https://docs.spring.io/spring-kafka/docs/2.1.6.BUILD-SNAPSHOT/reference/html/

[25] Rychlý, M.: *NoSQL databáze*. FIT VUT v Brně, Říjen 2013, [Online; navštíveno 02.04.2018].
URL http://www.fit.vutbr.cz/~rychly/public/docs/slides-nosql-databases/slides-nosql-databases.pdf

[26] Rychlý, M.: *Paradigma MapReduce a Apache Hadoop*. FIT VUT v Brně, Listopad 2017, [Online; navštíveno 05.04.2018].
URL https://www.fit.vutbr.cz/~rychly/private/pdi/PDI.hadoop_print.pdf

[27] Sadalage, P.: *NoSQL Databases: An Overview*. [Online; navštíveno 22.12.2017].
URL https://www.thoughtworks.com/insights/blog/nosql-databases-overview

[28] Stephens, B.: *What Is Digital Forensics?* [Online; navštíveno 28.10.2017].
URL https://www.interworks.com/blog/bstephens/2016/02/05/what-digital-forensics

[29] Tutorials Point (I) Pvt. Ltd.: *Apache Kafka Tutorial*. [Online; navštíveno 23.12.2017].
URL https://www.tutorialspoint.com/apache_kafka/

[30] Tutorials Point (I) Pvt. Ltd.: *Cassandra Introduction*. [Online; navštíveno 26.12.2017].
URL https://www.tutorialspoint.com/cassandra/index.htm

[31] Tutorials Point (I) Pvt. Ltd.: *Distributed DBMS Tutorial*. [Online; navštíveno 29.09.2017].
URL https://www.tutorialspoint.com/distributed_dbms/

[32] Tutorials Point (I) Pvt. Ltd.: *Hadoop Tutorial*. [Online; navštíveno 05.04.2018].
URL https://www.tutorialspoint.com/hadoop/index.htm

[33] Tutorials Point (I) Pvt. Ltd.: *MongoDB Tutorial*. [Online; navštíveno 27.12.2017].
URL https://www.tutorialspoint.com/mongodb/index.htm

[34] Vandeven, S.: *Forensic Images: For Your Viewing Pleasure*. [Online; navštíveno 28.10.2017].
URL https://www.sans.org/reading-room/whitepapers/forensics/forensic-images-viewing-pleasure-35447

[35] Yamada, K.: *Pcap4J*. [Online; navštíveno 01.04.2018].
URL https://github.com/kaitoy/pcap4j

Příloha A

Konfigurace

A.1 Systém distribuovaného úložiště

Konfigurace aplikace DistributedRepository se nachází v souboru `DistributedRepository/src/main/resources/application.properties` a obsahuje tyto položky:

```
# Cassandra
spring.data.cassandra.keyspace-name=structured_data
spring.data.cassandra.contact-points=192.168.99.100
spring.data.cassandra.port=9042
# Hadoop
spring.hadoop.fs-uri=hdfs://172.17.0.4:9000
# Kafka
spring.kafka.bootstrap-servers=192.168.99.100:9092
spring.kafka.consumer.auto-commit-interval=1000
spring.kafka.consumer.enable-auto-commit=true
spring.kafka.consumer.group-id=test
spring.kafka.consumer.key-deserializer=
   cz.vutbr.fit.communication.serialization.KafkaRequestDeserializer
spring.kafka.consumer.value-deserializer=
   org.apache.kafka.common.serialization.ByteArrayDeserializer
spring.kafka.producer.acks=all
spring.kafka.producer.batch-size=16384
spring.kafka.producer.bootstrap-servers=192.168.99.100:9092
spring.kafka.producer.buffer-memory=335544320
spring.kafka.producer.key-serializer=
   cz.vutbr.fit.communication.serialization.KafkaResponseSerializer
spring.kafka.producer.properties.max.request.size=500000000
spring.kafka.producer.retries=0
spring.kafka.producer.value-serializer=
   org.apache.kafka.common.serialization.ByteArraySerializer
input.topic=input_topic
output.topic=output_topic
error.topic=error_topic
# Logging
```

```
logging.level.cz.vutbr.fit=DEBUG
logging.level.org.pcap4j.core=INFO
# MongoDB
spring.data.mongodb.host=192.168.99.100
spring.data.mongodb.port=27017
spring.data.mongodb.database=metadata
# StorePcapHandler
packet.metadata.max.list.size=500
tmp.directory=tmp/
```

A.2 Klientská aplikace

Konfigurace aplikace ProducerDemo se nachází v souboru
`ProducerDemo/src/main/resources/application.properties`
a obsahuje tyto položky:

```
# Hadoop
spring.hadoop.fs-uri=hdfs://172.17.0.4:9000
# Kafka
spring.kafka.bootstrap-servers=192.168.99.100:9092
spring.kafka.consumer.auto-commit-interval=1000
spring.kafka.consumer.enable-auto-commit=true
spring.kafka.consumer.group-id=test
spring.kafka.consumer.key-deserializer=
   cz.vutbr.fit.communication.serialization.KafkaResponseDeserializer
spring.kafka.consumer.value-deserializer=
   org.apache.kafka.common.serialization.ByteArrayDeserializer
spring.kafka.producer.acks=all
spring.kafka.producer.batch-size=16384
spring.kafka.producer.bootstrap-servers=192.168.99.100:9092
spring.kafka.producer.buffer-memory=335544320
spring.kafka.producer.key-serializer=
   cz.vutbr.fit.communication.serialization.KafkaRequestSerializer
spring.kafka.producer.properties.max.request.size=500000000
spring.kafka.producer.retries=0
spring.kafka.producer.value-serializer=
   org.apache.kafka.common.serialization.ByteArraySerializer
input.topic=input_topic
output.topic=output_topic
error.topic=error_topic
# Logging
logging.level.cz.vutbr.fit=DEBUG
# StorePcapProducerDemo
cz.vutbr.fit.producerdemo.demo.StorePcapProducerDemo.dataSourceStorage=HDFS
# LoadPcapProducerDemo
cz.vutbr.fit.producerdemo.demo.LoadPcapProducerDemo.dataSourceStorage=HDFS
```

Příloha B

Běhové prostředí

Jako běhové prostředí pro systém distribuovaného repositáře byl zvolen projekt `Docker` [1] z důvodu jednodušší a oddělené správy použitých technologií. Každá technologie běží v odděleném kontejneru spuštěném z předem připraveného obrazu. Všechny soubory a skripty týkající se běhového prostředí jsou uloženy v adresáři `Docker/`.

B.1 Obrazy

Každou technologii reprezentuje samostatný obraz (angl. `image`), v němž je daná technologie nainstalována.

Kompletní přehled obrazů:

```
$ docker image ls
REPOSITORY                TAG                 IMAGE ID       CREATED         SIZE
martinfit/maven           3.5.2-jdk-9-slim    b78e23733813   3 hours ago     449MB
martinfit/cassandra       3                   9a1473c2fc55   3 hours ago     323MB
wurstmeister/kafka        1.0.1               2e0c23534746   2 weeks ago     330MB
mongo                     3.4                 9ad59b0c0624   3 weeks ago     360MB
cassandra                 3                   c6b513da2ff3   4 weeks ago     323MB
maven                     3.5.2-jdk-9-slim    58405637ffbb   8 weeks ago     392MB
wurstmeister/zookeeper    latest              351aa00d2fe9   16 months ago   478MB
sequenceiq/hadoop-docker  2.7.1               42efa33d1fa3   2 years ago     1.76GB
```

Většina výše uvedených obrazů pochází přímo z oficiálního repositáře `Docker Hub` [2]. Výjimku tvoří obrazy `martinfit/maven` a `martinfit/cassandra`. Tyto dva obrazy jsou založeny na obrazech z Docker Hub, ale byla do nich přidána dodatečná konfigurace. Obraz pro databázi Cassandra byl mírně upraven, aby bylo možné inicializovat schéma databáze při spuštění kontejneru. Do obrazu pro nástroj Maven byla doinstalována nativní knihovna `libpcap` kvůli korektnímu běhu knihovny Pcap4J 5.1.1.

Instalaci prostředí (tzn. stažení a sestavení obrazů) provádí nástroje `docker-compose pull` a `docker build` uvnitř skriptu `install-docker-enviroment.sh`. Spuštění kontejnerů z definovaných obrazů zajistí skript `run-docker-enviroment.sh`. Zastavit kontejnery lze pomocí skriptu `stop-docker-enviroment.sh`. Veškerá konfigurace obrazů a kontejnerů se nachází v souboru `Environment/docker-compose.yml`.

[1] https://www.docker.com/
[2] https://hub.docker.com/

Příloha C

Spuštění aplikací

Sestavení jednotlivých modulů zajišťují přiložené skripty `install.sh` v adresáři každého modulu. Jednotlivé cesty ke skriptům jsou:

1. `Communication/install.sh`
2. `Persistence/install.sh`
3. `DistributedRepository/install.sh`
4. `ProducerDemo/install.sh`

Kvůli závislostem mezi moduly je nutné skripty spustit ve výše uvedeném pořadí. Moduly jsou zkompilovány a sestaveny pomocí nástroje Maven, který provede stažení všech potřebných závislostí.

C.1 DistributedRepository

Systém distribuovaného úložiště lze spustit skriptem `DistributedRepository/run.sh`, který dynamicky zjistí IP adresy běžících kontejnerů v Docker, a na základě zjištěných hodnot přepíše aplikační proměnné.

C.2 ProducerDemo

Klientskou aplikaci lze spustit pomocí skriptu `ProducerDemo/run.sh`, který také dynamicky přepíše aplikační proměnné podle běžících kontejnerů.

Příloha D

Data měření výkonnosti

D.1 Prototyp

PCAP size (B)	Count of packets	Avg. packet size	Time (s)	Packets/s
91 340	382	239,1099476	7	54,57142857
1 955 172	1360	1437,626471	13	104,6153846
412 254	1879	219,4007451	19	98,89473684
420 869	2263	185,9783473	23	98,39130435
449 234	2563	175,276629	29	88,37931034

Tabulka D.1: Tabulka obsahuje data naměřená při testování výkonnosti prototypu.

D.2 Finální systém

PCAP size (B)	Count of packets	Avg. packet size	Time (s)	Packets/s
91 340	382	239,1099476	0,5	764
1 955 172	1360	1437,626471	0,5	2720
412 254	1879	219,4007451	0,5	3758
420 869	2263	185,9783473	1	2263
449 234	2563	175,276629	1,5	1709

Tabulka D.2: Tabulka obsahuje data naměřená při testování výkonnosti finálního systému.