VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

ZPRACOVÁNÍ BIG DATA Z ROZSAHLÝCH IOT SÍTÍ

PROCESSING BIG DATA FROM LARGE IOT NETWORKS

SEMESTRÁLNÍ PROJEKT
TERM PROJECT

AUTOR PRÁCE KRISZTIÁN BENKŐ
AUTHOR

VEDOUCÍ PRÁCE MARTIN KRČMA
SUPERVISOR

BRNO 2018

Ústav počítačových systémů (UPSY) Akademický rok 2018/2019

Zadání diplomové práce

Student: **Benkő Krisztián, Bc.**
Program: Informační technologie Obor: Informační systémy
Název: **Zpracování velkých dat z rozsáhlých IoT sítí**
Big Data Processing from Large IoT Networks
Kategorie: Informační systémy
Zadání:

1. Seznamte se s problematikou zpracování velkých dat.
2. Analyzujte vhodné datové struktury pro obecné uložení dat.
3. Analyzujte možnosti zpracování záznamů velkých dat v reálném čase.
4. Využijte předchozích analýz k realizaci sběru dat z hardware poskytovaného IQRF sítí.
5. Navrhněte a realizujte metodu upozornění v případě hodnot mimo rozsah a možnost jejich vizualizace a predikce.
6. Zhodnoťte vytvořenou realizaci a diskutujte možnosti pokračování projektu.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Krčma Martin, Ing.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 22. května 2019
Datum schválení: 9. května 2019

Abstrakt

Cieľom tejto diplomovej práce je návrh a vytvorenie systému pre zber, spracovanie a ukladanie dát z rosiahlych IoT sietí. Vytvorený systém predstavuje komplexné riešenie, umožňujúce spracovanie dát z rôznych IoT sietí, s využitím Apache Hadoop ekosystému. Dáta sú spracované v reálnom čase a ukladané do NoSQL databázy, ale ukladajú sa dáta aj do súborového systému pre prípadné neskoršie spracovanie. Systém je optimalizovaný a testovaný na dátach zo siete IQRF. Dáta uložené v NoSQL databázi sa vizualizujú a vykonávajú sa predikcie v pravidelných intervaloch. Používateľ je prepojený s týmto systémom cez informačný systém, kam mu v prípade hodnôt mimo rozsah chodia notifikácie.

Abstract

The goal of this diploma thesis is to design and develop a system for collecting, processing and storing data from large IoT networks. The developed system introduces a complex solution able to process data from various IoT networks using Apache Hadoop ecosystem. The data are real-time processed and stored in a NoSQL database, but the data are also stored in the file system for a potential later processing. The system is optimized and tested using data from IQRF network. The data stored in the NoSQL database are visualized and the system periodically generates derived predictions. Users are connected to this system via an information system, which is able to automatically generate notifications when monitored values are out of range.

Kľúčové slová

BigData, IoT, Apache, Hadoop, databáza, NoSQL, distribuovaný súborový systém, Java, real-time, HDFS, Flume, Spark, informačný systém, OpenTSDB, Grafana, HBase, Avro, dávkové spracovanie, lineárna regresia, predikcie, Ambari, IQRF

Keywords

BigData, IoT, Apache, Hadoop, database, NoSQL, distributed file system, Java, real-time, HDFS, Flume, Spark, information system, OpenTSDB, Grafana, HBase, Avro, batch processing, linear regression, predictions, Ambari, IQRF

Citácia

BENKŐ, Krisztián. *Zpracování Big Data z rozsahlých IoT sítí.* Brno, 2018. Semestrální projekt. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Martin Krčma

Zpracování Big Data z rozsahlých IoT sítí

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením pána Ing. Martina Krčmy. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

....
Krisztián Benkő
21. mája 2019

Poďakovanie

Rád by som poďakoval pánovi Ing. Martinovi Krčmovi a Ing. Michalovi Königovi, za vedenie tejto diplomovej práce a cenné rady pri dosahovaní cieľa. Ďalej by som sa chcel poďakovať kolegom a kolegyniam firmy Master s.r.o za motiváciu pri boji s touto prácou a špeciálne webovému tímu za pomocou pri vytváraní architektúry webovej aplikácie. V neposlednom rade ďakujem rodine a priateľom za podporu.

Obsah

1 Úvod **3**

2 Aspekty BigData a IoT **4**
2.1 BigData 4
2.2 IoT 6
2.3 Time series 8
2.4 IoT ekosystém 8
2.5 MQTT 10
2.6 WebSocket 11
2.7 Existujúce riešenie 11

3 Možnosti uloženia dát **13**
3.1 SQL a NoSQL databázy 13
3.1.1 Rozdiely medzi NoSQL a SQL 13
3.1.2 Rozdelenie NoSQL databáz 14
3.1.3 Spoločné vlastnosti NoSQL databáz 14
3.2 Distribuovaný súborový systém HDFS 16
3.2.1 Architektúra HDFS 16
3.2.2 Paralelné spracovanie dát 19

4 Hadoop ekosystém **20**
4.1 Základné komponenty Hadoop 20
4.1.1 MapReduce 20
4.1.2 YARN 21
4.1.3 Common 21
4.2 Vrstvy ekosystému 21
4.3 Distribúcie Apache Hadoop 23
4.4 Distribúcia Hortonworks 24
4.4.1 Apache Zookeeper 24
4.4.2 Apache HBase 24
4.4.3 Apache Hive 27
4.4.4 Apache Pig 27
4.4.5 Apache Oozie 27
4.4.6 Apache Sqoop 28
4.4.7 Apache Mahout 28
4.4.8 Apache Flume 28
4.4.9 Apache Spark 29
4.4.10 Apache Storm 30

4.4.11 Apache Solr . . . 31
4.4.12 Apache Ambari . . . 31
4.4.13 Apache Ranger . . . 32

5 Návrh systému **33**
5.1 Proces spracovania dát . . . 33
5.2 Plánovanie Hortonworks clustera . . . 35
5.3 Zber a predspracovanie údajov . . . 35
5.3.1 Architektúra Apache Flume . . . 35
5.3.2 Vstup dát z IQRF siete . . . 36
5.4 Spracovanie a ukladanie údajov . . . 37
5.4.1 Apache Spark . . . 38
5.4.2 Uloženie dát do OpenTSDB . . . 39
5.5 Zobrazenie údajov . . . 40
5.5.1 Informačný systém . . . 40

6 Realizovaný systém **44**
6.1 Hortonworks cluster . . . 44
6.2 Zber a predspracovanie údajov . . . 45
6.3 Spracovanie a ukladanie údajov . . . 47
6.3.1 Apache Spark . . . 48
6.4 Informačný systém . . . 49
6.4.1 Backend - Serverová aplikácia . . . 50
6.4.2 Frontend - Webová aplikácia . . . 51
6.4.3 Firebase notifikácie . . . 55
6.4.4 Predikcie . . . 55

7 Testovanie systému **57**
7.1 Konfigurácia systému . . . 57
7.2 Demonštrácia funkčnosti . . . 58
7.3 Výkonnostné testovanie . . . 59
7.4 Rýchlosť spracovania a priestorová náročnosť . . . 61
7.5 Informačný systém . . . 62

8 Záver **63**

Literatúra **65**

A IQRF DTO - dotaz a odpoveď **68**

B REST API Serverovej aplikácie **70**

C Obsah CD **71**

Kapitola 1

Úvod

Téma *IoT* sa stáva každým rokom populárnejšou. Dnes už pripojiteľné zariadenia na sieť pre komunikáciu s iným zariadením nájdeme v takmer každom odvetví. Množstvo týchto zariadení a interval generovania dát dosahuje vo väčších organizáciách vysoké hodnoty. Výrobcov IoT zariadení existuje mnoho, takže nie vždy sa všetky zariadenia v jednej organizácií nachádzajú v rovnakej *IoT* sieti, čo ovplyvňuje zber dát. Dáta nestačí zbierať, vyhodnocovať a vykonávať akciu, ale je potrebné ich aj spracovať a ukladať pre sledovanie historického stavu systému, monitorovanie, predikcie a ďalšie vyžadované akcie.

Cieľom tejto práce je navrhnúť a vytvoriť systém, ktorý bude primárne postavený na spracovávaní dát z *IQRF* siete, ale bude možné ho ľahko prispôsobiť aj pre iné IoT siete a teda bude schopný spracovávať dáta aj z iných sietí. K systému bude možné pripojiť brány jednotlivých IoT sietí pre získavanie a zber dát. Pozbierané dáta sa budú spracovávať *real-time* spracovaním, kde sa budú dáta spracovávať v pamäti a v menšom množstve, ale budú sa ukladať aj pre možnosti neskoršieho spracovania *dávkovým* spracovaním. Spracované dáta sa potom uložia v databázi alebo v distribuovanom súborovom systéme. Z týchto dát bude možné vytvárať predikcie a vizualizovať ich.

V práci sa zo začiatku pojednáva o aspektoch **BigData** a **IoT**, kde je snaha priblížiť problematiku tak, aby sa jej správne porozumelo a pochopili vlastnosti, ktoré k daným oblastiam patria. Predstaví sa **IoT ekosystém**, jeho vrstvy a používané protokoly na jednotlivých vrstvách, aby sme získali prehľad ako sa k IoT ekosystému pripojiť a aké dáta očakávať 2. Neskôr skúmame možnosti **uloženia potencionálnych dát** z IoT sietí a porovnávame možnosti 3. Dôležitou súčasťou tejto práce je samotná distribúcia **Hadoop** systému, ktorá je predurčená zvládať spracovávať dáta aj v **reálnom čase**. Predstavia sa jeho hlavné komponenty a následne po výbere distribúcie sa predstavia aj ďalšie dôležité komponenty 4. Po dôkladnej štúdii Hadoop systému sa vytvorí hrubý návrh, v ktorom sú zakomponované **požiadavky na výsledný systém** a možnosti Hadoop ekosystému. Tento **návrh** sa dekomponuje na menšie problémy a jednotlivé časti sa navrhnú 5. **Systém sa realizuje**, vychádza sa z návrhu, teda najprv sa realizujú menšie časti a nakoniec sa časti integrujú aby vytvorili výsledný systém 6. Výsledný systém sa potom **otestuje**, overí sa jeho stabilita, niektoré ďalšie vlastnosti systému a diskutuje sa **pokračovanie projektu** 7.

Kapitola 2

Aspekty BigData a IoT

Základnými stavebnými prvkami tohto projektu sú *BigData* a *IoT*. Tieto pojmy sú v spoločnosti už dnes veľmi rozšírené. V tejto kapitole sa uvedie význam týchto pojmov, ich charakteristika a vlastnosti. Ďalej sa popíše ako tieto prvky súvisia a ako s nimi správne narábať. Priblíži sa formát generovaných údajov z väčšiny IoT zariadení. Predstavíme si *IoT ekosystém* a popíšeme akým spôsobom v nej jednotlivé druhy zariadení komunikujú, aké protokoly sa využívajú a aké možnosti sú na pripojenie sa do tohto ekosystému za účelom vyčítania nameraných hodnôt. Nakoniec sa pozrieme na existujúce riešenie.

2.1 BigData

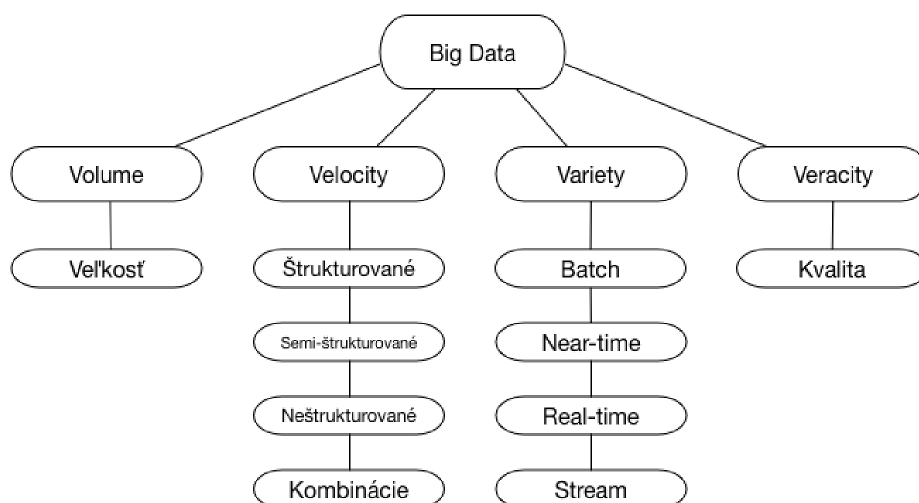
Pojem BigData je možné voľne preložiť ako veľké dáta (avšak pre jeho rozšírené použitie a veľmi špecifický význam ho v tejto práci nebudeme prekladať). Definícia BigData nie je v súčasnosti celkom jednoznačná. BigData pracuje na princípe, že čím viac viete o čomkoľvek alebo akejkoľvek situácii, tým spoľahlivejšie môžete získať nové poznatky a predpovedať, čo sa stane v budúcnosti. Porovnaním viacerých dátových bodov sa začnú objavovať vzťahy, ktoré boli predtým skryté, a tieto vzťahy nám umožňujú učiť sa a robiť inteligentnejšie rozhodnutia. Najčastejšie sa to robí prostredníctvom procesu, ktorý zahŕňa vytváranie modelov založených na údajoch, ktoré môžeme zhromaždiť, a potom spúšťať simulácie, zakaždým upravovať hodnotu dátových bodov a sledovať, ako to ovplyvňuje naše výsledky. Tento proces je zautomatizovaný - dnešná moderná analytická technológia spúšťa milióny týchto simulácií, vylepšuje všetky možné premenné, až kým nenájde vhodný model, ktorý pomôže vyriešiť problém, na ktorom sa pracuje.

Až do pomerne nedávnej doby boli údaje obmedzené na tabuľky alebo databázy, čo bolo krásne usporiadané a elegantné. Čokoľvek, čo nebolo ľahké usporiadať do riadkov a stĺpcov, bolo jednoducho príliš ťažké spracovať a bolo to ignorované. Teraz však pokroky v oblasti ukladania a analýzy znamenajú, že môžeme zachytiť, ukladať a pracovať s mnohými rôznymi typmi údajov. Výsledkom toho je, že údaje môžu v súčasnosti znamenať čokoľvek od databáz až po fotografie, videá, zvukové nahrávky, písomné textové a senzorové dáta.

Aby boli zmysluplné všetky tieto chaotické údaje, projekty BigData často používajú špičkové analýzy zahŕňajúce umelú inteligenciu a strojové učenie. Vyučovaním počítačov s cieľom zistiť, čo tieto údaje predstavujú - napríklad prostredníctvom rozpoznávania obrazov alebo spracovania prirodzeného jazyka sa môžu naučiť skopírovať vzorky oveľa rýchlejšie a spoľahlivejšie ako ľudia [21].

Charakteristika BigData

Aj keď samotný názov poukazuje na to, že sa skrátka jedná o veľké objemy dát, nemusí byť celkom jasné, kde je položená **hranica medzi bežnými dátami a BigData**. Preto sa riadime skôr charakterom dát. Teda, že ich "*veľkosť, rýchlosť rastu a rôznorodosť neumožňujú spracovanie pomocou doposiaľ známych a overených technológií v rozumnom čase*". BigData sú charakterizované akýsimi vlastnosťami, ktoré sa označujú 3V: **Volume, Velocity, Variety**. Avšak pri tejto charakteristike neexistuje úplná jednotnosť, lebo existuje viacero modelov, ktoré sa snažia popísať BigData vo viacerých rozmeroch. Model 3V sa všeobecne uznáva, ale pridávajú sa k nim ďalšie "V". Napríklad označenie 4V, ktoré vzniklo pridaním neistej vierohodnosti (**veracity**), obrázok 2.1. K nim sa pridávajú ďalšie: vysoká hodnota pre firmu (**value**), limitovaná doba platnosti (**validity**) a prechodná doba uchovávania (**volatility**) [13].



Obr. 2.1: Charakteristika BigData.

Volume – objem: veľké dáta logicky znamenajú veľký objem. To je ich hlavná charakteristika. Dnes sa špeciálne technológie pre veľké dáta používajú na dáta rádovo stoviek terabajtov až petabajty.

Velocity – rýchlosť: BigData technológie umožňujú v reálnom čase spracovávať dáta ktoré pribúdajú veľkou rýchlosťou. Dáta určené pre analýzu spracované klasickým spôsobom v takzvaných dátových skladoch (*data warehouse*) je potrebné najprv transformovať a až potom sa môžu pridať do analytických databáz. Veľké dáta môžu vstupovať do analytických procesov v "surovom" stave, v reálnom čase, tak ako prichádzajú.

- **Batch** - spracovanie dát sa vykonáva v určitých intervaloch
- **Real-time** - dáta sa spracovávajú okamžite ako sa príjmu
- **Near-time** - existuje malé oneskorenie medzi tým, ako sa dáta príjmu a spracujú, takmer *real-time*
- **Stream** - podobné *real-time* spracovaniu

Variety – rôznorodosť: BigData sú typické veľkou rôznorodosťou. Už to nie sú len klasické tabuľkové dáta na aké sme boli zvyknutí z klasických (relačných) databáz ale aj rôzne dokumenty, ktoré majú voľnejšiu štruktúru, ako napríklad web stránky, statusy sociálnych sietí, prepojenia v sociálnych sieťach, logovacie súbory z rôznych zariadení či dokonca obrázky alebo rôzne audio a video súbory.

- **Štrukturované** - dáta, ktoré sa bežne vyskytujú v relačných databázových systémoch
- **Semi-štrukturované** - sú to napríklad: textové dokumenty, *JSON*, *XML*. Štrukturované dáta, ktoré nemôžu byť organizované v tabuľke relačného modelu
- **Neštrukturované** - jedná sa napríklad o audio, video, obrázky, dáta zo senzorov a podobne. Tieto dáta sú ťažko analyzovateľné a skladovateľné, relačné databázové systémy nestačia.
- **Kombinácie** - kombinácie predchádzajúcich typov [8].

Veracity – neistá vierohodnosť: vzťahuje sa na skreslenosť, šum a odchýlky v údajoch. Predstavuje jak presné a pravdivé sú naše dáta. Presnejšie, pokiaľ ide o presnosť BigData, nie je to len kvalita samotných údajov, ale aj jak dôveryhodný je zdroj údajov, ich typ a ich spracovanie. Odstránením skreslenia, šumu, odchýlky alebo duplikácie údajov sa prispeje k zlepšeniu presnosti BigData.

2.2 IoT

Táto podkapitola vychádza zo zdroju [1] . Pojem IoT predstavuje širokú škálu vecí, ktoré sa menia na inteligentné predmety. Skratka IoT označuje *Internet of Things* (v preklade ako internet vecí, pre zaužívanosť tohto pojmu nebudeme používať jeho preklad). Zvyčajne sú to veci, ktoré bežne nie sú pripojené na internet a aby boli schopné získavať a spravovať údaje, sú vybavené počítačovými čipmi a snímačmi za účelom zhromažďovania údajov. IoT je o zariadeniach, údajoch a spojeniach. Je očakávané, že po pripojení veľkého počtu zariadení, IoT prinesie masívny prílev BigData. Kľúčovou výzvou aplikácií je vizualizácia a získanie poznatkov z rôznych typov údajov (štruktúrované, neštruktúrované, obrázky, kontextové, data v reálnom čase). Hlavným cieľom je využitie údajov a iných kontextových informácií zo senzorov, tak aby bolo možné v reálnom čase určiť modely údajov a ich vzájomné vzťahy s pridanou hodnotou pre zákazníkov. Existujúce technológie BigData je potrebné rozšíriť a vylepšiť, aby efektívne ukladali, spravovali a extrahovali skutočnú hodnotu z nepretržitých prúdov dát zo senzorov. Najväčšou výzvou bude správne identifikovať a vyhodnocovať údaje, a rozhodnúť sa ktoré údaje sa budú môcť spracovávať rýchlejšie pre vyvolanie odpovedajúcich udalostí. Použitie umelej inteligencie ako napríklad *deep-learning*[1] bude kľúčové pre získanie poznatkov z veľkých prúdov dát. V zhrnutí sú IoT zariadenia **zmysly**, Big Data je **pohonná látka** a umelá inteligencia je **mozog** pre realizáciu prepojeného inteligentného sveta.

[1] https://en.wikipedia.org/wiki/Deep_learning

Požiadavky v IoT

Požiadavky na BigData a ich analýzu v oblasti IoT sa časom exponenciálne zvýšili a sľúbili výrazné zlepšenia v rozhodovacích procesoch. V dôsledku toho sa zvýšili aj požiadavky na prispôsobenie analýzy BigData v oblasti IoT, čím sa zmenil spôsob zberu, ukladania a analýzy údajov. Analýza BigData má veľký potenciál získavať zmysluplné informácie z údajov, ktoré sú produkované snímačmi. Táto časť predstavuje kľúčové požiadavky na BigData a ich analýzu v prostredí IoT. Tieto požiadavky zohrávajú dôležitú úlohu pri zlepšovaní služieb IoT pomocou analýzy údajov.

Konektivita - Jednou z kľúčových požiadaviek IoT je poskytnutie spoľahlivého spojenia pre komunikáciu medzi BigData a ich analýzy, aby sa uľahčilo zoskupenie a integrácia obrovských objemov dát zo senzorov. Viaceré objekty okolo nás majú veľký potenciál byť pripojené k vysoko výkonným počítačovým infraštruktúram pre zlepšenie IoT služieb. S rastúcou prítomnosťou *WiFi* a *4G-LTE* sietí je už evolúcia vývoja smerom k všadeprítomným informačným a komunikačným sieťam zrejmá. Pred zavedením inteligencie do našich zariadení však musí byť zabezpečené bezproblémové prepojenie medzi rôznymi objektmi, ako sú IoT zariadenia, výpočet v *cloude*, BigData a ich analýzou.

Ukladací priestor - Rýchly nárast množstva IoT objektov vedie k ukladaniu veľkého množstva heterogénnych údajov na hardvér s nízkymi nákladmi v reálnom čase. Kľúčovými požiadavkami ukladania údajov v systéme IoT sú manipulácia s veľkým množstvom neštruktúrovaných údajov a poskytovanie nízkej latencie pre analytické služby. Aplikácie využívajúce technológie BigData pre IoT umožňujú efektívne ukladanie a spracovanie údajov s cieľom získať nové informácie, ktoré by pomohli zlepšiť iné služby. Väčšina služieb IoT je založená na komunikačných protokoloch *M2M*[2], ktoré by mali zvládnuť veľké množstvo dátových prúdov a využívajú rozšírené úložné kapacity *cloud computing*[3] infraštruktúry.

Kvalita služieb - QoS - Správa zdrojov IoT senzorov a mobilných zariadení je hlavnou požiadavkou na kvalitu služieb (QoS) pre efektívnu analýzu obrovského množstva údajov. Mnohé štúdie sa snažili splniť požiadavku QoS, ako zjednotiť a integrovať architektúru QoS do IoT s cieľom podporiť BigData a analýzu údajov. Svedčí o tom aj výskum [16]. Služba QoS poskytovaná sieťou IoT musí byť spoľahlivá a musí zaručiť mobilný a efektívny prenos údajov zo zdrojov, kde sa vytvárajú BigData. Podpora QoS v tejto sieti je nesmierne dôležitá pre BigData a jeho analýzy. Aby sa však vytvorila spoľahlivá sieť, musia sa do IoT zaviesť rozvíjajúce sieťové technológie, aby sa umožnil prenos v reálnom čase a zlepšili sa schopnosti spracovania údajov.

Real-time analýza - Jedným z najdôležitejších prvkov IoT je komunikácia v reálnom čase medzi prepojenými zariadeniami. BigData a ich analýza v oblasti IoT vyžaduje plynulé vysielanie udalostí a ukladanie prenášaných dát do databáze. Vzhľadom k tomu, že veľa z týchto neštruktúrovaných údajov je prenášaných priamo z webových technológií, implementácie BigData údajov musia vykonávať analýzu pomocou dotazov v reálnom čase s cieľom získať informácie pre zákazníkov rýchlo. Je potrebné rýchlo sa rozhodovať a komunikovať medzi ľuďmi a zariadeniami v reálnom čase.

[2] https://en.wikipedia.org/wiki/Machine_to_machine

[3] **Cloud computing** - model vývoja a používánia počítačových technologií cez internet, poskytovanie služieb či programov s tým, že užívatelia k nim môžu pristupovať vzdialene

Benchmark - Organizácie, ktoré sa rozhodli využívať BigData a analyzovať ich, čelia určitým problémom pri ukladaní a analýze obrovského množstva údajov, ktoré sa zhromažďujú prostredníctvom senzorov v prostredí IoT. Riešenie týchto problémov si vyžaduje pochopenie, ktoré je možné dosiahnuť pri práci s BigData a analytickej platformy. Benchmark hrá dôležitú úlohu v tomto kontexte tým, že poskytuje organizáciám spôsob, ako posúdiť kvalitu BigData a analytických riešení. Vynikajúci systémový benchmark môže poskytnúť jednoduché a priame porovnanie rôznych riešení.

2.3 Time series

V preklade - časové rady, sú generované dáta v IoT, ktoré predstavujú pozorované hodnoty usporiadané v čase tak, že každá hodnota je označená časovou známkou (*timestamp*). Tieto dáta môžu byť získavané z *RFID*[4] alebo *NFC*[5] tagov, zo zariadení, senzorov, inteligentných domov, vozidiel a z podobných inteligentných prostredí. Temporálne dáta sú v IoT prostredí najviac rozšíreným typom dát. Významná je aj efektívnosť výpočtových zdrojov. Databázový systém, ako napríklad *RDBMS* 3.1, nie je vhodný na spracovanie údajov časových radov. Časové rady temporálnych dát v surovom stave nie sú vhodné pre analýzu. Vyžaduje sa predspracovanie pozbieraných dát, ktorých výstupom budú dáta vhodné pre analýzu a ďalšie spracovanie. Výstupom tejto fáze sú časovo označené dáta v temporálnej databázi, ktoré sa rozrastajú horizontálne, a nie vertikálne ako v relačných databázach [2].

2.4 IoT ekosystém

V tejto sekcii sa popisuje zabezpečenie komunikácie medzi IoT zariadeniami a bránou (*gateway)* a smerovanie dát na výstup brány pre spracovanie. Samotná vrstva je zobrazená na obrázku 2.2. Máme zobrazené vrstvy: *datalink* (linková vrstva), *network* (sieťová vrstva) a *transport/session* (transportná/relačná vrstva). Linková vrstva spája 2 IoT prvky, čo môže byť spojenie medzi snímačmi navzájom alebo snímačmi a bránou. Často viacero snímačov potrebuje zoskupiť informácie pred odoslaním informácie. Pre tento prípad existujú smerovacie protokoly (v tabuľke - *routing*). Relačná vrstva je zodpovedná za komunikáciu medzi subsystémami IoT ekosystému. Ďalej sú na obrázku zobrazené ďalsie protokoly týkajúce sa zabezpečenia a manažmentu [27].

Pre prehľad si predstavíme niektoré protokoly linkovej vrstvy a nakoniec sa zameriame na protokol IQRF. Predstavíme si aj protokoly relačnej vrstvy *MQTT* 2.3, *WebSocket* 2.4, keďže sa veľmi často vyskytujú v IoT a sú aj súčasťou IQRF siete.

IEEE 802.15.4

Najrozšírenejší IoT štandard. Definuje formát rámca, hlavičky, zdrojovej a cieľovej adresy, a ako jednotlivé uzly navzájom komunikujú. Formát rámca v klasických sietiach nevyhovuje v nízko-energetických *multi-hop* sietiach, ako sú napríklad IoT siete, kvôli réžiám.

IEEE 802.11 AH

IEEE 802.11 AH je nízko-energetická verzia originálneho štandardu **IEEE 802.11**. Bolo to navrhnuté s menšími réžiami, tak aby naplnil IoT požiadavky. IEEE 802.11 (WiFi)

[4] https://www.abr.com/what-is-rfid-how-does-rfid-work/
[5] https://www.techradar.com/news/what-is-nfc

Session		MQTT, SMQTT, CoRE, DDS, AMQP , XMPP, CoAP, …
Network	Encapsulation	6LowPAN, 6TiSCH, 6Lo, Thread, …
	Routing	RPL, CORPL, CARP, …
Datalink		WiFi, Bluetooth Low Energy, Z-Wave, ZigBee Smart, DECT/ULE, 3G/LTE, NFC, Weightless, HomePlug GP, 802.11ah, 802.15.4e, G.9959, WirelessHART, DASH7, ANT+, LTE-A, LoRaWAN, …

Security	Management
TCG, Oath 2.0, SMACK, SASL, ISASecure, ace, DTLS, Dice, …	IEEE 1905, IEEE 1451, …

Obr. 2.2: IoT protokoly. Zdroj: [27]

je najrozšírenejší bezdrôtový štandard. Je implementovaný takmer v každom digitálnom zariadení. Avšak, kvôli réžiam nemohol byť použitý v IoT, tak to bolo potrebné prispôsobiť a vzniklo **IEEE 802.11 AH** [27].

ZigBee

ZigBee je bezdrôtová komunikačná technológia vystavaná na štandarde **IEEE 802.15.4**. Podobne ako *Bluetooth* je určená na spojenie *low-power zariadení* na malé vzdialenosti. *Multi-skokové ad-hoc smerovanie* umožňuje komunikáciu aj na dlhšie vzdialenosti bez priamej rádiovej viditeľnosti jednotlivých zariadení. Technológia má nájsť primárne využitie v priemysle a senzorových sieťach. ZigBee je navrhnutý ako jednoduchá a flexibilná technológia pre tvorbu rozsiahlejších bezdrôtových sietí, u ktorých nie je požadovaný prenos veľkého objemu dát. Vlastnosti: spoľahlivosť, jednoduchá a nenáročná implementácia, veľmi nízka spotreba energie a priaznivá cena [24].

Z-Wave

Z-Wave je jedna z ďalších bezdrôtových komunikačných sietí, primárne určená na automatizáciu domácnosti. Jej základom je *mesh sieť*[6], pomocou ktorej komunikujú jednotlivé zariadenia medzi sebou [27].

WiFi HaLow

Wi-Fi HaLow je založená na technológií **IEEE 802.11AH**. Na rozdiel od bežných Wi-Fi sietí, pracujúcich na frekvenciách 2,4 a 5 GHz, je prevádzkovaná na frekvencií 900 Mhz, čo zaručí širšie pokrytie a menšiu náchylnosť na rušenie signálu z obvyklého pásma. Prenos dát neprebieha kontinuálne, ale v pravidelných dávkach, ktorých interval môžeme nastaviť. Nižší vysielací výkon a odlišné schéma tak dovoľujú chod na batériu. Wi-Fi HaLow má byť priamym konkurentom Bluetooth, iba s väčším dosahom [23].

[6] https://www.iqrf.org/technology/iqmesh

IQRF

IQRF (*Inteligence Quocient Radio Frequency*) je platforma českého výrobcu, určená pre nízko-energetické siete s malým objemom dát a nízkou rýchlosťou. Je to kompletný systém dodávaný od jednej firmy vrátane hardvéru, softvéru, protokolov a služieb. Dosah komunikácie je v radách desiatok až stoviek metrov, v zriedkavých prípadoch, alebo v špeciálnych sieťach, až niekoľko kilometrov. Využitie nachádza predovšetkým v sieťach IoT, najmä pre telemetriu, priemyslové riadenie a automatizáciu budov a miest. Prvky IQRF môžu byť použité s ľubovoľným elektronickým zariadením (nutná kompatibilita rozhrania). Typicky sa používa na diaľkové ovládanie, monitoring diaľkovo získaných dát alebo pripojenie viacerých zariadení k bezdrôtovej sieti. IQRF je založené na vysielačoch so vstavaným operačným systémom a voliteľnou komunikačnou vrstvou *DPA*[7].Vhodný je aj pre jednoduchú komunikáciu *peer-to-peer*, ale najvyššiu robustnosť dosahuje v komplexných *mesh sieťach.*

Najvýznamnejšou komponentou IQRF siete je pre nás brána (*gateway.* Je to v podstate softvér nasadený na výpočetnom stroji s operačným systémom **Linux**. Tento výpočetný stroj je väčšinou vstavaný systém ako napríklad *Raspberry Pi*[8]. Takúto bránu dokážeme vytvoriť pomocou *IQRF Gateway Daemonu*, čo je softvérový balíček, ktorý vytvorí bránu aj s možnosťou pripojenia na internet a cloud. Brána využíva pre komunikáciu viacero kanálov: *UDP, MQ, WebSocket a MQTT* [30].

Komunikácia prebieha pomocou DPA správ. IQRF definuje formát JSON dotazov a odpovedí, pomocou ktorých sa komunikuje s bránou z obecnej aplikácie.

- `JsonDpaRequest` - posiela sa od odosielateľa na démon
- `JsonDpaResponse` - posiela sa od démona k príjemcovi, ktorý poslal dotaz, ako priama odpoveď alebo ako asynchrónna odpoveď.

Parametre, ktoré sa v týchto správach používajú sú bežné pre viacero typov správ. V nasledujúcom repozitári sa však popisujú parametre pre konkrétny typ správ JsonStructureDpa.

2.5 MQTT

MQTT je skratka pre *Message Queue Telemetry Transport* a je to jednoduchý a nenáročný protokol pre predávanie správ medzi klientami prostredníctvom centrálneho bodu – *brokeru.* Vďaka tejto nenáročnosti a jednoduchosti je jednoducho implementovateľný aj do zariadení s „malými“ procesormi a pomerne rýchlo sa rozšíril. Navrhnutý bol v *IBM*, dnes za ním stojí *Eclipse foundation* a pred nedávnom prebehla štandardizácia OASIS[9].

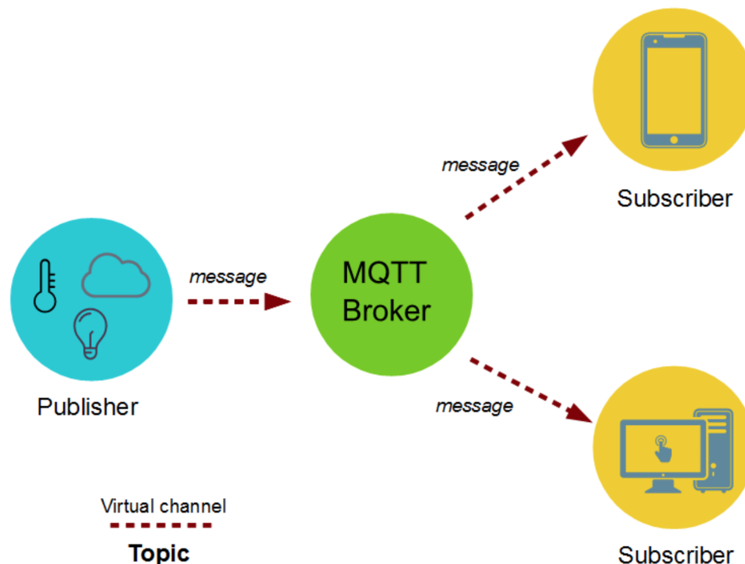
U protokolu MQTT prebieha prenos pomocou TCP a používá návrhový vzor *publisher – subscriber*, viď obrázok 2.3. Teda existuje jeden centrálny bod (*MQTT broker*), ktorý sa stará o výmenu správ. Správy sú triedené do tzv. tém (*topic*) a zariadenie buď publikuje v danej téme (*publish*), to znamená, že posiela dáta brokeru, ktorý ich ukladá a distribuuje dalším zariadeniam, alebo je prihlásený k odberu správ (*subscribe*), a broker potom všetky správy s danou témou posiela do zariadenia. Jedno zariadenie samozrejme môže naraz byť v niektorých témach publisher, v iných subscriber.

[7] `https://www.iqrf.org/technology/dpa`

[8] `https://www.raspberrypi.org/`

[9] **Organization for the Advancement of Structured Information Standards (OASIS)** - Organizácia na presadzovanie noriem pre štruktúrované informácie - je neziskové, globálne konzorcium, ktoré je hnacou silou vývoja, spájania a adaptovania noriem elektronického obchodu a webových služieb.

Z pohľadu IoT, publikujúci sú senzory, ktoré sa pripájajú na broker, aby tam mohli poslať svoje dáta a vrátiť sa do režimu spánku. Odoberajúci sú aplikácie, ktoré majú záujem o dáta z danej témy, takže sa pripájajú na broker aby boli informovaní, keď nejaké nové dáta príjmu. Broker dáta zo senzorov klasifikuje a posiela ich odoberajúcim [27].



Obr. 2.3: IoT protokoly. Zdroj: IoT MQTT prakticky v automatizaci

2.6 WebSocket

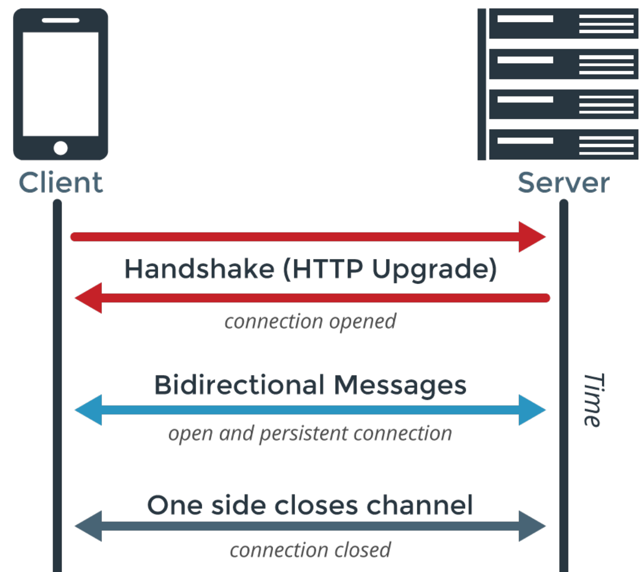
WebSocket je obojsmerný komunikačný protokol, ktorý pracuje cez TCP a vznikla ako inovácia pre štandardné HTTP spojenie. Toto umožňuje plnú duplexnú komunikáciu medzi klientom a serverom pomocou jedného socketu, ktorý je vystavený cez rozhranie *JavaScriptu*. Tento protokol nie je iba rozšírením súčasného protokolu HTTP, ale je nesmierne pokročilý, najmä pre komunikáciu riadenú udalosťami v reálnom čase.

Pri používaní WebSocketu nemusíme zhromažďovať komunikačné dáta ako pri konvenčnom volaní API dotazov. Naopak, môžeme mať nastavenú *push komunikáciu*[10] v reálnom čase medzi serverom a zariadením. Preto môžeme posielať a prijímať veľké množstvo dát v mikrosekundách. Správy sa prenášajú pomocou binárnych údajov, ktoré sú zakódované pomocou JSON. Na obrázku 2.4 je znázornená komunikácia [11].

2.7 Existujúce riešenie

Bashir a Gill [5] vymysleli framework pre analýzu BigData z IoT sietí ako riešenie na problémy spojené s ukladaním a analýzou veľkého množstva údajov pochádzajúcich z **inteligentných budov**. Navrhovaný framework sa skladá z troch častí, ktorými sú správa BigData, IoT senzory a analýza údajov. Analýzy sa vykonávajú v reálnom čase, aby sa mohli používať v rôznych častiach inteligentnej budovy na riadenie hladiny kyslíka, dymu/nebezpečných plynov a svietivosti. Framework je implementovaný pomocou distribúcie

[10]**Push komunikácia** - Typ komunikácie, ktorá smeruje od odosielateľa k príjemcovi, bez vyžiadania odpovede.



Obr. 2.4: HTTP and Websockets Zdroj: Schopnosti dnešných webových komunikačných technológií

Cloudera Hadoop, kde sa analýza vykonáva pomocou *PySparku*. Výsledky ukazujú, že framework môže byť využitý pre analýzu BigData v rámci IoT. Navrhovaný framework je špeciálne navrhnutý pre inteligentné budovy, ktorý je možné rozšíriť pre inteligentné mestá a inteligentné letiská.

Kapitola 3

Možnosti uloženia dát

Uvažujme rôzne typy údajov IoT dát, vrátane **štruktúrovaných**, **čiastočne štruktúrovaných** a obrovského množstva **neštruktúrovaných údajov**, a rôzne druhy databáz alebo súborových systémov, ako napríklad distribuovaný súborový systém *HDFS*, systém riadenia relačných databáz (*RDBMS*), rovnako jak *SQL*, tak aj *NoSQL*. Databáza NoSQL poskytuje mechanizmus pre ukladanie a vyhľadávanie dát, ktorý je modelovaný inými prostriedkami ako sú tabuľkové vzťahy používané v relačných databázach.

3.1 SQL a NoSQL databázy

Mnohé platformy na ukladanie štruktúrovaných dát sú založené na RDBMS, v preklade je to systém riadenia báze dát. RDBMS je softvérový systém, ktorý používateľovi umožňuje definovať, vytvárať a udržovať databázu. Poskytuje riadený prístup k tejto databáze a operácie, ktoré umožňujú používateľom vkladať, aktualizovať, mazať a získavať dáta z databázy. Na tieto dáta sa dotazuje pomocou dotazovacieho jazyka (SQL - Structured Query Language).

NoSQL databázy bývajú často označované ako *"bezschémové"*, ale aby boli dáta pre stroj zrozumiteľné a riešiteľné, je nutné schému vytvárať. Táto schéma nemusí byť nijak špeciálne dopredu definovaná a skôr jak o schéme hovoríme o meta informáciách dát. Oproti relačným databázam, NoSQL databáza často prenáša zodpovednosť za správu schémy na aplikačnú logiku[7].

3.1.1 Rozdiely medzi NoSQL a SQL

Účelom NoSQL databáz je vytvoriť databázový systém, ktorý bude lepšie reagovať na záťaž spojenú s veľkým množstvom semi-štruktúrovaných dát. Ich použitie by však malo byť premyslené a cielené, pretože za cenu škálovateľnosti prichádzame o mnohé výhody SQL databáz, ktoré majú jasnú štruktúru a overené teoretické pozadie. SQL a NoSQL databázy je možné kombinovať a používať oddelene.

Dole v nižšie uvedenej tabuľke 3.1 sa popisujú najčastejšie rozdiely vlastností daných typov databáz, na základe ktorých sa pri výbere databázy rozhodujeme.

Vlastnosť	Relačné databázy	NoSQL databázy
Distribuovanosť	Zložitá implementácia	Distribúcia dát cez viaceré uzly
Škálovateľnosť	Vertikálna	Horizontálna
Robustnosť	Pravidelné zálohy dát	Replikácia dát vo viacerých uzloch
Konzistencia	Silná	Eventuálna
Dotazovací jazyk	Jazyk SQL	Vlastný dotazovací jazyk
Znalosť schémy	Data definition language (DDL[1])	Znalosť udržuje aplikácia
Schéma	Dobre definovaná	Voľná
Charakter dotazov	Častý update	Jeden zápis, čítanie viacnásobné
Normalizácia dát	Normálna forma	Denormalizované dáta
Trvalosť dát	Trvalé ukladanie	Dynamické mazanie záznamov
Rast dát	Lineárny	Exponenciálny

Tabuľka 3.1: Rozdiely vo vlasnostiach medzi relačnými a NoSQL databázami

3.1.2 Rozdelenie NoSQL databáz

NoSQL databázý rozdeľujeme na základe modelu ukladania dát. V súčastnosti sa používajú 4 koncepty:

- **Databáza kľúč-hodnota** - typ databáze, ktorý je založený na modeli pár kľúč-hodnota. Hodnota je priradená ku kľúču, na základe ktorej je možné hodnotu vyhľadať.
- **Stĺpcové databázy** - dátova sada sa skladá z niekoľkých riadkov, pri čom každý riadok je adresovaný primárnym kľúčom a je zložený zo skupiny stĺpcov. Rôzne riadky môžu mať rôzne skupiny stĺpcov. Vhodný pre dáta prichádzajúce v rýchlom prúde, napríklad dáta zo senzorov.
- **Dokumentové databázy** - kľúč je pri tomto type databáz použitý pre vyhľadanie pozície dokumentu. Dokumenty sú najčastejšie formátu JSON, YAML alebo XML. Je možné vyhľadávať aj podľa obsahu dokumentov a niektoré databázy tohto typu umožňujú aj vlastnú implementáciu *MapReduce* 4.1 funkcie.
- **Grafové databázy** - graf je použitý ako dátový model. Tento typ databáz vznikol skôr pre správu dát s komplexnou štruktúrou a nie pre BigData [13].

3.1.3 Spoločné vlastnosti NoSQL databáz

Konzistencia

Z pohľadu *CAP teorému* [6] väčšinu distribuovaných RDBMS označujeme primárne za *CP* (konzistentné s odolnosťou voči rozpadu siete) s eventuálnou dostupnosťou. NoSQL databázy sa označuje *AP* (dostupné s odolnosťou voči rozpadu siete) s eventuálnou konzistenciou. Úplnú konzistenciu je možné zabezpečiť len na úkor dostupnosti. S tým súvisí vykonávanie transakcií. U relačných databázach sa vlastnosti transakcií označujú ako *ACID*[2], pri NoSQL databázách sú to *BASE vlastnosti*.

[1] jazyk pre definíciu dátových štruktúr
[2] ACID transakcie

BASE vlastnosti:

- **Basic Avialability** - systém garantuje odpoveď na každú požiadavku, ale nie je zaručená konzistencia.
- **Soft state** - systém sa môže kedykoľvek meniť, nie je konzistentý po celú dobu.
- **Eventual consistency** - časom však vždy dosiahne konzistentného stavu [13].

Horizontálne škálovanie

Pri horizontálnom škálovaní (*scaling out*) sa dáta distribuujú cez viaceré uzly a pri zvyšovaní nárokov sa navyšuje počet uzlov. Skupina takýchto uzlov vytvára *cluster*. V rámci clustrov sa aplikuje **replikácia** a rozdelenie dát. Je protipólom vertikálneho škálovania (*scaling up*), ktoré vo väčšine prípadov využívajú relačné databázy a ktoré predstavuje zvyšovanie výpočtového výkonu jediného uzlu (servera) pomocou výkonnejšieho hardvéru. Nákup výkonnejšieho hardvéru je však drahší ako použitie veľkého množstva komoditného hardvéru a taktiež má svoju hornú hranicu výkonu. Pri horizontálnom škálovaní zas narazíme na problém so stabilitou sieťového prepojenia a synchronizáciou pri paralelnom spracovaní. **Elasticita** je však stále jednou z najsilnejších výhod NoSQL databáz. Pri správnom použití distribúcie dát môžeme efektívne rozložiť záťaž a teda zvýšiť priepustnosť systému a taktiež mať k dispozícií v podstate neobmedzenú veľkosť úložiska [13] [25].

Distribuovanosť

Distribuované spracovanie dát so sebou prináša aj problémy, ktoré je potrebné riešit. Obecne riešime 2 problémy:

- **Replikácia** - rieši sa základný problém, ktorým sú výpadky spojenia uzlov. Teda uložia sa dáta na viacerých uzloch v rôznych častiach siete. Týmto vzniká problém udržovania konzistencie kópií v jednotlivých uzloch
- **Rozdelenie (sharding)** - rozdelenie dát (*shards*) na uzly pričom sa snažíme nájsť kompromis medzi rovnomerným rozmiestnením dát a minimalizáciu počtu uzlov, na ktoré musí užívateľ pristupovať, vrátane optimálneho geografického rozmiestnenia dát. Tento prístup zlepšuje škálovateľnosť výkonu, úložiskového priestoru aj odolnosť systému [13].

Architektonické prístupy pri návrhu NoSQL databáze

Pri návrhu dátovej schémy v NoSQL databázach sa využívajú odlišné prístupy než v relačných databázach. Je potrebné vopred navrhnúť databázu tak, aby čo najviac spĺňala požiadavky systému, pretože neskoršie zmeny môžu byť náročné a môžu spôsobovať chyby.

- **Dátový model orientovaný na agregáty** - pri návrhu dátového modelu v NoSQL databázach sa vytvárajú agregáty (celky), narozdiel od relačných databáz, v ktorých sa vytvárajú entity a vzťahy. Správne určenie agregátov je nevyhnutné pre efektívnu prácu s databázou. Pri vytváraní agregátu sa môžeme riadiť buď relačným modelom, kde agregačné vzťahy môžu byť ukladané spolu. Alebo sa riadime aplikáciou a dáta, na ktoré sa chceme dotazovať spolu, uložíme do jedného agregátu.

- **Bezschémový model** - vieme, čo očakávať od agregátu. V dokumentovej databázi budú mať rôzne dokumenty rozdielnu štruktúru, v stĺpcovej databázi bude mať každý riadok rozdielny počet a typ stĺpcov.

- **Materializovaný pohľad** - slúži pre prístup k dátam uložených vo viacerých agregátoch. Tieto pohľady sa následne uložia do databáze pre opätovné použitie.

- **Lambda architektúra** - dotazovanie nad rôznymi agregátmi pri vytváraní materializovaného pohľadu môže byť výpočtovo náročné. Lambda architektúra rieši tento problém v troch vrstvách. V **dávkovej vrstve** sa spravuje dátová sada, do ktorého sa dáta maximálne len pridávajú a vytvárajú sa materializované pohľady. V **rýchlostnej vrstve** sa priebežne spracovávajú prúdové dáta, ktoré neboli spracované dávkovým spracovaním, čím sa vracajú potom dáta v materializovaných pohľadoch efektívnejšie. V **obsluhovacej vrstve** sa kombinujú dáta z predchádzajúcich dvoch vrstiev, tak aby dotazy pracovali s aktuálnymi dátami. [13]

3.2 Distribuovaný súborový systém HDFS

Apache Hadoop je *open-source* softvérová platforma, ktorá umožňuje prácu s BigData a ich ukladanie na bežne dostupný hardvér. K jeho základnej funkcii teda nepotrebujeme žiadne špeciálne upravené stroje, ktoré by nám umožnili funkčnosť tejto platformy. Dokáže prepojiť viac staníc, s ktorými potom pracuje, ale každá stanica zapojená do tejto platformy pracuje nezávisle na sebe. Vzniká nám teda distribuovaný súborový systém *HDFS*.

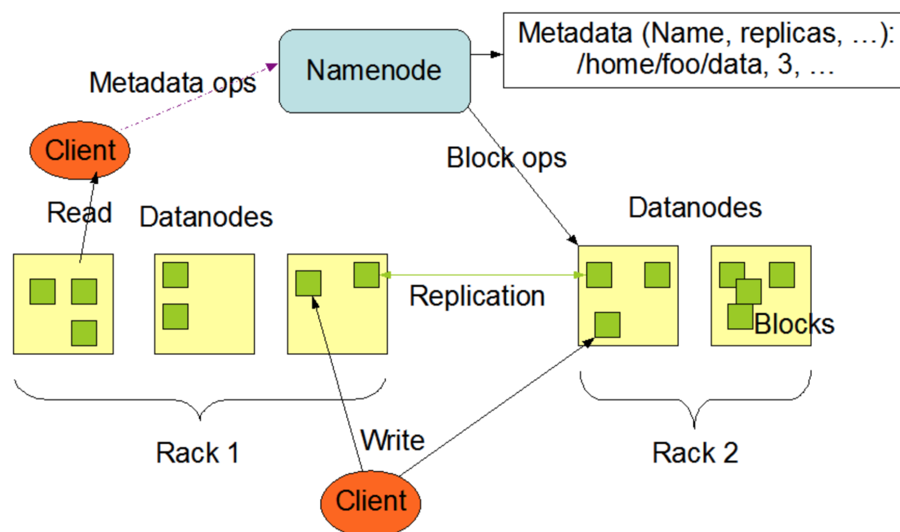
3.2.1 Architektúra HDFS

Architektúra Hadoop spočíva v zapojení bežne dostupných staníc do HDFS. Hlavný princíp tejto architektúry je zabezpečiť rýchly prístup k dátam, jej spoľahlivosť a jednoduchá rozšíriteľnosť pre budúce potreby. Samotný Hadoop je súborom rôznych programov, každá zameriavajúca sa na iné časti použitia. Nasledujúca sekcia vychádza z oficiálnej dokumentácie [3].

Hadoop cluster je špeciálny typ výpočtového klastra navrhnutý špeciálne pre ukladanie a analyzovanie veľkých objemov **neštruktúrovaných dát** v distribuovanom výpočtovom prostredí. Tento systém sa skladá z veľkého počtu bežných staníc, na ktoré sa autoritatívne ukladajú dáta, ktoré potrebujeme uskladniť. Každá stanica, *DataNode*, má svoj vlastný výpočtový výkon a svoje úložisko. Stanice medzi sebou nespolupracujú na dosiahnutie cieleného výsledku. Každá má za úlohu spracovať svoju pridelenú úlohu, ktorú jej zaslal jej nadriadený. Tomuto nadriadenému prvku sa hovorí *NameNode* [26], viď obrázok 3.1.

NameNode

NameNode umožňuje tvorbu distribuovaných výpočtov v HDFS na princípe *master-slave*. Riadi tok dát medzi DataNodami a uchováva si metadáta o súborovom systéme, ktorý je k NameNodu pripojený. Metadáta obsahujú informácie o tom, kam sa dáta uložili, prípadne ich kópie na inom DataNode. NameNode opakovane testuje, či každý DataNode je aktívny, aby získal aktuálnu štruktúru HDFS. NameNode si neustále žiada odozvu od DataNodov a ak od nejakého DataNodu neobdrží odpoveď, označí ho za neaktívny. Dáta, ktoré sa nachádzali na neaktívnom DataNode sa replikujú na iný DataNode pomocou metadát, ktoré sú uložené v NameNode. NameNode sa neustále snaží udržiavať viac kópií jedného



Obr. 3.1: HDFS architektúra. Zdroj: HDFS architecture

súboru v HDFS, hlavne kvôli výpadkom nejakého DataNodu. Nepríde teda o dáta, ktoré sa na odstavenom DataNode nachádzali. Tomuto procesu sa hovorí replikačný faktor (*RF*). NameNode sa neustále snaží **rozprestierať dáta** v HDFS medzi jednotlivé DataNody tak, aby boli čo najrovnomernejšie zaplnené. Ak sa teda pripojí nový prázdny DataNode do HDFS a NameNode si ho označí za aktívny, okamžite sa replikujú dáta z ostatných DataNodov tak, aby opäť boli všetky rovnomerne zaplnené. Umožní tým **rozloženie náročnosti** jednotlivých úloh medzi viacerými DataNodmi. Tento proces je síce náročnejší, ale plne automatický zo strany NameNodu. Tým nám vzniká úložisko, ktoré je veľmi **jednoducho rozšíriteľné** o ďalšie stroje pre zväčšenie úložiskovej a výpočtovej kapacity HDFS.

DataNode

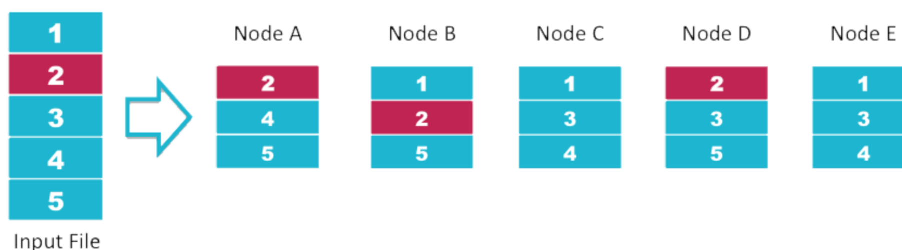
DataNode slúži ako jednotka pre ukladanie a prácu s dátami v HDFS. Každý DataNode má vlastný blok s úložiskom a vlastný výpočtový výkon pre prácu nad dátami vo vlastnom bloku. NameNode riadi ukladanie a prácu s dátami, ktorú majú jednotlivé DataNody spracovať na svojom bloku. DataNody spolu vzájomne komunikujú v prípade replikovania dát, aby bol dodržaný RF. Tento proces síce stále riadi NameNode, ale replikované dáta si medzi sebou preposielajú jednotlivé DataNody. NameNody po ukončení procesu zašlú **správu ohľadom úspešnosti** replikácie dát. DataNode musí neustále oznamovať NameNodu svoju aktivitu, tzv. *Heartbeat*. Každé 3 sekundy DataNode oznamuje, že nevykazuje žiadny problém a je v bezproblémovej prevádzke. V každej desiatej správe DataNode oznamuje stav vlastného bloku, aby NameNode mohol aktualizovať svoje metadáta a podľa potreby zabezpečiť dodržanie *RF*.

Replikácia

Základný princíp pri replikácii dát je, aby sa jeden súbor nenachádzal na tej istej stanici ako pôvodný súbor, z ktorého kópia vzniká. Dôvodom je, že ak prídeme o stanicu, kde sa nachádza zdrojový aj replikovaný súbor, dáta budú nenávratne preč. Základným štandar-

dom býva nastavenie RF na 3. Pri nastavenom **RF 3** sa nachádza jeden súbor na troch rôznych DataNodoch v HDFS, teda jeden pôvodný zdrojový súbor a jeho dve kópie, viď obrázok 3.2.

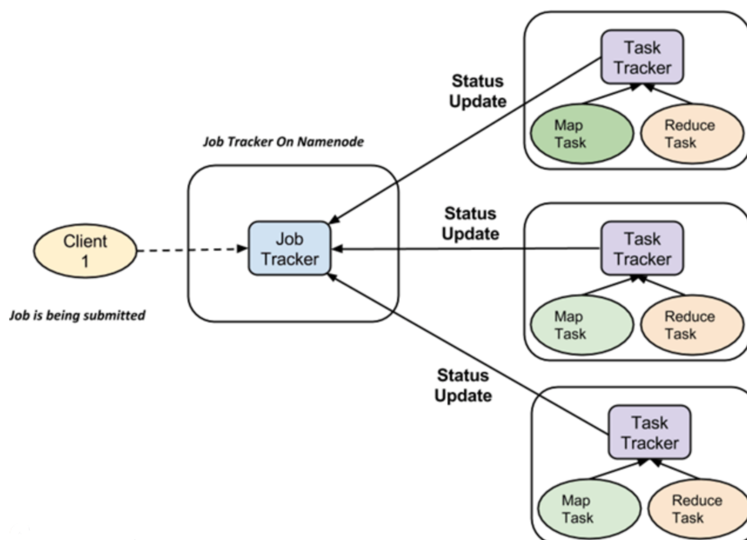
RF je možné meniť podľa požiadaviek, ktoré sú na systém kladené. Čím nižšie je nastavený RF, tým sa kladú menšie požiadavky na úložný priestor pre dáta. Napríklad ak máme dostupných 300 TB úložného priestoru, tak pri použití RF 3 sa nám zmenší reálna kapacita úložiska na 100 TB.



Obr. 3.2: HDFS distribúcia dát. Zdroj: Hadoop – What is Hadoop

JobTracker

JobTracker je služba, ktorá zabezpečuje priame spojenie medzi klientskou aplikáciou a Hadoop clusterom. Táto služba ovláda úlohu *MapReduce* 4.1 a má nad ňou plnú kontrolu. Obdrží prácu (*Job*) od klienta, rozdelí to na menšie úlohy (*Task*), priraďuje úlohy jednotlivým uzlom (*TaskTracker*) a monitoruje ich súčasný stav. Pokiaľ nastane na nejakom uzle zlyhanie pri riešení úlohy, reštartuje riešenú úlohu a pošle požiadavku inému uzlu. Spravidla beží služba JobTracker na rovnakej stanici ako NameNode, ktorý priamo zadáva a kontroluje stavy jednotlivých uzlov. JobTracker má teda neustále aktuálne informácie ohľadom HDFS. Tento proces je znázornený na obrázku 3.3.



Obr. 3.3: JobTracker a TaskTracker. Zdroj: MapReduce - How it works

Priebeh JobTrackeru:

- Klientská aplikácia pošle požiadavku na JobTracker
- JobTracker požiada NameNode k určeniu pozície požadovaných dát
- JobTracker zistí pozíciu TaskTracker uzlov s dostupnými slotmi v blízkosti dát
- JobTracker pošle vybraným uzlom TaskTrackeru prácu
- Uzly TaskTrackeru sú sledované. Pokiaľ uzol neposiela heartbeat v pravidelnom intervale, tak je považovaný za neaktívny a úloha je priradená inému TaskTrackeru.
- Pokiaľ zlyhá nejaká operácia na TaskTrackeri, oznámi chybu JobTrackeru. JobTracker danú úlohu buď prepošle na iný uzol, zamietne vykonať akciu alebo označí daný uzol ako neaktívny.
- Pokiaľ je zadaná úloha na TaskTrackeri hotová, tak JobTracker aktualizuje svoju dostupnosť.
- Klientská aplikácia môže získať informácie z JobTrackeru [18].

3.2.2 Paralelné spracovanie dát

Paralelné spracovanie dát môžeme rozdeliť na základe dvoch architektúr. Paralelné spracovanie na zdielanej pamäti a paralelné spracovanie na distribuovanej pamäti.

Systémy so zdieľanou pamäťou

Pri paralelnom spracovaní na zdieľanej pamäti komunikujú procesory prostredníctvom jednej pamäti, ku ktorej majú spoločný prístup. Jednotlivé procesory tak môžu byť špecializované na určitú úlohu. Nevýhoda tohto systému je, že procesory musia synchronizovať svoje aktivity na spoločnej linke, pretože v jednu chvíľu môže do pamäti vstupovať iba jeden procesor, Rieši sa problém exkluzívneho prístupu do pamäti.

Systémy s distribuovanou pamäťou

Ide spravidla o veľké množstvo strojov, ktoré sú medzi sebou spojené komunikačnou linkou. Každý procesor pracuje s vlastnou pamäťou, teda odpadá problém riešenia exkluzívneho prístupu do pamäte. Musí sa ale riešiť problém v komunikácii medzi jednotlivými procesormi. A ako už názov napovedá, tak tento typ systému s distribuovanou pamäťou je využívaný v HDFS.

Kapitola 4

Hadoop ekosystém

Táto kapitola je zameraná na technológie softvérového systému Apache Hadoop, ktorý nám bude uľahčovať prácu pri spracovaní BigData. Hadoop má ekosystém, ktorý sa vyvinul z jeho troch základných vrstiev: *spracovanie, riadenie zdrojov a ukladanie*. Predstavené budú komponenty ekosystému Hadoop a spôsob, akým vykonávajú svoje úlohy počas spracovania BigData. Ďalej si v jednotlivých vrstvách predstavíme obecné funkcie komponentov, čím sa priblíži význam každej vrstvy. Potom si porovnáme najznámejšie a najvyužívanejšie distribúcie postavené na Apache Hadoop, aby sme si vybrali najvhodnejší pre náš systém.

4.1 Základné komponenty Hadoop

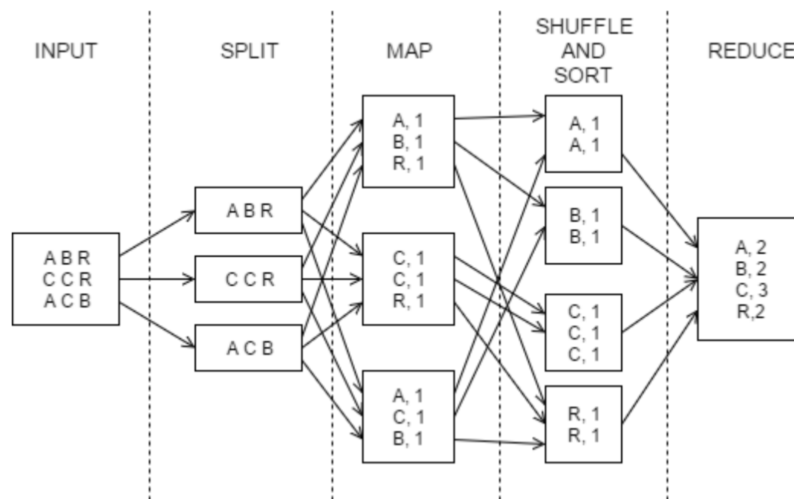
Ekosystém Hadoop neustále rastie a rozširuje sa, aby uspokojil potreby spracovávania BigData. Avšak skladá sa zo štyroch základných komponentov a tie ostatné závisia aj od konkrétnej distribúcie.

HDFS a jeho architektúru sme už podrobne rozobrali v predchádzajúcej časti 3.2. Je hlavnou komponentou alebo inak povedané chrbticou Hadoop ekosystému. HDFS je ten, ktorý umožňuje ukladať rôzne typy veľkých dátových súborov (t.j. štruktúrované, neštruktúrované a čiastočne štruktúrované dáta). HDFS vytvára úroveň abstrakcie nad zdrojmi, odkiaľ môžeme vidieť celý systém HDFS ako jednu jednotku.

4.1.1 MapReduce

MapReduce je programovací model, ktorý je navrhnutý pre paralelne distribuované spracovanie veľkých dátových sád. Principiálne vychádza z bežne používaných funkcií **mapovania a redukcie** v kombinácii s paralelným prevedením metódy zvanej "*rozdeľuj a panuj*". Procesy založené na tomto modeli postupne prechádzajú štyrmi stavmi: rozdelenie vstupných dát, mapovanie, zamiešanie/zoradenie a redukcia, viď obrázok 4.1.

Z programátorského hľadiska sú však dôležité iba mapovacie a redukčné funkcie, pretože iba tie je možné užívateľsky špecifikovať. Behové prostredie sa stará o detaily rozdelenia vstupných dát, riadenie, vyrovnanie sa s chybami a riadenie vnútornej komunikácie medzi strojmi. Mapovacia funkcia spracováva páry kľúč/hodnota na vytvorenie množiny nezávislých ďalších párov kľúč/hodnota. Redukčná funkcia má všetky hodnoty priradené k rovnakému kľúču. Hlavnou výhodou tohto modelu je schopnosť *shared-nothing*, čo znamená, že všetky mapovacie funkcie môžu byť vykonávané nezávisle na sebe. Táto schopnosť umožňuje paralelný beh programu na viacerých nespoľahlivých strojoch s tým, že sa úloha vyrieši vo veľmi krátkom čase [17].



Obr. 4.1: MapReduce princíp

4.1.2 YARN

YARN sa považuje za mozog Hadoop ekosystému. Vykonáva všetky úlohy určené na spracovanie, tým že alokuje zdroje a plánuje úlohy. Skladá sa z dvoch hlavných častí, ktorými sú ResourceManager a NodeManager.

ResourceManager je hlavný uzol pri spracovávaní. Obdrží požiadavku na spracovanie a posunie časti danej požiadavky ďalej na odpovedajúce NodeManagery, kde sa vlastne spracovanie vykoná. **NodeManager** je nainštalovaný na každom jednom DataNode a je zodpovedný za spúštanie úloh. **Scheduler** vykonáva plánovacie algoritmy a alokuje zdroje. **ApplicationManager** akceptuje odoslanie úlohy, vyjednáva kontajnery (DataNode, kde sa proces vykonáva) na vykonanie *ApplicationMaster* na základe špecifickej aplikácie a monitoruje progres. *ApplicationMaster* je deamon, ktorý sa nachádza na kažom DataNode a komunikuje s kontajnermi na vykonávanie úloh [17].

4.1.3 Common

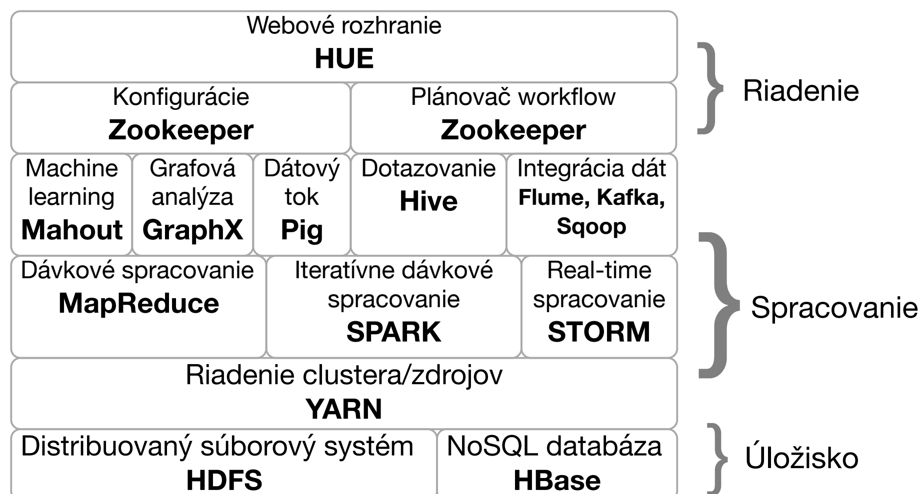
Súbor spoločných nástrojov potrebných pre ostatné moduly Hadoop. Má natívne zdieľané knižnice, ktoré obsahujú implementácie jazyka Java, I/O nástroje a detekciu chýb. Zahrnuté sú aj rozhrania a nástroje na konfiguráciu *rack awareness*[1], autorizáciu používateľov *proxy serverov*, autentifikáciu, autorizáciu na úrovni služby, dôvernosť údajov a server Hadoop *Key Management Server (KMS)*[2] [3].

4.2 Vrstvy ekosystému

Ekosystém Hadoopu je postavený z komponentov jadra, ktoré sme predstavili v predchádzajúcej sekcii a z množstva projektov, ktoré sú postavené na týchto komponentách. Tieto projekty boli navrhnuté pre pomoc výskumníkom a praktikantom vo všetkých aspektoch typickej analýzy dát a strojového učenia. Niektoré spoločnosti ako napríklad *Cloudera*[3],

[1] Odolnosť voči chybám, keď sa jeden blok repliky z jedného nodu kopíruje na druhý
[2] https://hadoop.apache.org/docs/current/hadoop-kms/index.html
[3] https://www.cloudera.com/

MapR[4] alebo *Hortonworks*[5] ponúkajú distribúcie, ktoré využívajú a spájajú niekoľko týchto projektov.



Obr. 4.2: Vrstvy Hadoop ekosystému

Ako už bolo spomenuté, obecná štruktúra ekosystému sa skladá z **3 vrstiev**. Príklad toho, aké technológie sa môžu využiť v jednotlivých vrstvách je zobrazené na obrázku 4.2. Špecifické projekty uvedené na tomto obrázku sú uvedené ako vzorové projekty, ktoré sú často v rôznych distribúciách využívané. Samozrejme vytvorených projektov vykonávajúce dané úlohy je oveľa viac, ale nás budú zaujímať len technológie vybranej distribúcie.

Úložisková vrstva

Úložisková vrstva, kde prebieha ukladanie spracovaných dát sa nachádza na najnižšej vsrtve ekosystému. Štandardne je tu použité HDFS, ktoré už bolo popísané 3.1. Ale sú aj iné možnosti distribuovaných systémov, ktoré by sa mohli použiť, či už systémy postavené na HDFS alebo samostatné systémy. Avšak HDFS je známe pre svoje skvelé vlastnosti pri škálovaní, pri odolnosti voči chybám a pre ukladanie historických dát, ktoré nepotrebujú byť často používané.

Spracovávacia vrstva

Pri spracovávaní dát prebieha analýza. Základom tejto vrstvy je *YARN* 4.1.2, ktoré umožňuje jednému alebo viacerým spracovávacím jednotkám bežať na Hadoop clustri. Navyše k spracovávacím jednotkám, táto vrstva zahŕňa aj niekoľko ďalších iných nástrojov, ktoré môžu byť využité pre *machine learning* alebo analýzu dát. Okrem toho obsahuje táto vrstva nástroje na pohyb a interakciu údajov. Vykonávajú sa tu operácie ako **zber dát, agregácie, sprostredkovanie logových údajov** do HDFS a prenos dát medzi HDFS a relačnou databázou. Ďalej sa v tejto vrstve nachádzajú dotazovacie jednotky, pomocou ktorých sa dotazujeme na dáta v HDFS a v NoSQL databáze. Nevýhodou pri využívaní MapReduce je, že mnoho algoritmov sa ťažko prekladá na vzor MapReduce. Preto v tejto vrstve existujú nástroje, ktoré vytvárajú vysoko-úrovňovú abstrakciu, ktorá skrýva zložitosť MapReduce

[4] https://mapr.com/
[5] https://hortonworks.com/

modelu, čím sa zjednodušuje programovací proces. Podobné nástroje existujú aj pre *real-time* spracovanie údajov.

Riadiaca vrstva

Vrstva riadenia obsahuje nástroje na interakciu s používateľmi a na vysoko-úrovňovú organizáciu. Patrí sem plánovanie, monitorovanie, koordinácia a používateľské rozhranie. **Plánovač workflow** spravuje práce pre rôzne nástroje v riadiacej vrstve, vrátane riadiacich jednotiek. Pre komplexné procesy, ktoré vyžadujú viacero prác a nástrojov, špecifikuje sekvenciu akcií a skoordinuje ich aby sa mohli úlohy správne vykonať. Ďalšia služba poskytuje nástroje pre riadenie koordinácie údajov a protokolov, a je schopný zvládnuť čiastočné poruchy siete, ktoré sú bežné v distribuovaných systémoch. Pre väčšinu komponentov Hadoop ekosystému existuje **webové rozhranie**.

4.3 Distribúcie Apache Hadoop

Hadoop predstavil nový spôsob, ktorý zjednodušuje analýzu veľkých dátových súborov a vo veľmi krátkom čase zmenil trh s BigData. V súčastnosti je Hadoop často synonymom pre výraz BigData. Keďže projekt Hadoop je open-source projekt, viaceré spoločnosti vyvinuli vlastné distribúcie, pridali nové funkcie alebo zlepšili jadro. Pokiaľ ide o distribúcie spoločnosti Hadoop, existujú tri spoločnosti, ktoré vynikajú v ich práci, sú to: *Cloudera, MapR a Hortonworks.*

Cloudera existuje najdlhšiu dobu. Hortonworks prišiel neskôr. Zatiaľ čo Cloudera a Hortonworks sú open-source, väčšina verzií MapR je vybavená **proprietárnymi modulmi**. Každý distribútor má svoje silné aj slabé stránky, z ktorých každá má určité prekrývajúce sa funkcie. Aby sme mali možnosť najväčšej využiteľnosti Hadoopu pri spracovaní dát, je zmysluplné tieto **top 3 distribúcie** porovnať.

Všetky 3 distribúcie ponúkajú konzultácie, tréning a technickú podporu pre svoje produkty. Avšak narozdiel od svojich 2 rivalov, distribúcia od Hortonworks sa ako jediná považuje byť za **100% open-source**. Cloudera obsahuje vo svojej distribúcii verzii *Enterprise 4.0* množstvo vlastných patentovaných prvkov, ktoré pridávajú vrstvy administratívnych a riadiacich funkcií do jadra systému Hadoop. Keď sa pozrieme na MapR, tak zistíme, že v jeho distribúcii je HDFS nahradené vlastným **patentovaným** súborovým systémom *MapRFS*, ktoré by malo zabezpečiť lepšie vlastnosti. Tým sa MapR stáva lepším pri nasadení do produkcie. Po edíciu *M3*, MapR je zdarma, ale v neplatenej verzii chýbajú niektoré základné komponenty. Tým by sme **MapR vylúčili** a pozreli sa na spoločné a odlišné vlastnosti Cloudera a Hortonworks distribúcií.

Výber medzi distribúciami Cloudera a Hortonworks

Pri rozhodovaní hrajú najväčšiu rolu odlišné vlastnosti dvoch distribúcií. Cloudera a Hortonworks sú odlišné v týchto vlastnostiach:

- Cieľ Cloudery je zbaviť sa potreby dátového skladu. Kým Hortonworks chce zostať poskytovateľom Hadoop distribúcie.
- Distribúcia Cloudery obsahuje komponenty, ktoré by teoreticky mali zvládať niektoré riadiace a monitorovacie úlohy lepšie. Hortonworks však tiež obsahuje komponenty, ktoré podobne zvládajú tie isté úlohy.

- Kým niektoré komponenty Cloudery nie sú zadarmo, všetky kompontenty distribúcie Hortonworks sú neplatené. Avšak Cloudera ponúka 60 dňovú trial verziu, v ktorej sa dané komponenty môžu vyskúšať.

Cloudera je staršia a je viac vyvinutá než Hortonworks. Ale Hortonworks rýchlo dobieha a v nedávnej minulosti priniesla **viac inovácií** v ekosystéme Hadoop. Čo je pre nás dôležité - Hortonworks je neplatené a 100% open-source [28].

4.4 Distribúcia Hortonworks

`Hortonworks Data Platform (HDP)` je distribúcia Hadoop systému, ktorého cieľom je zjednodušiť nasadenie a riadenie Hadoop clustra, a zabezpečuje kompatibilitu jednotlivých komponentov. Architektonickým centrom tohto systému je YARN, vďaka ktorému poskytuje rôzne metódy spracovania údajov - dávkové aj real-time spracovanie. Podporované sú aj funkcie dôležité pre podnikový produkt, ako napríklad: riadenie, bezpečnosť a prevádzka. HDP podporuje aj experimenty a vyhodnocovanie, tak že poskytuje virtuálny stroj `The Hortonworks Sandbox`, ktorý predstavuje implementáciu jedného uzla. Zahŕňa aj výukový program a aplikácie, pomocou ktorých získate prehľad o komponentách a naučíte sa s nimi pracovať. Zistíte ako vložiť údaje do systému a ako ich spracovať pomocou nástrojov. Všetky komponenty tohto systému sú *open-source*. Popis nasledujúcich komponent vychádza z oficiálnej dokumentácie HDP [15]. Na nasledujúcom obrázku 4.3 je zobrazené vzorové použitie systému Hortonworks s použitím niektorých komponent ekosystému a tok dát medzi vrstvami.

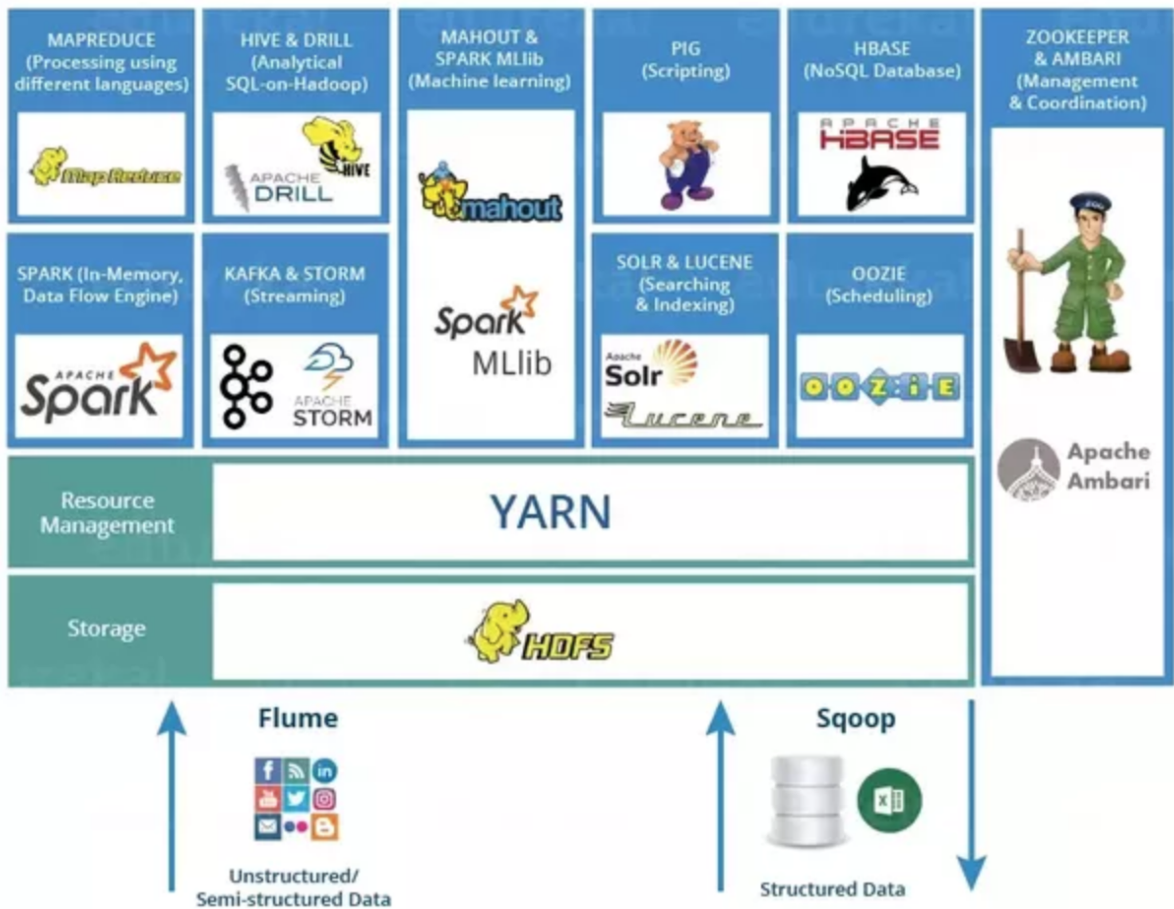
4.4.1 Apache Zookeeper

Po inštalácii HDP by sa malo ako ďalšie inštalovať Apache Zookeeper, kvôli závislostí ostatných komponentov na tejto komponente. Apache Zookeeper je server, ktorý spoľahlivo koordinuje distrubuované procesy. Poskytuje službu distribuovanej konfigurácie, synchronizačnú službu a register názvov pre distribuované systémy. Distribuované aplikácie používajú Zookeeper na ukladanie a sprostredkovanie aktualizácií ohľadom dôležitých informácií o konfigurácii.

4.4.2 Apache HBase

Po inštalácii Apache Zookeeper by sa mali inštalovať komponenty jadra Hadoopu, ktoré boli predstavené v sekcii 4.1. Ďalšou komponentou pri inštalácii je potom Apache HBase. NoSQL databáza stĺpcového typu 3.1.2, ktorá je postavená nad HDFS a poskytuje real-time prístup k dátam. Je navrhnutý tak, aby bol ľahko škálovateľný a podporoval veľký objem dát s náhodným prístupom pre rôzne aplikácie. Dokáže kombinovať rôzne štruktúry dát. Vďaka tomu, že je HBase integrovaný v Hadoope, dokáže bez problémov spolupracovať s inými komponentami **sprístupňujúce dáta cez YARN**. Nad HBase existuje nadstavba - komponenta, ktorá vytvára abstrakciu nad úložiskom a umožňuje SQL dotazovanie pomocou *JDBC*[6] ovládača. `Apache Phoenix` poskytuje funkcie, ako sú sekundárne indexy, ktoré pomáhajú urýchliť dotazy bez závislosti na konkrétny návrh riadkov a kľúčov.

[6] **JDBC** - je API pre programátorov v programovacom jazyku Java, ktoré definuje jednotné rozhranie pre prístup k relačným databázam.



Obr. 4.3: Hortonworks ekosystém. Zdroj: What is a Hadoop ecosystem

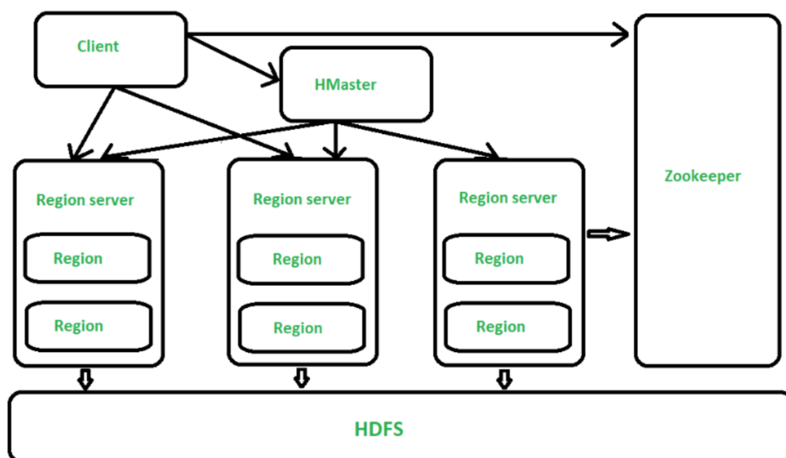
Uloženie dát v tabuľkách by sa dalo prirovnať k uloženým dátam v bežných SQL databázach. Tabuľka sa skladá zo stĺpcov, ktoré sú spojené do tzv. *column families*. Tieto skupiny vytvárajú sémantické hranice medzi dátami riadkov. Pri vytváraní HBase tabuľky sa definujú práve tieto column families. Samotné stĺpce sa vytvárajú až pri vkladaní dát. Každý riadok má svoj jedinečný identifikátor, tzv. *row key*. Riadky sú vždy usporiadané lexikálne práve podľa tohto identifikátoru. Row key má v HBase tabuľke rovnakú úlohu ako primárny index v klasickej relačnej databáze. HBase neodporúča meniť už vytvorené column families a mať z nich príliš veľa. Ideálne množstvo column families je 3. Ale jeden column family môže mať v sebe aj milióny stĺpcov. Štruktúra tabuľky je znázornená na obrázku 4.4 [4].

RowKey	ColumnFamily1		ColumnFamily2	
	Column1	Column2	Column3	Column4
Row1	KeyValue1	KeyValue2	KeyValue3	KeyValue4

Obr. 4.4: Štruktúra tabuľky v Hbase.

Architektúra

Systém sa skladá z troch druhov zariadení: *RegionServer*, *HMaster* a *Zookeeper*. Architektúra HBase databáze je založená na princípe *master-slave*. Regióny sú distribuované medzi uzlami Hadoop clusteru. *RegionServer* je zodopovedný za ukladanie regiónov a tiež poskytuje regióny pre úpravy a čítanie. Jeden server môže spravovať viacero regiónov. *HMaster* rozhoduje o tom, ktoré regióny sa uložia do ktorého RegionServeru. Ďalej je zodpovedný za vytváranie, upravovanie a mazanie HBase tabuľky. Pre správne fungovanie distribuovaného systému je nutné udržiavať stav jednotlivých uzlov clusteru. Zisťovanie informácií o stave uzlov je úlohou komponenty tzv. *Zookeeper* 4.4.1. Dáta sú fyzicky uložené v RegionServeroch. HBase systém obsahuje viacero RegionServerov, kde je každý zodpovedný za svoje regióny. RegionServer umožňuje klientom čítať dáta, ktoré obsahuje a zapisovať nové dáta do regiónov. Keďže klient komunikuje priamo so servermi obsahujúcimi fyzické dáta, čítanie aj zápis je možné vykonávať paralelne.



Obr. 4.5: Hbase architektúra. Zdroj: Architecture of HBase

OpenTSDB

OpenTSDB je nástroj na ukladanie a vyhľadávanie v dátach časových radov 2.3. Dáta časových radov je druh dát, ktoré spravidla predstavujú vlastnosti pozorovaného objektu zbieraných opakovane v rámci krátkych časových úsekov, čo umožňuje sledovať vývoj pozorovaného objektu v čase. OpenTSDB sa skladá z niekoľkých častí. Hlavnú časť tvorí démon *tsd*, ktorý slúži ako zberný uzol pre prichádzajúce dáta. Dáta sú získavané sadou sond nazývaných kolektory. **Kolektory** zbierajú dáta z výpočtových uzlov, na ktorých sú nasadené a za pomoci jednoduchého HTTP API odosielajú dáta tsd démonu. OpenTSDB neprichádza s vlastným úložným riešením. Na ukladanie získaných dát **využíva NoSQL databázu HBase**, ktorá slúži na ukladanie veľkých dát. Pri získavaní uložených dát ponúka OpenTSDB možnosť využitia **agregačných funkcií** (suma, priemer, maximum, minimum, atď.). Výpočet prebieha priamo nad HBase za pomoci *MapReduce* úloh 4.1, čo prináša efektívnu paralelizáciu a vysoký výkon. Veľkosť väčšiny databáz na ukladanie dát časových radov je fixná. To znamená, že v určitom momente na uloženie nových dát musia byť najskôr staré dáta zmazané alebo musí byť znížená ich granularita. OpenTSDB takto **obmedzená nie je**. Veľkosť dát, ktoré môže uchovávať je limitovaná len veľkosťou HBase

databáze, ktorú využíva. HDFS, na ktorom HBase beží, sa zároveň automaticky stará o **replikáciu databáze** na niekoľko nezávislých uzlov, čo zabraňuje strate dát pri výpadku niektorého z nich.[22]

4.4.3 Apache Hive

Nástroj na vytváranie SQL dotazov pomocou jazyka *HiveQL*, ktorý je v Apache Hive využívaný a môže byť tiež kompilovaný do sekvencie príkazov v MapReduce programe. Apache Hive je vlastne štandard pre SQL dotazy v Hadoop systéme. Analytici využívajú Hive na dotazovanie, sumarizáciu, preskúmanie a analýzu údajov, ktoré sú uložené v Hadoope v rôznych formátoch a veľkostiach, aby potom mohli vytvoriť prehľady. Spolu s Apache Hive sa zvykne inštalovať aj komponent `Apache HCatalog`, ktorý vytvára abstrakciu metadát a teda oddeľuje užívateľov od informácií, kde a ako sú dáta fyzicky uložené.

4.4.4 Apache Pig

Pig vznikol kvôli niektorým nedostatkom MapReduce systému. Medzi nedostatky MapReduce patria:

- veľká jednoduchosť, ktorá sa stáva problémom pri písaní mnoho-krokových transformácií
- nepodporuje komplexné mnoho-krokové dátové prúdy a kombinované spracovanie viacerých súborov dát
- problém písania každej jednoduchej operácie, to vytvára miesto pre vznik chýb a zbytočne zvyšuje trvanie analýzy dát

Tieto nedostatky vytvorili priestor pre vznik systému Pig, ktorý slúži na písanie komplexných MapReduce transformácií použitím jednoduchého skriptovacieho jazyka zvaného *Pig Latin*[7]. Pig bol navrhnutý aby dokázal vykonávať série operácií a preto je veľmi vhodný pre extrahovanie, transformáciu a ukladanie dát, iteratíve spracovanie dát a prieskum neštrukturovaných dát. Pig je voľne rozšíriteľný vlastnými funkciami, jednoduchý na programovanie a má vlastnú optimalizáciu vykonávania úloh. Program sa skladá z **troch základných častí**, Pig Latin programovacieho jazyka, prostredia pre vykonávanie programov Pig a repozitára užívateľom definovaných funkcií. Po komponente Pig sa inštaluje `Apache WebHCat`, ktorý poskytuje rozhranie *REST API*, cez ktoré sa dotazujeme na služby HCatalogu.

4.4.5 Apache Oozie

Apache Oozie je webový nástroj pre Hadoop, ktorý slúži na plánovanie a koordináciu jednotlivých úloh. Pôvodne bol vyvinutý pre **správu zložitých úloh**. Príklad takejto úlohy môže byť nasledovný. Najprv sa spracuje MapReduce úloha, následne Pig úloha a nakoniec sa zobrazí výsledok pomocou Hive dotazu. Oozie spolupracuje priamo so službou JobTracker, takže úloha bude rovnomerne rozdelená medzi všetky uzly. Plány úloh na seba môžu naväzovať, môžu byť spustené manuálne alebo mať nastavený časovač spustenia. Plány sa zapisujú do súboru XML pomocou jazyka *hPDL - Hadoop Process Definition Language*[8].

[7] https://pig.apache.org/docs/latest/basic.html

[8] Jazyk, ktorý bol špeciálne navrhnutý pre Oozie, je podobný jazyku XML.

4.4.6 Apache Sqoop

Sqoop je nástroj, ktorý je navrhnutý pre efektívne **presúvanie veľkých objemov dát** medzi systémom Hadoop a štruktúrovanými dátovými úložiskami ako sú napríklad relačné databázy. Na rozdiel od systému *Flume* 4.4.8 tento program funguje aj opačným smerom, čo znamená že dokáže vyberať dáta so systému Hadoop a ukladať ich do štruktúrovaných dátových úložísk. Sqoop je navrhnutý s ohľadom na import dát z externých zdrojov do systému Hadoop, paralelné prenosy pre vysoký výkon pre optimálne vyťaženie systému a presunutie nadmerného zaťaženia na externé systémy. Priebeh prenosu je veľmi jednoduchý, pretože systém Sqoop narozdiel od systému *Flume* **presúva dáta s pevnou štruktúrou** a tak je len malá šanca, že nastane niečo neočakávané počas prenosu. Taktiež architektúra tohto systému je veľmi jednoduchá z rovnakých dôvodov. Program sa skladá so súboru obsahujúcej metadáta o štruktúre prenášaných dát a z Hadoop úlohy `map()`, ktorú Sqoop odošle následne na cluster pre vykonanie. Sqoop pracuje s relačnými databázami ako napríklad: *MySQL*[9], *Postgres*[10], *Teradata*[11] a ďalšie.

4.4.7 Apache Mahout

Mahout je knižnica, ktorá obsahuje algoritmy zaoberajúce sa **strojovým učením**. V súčasnej dobe existuje táto knižnica v jazyku Java. Zameriava sa na tri okruhy strojového učenia: doporučujúce algoritmy, zhlukovanie dát a klasifikácia. Výhodou Mahoutu je jeho škálovateľnosť - môžeme ho využiť jak na malé objemy dát (rádovo KB), tak aj na veľké (rádovo TB), pretože k spracovávaniu môžeme využívať viac strojov zároveň.

4.4.8 Apache Flume

Flume je definovaný ako distribuovaná, spoľahlivá a dostupná služba pre efektívne zbieranie, agregáciu a presúvanie veľkého toku dát z rôznych zdrojov do centralizovaného úložiska. Najbežnejšie sa používa pre **transport záznamových dát**, avšak nie je obmedzený iba na to. Dokáže presúvať takmer akékoľvek dáta z akéhokoľvek zdroja, taktiež dokáže ukladať súbory do rôznych systémov Hadoop, ako napríklad do HDFS alebo do HBase. Používateľom tohto systému ponúka ukladanie dátových tokov z viacerých zdrojov do distribuovaného systému na analýzu, zbieranie záznamových súborov **veľkého objemu v reálnom čase**, garanciu doručenia dát, protekciu pred výkyvmi v prenose ktoré môžu presiahnuť zapisovaciu schopnosť cieľa a vysokú pružnosť pri prenose.

Prúd toku dát v komponente Flume sa skladá z nasledujúcich častí:

- **Event** *(udalosť)* - základná jednotka údajov, ktorá sa presúva pomocou komponenty Flume. Je to zvyčajne malá správa, ktorá sa skladá z *hlavičky* a tela *byte-array*.
- **Source** *(zdroj)* - prijíma udalosti z externého zdroja, ktoré potom ukladá do kanála. Typ udalosti, ktorý sa bude prijímať musí byť definovaný.
- **Channel** *(kanál)* - interné pasívne úložisko so špecifickými vlastnosťami. Udalosti je možné presúvať v pamäti, v súboroch alebo pomocou využitia externej komponenty. Vďaka tomuto kanálu môže Sink a Source bežať asynchrónne.

[9] https://www.mysql.com/
[10] https://www.postgresql.org/
[11] https://www.teradata.com/

- **Sink** *(prepadlisko)* - odoberá udalosti z kanálu a predáva ich ďalej, buď do HDFS úložiska alebo ďalšiemu procesu.

- **Agent** *(agent)* - kontajner pre tok dát komponenty Flume. Je to bežiaci proces `flume-ng` v JVM. Každý agent musí obsahovať minimálne 1 source, channel a sink, ale môže ich obsahovať aj viac.

4.4.9 Apache Spark

Apache Spark je *framework* pre real-time výpočet v distribuovanom prostredí. Jeho výpočty prebiehajú v pamäti (*in-memory*) pre dosiahnutie vyšších rýchlostí než MapReduce. Uvádza sa, že je **100-krát rýchlejší než MapReduce** vďaka spracovaniu v pamäti a ďalším optimalizáciám [29]. Na obrázku 4.6 je zobrazená štuktúra komponenty Apache Spark. Knižnica je dostupná aj v jazyku Java a ponúka ešte ďalšie knižnice ako nadstavbu nad Sparkom pre uľahčenie práce. Jadro Sparku je tvorené štyrmi základnými knižnicami : `Spark SQL, Spark Streaming, MLlib, GraphX`.

- **Spark SQL** - knižnica na prácu so štrukturovanými dátami. Jej súčasťou sú konektory na rôzne databázy, či už relačné alebo NoSQL. Spark SQL prináša do Sparku možnosť písania dotazov nad dátami v SQL jazyku. Vo výsledku sa SQL dotazy transformujú na MapReduce programy a Spark sa postará o ich výpočet.

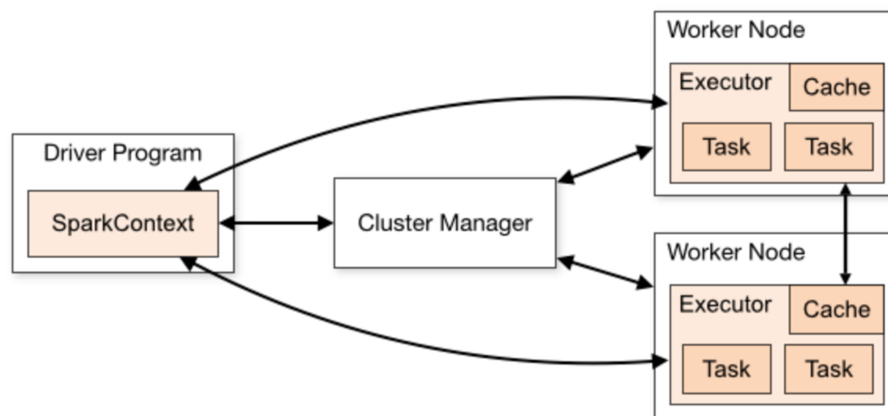
- **Spark Streaming** - knižnica, ktorá umožňuje spracovať a analyzovať dáta v reálnom čase. Vstupné dáta, ktoré vchádzajú na spracovanie do Spark Streamingu sa rozdelia na snímky zvané *DStream (discretized stream)*. Každá snímka obsahuje dáta za určené časové obdobie. Nad DStreamami môžeme následne robiť *analýzy a agregácie*, pričom je možné využiť *akcie a transformácie* 4.4.9. Jednotlivé výsledky môžeme ukladať na rôzne úložiská, alebo ich ďalej spracovávať.

- **MLlib** - knižnica, ktorá obsahuje distribuované algoritmy pre strojové učenie a štatistiku. Vzhľadom na to, že väčšina algoritmov pre strojové učenie je založená na princípe iteratívneho prechádzania tej istej dátovej štruktúry, môžu tieto algoritmy ťažiť zo schopnosti Sparku - uložiť všetkých dáta do operačnej pamäte. Čím sa vyhýbajú potrebe dáta znovu načítať z diskov. Knižnica obsahuje implementácie mnohých algoritmov pre účely klasifikácie, regresie a podobne. Má zakomponované algoritmy pre rozdelenie dát na trénovacie a testovacie množiny, ako aj implementáciu cross-validácie.

- **GraphX** - knižnica obsahujúca množstvo distribuovaných algoritmov pre prácu s grafmi.

RDD a operácie

Základným konceptom práce s dátami je odolná distribuovaná sada dát (*Resilient Distributed Dataset, RDD*). Ide o kolekciu dát rozdelených medzi jednotlivé uzly clusteru umožňujúce **paralelné operácie**. RDD môže vzniknúť zo súboru v systéme HDFS, alebo iných vhodných zdrojov, či transformáciou kolekcie v aplikácii. Je možné RDD následne uložiť do pamäti pre ďalšie spracovanie a RDD sa dokáže aj obnoviť po zlyhaní uzlu. V RDD sú 2 základné operácie: transformácie a akcie. Ako bolo v sekcii MapReduce 4.1 popísané, MapReduce je založený na operáciách `Map`, ktorá vykonala filtrovanie a triedenie a `Reduce`



Obr. 4.6: **Štruktúra komponenty Spark** - Na jednom uzle je nasadený riadiaci program (*driver*), ktorý si drží tzv. *SparkContext* a komunikuje so správcom clustera, a na dostupných strojoch (*worker nodes*) potom spustí proces (*executor*) pre každú bežiacu aplikáciu. *Driver* potom rozdeľuje dáta a úlohy medzi *executory.* Zdroj: Cluster Mode Overview

následne vykonala operáciu, ktorá vrátila výsledok. V Sparku je tento prístup **pomocou RDD** a možnosti uloženia sady do pamäti rozšírený a miesto dvojfázového spracovania je možné vykonať postupne viac výpočtov, čo v niektorých prípadoch rádovo urýchľuje čas spracovania. Vďaka tomu stále stúpa obľuba tohto nástroja pri spracovaní veľkého množstva dát.

Transformácia

Transformácie sú metódy, ktoré vracajú ako výsledok ďalšie, upravené RDD. Sú tzv. lenivo vyhodnocované (*lazy evaluation*), čo znamená, že kým nie je potreba vrátiť výsledok, je uložené iba poradie transformácií, ktoré majú byť vykonané, a nie výsledok každej z nich. Patria sem metódy ako `map` (aplikácia funkcie na každý prvok), `filter` (výber tých prvkov, ktoré spĺňajú podmienku), prieniky a zjednotenia viacerých RDD (`union, intersection`), `groupByKey`, a mnoho ďalších.

Akcia

Tieto metódy vracajú výsledok riadiacemu programu a patria sem `reduce` (agregácia pomocou pridanej funkcie), `collect` (pozbieranie napríklad vyfiltrovaných dát), `count, first, take` (výber prvých n prvkov), alebo `foreach`, ktorý pracuje s každým prvkom.

4.4.10 Apache Storm

Apache Storm je oproti Sparku **plne real-time** výpočtový systém, prebieha teda spracovanie každého záznamu hneď, keď je prijatý (tento prístup býva nazývaný *one at a time*). Štruktúra aplikácie je definovaná grafom nazvaným *topológia*. V topológii rozlišujeme *spout*, ktorý prijíma dáta z prúdu a preposiela ich ako dvojicu *kľúč-hodnota (tuple)* do *boltov*, kde prebiehajú transformácie a dáta je možné smerovať do ďalších *boltov*. Storm je z veľkej časti naprogramovaný v jazyku *Clojure*[12], ale bol navrhnutý tak, aby bolo možné pre výpočty

[12] https://clojure.org/

použiť akýkoľvek programovací jazyk. V základe nie je zaručené doručenie každej správy presne jedenkrát, ako napríklad v Sparku, ale je možné, že niektoré **správy sa doručia viackrát**. Tiež nie je zavedená žiadna správa ukladania stavu výpočtu. Tieto dve požiadavky možno splniť použitím vlastných nástrojov, alebo dostupnej vyššej úrovne abstrakcie menom `Trident`[13].

4.4.11 Apache Solr

Apache Solr a `Apache Lucene`[14] sú dve služby, ktoré sa používajú na vyhľadávanie a indexovanie v Hadoop ekosystéme. Apache Lucene je založená na jazyku Java a pomáha aj pri kontrole pravopisu. Ak Apache Lucene je motor, Apache Solr je vozidlo postavené okolo neho. Solr je kompletná aplikácia postavená na projekte Lucene. Využíva *vyhľadávaciu knižnicu Lucene Java* ako jadro pre vyhľadávanie a indexovanie.

Dokumenty sú v aplikácii Apache Solr uložené indexovaním pomocou XML, JSON, CSV alebo binárne cez protokol HTTP. Užívatelia sa môžu dotazovať na tieto dáta cez `HTTP GET` dotazy. V odpovedi môžu prijať *XML, JSON, CSV* alebo binárny súbor. Apache Solr je optimalizovaný pre vysokú návštevnosť webu.

4.4.12 Apache Ambari

Komponent, ktorý uľahčuje prácu s Hadoop ekosystémom, tým že poskytuje softvér pre *nastavenia, riadenie a monitorovanie* Apache Hadoop clustera.

- **Nastavenia** - poskytnutie postupu inštalácie (krok za krokom) pre služby Hadoopu medzi viacerými uzlami. Takisto sa tu rieši konfigurácia služieb Hadoop clustra.
- **Riadenie** - centrálna riadiaca služba, pomocou ktorej sa spúšťajú, zastavujú služby Hadoopu.
- **Monitorovanie** - poskytuje *dashboard* (v preklade "nástenka"), cez čo môžeme monitorovať stav jednotlivých služieb. Je možné aj zasielanie notifikácií v prípade zmien stavov jednotlivých uzlov.

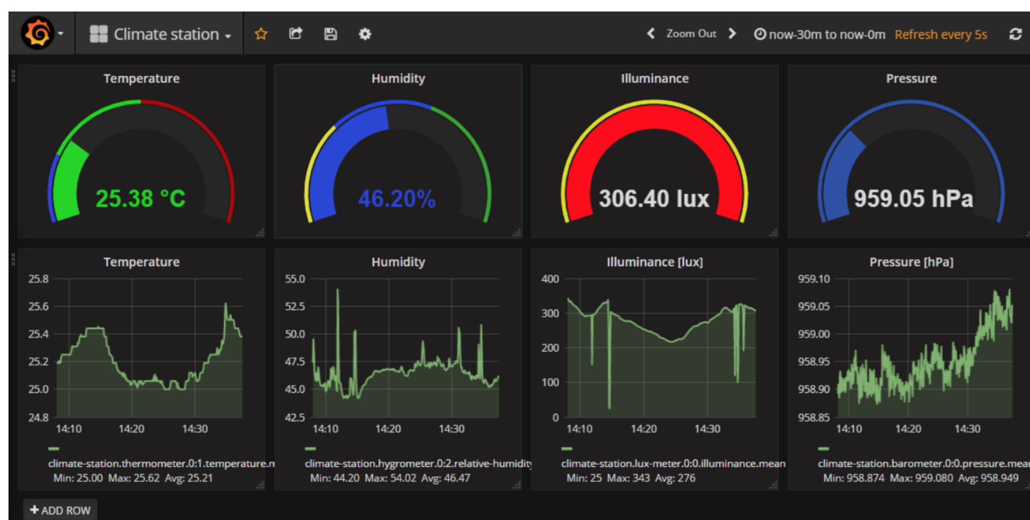
Grafana

Spolu s Ambari je v HDP nainštalovaný aj nástroj Grafana. Grafana je jedným z najrozšírenejších a najprepracovanejších viaczdrojových webových frontendov určených pre **zobrazovanie časovo usporiadaných a číselných údajov**. Podporuje integráciu viacerých úložísk určených pre časovo usporiadané dáta, ako sú napríklad *InfluxDB*[15], *OpenTSDB* 4.4.2, *Elasticsearch*[16] a ďalšie.

Grafana umožňuje vytváranie si vlastných dashboardov, do ktorých možno vkladať grafy či dlaždice s konkrétnymi hodnotami, čo je ukázané na obrázku 4.7. Podporuje aj vytváranie základných oznamovacích pravidiel pre jednotlivé grafy. V Grafane možno kombinovať viacero zdrojov dát medzi grafmi, ale aj v jednotlivých grafoch. Pomocou rozličných užívateľských modulov možno Grafanu rozšíriť o mnohé funkcionality či ďalšie zdroje dát. V Grafane je možné spustiť takzvaný **real-time mód**, ktorý začne všetky zobrazené grafy

[13] `http://storm.apache.org/releases/current/Trident-tutorial.html`
[14] `https://lucene.apache.org/core/`
[15] https://www.influxdata.com/
[16] https://www.elastic.co/

aktualizovať o nové dáta v zadanom intervale. Ďalšou užitočnou vlastnosťou nástroja Grafana je využitie metadát pre **filtrovanie hodnôt** v zobrazovaných grafoch, a ktorú sprostredkovávajú niektoré databázy určené pre časovo usporiadaná a číselné dáta [12].



Obr. 4.7: **Grafana** - na obrázku je zobrazené webové rozhranie Grafany, na ktorom sa nachádza dashboard s rôznymi grafmi. Zdroj: Grafana for Visualization

4.4.13 Apache Ranger

Apache Ranger prináša komplexný prístup pre bezpečnosť v Hadoop clusteri. Poskytuje centralizovanú platformu pre definovanie a **správu bezpečnostných politík** v rámci komponentov Hadoopu. Pomocou konzoly Apache Ranger môžu správcovia zabezpečenia ľahko spravovať pravidlá prístupu k súborom, priečinkom, databázam, tabuľkám alebo stĺpcom. Tieto pravidlá je možné nastaviť pre jednotlivých používateľov alebo skupiny a následne presadzovať konzistentne v rámci balíka *HDP*.

Pomocou konzoly Apache Ranger môžu správcovia zabezpečenia ľahko spravovať práva prístupu k súborom, priečinkom, databázam, tabuľkám alebo stĺpcom. Tieto **pravidlá** je možné nastaviť pre jednotlivých používateľov alebo skupiny a následne šíriť konzistentne v rámci balíka HDP.

Kapitola 5

Návrh systému

Táto práca sa zameriava na spracovanie dát produkovaných z rôznych IoT sietí, s bližším zameraním na sieť *IQRF* 2.4. Dáta produkované z IoT sietí ukladáme a spracovávame z rôznych dôvodov, napríklad chceme: sledovať zmeny v systéme, vytvárať štatistiky z historických dát, vykonávať strojové učenie z pozbieraných dát a podobne.

V tejto kapitole sa priblíži proces spracovania dát z IoT zariadení. Zoznámite sa so schémou spracovania takýchto údajov a postupne v ďalších sekciách rozoberieme jednotlivé časti schémy podrobnejšie.

Pre vytvorenie systému pre zbieranie BigData z rozsiahlych IoT sietí sme vybrali distribúciu *Hortonworks* 4.4. Pre správny návrh systému je potrebné si naštudovať celý ekosystém, aby sme dokázali vybrať pre nás potrebné komponenty a správne ich navrhnúť.

5.1 Proces spracovania dát

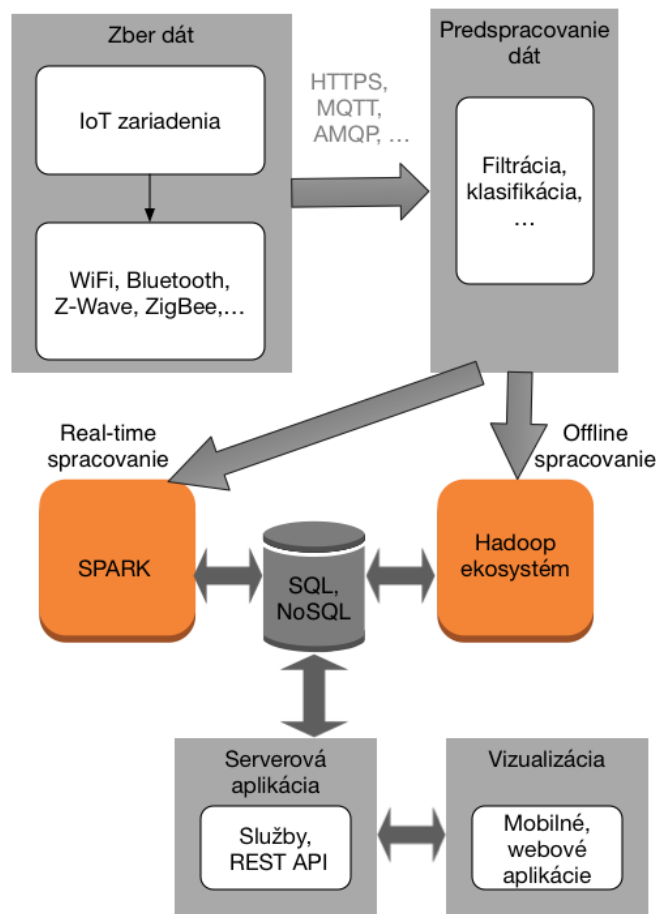
Samotný proces spracovania dát zahŕňa: zber dát, predspracovanie, real-time alebo offline spracovanie, ukladanie dát, analýzu nad uloženými dátami a následne serverovú aplikáciu, ktorá s týmito dátami bude ďalej pracovať a vystavovať ich pre aplikácie slúžiace na vizualizáciu dát. Proces je zobrazený na obrázku 5.1, kde šípky znázorňujú tok dát.

Proces začína **zberom dát** z IoT zariadení. Tieto zariadenia sa podľa zadania práce môžu nachádzať v rôznych IoT sietiach. Každá sieť má svoju bránu, kam zariadenia odosielajú svoje dáta. S bránou sa potom komunikuje napríklad cez protokoly *HTTPS*[1], *AMQP*[2], *MQTT 2.3*, *WebSocket 2.4*. V tejto práci nás bude zaujímať formát dát, ktorý sa cez tieto protokoly príjma z rôznych IoT sietí.

Existujú spôsoby, ktoré môžu byť použité pre dosiahnutie lepšej kvality vstupných dát z IoT zariadení. Pri **predspracovaní** dát sa napríklad snažíme odstrániť často vyskytujúci sa nežiaduci šum. Na **odstránenie šumu** môžeme aplikovať napríklad filtrovanie dát, vyhladzovanie pomocou plávajúceho priemeru (*moving average*), frekvenčnou analýzou aplikovaním *Fourierovej transformácie* a použitím ďalších tranformačných algoritmov. Použitím regresnej analýzy môžeme zisťovať trend vývoja a predpovedať vývoj hodnôt do budúcnosti [2].

Pri **spracovaní dát** použijeme dva prístupy: real-time spracovanie (*prúdové spracovanie*) a offline spracovanie (*dávkové spracovanie*). Niektoré údaje je jednoduchšie spracovávať v pravidelných intervaloch, dávkach, raz za určitý čas, napríklad pre analýzu histórie alebo

[1]https://dev.to/danielkun/where-is-https-for-iot-49ao
[2]https://www.amqp.org/



Obr. 5.1: Schéma procesu spracovania Big Data z IoT zariadení.

vyhľadávania udalosti , ktorá sa stala v minulosti. Avšak z niektorých zariadení potrebujeme mať výstupy dostupné rýchlejšie, v ideálnom prípade ihneď. Tento problém sa bude riešiť **prúdovým spracovaním**.

Prístup prúdového spracovania oproti **dávkovému** sa líši. Pri dávkovom spracovaní sa dáta dlhšiu dobu zhromažďujú a potom sa spracujú naraz, pri prúdovom spracovávaní sa dáta príjmajú zo vstupného prúdu v reálnom čase a ihneď sa s nimi pracuje. Vďaka tomu môžu byť výsledky okamžite k dispozícii. Filozofia práce s dátami je teda iná, kým pri dávkovom spracovaní máme k dispozícii obvykle veľké množstvo dát, pri prúdovom spracovaní pristupujeme k jednému prvku, alebo k veľmi malému množstvu dát. Pri dávkovom spracovaní máme na výpočet viac času, narozdiel pri prúdovom spracovaní je potrebné rýchlo vrátiť výsledok operácie. Pri prúdovom spracovaní nemáme možnosť pozrieť sa na dáta z minulosti, teda je potrebné spracovať všetky položky zo záznamu alebo tieto údaje spracovať aj pri dávkovom spracovaní. V prípade prúdového spracovania, pokiaľ spracovávané dáta nie sú navzájom na sebe závislé, tak je možné využiť paralelné spracovanie, keďže je operácia dopredu známa. Pri dávkovom spracovaní je pravdepodobná závisloť záznamov, ale tiež je možná paralelizácia. Výstup je potom **ukladaný** v *NoSQL a SQL databázach* 3.1, prípadne je preposlaný znovu na spracovanie [20].

Uložené a spracované dáta z IoT zariadení je potrebné **zobraziť pre užívateľa**. Jeden z prístupov je vytvorenie informačného systému pomocou frameworku *Spring Boot* 6.4.1,

v ktorom sa budú tieto údaje sprístupňovať a podľa potreby ďalej spracovávať v službách. Cez *REST API*[3] sa sprístupnia služby pre klientské aplikácie.

5.2 Plánovanie Hortonworks clustera

Z požiadavok zadania plynie, že výsledný systém by mal zvládnuť spracovávať dáta z veľkého množstva *IQRF čidiel*, čo pre nás znamená BigData a teda použitie Hadoop ekosystému pre spracovanie týchto dát. Na základe odporúčaní na oficálnej stránke Hortonworks [15] je správne **začať s malým clusterom** a postupne ho podľa potrieb škálovať na základe meraní skutočnej pracovnej záťaže počas pilotného projektu. Týmto spôsobom sa môže ľahko prispôsobiť pilotné prostredie bez toho, aby došlo k významným zmenám existujúcich serverov, softvéru a sieťového pripojenia.
V Hadoop ekosystéme budeme predpokladať tieto tri typy komponent:

- **Master** - HDFS NameNode, YARN ResourceManager a HBase Master
- **Slave** - HDFS DataNodes, YARN NodeManagers a HBase RegionServers
- **Single** - Zookeeper, Oozie, Flume, atď.

DataNody, NodeManagery a *HBase RegionServery* by mali byť umiestnené spoločne umiestnené pre **optimálnu lokalitu dát**. Master a slave komponenty by mali byť inštalované na separátnych uzloch, pretože sa vyskytuje **vysoká záťaž na slave uzloch** pri spracovávaní úloh a to by mohlo mať negatívny dopad na master uzol. Najmenší možný cluster, ktorý sa dá vytvoriť, sa skladá z 2 uzlov, kde sa komponenty *NameNode* a *ResourceManager* nachádzajú na master uzli, a DataNode s NodeManager na slave uzle. Clustere o veľkosti 3 a viac uzlov obvykle používajú jediný uzol pre *NameNode* a *ResourceManager* a všetky ostatné uzly sú slave uzly. Single komponenty nemajú špecifické zadelenie na ktorom uzle by sa mali nachádzať, môžu byť teda pre začiatok rovnomerne rozmiestnené na ľubovoľných uzloch v clusteri.

5.3 Zber a predspracovanie údajov

Ako už bolo spomenuté v sekcii 4.4.8, *Apache Flume* slúži pre vstup údajov do Hadoop ekosystému z vonkajších zdrojov, preto ho na tieto účely aj využijeme. Vstup údajov by mal byť umožnený z rôznych zdrojov, cez rôzne protokoly. To spôsobí, že do systému budú tiecť údaje v **rôznych formátoch**. Tieto údaje sa následne budú smerovať ďalej do *Apache Spark* 4.4.9 komponenty, kde sa budú spracovávať v reálnom čase a mali by sa aj priamo ukladať do *HDFS* 3.2 úložiska. Tieto problematiky by sa mali riešiť pri návrhu tejto komponenty.

5.3.1 Architektúra Apache Flume

Jednotlivé časti komponenty Flume sme si už rozdelili v sekcii 4.4.8. Táto sekcia sa bude zaoberať ako správne navrhnúť a nastaviť túto komponentu aby spĺňala naše požiadavky a bola škálovateľná.

[3] https://searchmicroservices.techtarget.com/definition/RESTful-API

Konfigurácia Flume agenta je uložená v lokálnom konfiguračnom súbore, ktorá je podobná konfiguračnému súboru v Jave a je uložený ako textový súbor. V jednom konfiguračnom súbore sa môže nachádzať konfigurácia jedného alebo viacerých agentov. Konfiguračný súbor obsahuje konfiguráciu každého *source, sink* a *channel* v agente a spôsob, akým sú prepojené pre vytvorenie dátového toku.

Našim plánom je vytvoriť malý **distribuovaný systém** komponenty Flume, ktorý bude zbierať udalosti v logoch. Agenty sa budú využívať ako uzly, ktoré budú zbierať dáta z externého zdroja (v našom prípade to bude *IoT gateway*). Tieto agenty budú dáta posielať ďalej na uzol, ktorý sa v tomto prípade bude tváriť ako **kolektor udalostí** a bude mať za úlohu posunúť údaje ďalej do HDFS úložiska alebo do komponenty Apache Spark. Na začiatok sa použije jeden *Agent* uzol a jeden *Collector* uzol. V nasledujúcej tabuľke 5.1 sú znázornené tieto 2 uzly.

Uzol	Source	Sink
Agent uzol	IoT GW	Collector
Collector uzol	Agenty	Spark, HDFS

Tabuľka 5.1: Source a sink pri agent a collector uzle.

Z dôvodu, že na vstup Agent uzla môžu prichádzať rôzne formáty dát z IoT gatewayu, je potrebné tieto dáta konvertovať na jednotný formát. Pre **reprezentáciu udalostí v systéme** sme vybrali binárny formát *Apache Avro*[4]. Je multiplatformový a práca s ním je vo väčšine programovacích jazykov veľmi jednoduchá, navyše je to **odporúčaný dátový formát** v komponente Apache Spark pre analýzu dát. Používa sa pre reprezentáciu základných typov hodnôt dát, ako napríklad reťazcov, desatinných čísel alebo polí obsahujúcich primitívne typy.

Oproti ostatným dátovým formátom sa vyznačuje s nasledovnými znakmi:

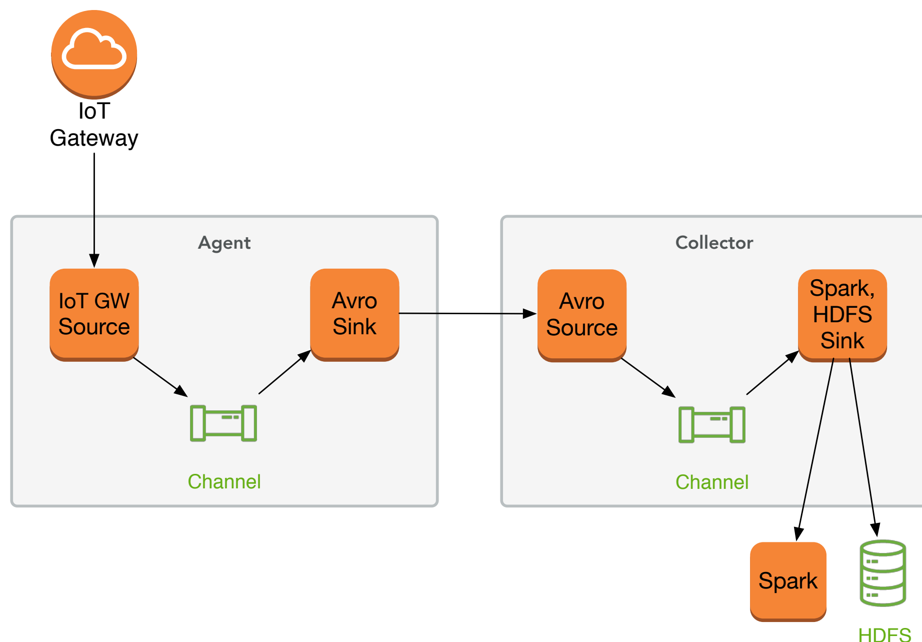
- Na serializáciu a deserializáciu dát reprezentovaných pomocou Apache Avro je potrebná schéma vo formáte JSON, ktorá jednoznačne popisuje ich štruktúru.
- Udalosti serializované pomocou Apache Avro sú niekoľkonásobne menšie ako rovnaké udalosti, ktoré boli reprezentované niektorým z iných formátov (JSON, XML).
- Apache Avro nepotrebuje na serializáciu a deserializáciu žiadne zbytočné statické typy a triedy.

Výsledná architektúra je zobrazená na obrázku 5.2.

5.3.2 Vstup dát z IQRF siete

V dobe realizácie tejto práce pribudol do IQRF siete *WebSocket protokol* 2.4 pre komunikáciu s IoT gatewayom. Namiesto pôvodného plánu, podľa ktorého sa mal jednoznačne využiť protokol *MQTT* 2.3 sme museli zvážiť aj ďalšiu možnosť. Aplikácie založené na IoT vyžadujú spoľahlivú komunikáciu v reálnom čase s takmer nulovou latenciou; nie len vysielanie, ale obojsmerná komunikácia je to, čo sa vyžaduje a WebSocket pre toto poskytuje perfektné prostredie. WebSocket je mimoriadne dôležitý **nástroj pre komunikáciu v reálnom čase** cez internet. Takže sme sa rozhodli použiť WebSocket protokol.

[4]https://avro.apache.org/



Obr. 5.2: Architektonický návrh Flume komponenty.

Serializácia vstupných dát

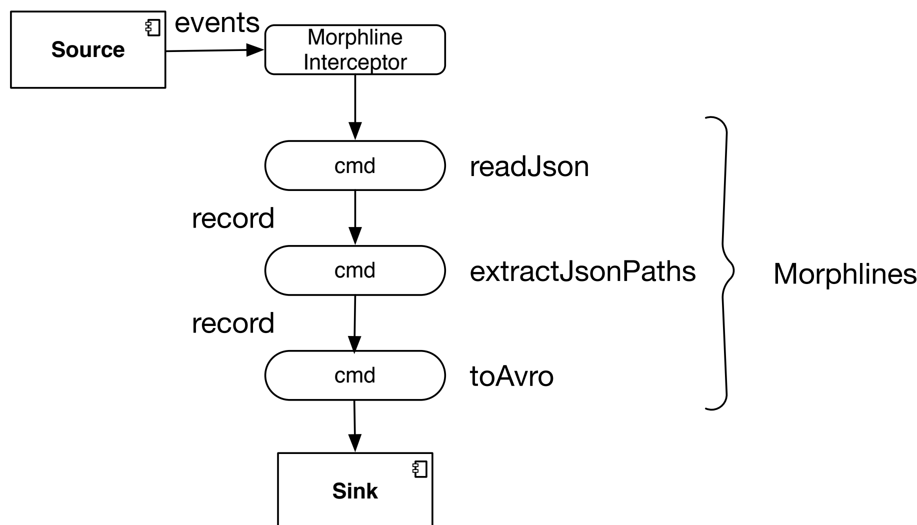
Apache Flume umožňuje **modifikovať/filtrovať prichádzajúce udalosti** pred vstupom do kanálu pomocou parametru tzv. *interceptor*[5]. Interceptor môže modifikovať alebo filtrovať udalosti na základe akýchkoľvek kritérií, ktoré si vybral vývojár interceptoru. Flume podporuje reťazenie interceptorov. Udalosť spracovaná jedným interceptorom je poslaná ďalej ďalšiemu interceptoru. **Flume natívne nepodporuje konvertovanie** niektorých formátov súborov, ale podporuje využitie externej komponenty, ktorá to dokáže. Táto komponenta sa volá *Morphlines*[6]. Použitie Morphlines interceptoru poskytuje potrebnú flexibilitu pri analyzovaní, pridávaní, transformácii, premenovaní, odstraňovaní a iných operáciách na dátach v rámci udalostí. Myšlienkou tohto frameworku je nahradenie programovania v Jave konfiguračným súborom, avšak v rámci konfigurácie umožňuje aj programovanie. Morphlines je akokeby kontajner príkazov v pamäti, ktoré slúžia na transformáciu udalostí a sú použité v sekvencii - jedna za druhou.

Našou úlohou je pomocou tohto framework previesť vstupné dáta z IQRF siete, ktoré budú v JSON formáte na Avro formát. Morphline reťazec príkazov by mohol zhruba vyzerať ako na nasledujúcom obrázku 5.3. Najprv sa použije príkaz `readJson` pomocou ktorého sa načíta JSON udalosť, ďalej `extractJsonPaths` sprístupní hodnoty z JSON objektu pre používanie. Nakoniec sa pomocou `toAvro` skonvertuje pôvodný Morphlines udalosť na Avro záznam a pošle sa ďalej na Sink.

5.4 Spracovanie a ukladanie údajov

V predchádzajúcej sekcii 5.3 sme navrhli 2 spôsoby ukladania dát z Apache Flume zdroja. Prvou možnosťou bolo ukladanie dát do súborového systému HDFS. Pri konfigurácii ulo-

[5]https://flume.apache.org/FlumeUserGuide.html
[6]http://kitesdk.org/docs/1.1.0/morphlines/morphlines-reference-guide.html



Obr. 5.3: Morphlines reťazec príkazov.

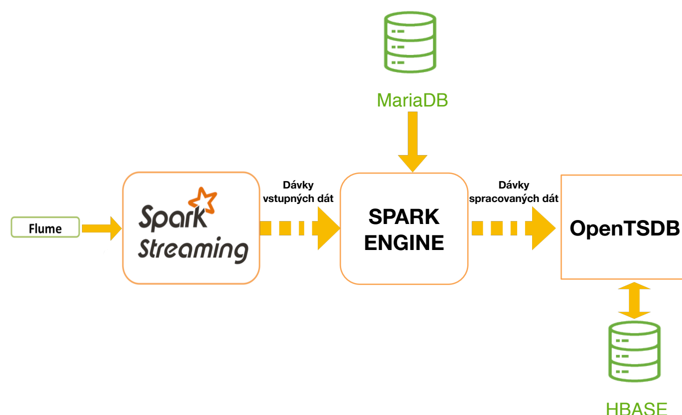
ženia dát je možné nastaviť aby sa **do jedného súboru uložil určitý počet záznamov**, alebo aby sa záznamy ukladali do jedného súboru počas určitej doby, veľkosť súboru a ďalšie nastavenia, ktoré určujú umiestnenie záznamov v súboroch. Jednotlivé súbory sú potom smerované na NameNode uzol, ktorý ich rozmiestni medzi uzle clustera Hadoopu podľa stratégie ukladania, ktorá bola predstavená v sekcii 3.2. Uloženie V HDFS môže mať význam v prípade Flume udalostí, ktoré síce pochádzajú z rovnakého zdroja, ale ich obsah je pri niektorých udalostiach odlišný. Potom majú tieto udalosti **odlišnú dobu spracovania** a spomalujú tým dobu spracovania ostatných udalostí. Takéto udalosti sa môžu napríklad odhaliť pomocou *benchmarkingu* 2.2 a následne indexovaním, kde sa zistí, spracovanie ktorej udalosti, koľko času zabralo. Následne sa môžu udalosti s dlhšou dobou spracovania odfiltrovať ešte vo fáze predspracovania a presmerovať na iný *Flume channel*, a tým nebudú potom smerovať do komponenty, ktorá vykonáva real-time spracovanie, ale sa budú spracovávať dávkovou metódou. Zadaním tejto práce bolo real-time spracovanie dát, takže sa využije komponenta *Apache Spark*, ktorá je na to určená, s tým, že sú pripravené súbory pre prípad potreby dávkového spracovania.

5.4.1 Apache Spark

Komponenta bola predstavená v sekcii 4.4.9. Flume je navrhnutý tak, aby smeroval dáta medzi Flume agentami. V tomto prípade Spark Streaming 4.4.9 v podstate funguje ako prijímač, ktorý pôsobí ako Avro agent pre Flume, na ktorý môže Flume posielať údaje. Tento model komunikácie sa nazýva *Push-based* prístup, ktorý je možné vidieť aj na obrázku 5.4.

Prúd dát - DStream

Ako už bolo spomenuté v sekcii 4.4.9, **DStream je abstrakciou Sparku**, ktorá reprezentuje prúd dát, ktoré vstupujú z vonkajšieho zdroja, v našom prípade Flume collectoru. Pôvodný DStream bez spracovania predstavuje Avro udalosti, ktoré je nutné previesť na spracovateľnú podobu. Tým vytvoríme sekvenciu **RDD, ktoré budú tvoriť DStream**



Obr. 5.4: Návrh spracovania dát v Apache Spark a ich uloženie.

a môžeme ich jednoduchšie spracovávať pomocou transformačných operácií. Po spracovaní sa jednotlivé RDD v DStreame uložia do OpenTSDB.

Pre demonštráciu a tiež pre splnenie zadania sa pracuje s IQRF sieťou, odkiaľ sa príjma *DTO(Data transfer object)*[7], ktorý ma pevne stanovenú veľkosť aj atribúty. Príklady dotazu aj odpovede sa nachádzajú v prílohe A. Po odoslaní dotazu sa v zadanom časovom intervale začnú posielať na Apache Flume cez WebSocket udalosti v špecifikovanom tvare. Čo je pre nás dôležité, jedno **DTO obsahuje namerané hodnoty z viacerých čidiel**. Čiže v našom prípade namerané teploty z viacerých miestností. Pri tansformácii udalosti bude teda potrebné oddeliť tieto hodnoty teploty ako doležitú časť udalosti a ju v DStreame posunúť ďalej pre ďalšie spracovanie. Pri spracovaní sa musia tieto hodnoty teploty orazítkovať časovou známkou a následne sa uložiť do OpenTSDB databáze.

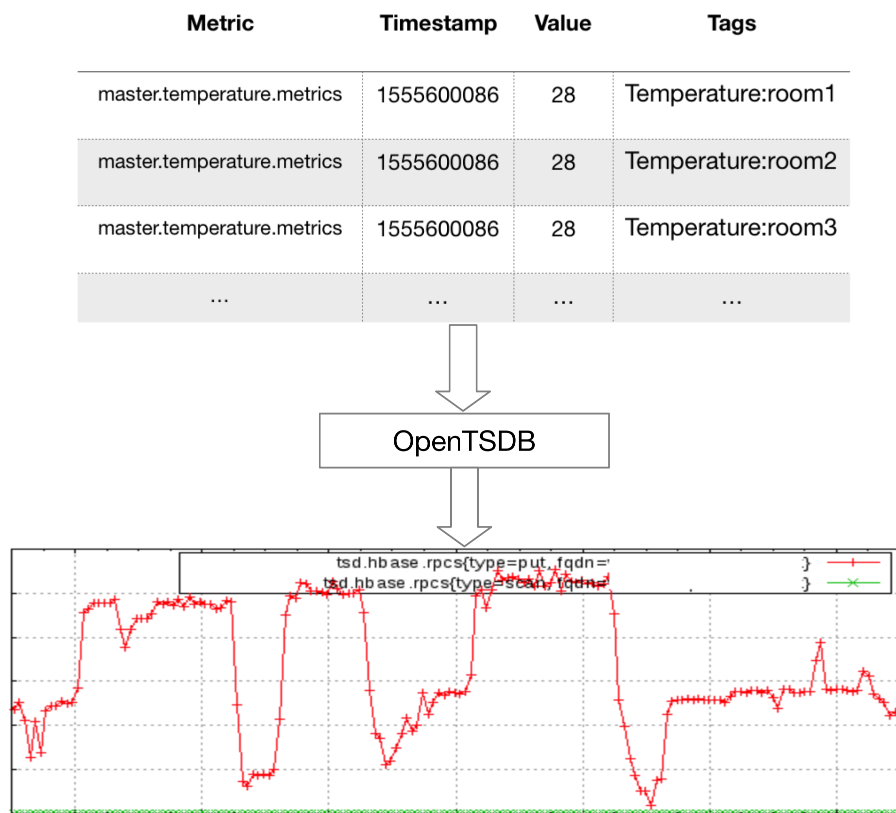
5.4.2 Uloženie dát do OpenTSDB

Našim cieľom je ukladať údaje z rôznych zdrojov, v rôznom formáte a mať možnosť ľahko škálovať databázu horizontálne. Na toto sa nám jednoznačne najviac sedela NoSQL databáza HBase, ktorá je už súčasťou Hadoop ekosystému. V samotnom HBase sa môžu ukladať aj **časové rady údajov**, avšak existuje nástroj, ktorý sa volá OpenTSDB a značne uľahčuje prácu s týmito údajmi, ktoré sa v IoT najčastejšie vyskytujú. Pomocou tohto nástroja sa údaje ukladajú tiež do HBase. OpenTSDB databáza bola predstavená v sekcii 4.4.2. Ďalším dôvodom použitia tohto nástroja bol fakt, že je možné z OpenTSDB vyčítať údaje pomocou nástroja Grafana 4.4.12. Jedným zo spôsobov, ako **odoslať dáta do OpenTSDB**, je použitie jeho REST API. Pri odoslaní dát, musia byť dáta vo formáte JSON, takže je potrebné ich skonvertovať. Následne sa potom využije knižnicu pre vytvorenie HTTP spojenia medzi komponentami Spark a OpenTSDB.

OpenTSDB očakáva, že vstupné dáta budú mať štyri hodnoty, t.j. **metriku, časovú známku, hodnotu a značky**. Vzhľadom na túto skutočnosť a architektonický prehľad OpenTSDB môžeme navrhnúť systém na ukladanie a analýzu dát nameraných pomocou teplomerov, tak ako je to zobrazené na obrázku 5.5. Teda metrika môže byť označená ako teplota firmy Master, časová známka označuje čas spracovania, hodnota je aktuálna teplota a značka značí miestnosť, v ktorej hodnota teploty bola nameraná. Každá jedna metrika sa skonvertuje na JSON a uloží do OpenTSDB, kde sa overí správnosť formátu a uloží sa do

[7]https://en.wikipedia.org/wiki/Data_transfer_object

HBase. OpenTSDB ďalej má aj **svoje vlastné grafické rozhranie**, ktoré je tiež zobrazené vzorovo na obrázku 5.5 a môžu sa pomocou neho zobraziť dáta.



Metric	Timestamp	Value	Tags
master.temperature.metrics	1555600086	28	Temperature:room1
master.temperature.metrics	1555600086	28	Temperature:room2
master.temperature.metrics	1555600086	28	Temperature:room3
...	...	...	...

Obr. 5.5: Návrh uloženia dát do OpenTSDB.

5.5 Zobrazenie údajov

Predvolené užívateľské rozhranie OpenTSDB je veľmi jednoduché a nie je vhodné pre vytváranie vstavaných dashboardov. Okrem toho sú grafy vytvárané pomocou nástroja zvaného *Gnuplot*[8], ktorého predvolený formát grafu vyzerá veľmi zastaralo. Je žiaduce iné, **novšie vizualizačné rozhranie**.

Jedným z dobrých riešení je open-source dashboard editor známy ako *Grafana* 4.4.12. OpenTSDB poskytuje prístup k údajom cez HTTP a Grafana využíva tento prístup na vytvorenie vysoko-kvalitného vizualizačného rozhrania pre OpenTSDB a ďalšie. Grafana ponúka zdieľanie dashboardov, čo sa nám veľmi zíde pri použití v informačnom systéme.

5.5.1 Informačný systém

Jednou z požiadaviek v tejto práci bolo vytvorenie informačného systému, kde bude **správa užívateľov a miestností**, v ktorých sa užívatelia nachádzajú. Takisto by údaje spracované v Hadoop ekosystéme nemali byť prístupné širokej verejnosti. Ďalej v tomto konkrétnom prípade, keď ide o meranie teploty, je potrebné umožniť užívateľovi nastaviť hodnotu teploty,

[8]http://www.gnuplot.info/

pri ktorej chce byť **notifikovaný**, že daná hodnota teploty v miestnosti je mimo zvolený rozsah. V tomto prípade by mala byť užívateľovi doručená notifikácia a on by si mal ideálne miestnosť napríklad vyvetrať alebo zapnúť klimatizáciu.

Aj keď sa jedná o pomerne malý systém, tak by sa z hľadiska návrhu mala predpokladať možnosť **rozšírenia tohto systému** v budúcnosti, teda je potrebné zvážiť vyhovujúce technológie pre prípad väčšieho systému v budúcnosti.

Backend - Serverová aplikácia

Informačný systém je len menšou časťou tejto práce, takže podrobnosti návrhu musíme vynechať, pretože v tejto práci pre to nie je priestor.

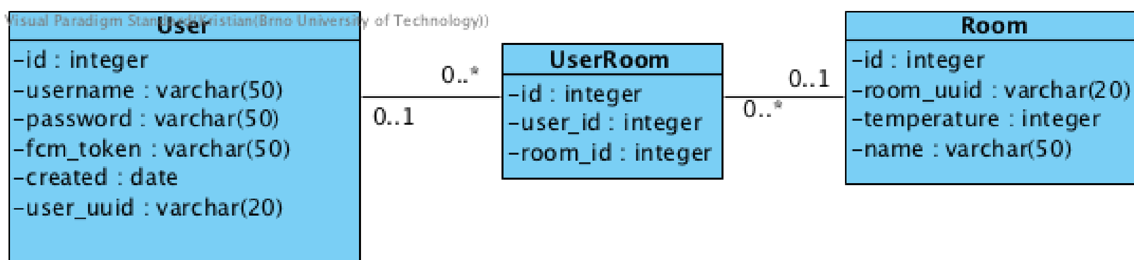
Na nasledujúcom obrázku sa nachádza *diagram prípadov použitia (angl. Use case diagram)*, z ktorého sa odvíja aj **návrh serverovej aplikácie**. Z daného diagramu je jasné, čoho všetkého by mal byť používateľ schopný dosiahnuť v našom systéme. Takže po prihlásení si používateľ vyberie miestnosť, v ktorej si bude vedieť nastaviť maximálnu teplotu, pri ktorej keď sa daná hodnota dosiahne tak sa pošle notifikácia. Notifikácie môže začať odoberať pridaním sa do miestnosti, resp. ukončiť odber odobratím z miestnosti. Zmenu stavu teploty vykonáva výlučne Hadoop ekosystém pomocou HTTP dotazu, po ktorej sa posiela notifikácia.

Obr. 5.6: Diagram prípadov použitia.

Na základe Use case diagramu sa vytvoril *ER diagram* 5.7 pre návrh entít v databáze. Entita `User` mimo iných atribútov obsahuje atribút `fcm_token`, ktorý slúži k identifikácii v systéme *Firebase* 6.4.3. Keďže používateľ môže odoberať notifikácie z viacerých miestností a zároveň v jednej miestnosti (entita `Room`) môže byť viacero používateľov, tak bola potrebná väzobná tabuľka (entita `UserRoom`).

Upozornenie užívateľa v prípade hodnôt mimo rozsah

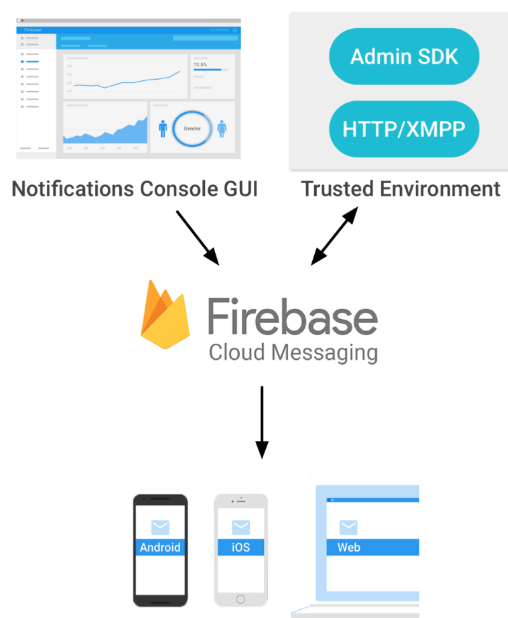
Jednou z požiadaviek na vysledný systém bolo informovanie užívateľa v prípade hodnôt mimo špecifikovaný rozsah. V ďalšej časti tejto sekcie sa popisuje **frontend** informačného systému, ktorým je webová aplikácia. Užívatelia tam budú prihlásení a ideálne by tam mali byť notifikovaný v prípade **výnimočného stavu**. Jednou z technológií, ktoré sa na takéto notifikácie využívajú je *Firebase* [9].



Obr. 5.7: Návrh databázového modelu - ER diagram.

Firebase je komplexná mobilná a webová vývojárska platforma unifikovaná skrz Android, iOS a web. Vývojárom pomáha tvoriť kvalitné aplikácie a rozširovať užívateľskú základňu. Platforma Firebase obsahuje množstvo nástrojov, ako napríklad databázu v reálnom čase, úložisko dát či zasielanie notifikácií. Jej spektrum funkcií je tak rozmanité, že zbavuje vývojára potreby implementovať vlastný server. Náš **informačný systém** implementuje spojenie s platformou Firebase a ako kľúčový nástroj používa *Cloud Messaging (FCM)*. Ten umožňuje bezplatné doručovanie správ a notifikácií medzi zariadeniami.

FCM poskytuje webovým a mobilným aplikáciám možnosť prijímať správy, ktoré im boli poslané zo servera, či už je alebo nie je klientská aplikácia v čase odoslania správy spustená. Toto umožňuje posielať **asynchrónne notifikácie** a aktualizácie používateľom.

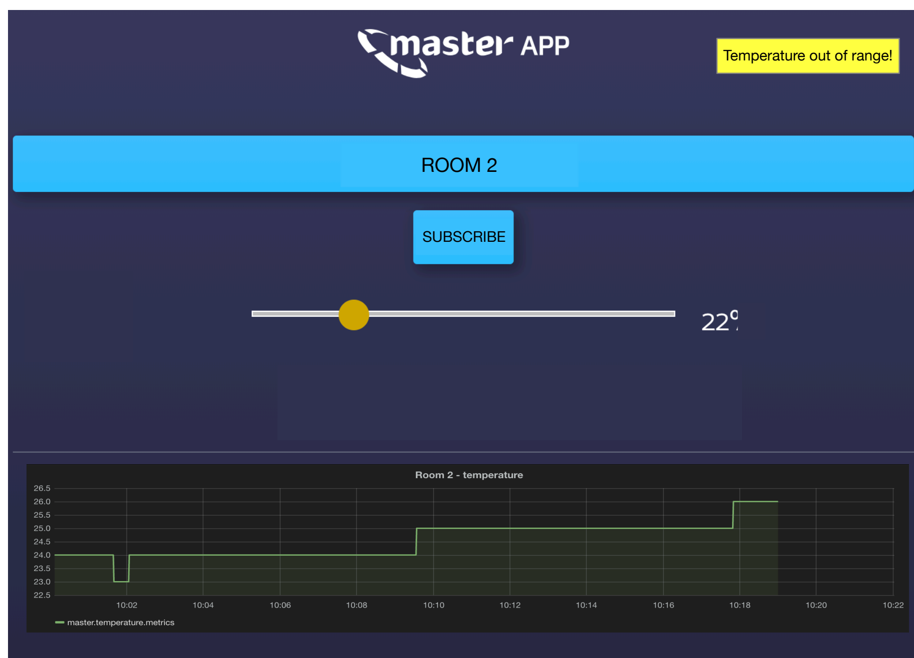


Obr. 5.8: **Firebase Cloud Messaging** - na obrázku sú zobrazené 2 hlavné komponenty odosielania notifikácií - serverová aplikácia, ktorá odošle notifikáciu a klientské aplikácie, ktoré ju príjmu. Zdroj: [9]

Frontend - Webová aplikácia

Na obrázku 5.9 je zobrazený návrh hlavnej stránky webovej aplikácie, na ktorej budú zobrazené pre užívateľa potrebné informácie. Stránka umožní užívateľovi **vybrať miestnosť**,

do ktorej sa bude môcť pridať a začať odoberať notifikácie pomocou tlačidla `Subscribe`. Ďalej si tam môže užívateľ **nastaviť hodnotu teploty**, pri ktorej bude notifikovaný pokiaľ sa daná hodnota presiahne. Notifikácie budú zobrazované v pravom hornom rohu. V dolnej časti stránky sa potom nachádza **graf, ktorý sa pravidelne aktualizuje** a zobrazujú sa tam hodnoty teploty v miestnosti a taktiež predikované hodnoty teploty na nasledujúce 3 hodiny. Webová aplikácia bude prepojená so serverovou aplikáciou cez rozhranie *REST API*.



Obr. 5.9: **Mock-up webovej aplikácie**

Jednoduchá lineárna regresia

Keď sa pozrieme na **vzťah medzi hodnotami**, z ktorých by sa mal predikovať budúci stav, tak sa nám najviac hodí metóda lineárnej regresie. Lineárna regresia je štatistická metóda, ktorá kvantifikuje závislosť medzi dvoma spojitými premennými: `závislou` (snažíme sa predikovať) a `nezávislou` – prediktívna premenná. Princípom je nájdenie priamky, ktorá prechádza jednotlivými bodmi. V našom prípade budeme mať `časovú známku (X)` ako nezávislú premennú a `hodnotu teploty (Y)` ako závislú premennú, ktorú budeme predikovať. Medzi X a Y platí všeobecná lineárna závislosť, teda:

$$y = a_0 + a_1 x \tag{5.1}$$

Kde y predstavuje očakávanú hodnotu, ktorá rastie alebo klesá v priemere podľa nezávislej premennej x. Regresný koeficient a_0 predstavuje priesečník regresnej priamky s osou o_y. Regresný koeficient a_1 vyjadruje smernicu regresnej priamky, teda sklon priamky k osi o_x. Charakterizuje zmenu závislej premennej, ktorá zodpovedá zmene nezávislej premennej o jednu jej jednotku.

Za využitia tejto metódy sa môžu **predikovať hodnoty** teploty na najbližšie hodiny.

Kapitola 6

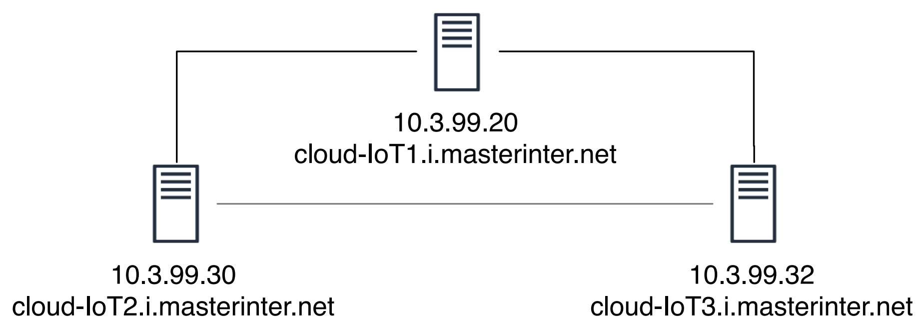
Realizovaný systém

Vychádzajúc z predchádzajúcej kapitoly 5, kde sa spravil hrubý návrh celého systému na základe požiadaviek na systém a následne sa problém dekomponoval na menšie časti, sa teraz jednotlivé navrhnuté časti realizujú. Začne sa inštaláciou *Hortonworks* distribúcie 4.4 na cluster serverov. Potom sa realizuje zber a predspracovanie dát, využitím komponentov z *HDP* a IoT ekosystému. Predspracované dáta sa budú ukladať pre možnosti neskoršieho spracovania alebo sa posunú ďalej na spracovanie v reálnom čase, kde sa nakoniec tiež uložia. Dáta spracované v reálnom čase sa budú vizualizovať v grafoch. Pre prístup k Hadoop ekosystému sa vytvorí informačný systém pre používateľov, kde sa budú implementovať notifikácie pre hodnoty mimo rozsah a predikcie pre predikovanie budúcej hodnoty.

6.1 Hortonworks cluster

Vytvorený cluster serverov 6.1, na ktorom pobeží Apache Hadoop ekosystém a na ktorom sa bude testovať sa skladá z 3 uzlov s nasledujúcou počiatočnou konfiguráciou:

- **cloud-IoT1.i.masterinter.net** - Master uzol, disková kapacita 40GB, operačná pamäť 8GB
- **cloud-IoT2.i.masterinter.net** - Slave uzol, disková kapacita 40GB, operačná pamäť 8GB
- **cloud-IoT3.i.masterinter.net** - Slave uzol, disková kapacita 40GB, operačná pamäť 8GB



Obr. 6.1: Zapojenie serverov v clusteri.

Na týchto uzloch je nainštalovaná verzia HDP 2.6.5. Aktuálna najnovšia verzia v dobe inštalácie je 3.1.0, avšak je úplne bežné **použitie staršej, ale stabilnej verzie**, v ktorej sú už známe problémy vyriešené a k jednotlivým komponentám existuje spoľahlivá podpora. V nasledujúcej tabuľke 6.1 sú predstavené komponenty, ktoré boli nainštalované, ich verzie a ich rozloženie na jednotlivých uzloch.

Komponenta	**cloud-IoT1**	**cloud-IoT2**	**cloud-IoT3**
HDFS (v2.7.3)	NameNode	DataNode	DataNode
YARN (v2.7.3)	ResourceManager	NodeManagers	NodeManagers
Oozie (v4.2.0)	Server, Client	Client	Client
MapReduce2 (v2.7.3)	Client	History server, Client	Client
Pig (v0.16.0)		Client	Client
Hive (v2.1.0)	Metastore	Server, WebHCat, Client	Client
Tez (v0.7.0)	Client	Client	Client
Sqoop (v1.4.6)			Client
Spark2 (v2.3.0)		History server	Client
Zeppelin (v0.7.3)	Notebook		
HBase (v1.1.2)	Master	RegionServer	RegionServer
Ambari (v2.6.2)	Server, Metrics		
Zookeeper (v3.4.6)	Server	Server, Client	Server, Client
Flume (v1.5.2)		Agent	Agent
OpenTSDB (v2.4.0)	Server		
Grafana (v2.6.0)	Server		

Tabuľka 6.1: Zoznam komponentov na jednotlivých uzloch

6.2 Zber a predspracovanie údajov

Predchádzajúci návrh 5.3 využijeme k realizácii zberu dát z IQRF siete. Ako už bolo spomenuté, najlepšou voľbou pre naše riešenie bude použitie WebSocket protokolu pre komunikáciu s IoT gatewayom. Apache Flume neumožňuje natívne **použitie WebSocket**

protokolu, avšak umožňuje vlastnú konfiguráciu. Vo vlastnej konfigurácii sa môže použiť ľubovoľný *plugin* spustiteľný v JVM. Tento plugin, ktorý sa použije pri konfigurácii *Source* časti komponenty Flume, musí byť pridaný do priečinku `FLUME_HOME/plugins.d`, čo je špeciálny priečinok pre pluginy. Pri spustení, skript `flume-ng` začne v adresári `plugins.d` vyhľadávať pluginy, ktoré vyhovujú formátu a spustí ich spolu s agentom.

Využil sa existujúci plugin Websocket Source for Apache Flume. Tento plugin poskytuje tieto vlastnosti:

- Spojenie cez `ws://` alebo zabezpečené spojenie `wss://`
- Dokáže preposielať správy na Flume channel
- Automaticky naväzuje opätovné spojenie pri nastavenom parametri `retryDelay`

Flume agent sa spúšťa spolu s konfiguráciou `flume-agent.conf`, jednou časťou tejto konfigurácie je konfigurácia WebSocket pripojenia 6.2, kde parameter `initMessage` znázorňuje inicializačnú správu pomocou ktorej definujeme o aké dáta z daného IoT gatewayu žiadame.

```
agent.sources.iot_source.type          = com.deniscoady.flume.websocket.WebSocketSource
agent.sources.iot_source.endpoint      = ws://10.2.33.208:1338
agent.sources.iot_source.retryDelay    = 5
agent.sources.iot_source.initMessage   = <initMessage>
```

Obr. 6.2: Konfigurácia WebSocket protokolu vo Flume.

Nastavenie parametrov Flume channelu sme nechali na štandardných hodnotách, ktoré sa neskôr v prípade potreby upravia na potrebné hodnoty na základe vyťaženosti. Ako už bolo v návrhu spomenuté, našu architektúru Flumu tvoria komponenty *agent* a *collector*. Prepojenie týchto dvoch komponentov do systému, ktorý sa nazýva *multi-hop* je zobrazené v konfigurácii 6.3.

```
agent.sinks.avro-forward-sink.type = avro
agent.sinks.avro-forward-sink.hostname = 10.3.99.32
agent.sinks.avro-forward-sink.port = 60000

collector.sources.avro-collection-source.type = avro
collector.sources.avro-collection-source.bind = 10.3.99.32
collector.sources.avro-collection-source.port = 60000
```

Obr. 6.3: Multi-hop konfigurácia.

Nakoniec všetky tieto dáta smerujú na *Sink collectoru*, ktorých približná konfigurácia je viditeľná na obrázku 6.4. V rámci jedného Flume agentu, ktorý sa v tomto prípade tvári ako collector, sa vytvoria **dva sinky a dva kanály** pre duplikáciu *avro udalostí*. Udalosti, ktoré sa uložia priamo do HDFS, sa ešte najprv overia pomocou takzvaného *serializeru*, či spĺňajú zadanú schému avro udalosti a ak nespĺňajú, tak sa automaticky zahadzujú. Tieto údaje sú smerované na uzol, na ktorom sa nachádza HDFS NameNode, ktorý predáva dáta ďalej DataNode-om a neskôr sa môžu tieto dáta dávkovým spôsobom vyhodnocovať. Údaje sa ukladajú bez komprimácie. V prípade komponenty Spark bude potrebné vytvoriť Spark program, ktorý pobeží na danom porte a serveri, a bude prijímať Flume udalosti.

Serializácia dát z IQRF siete

Použitie špeciálneho interceptoru - frameworku *Morphlines*, ktorá nie je súčasťou Flume komponenty, v konfigurácii Flume agenta môže vyzerať približne, tak ako je to zobrazené na

```
collector.sinks.spark-sink.type = org.apache.spark.streaming.flume.sink.SparkSink
collector.sinks.spark-sink.hostname = localhost
collector.sinks.spark-sink.port = 1477

collector.sinks.hdfs-sink.type=hdfs
collector.sinks.hdfs-sink.serializer=AvroEventSerializer$Builder
collector.sinks.hdfs-sink.hdfs.path=hdfs://cloud-iot1.i.masterinter.net:8020/
collector.sinks.hdfs-sink.hdfs.fileType=DataStream
collector.sinks.hdfs-sink.hdfs.writeFormat=Text
```

Obr. 6.4: HDFS a Spark sink.

obrázku 6.5. V konfigurácii je vidieť pripojenie Morphlines frameworku, ďalej je tam vidieť použitie konfiguračného súboru `morphline.conf`. Morphlines vo vlastnej konfigurácii tiež **podporuje reťazenie**, takže je potrebné definovať aj `id` konfigurácie.

```
agent.sources.iot_source.interceptors = morphlines
agent.sources.iot_source.interceptors.morphlines.type = MorphlineInterceptor$Builder
agent.sources.iot_source.interceptors.morphlines.morphlineFile = morphline.conf
agent.sources.iot_source.interceptors.morphlines.morphlineId = convertJsonToAvro
```

Obr. 6.5: Pripojenie frameworku Morphlines do komponenty Flume.

Pri vytváraní konfigurácie pre Morphlines sme postupovali podľa návrhu, ktorý je zobrazený na obrázku 5.3. Narazili sme však na komplikáciu hneď pri prvom príkaze `readJson`, kde príkaz nebol schopný spracovať ľubovoľný JSON súbor. Problém tvorilo **viacnásobné zanorenie položiek**. Toto bolo vyriešené pomocou knižnice `jackson`[1], ktorý bol použitý pri implementácii parseru JSON súboru v Morphlines konfigurácii. Sparsované JSON objekt je potom možné previesť na avro reprezentáciu, ale je potrebná schéma JSON objektu, ktorý sa ide na avro reprezentáciu prevádzať. Túto schému je možné ľahko vytvoriť pomocou online nástroja Avro from JSON, kde sa zadá vzorový JSON vstupný objekt a získame potrebnú schému pre prevod na avro. Nakoniec sa avro udalosť **uloží do kanálu** pomocou príkazu `writeAvroToByteArray`, čiže v bajtovej reprezentácii.

6.3 Spracovanie a ukladanie údajov

Zase sa odkážeme na návrh v sekcii 5.4, z ktorého vyplýva použitie komponenty Apache Spark. Pre vývoj Apache Spark Streaming aplikácie je potrebné prostredie, v ktorom je **Spark nainštalovaný**. V Hadoop ekosystéme sa Spark už nachádza, avšak odporúča sa vyvíjať a debugovať aplikácie na jednom uzle, takže bolo nainštalované virtuálne prostredie Hortonworks Hadoopu, tzv. *Sandbox*[2]. Po odladení aplikácie sa vytvoril *Spark Job*, ktorý bol spustený na HDP clustri.

Spark bol nakonfigurovaný pre spustenie s nasledujúcimi parametrami:

- **Typ:** Avro
- **Hostname:** localhost
- **Port:** 1177

Pretože Spark aplikácia bude bežať na rovnakom node ako *Flume collector*, pobeží na *localhoste*. *Spark job* musí byť spustený ako prvý, lebo následne sa pripojí Flume collector, na ktorý sa potom pripoja Flume agenti.

[1] https://www.tutorialspoint.com/jackson/
[2] https://hortonworks.com/tutorial/hortonworks-sandbox-guide/

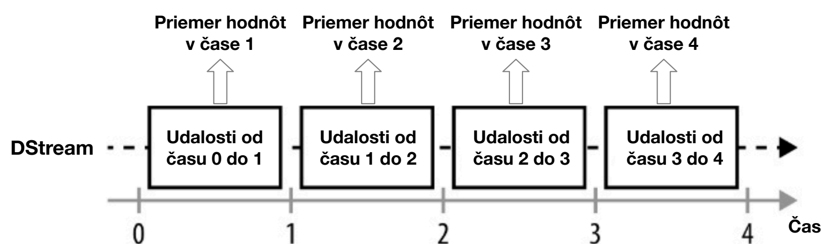
6.3.1 Apache Spark

Je potrebné poznamenať, že Spark je len **čiastočne real-time** výpočtový systém a teda, že vstupujúci prúd dát sa rozdeľuje do dávok na základe časového intervalu. Nespracováva sa teda každý záznam po jednom, ale môžeme **viacero záznamov spracovať naraz** v časovom intervale, ktorý sme si zvolili. Na začiatku aplikácie je nutné vytvoriť a správne nastaviť `SparkSession` a `StreamingContext`, nasledujúce parametre sú dôležité:

- `setMaster("local[N]")` - pre spustenie vo virtuálnom režime sa nastavil master na lokálny režim s aspoň 2 vláknami, pretože 1 vlákno sa používa na zber prichádzajúceho prúdu dát a ďalšie vlákna na spracovanie dát.
- `setMaster("yarn")` - v clustri sa pripája na YARN cluster 4.1.2.
- `Durations.seconds(S)` - Čas nastavený pre každú dávku je na začiatok nastavený na 1 sekundu. Inými slovami, všetky zozbierané udalosti budú dávkované po 1 sekunde pred odoslaním na spracovanie.

Realizácia Spark aplikácie a spustenie

Aplikácia začína konfiguráciou, ktorú sme spomenuli vyššie. Prúd dát sa začína spracovávať v triede `IqrfStreamingContext`, kde sa využívajú *lambda výrazy*[3] z triedy `org.apache.spark.api.java.function`. Do tejto funkcie vstupujú už len udalosti, ktoré spĺňajú formát IQRF DTO objektu z prílohy A, keďže už prešli filtrom v komponente Flume. V prvom kroku sa Avro udalosť prevedie na **reťazec znakov**, aby ho bolo možné spracovať. Keďže udalosť obsahuje veľa metadát, ktoré pre nás nie sú podstatné, tak ich odstránime a vo výsledku nám zostanú len `frcData`, čo sú vlastne združené namerané hodnoty z viacerých čidiel. Avro udalosť s IQRF DTO objektom nenesie explicitnú informáciu o tom, z ktorej miestnosti nameraná hodnota pochádza, dokážeme to zistiť len na základe indexu hodnoty v poli. Daný **index určuje konkrétne čidlo**. Podľa tohto zistenia sa v ďalšom kroku priradia hodnotám miestnosti z ktorých sa hodnoty namerali a časové známky podľa aktuálneho času. Keďže nie je na každý index pripojené čidlo, odčítajú sa len hodnoty, ktoré boli namerané a ostatné sa ignorujú. Behom 1 sekundy - čas, ktorý sme nastavili pre dávkovanie, nám príde viacero udalostí, ale my chceme ukladať teplotu len raz za 1 sekundu, takže je **potrebné udalosti zredukovať**. Momentálne sú už udalosti spracované do reprezentácie s ktorým Spark pracuje a teda RDD. Vykonáme `reduce` funkciu nad DStreamom a namerané hodnoty spriemerujeme. Takto do posledného kroku už vstupuje len jedno RDD, ktoré sa vyhodnotí a uloží. Tento proces je zobrazený na obrázku 6.6.

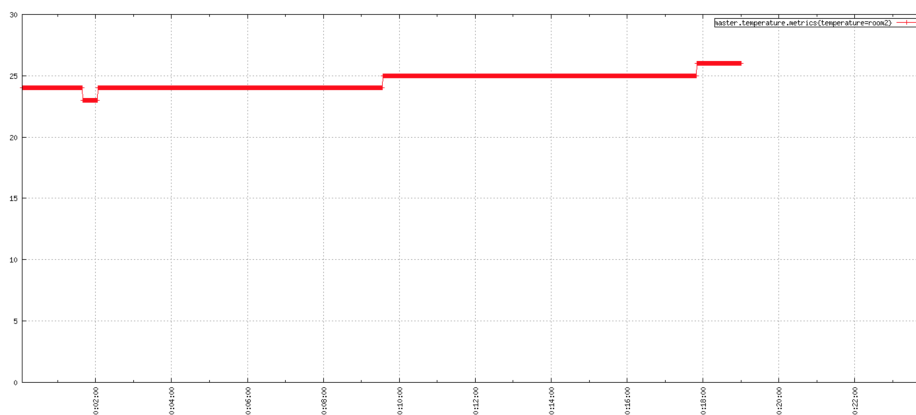


Obr. 6.6: Spracovanie udalostí v Sparku.

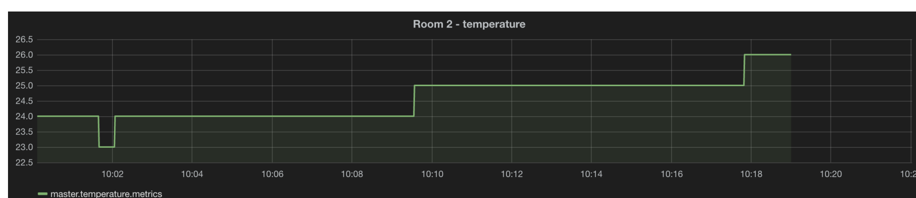
[3]https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

Pred vyhodnotením sa RDD skonvertuje na DTO objekt `OpenTSDBTemperatureDTO` pre ľahšiu manipuláciu s dátami. Tento objekt spĺňa formát OpenTSDB, čiže ho stačí skonvertovať na JSON a uložiť cez REST API. Vyhodnotenie prebieha pripojením sa na MariaDB databázu nášho informačného systému a získanie intervalu, ktorý zadal používateľ ako rozsah teploty. Hodnoty teplôt sú teda zvalidované, či sa nachádzajú v tomto intervale a pokiaľ nie, tak sa cez REST API informačného systému posiela dotaz na notifikáciu používateľov, ktorých sa notifikácia týka, že namerané hodnoty sú mimo rozsah.

`OpenTSDBTemperatureDTO` objekty sa teda skonvertujú na JSON a pomocou triedy `OpenTSDBService` sa vytvorí HTTP spojenie, pomocou ktorého sa pošlú dáta do OpenTSDB na endpoint `http://<host>:9999/api/put`. Na nasledujúcich obrázkoch 6.7 a 6.8 sú zobrazené uložené namerané dáta za určité obdobie.



Obr. 6.7: Zobrazenie nameraných dát v OpenTSDB.



Obr. 6.8: Zobrazenie nameraných dát v Grafane.

Spúšťanie aplikácie v clustri je štandardne vykonané pomocou príkazu `spark-submit` dostupného v adresári s inštaláciou Sparku. Ako parametre sa odovzdávajú: názov spúšťanej triedy, použité knižnice a ďalšie parametre. Posledným argumentom je potom cesta k súboru so spúšťaným programom.

6.4 Informačný systém

Vzhľadom k tomu, že Apache Hadoop ekosystém je rozsiahly systém, ktorý pobeží na viacerých strojoch a spracováva dáta, ktoré môžu byť citlivé, nie je žiadny dôvod aby bol tento systém dostupný z vonkajšej siete. Hadoop ekosystém pobeží na **vnútornej sieti** firmy *Master Internet,s.r.o*[4]. Informačný systém bude dostupný jak z **vonkajšej siete**, tak aj

[4] https://www.master.cz/

z vnútornej siete. Navyše umožní správu užívateľov, vizualizáciu dát a zabezpečí autorizované prístupy. Informačný systém bude pripojený na Hadoop ekosystém pomocou *HTTP* protokolu a dotazovať sa na potrebné dáta.

6.4.1 Backend - Serverová aplikácia

Serverová aplikácia bude napísaná v programovacom jazyku *Java*, s použitým frameworkom *Spring Boot* 6.4.1, cez ktorý sa napojí relačná databáza *MariaDB*. V tejto časti si predstavíme použité technológie, oddvôvodníme si ich použitie a popíšeme výslednú architektúru systému.

Programovací jazyk Java

Ako sme sa už aj v škole učili, v oblasti informačných systémoch programovací jazyk `Java`[5] v serverových aplikáciách vyniká z viacerých dôvodov. Java má rozsiahlu škálu vynikajúcich knižníc na riešenie väčšiny bežných problémov, ktoré je potrebné riešiť pri **vývoji informačných systémov**. Keďže Java beží vo virtuálnom stroji *JVM*[6], stačí napísať kód raz a v každom prostredí sa bude softvér správať rovnako a bude sa dať dobre ladiť. Je to vyspelý jazyk, ktorý bol použitý už na viacerých veľkých projektoch, existuje skvelá podpora a veľa materiálov pre štúdium.

Navyše **Hadoop je napísaný v jazyku Java**. Pre nás to znamená, že programovateľné časti ekosystému budú pri znalosti tohto jazyku ľahšie zvládnuteľné. Skúsenosti s jazykom Java sú potrebné aj pre ladenie Hadoop aplikácií [19].

Framework Spring Boot

Spring Boot uľahčuje vytváranie samostatných, produkcie-schopných serverových aplikácií, ktoré sú jednoducho spustiteľné. *Framework* je v prvom rade prostriedok na **zjednodušenie práce**. Je to súbor knižníc a kódu, ktorý sa snaží pokryť čo najviac funkčných požiadaviek spoločných pre rôzne aplikácie. Tento framework vznikol ako **nadstavba** na *Spring Framework*. Spring framework je veľmi populárny, ale vytvorenie jednoduchej aplikácie zaberie veľa času. Spring framework vychádza z myšlienky, že všetko, čo je potrebné, si nastaví autor sám. K tomu, aby to všetko nastavil, je potrebné detailnejšie zoznámenie sa s touto technológiou. Vo väčšine situácií sú tieto nastavenia nastavené na východzie hodnoty. Tu vstupuje Spring Boot. Vychádza z myšlienky, že vo **východzom stave** je všetko nastavené tak, aby to fungovalo. Samozrejme, je tu možnosť všetko manuálne prepísať. Pre Spring Boot existuje incializátor, v ktorom je možné si vybrať, ktoré **závislosti** budú v projekte použité a následne sa vygeneruje celý projekt, ktorý v sebe obsahuje tieto závislosti, a ktorý je pripravený na implementáciu požadovanej aplikácie [31].

MariaDB

Informačný systém využíva relačný databázový systém MariaDB, ktorý je open-source. Niektoré vlastnosti relačných databáz sú spomenuté v sekcii 3.1.1. MySQL a MariaDB sú takmer identické databázové systémy, ktoré sa v istom momente oddelili pre zachovanie *licenčne slobodného softvéru*[7]. **Rozdielne a spoločné vlastnosti** medzi MySQL a MariaDB

[5] https://www.java.com/en/
[6] **JVM (Java Virtual Machine)** - virtuálny stroj, ktorý vykonáva medzikód (*bytecode*) Javy.
[7] https://cs.wikipedia.org/wiki/GNU_General_Public_License

nájdete na stránke Porovnanie MariaDB a MySQL. Na dotazovanie sa využíva tradične SQL jazyk. Pri použití **MariaDB v Spring Boot** frameworku však nie je nutné používať priamo SQL jazyk pri jednoduchších operáciách. Existuje rozhranie tzv. `Repository` cez ktoré sa dotazujeme pomocou *CRUD*[8] operácií. Jednotlivé tabuľky v klasickom SQL jazyku sú reprezentované ako dátové objekty, na základe ktorých sa **generujú tabuľky** v databázi. Veľká výhoda je, že tým sa stáva aplikácia nezávislá na databázi. Typ relačnej databázy sa môže jednoducho zmeniť v konfiguračnom súbore.

Architektúra serverovej aplikácie

Pri popise architektúry serverovej aplikácie budeme vychádzať z diagramu tried na obrázku 6.9. Ako už bolo aj vyššie spomenuté, bežné databázové tabuľky sú vo frameworku prezentované datovými objektmi, ktoré sú označené anotáciou `@Entity` a z nich sa generujú potom tabuľky. To je znázornené aj na obrázku, kde je zobrazený návrhový vzor **Singleton**[9] (v preklade: Jedináčik), pre vytvorenie spojenia s databázou. Pre vytvorenie jednotlivých entít sa používa návrhový vzor **DAO**[10]. Pre umožnenie vytvárania *CRUD* operácií nad entitami je potrebné vytvoriť rozhranie s anotáciou `@Repository`, kde sa definujú CRUD operácie a v prípade komplexnejších operácií aj tzv. *HQL*[11] dotaz. V našom prípade sa dedí od nadtriedy `GeneralRepository`, kde sa definujú základné operácie. Je možné jednotlivé rozhrania aj implementovať v prípade zložitejších problémov. Následne sa implementujú služby, ktoré používajú anotáciu `@Service` a tu sa implementuje tzv. **bussines logika**. Pre redukciu závislostí sa používa návrhový vzor **Dependency injection**[12] pomocou anotácie `@Autowired`. Využíva sa to hlavne pri závislostiach medzi službami a medzi službami a kontrolérmi. Posledná vrstva `@Controller` vytvára *REST API* rozhranie pre frontendové aplikácie za použitia služieb. Pre zjednodušenie diagramu sa nenachádzajú všetky triedy v diagrame, taktiež sa tam nenachádza architektúra predikcií, ktoré sú zvlášť rozobraté v sekcii 6.4.4. Používateľ po registrácii a prihlásení sa (`UserController`) do systému obdrží **JWT token**[13], ktorý bude slúžiť na autentizáciu používateľa pri pristupovaní do autorizovaných častí systému. Potom môže vykonávať zmeny teploty v jednotlivých miestnostiach a pridávať/odobrať sa ľubovoľne do/z miestností (`RoomController`). Kontrolér `NotificationController` je využívaný systémom Hadoop pre zasielanie notifikácií (6.4.3). Aplikácia je nasadená na server, na ktorom beží *Apache Tomcat*[14], ktorý slúži na spúšťanie a správu Java aplikácie, pomocou nástroja *Maven*[15], ktorý slúži na *buildenie* aplikácií.

6.4.2 Frontend - Webová aplikácia

Aj keď sa v tejto práci jedná o rozsahovo malú webovú aplikáciu, do budúcna sa plánuje jeho rozšírenie. Preto treba uvažovať nad použitím programovacieho jazyku a frameworku, ktorý umožňuje vytvoriť dobre spravovateľnú webovú aplikáciu. Podobne ako aj pri implementácii

[8]**CRUD** - Create, Read, Update, Delete

[9]**Singleton** - je názov návrhového vzoru, ktorý sa využíva v prípade, kedy je potrebná iba jedna inštancia objektu počas behu programu.

[10]https://www.baeldung.com/java-dao-pattern

[11]**Hibernate query language** - v porovnaní s SQL pracuje s objektami a zvláda aj dedenie, polymorfizmus a asociácie

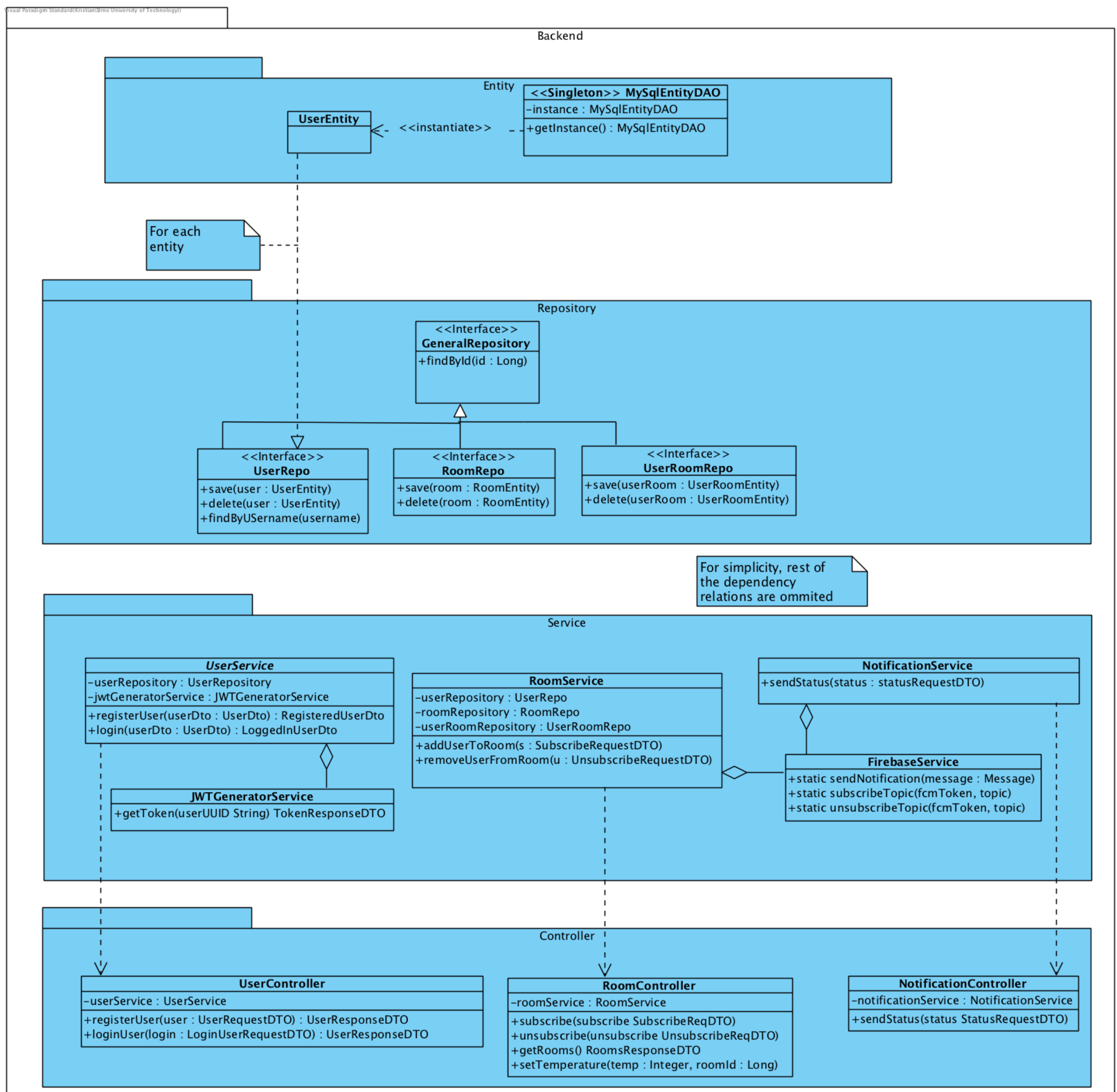
[12]https://www.baeldung.com/spring-autowire

[13]https://jwt.io/

[14]http://tomcat.apache.org/

[15]https://maven.apache.org/

serverovej aplikácii 6.4.1, aj tu si predstavíme hlavné použité technológie, oddvôvodníme si ich použitie a nakoniec predstavíme výslednú architektúru systému.



Obr. 6.9: Diagram tried serverovej aplikácie.

Programovací jazyk PHP

PHP (*PHP: Hypertext Preprocessor*) je rozšírený dynamický open-source skriptovací jazyk, ktorý je populárny pri tvorbe webových stránok bežiacich na webovom serveri. Najčastejšie sa začleňuje priamo do štruktúry jazyka *HTML*, *XHTML* či *WML*[16], čo sa dá dobre využiť pri tvorbe webových aplikácií. PHP sa dá použiť aj k tvorbe konzolových a desktopových aplikácií. V prípade **dynamických stránok** sú skripty prevádzané na strane serveru – k užívateľovi je prenesený až výsledok ich činnosti. Syntax jazyka je inšpirovaný niekoľkými

[16] https://www.developershome.com/wap/xhtmlmp/xhtml_mp_tutorial.asp?page=devWirelessMarkup

programovacími jazykmi (*Perl, C, Pascal a Java*). PHP je nezávislý na platforme, rozdiely v rôznych operačných systémoch sa obmedzujú len na niekoľko OS-závislých funkcií a skrípt, a dá sa väčšinou medzi operačnými systémami portovať bez akýchkoľvek úprav [14].

Framework Nette

Nette [10] je kompletný framework pre PHP, ktorý výrazne zjednodušuje tvorbu webových aplikácií. Jeho autorom je český vývojár David Grudl. Podporuje tvorbu aplikácií založených na architektúre MVP a využíva možnosti *OO programovania*[17]. Framework tiež obsahuje nástroj *Tracy*[18], ktorý využijeme na ladenie kódu a zobrazovanie chýb pri vývoji. Jazyku *JavaScript*[19] sa využíva na obnovenie menšej konkrétnej časti stránky, aby nebolo potrebné vykonávať vždy presmerovanie.

Architektúra MVP

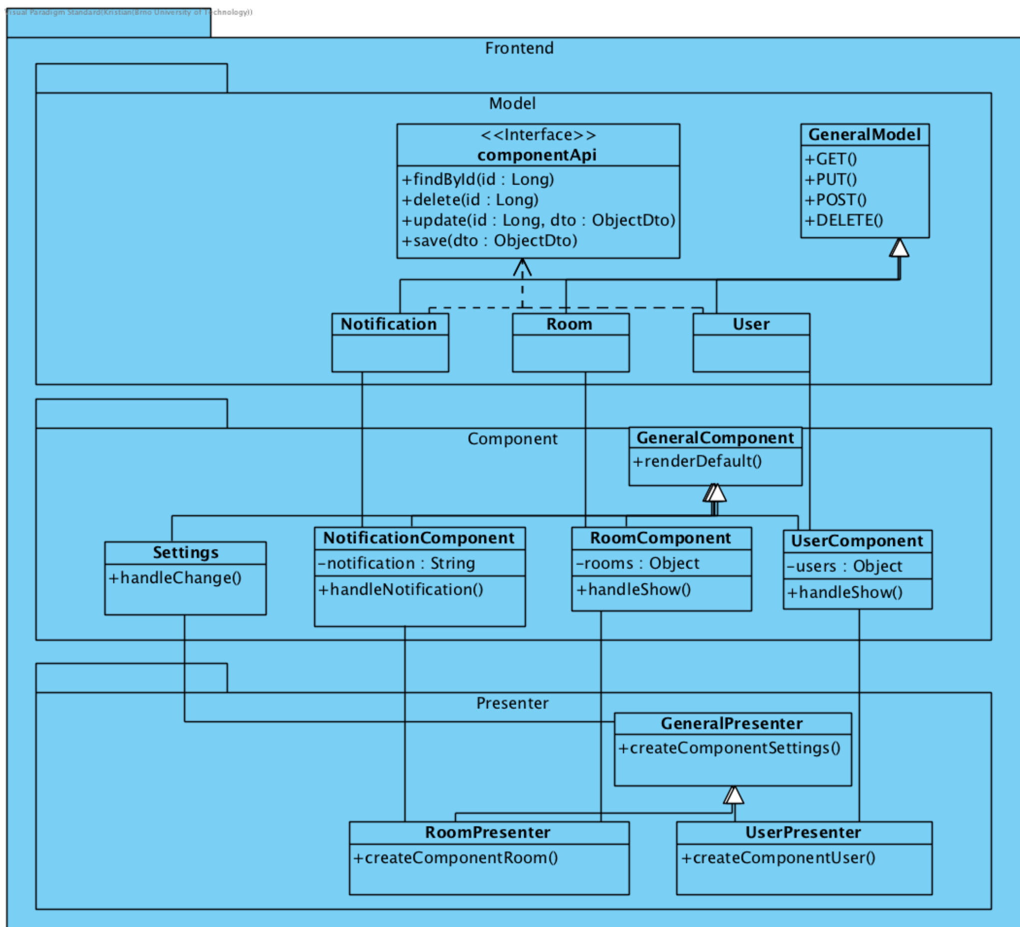
Architektúra Nette je veľmi podobná *MVC frameworku*[20] a princíp je takmer rovnaký. Aplikácia stojí na troch typoch komponent, ktoré sa v aplikácii delia o 3 základné úlohy - riadenie, logiku a výstup. Len takto rozdelená aplikácie je totiž prehľadná a rozšíriteľná.

- **Model, logika** - Obsahujú logiku aplikácie, ako napr. prácu s databázou alebo výpočty. Každá dátová entita má väčšinou svoj model.
- **View (Pohľad), výstup** - Obsahujú šablóny s HTML kódom. *Latte* [21] je šablónovací jazyk, ktorý do HTML šablón umožňuje vkladať dáta z PHP pomocou špeciálnych značiek.
- **Presenter, riadenie** - Presenter je komponenta, s ktorou komunikuje užívateľ. Odovzdá jej parametre a ona mu vráti HTML stránku. Presenter typicky parametre odovzdá modelom, od ktorých získa dáta. Tieto dáta odovzdá pohľadom (šablónam), ktoré dáta začlení do nejakého HTML kódu. Tento HTML kód pošle presenter užívateľovi do prehliadača. Funguje teda ako taký prostredník.

Architektúra aplikácie

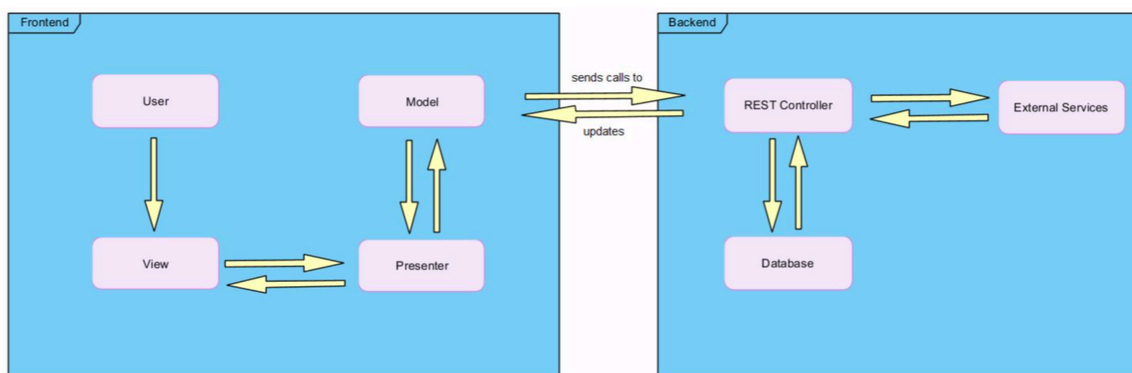
Architektúra výslednej webovej aplikácie je zobrazená na obrázku 6.10. Jednotlivé vrstvy architektúry sme si popísali už vyššie. V našom prípade `Model` predstavuje **pripojenie sa na REST API** serverovej aplikácie a prevolanie jednotlivých *endpointov*. Na tieto modely sú potom pripojené komponenty. `Component` predstavuje určitú malú časť stránky, každý komponent má svoj *Javascript* súbor aj *HTML a CSS* súbor[22]. Komponenty teda slúžia na **dekompozíciu** jednej stránky na menšie časti. Nakoniec sa tieto komponenty združujú do `Preseneter`, čo predstavuje celú jednu stránku a chová sa ako *kontajner*. Podobne ako v prípade serverovej aplikácie, aj táto aplikácia je nasadená na serveri, ale pomocou *Apache*[23] serveru.

[17] https://searchmicroservices.techtarget.com/definition/object-oriented-programming-OOP
[18] https://tracy.nette.org/cs/
[19] https://www.javascript.com/
[20] https://www.tutorialspoint.com/mvc_framework/
[21] https://latte.nette.org/en/
[22] https://www.w3schools.com/html/html_css.asp
[23] https://www.php.net/manual/en/book.apache.php



Obr. 6.10: Diagram tried webovej aplikácie.

Na obrázku 6.11 je zobrazená architektúra výsledného informačného systému. Konkrétne prepojenie serverovej a webovej aplikácie pomocou REST API.

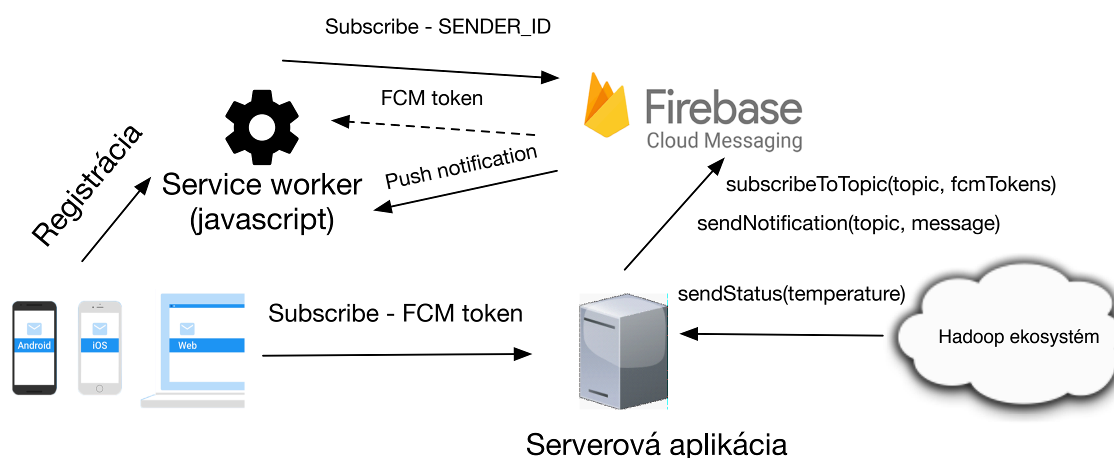


Obr. 6.11: **Architektúra informačného systému.**

6.4.3 Firebase notifikácie

Systém Firebase notifikácií prepája Hadoop ekosystém, serverovú a webovú aplikáciu. Na obrázku 6.12 je zobrazená architektúra implementácie tohto externého systému do našej práce a je zobrazená aj komunikácia medzi jednotlivými komponentami.

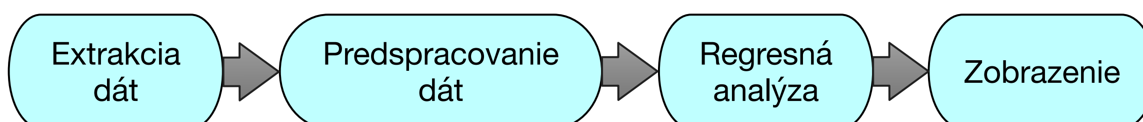
Proces začína vytvorením `Service workeru` na strane webovej aplikácie. Service worker je *script*, ktorý beží na pozadí, je ovládaný udalosťmi a je akýsi medzník medzi prehliadačom, aplikáciou a sieťou. To mu umožňuje vykonávať asynchrónne operácie. Aby sa webová aplikácia mohla k service workeru pripojiť, musí sa pripojiť cez *HTTPS*. Aplikácia sa zaregistruje na service worker a posiela svoje `SENDER_ID` na FCM, odkiaľ ako odpoveď dostane **FCM token**, ktorý mu slúži na identifikáciu v FCM a umožňuje aplikácii príjmať notifikácie. Pokiaľ používateľ v aplikácii obrdží FCM token, tak sa môže prihlásiť na odber notifikácií pomocou dotazu na backend, v ktorom pošle miestnosť a svoj FCM token. Na strane backendu sa pridá FCM token používateľa do zoznamu tokenov pre danú miestnosť a pošle to na FCM, kde sa to uloží. Následne ak príde dotaz na backend z Hadoop ekosystému, v prípade prekročenia nastavenej hodnoty, tak sa pošle zmena stavu na FCM a pošlú sa notifikácie príslušným používateľom.



Obr. 6.12: Architektúra posielania notifikácií pomocou Firebase.

6.4.4 Predikcie

Podľa nasledujúcej schémy 6.4.4 si predstavíme jednotlivé fázy, ktoré povedú od získaniu historických dát až po zobrazenie predikcií, ktoré boli vytvorené na základe nich.



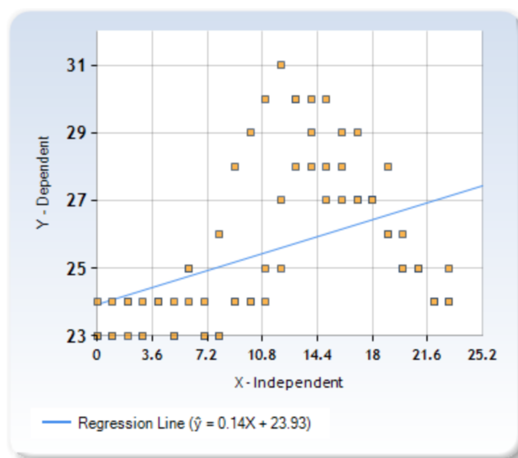
Obr. 6.13: Fáze pri predikovaní hodnôt teploty.

- **Extrakcia dát** - Na výber sú v podstate 2 možnosti. Buď sa dáta extrahujú z HDFS úložiska alebo z OpenTSDB databáze. V HDFS úložisku sú dáta nespracované, čo by skomplikovalo proces spracovania. Naopak u OpenTSDB existuje API, cez ktoré

vieme získať uložené dáta v štrukturovanom formáte. Navyše nám API umožňuje vybrať špecifické rady údajov za určité obdobie a ponúka aj *downsampling* údajov, teda môžeme údaje získať spracované podľa ponúkanej agregačnej funkcie za zvolený časový úsek.

- **Predspracovanie dát** - Po získaní dát z OpenTSDB máme priradené ku každej unikátnej časovej známke jednu hodnotu. Našim cieľom je predikovať hodnotu teploty na blízku dobu, teda predpokladajme zopár hodín dopredu. Pokiaľ sme pri extrakcii dát zvolili downsampling 1 hodinu, tak by sme mali časové známky normalizovať tiež na hodiny. Takže sa z časových známok získajú hodiny dňa. Tieto údaje posúvame ďalej do regresnej analýzy.

- **Regresná analýza** - Regresnú analýzu sme zakomponovali do nášho informačného systému. Zo Spark frameworku sme využili 2 knižnice: **Spark SQL** a **MLib** 4.4.9. Z predspracovaných dát sa vytvorí `Dataset` sada, pomocou ktorého nahráme údaje do Sparku. Pomocou Spark SQL vytvoríme pohľad a vykonáme dotaz v ktorom sa nastavia typy jednotlivým stĺpcom, pretože sa dátové typy definujú predvolene ako reťazec. Teraz potrebujeme určiť dáta, ktoré budú tvoriť množinu `features` a vytvoriť z nich vektor. `Features` tvorí množina nezávislých premenných, teda v našom prípade len jedna premenná, ktorou sú hodiny. Teplota bude tvoriť stĺpec `label`. Dátová sada sa rozdelí v pomere 30% pre testovacie dáta a 70% pre trénovacie dáta. Vytvorí sa model a pomocou triedy `LinearRegression` sa model trénuje, po trénovaní je možné model uložiť. Testovaním sa model transformuje a vytvoria sa predikcie. Pokiaľ ide o teplotu v miestnosti, tak tá čiastočne závisí aj od vonkajšieho počasia a teda v rôznych ročných obdobiach bude v miestnosti iná teplota, čiže by sa mal model ukladať len v prípade, ak máme namerané hodnoty aspoň za jeden rok, ktoré ale momentálne nemáme, čiže sa model ukladať nebude. Na obrázku 6.14 sa nachádza príklad lineárnej regresie na údajoch ktoré boli namerané v jednej z miestností. Predikcie sa budú vytvárať vo zvolenom pravidelnom intervale pomocou plánovača úloh `Scheduler` v *Spring Boot* frameworku.



Obr. 6.14: Lineárna regresia z údajov nameraných za 3 dni.

- **Zobrazenie** - Z predikovaných hodnôt sa vytvoria dátové objekty, ktoré sa cez OpenTSDB REST API nahrajú do OpenTSDB databáze ako nová časová rada a bude možné zobrazovať predikcie v rámci grafu s aktuálnym stavom.

Kapitola 7

Testovanie systému

Realizovaný systém je už hotový, nakonfigurovaný a zabezpečený tak aby sa k dátam dostali len oprávnení užívatelia. V kapitole je predstavený výsledný systém a jednotlivé metódy testovania, ktoré boli použité pre riadne overenie funkčnosti systému a splnenia požiadavok. V jednotlivých sekciách sa postupne predstaví výsledný systém a spomenú sa konfigurácie systému, ktoré museli byť z určitých dôvodov zmenené. Vykoná sa demonštrácia funkčnosti od zberu dát z IQRF siete až po ich vizualizáciu. Odskúšaný systém sa použije pre testovanie pod určitou záťažou a vyhodnotí sa stabilita systému. Vyhodnotí sa rýchlosť spracovania jednej udalosti a priestorová náročnosť zbieraných dát. A na záver sa otestuje informačný systém.

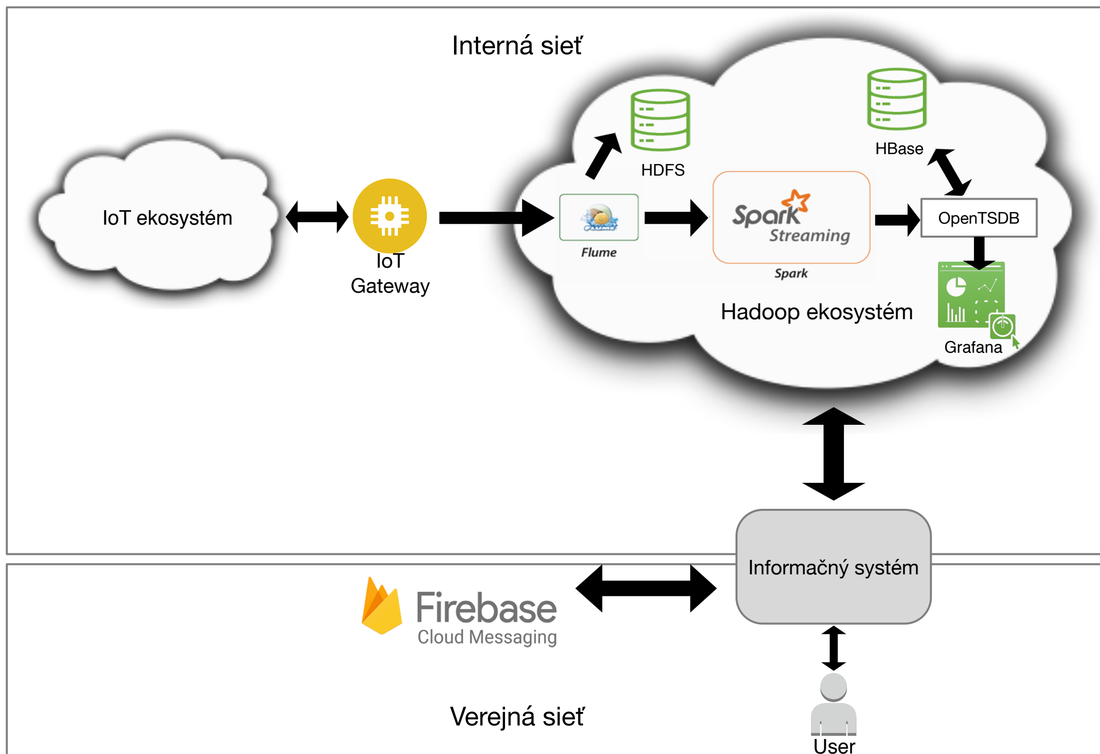
7.1 Konfigurácia systému

Výsledný systém je možné vidieť na obrázku 7.1, jeho jednotlivé časti boli predstavené podrobnejšie v kapitole 6. Ako je možné vidieť na obrázku celý systém je rozdelený na 2 časti. IoT ekosystém a Hadoop ekosystém ktoré sú v internej sieti (neprístupné z verejnej siete) a informačný systém, ktorý je prístupný z internej aj verejne siete, teda prepája užívateľa s Hadoop ekosystémom. Informačný systém je prístupný cez zabezpečený protokol *HTTPS* cez odkaz IoT-offices. V tejto časti bude predstavená konfigurácia niektorých komponentov, ktorá sa môže čiastočne líšiť pri použití systému na iný účel, resp. rozšírenie systému. Na tomto systéme sa bude demonštrovať aj funkčnosť celého systému.

Konfigurácia serverov v Hadoop clustri 6.1 sa menila po inštalácii *HDP* systému, keď pôvodných 4GB RAM pamäti nebolo dostatočných na `Master` uzli - navýšilo sa na 8GB. Na ďalších uzloch sa navýšilo na 8GB po tom, čo integrácia *Flume* komponenty s *Morphline* pluginom vyžadovala väčšiu pamäť RAM. Okrem toho nebolo potrebné meniť konfiguráciu serverov.

Flume agent pri inicializácii spojenia cez *WebSocket* s *IoT gatewayom* zasielal pôvodne správu s `crontabom`[1] na zasielanie správ každú sekundu. Avšak po 3 dňoch testovania sa zahltila fronta na strane IoT gatewayu a bolo nutné reštartovať systém. Zistilo sa, že pre typ príkazu, s ktorým zisťujeme namerané hodnoty je nastavená 1 sekunda veľa a nestíha spracovať príkaz v IoT ekosystéme. Takže bolo nutné nastaviť crontab na 10 sekúnd. Na IoT gateway bolo pripojených 5 senzorov, rozložených v piatich miestnostiach. Flume bol nakonfigurovaný podľa konfigurácie, ktorá bola spomenutá v kapitolách 6 a 5. Pre uloženie súborov na *HDFS* bolo nastavené ukladanie udalostí po 1000 nazbieraných udalostí. Kom-

[1] https://crontab.guru/



Obr. 7.1: Výsledný kompletný systém použitý pri testovaní.

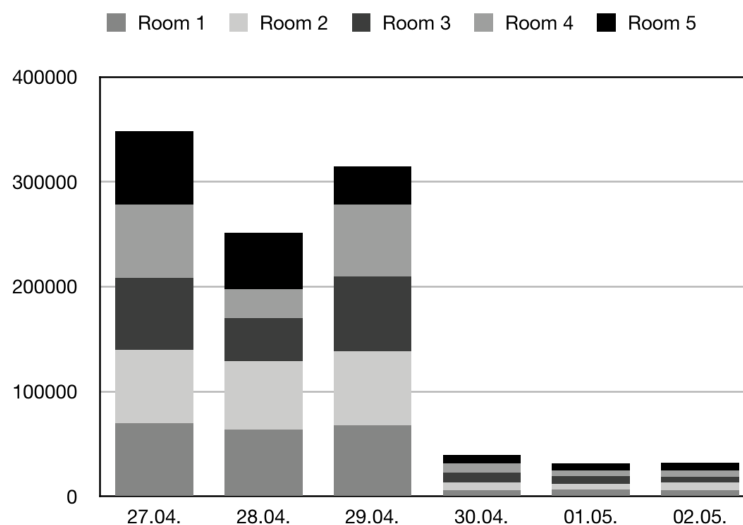
pletnú konfigúraciu nájdete na pamäťovom médiu C. Rovnako tam nájdete aj konfiguráciu Morphline pluginu.

Spark Streaming a ďalšie komponenty boli nakonfigurované podľa kapitoly 6. *Grafana* bola nastavená aby zobrazovala vývoj teploty v každej miestnosti na odlišnom `dashboarde` spolu s predikciou (metriky: `master.temperature.metrics`, `master.temperature.prediction`).

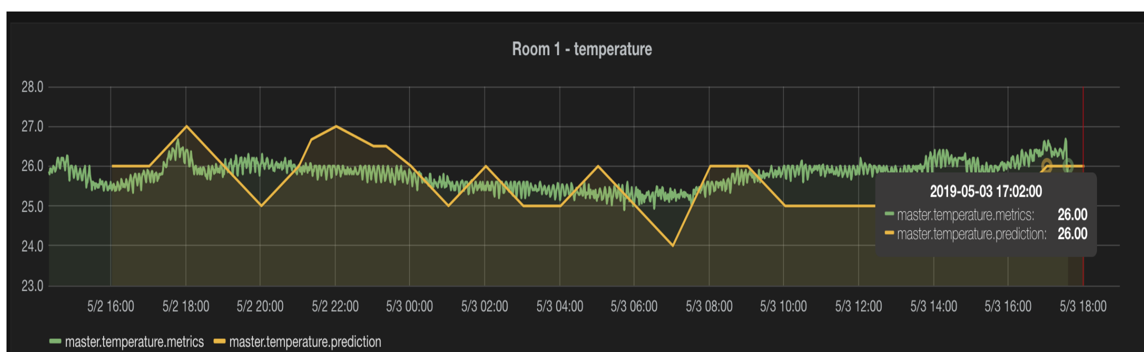
7.2 Demonštrácia funkčnosti

Namerané dáta sa zbierali **jeden týždeň**. Ako už bolo spomenuté v istom momente sa zahltila fronta na IoT gatewayi a bolo nutné zmeniť frekvenciu posielania dát z 1 sekundy na 10 sekúnd, čo sa odrazilo aj na počte nameraných hodnôt, viď graf 7.2. Počet nameraných hodnôt momentálne za deň nepresiahne **10000 nameraných hodnôt**. Pokiaľ je meranie hodnoty v IoT ekosystéme neúspešné, tak sa obdrží hodnota 0 a v takom prípade sa meranie neukladá, čo spôsobuje rôzne počty nameraných údajov v jednotlivých miestnostiach.

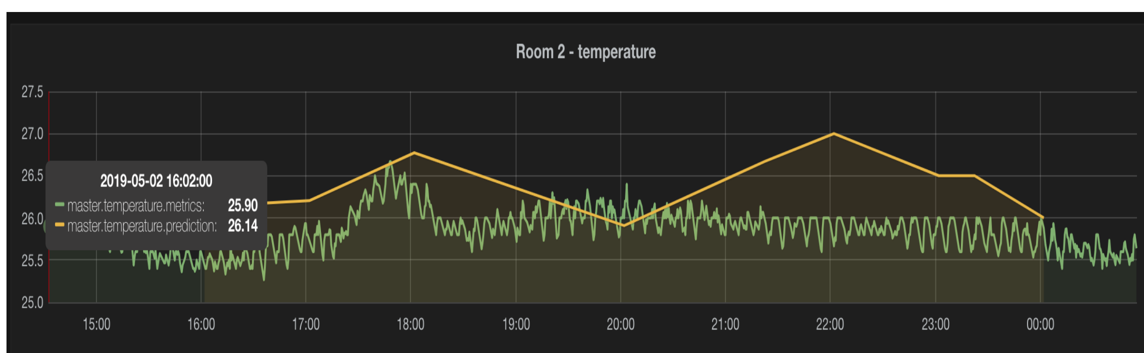
Počet nameraných hodnôt však nemá významnú rolu v prípade **predikcií** budúcej teploty v miestnosti. Hodnoty sa totiž priemerujú v rámci jednej hodiny. A následne predikované hodnoty sa zaokrúhlujú na celé čísla. Predikcie môžu byť vylepšené pomocou väčšej histórie dát, resp. použitím ďalších premenných, ktoré momentálne nemáme dostupné. V dvoch rôznych miestnostiach sú zobrazené namerané hodnoty a takisto aj predikované hodnoty za rôzne časové úseky. Vývoj hodnôt aktuálnej teploty a predikovanej teploty je možné vidieť na obrázkoch 7.3 a 7.4 vytvorené z nástroja *Grafana*.



Obr. 7.2: Počet nameraných hodnôt ze jednotlivé dni v miestnostiach.



Obr. 7.3: Vývoj hodnôt teploty a predikcie v miestnosti 1.



Obr. 7.4: Vývoj hodnôt teploty a predikcie v miestnosti 2.

7.3 Výkonnostné testovanie

Výkonnostné testovanie bolo zamerané na zaťaženie spracovacej jednotky, ktorou je *Apache Spark*. Zaťaženie prebiehalo vytvorením **viacerých procesov** komponenty *Flume agent*, ktoré navyše vytvárali plánované úlohy v *IQRF sieti* na každú sekundu. Je potrebné zmieniť,

že naplánovaná úloha zotrvá v IQRF sieti až do manuálneho zmazania úlohy priamo v IQRF *gatewayi*. Udalosti sú posielané bezohľadu na to, či sa Flume agent vypne, teda počet Flume agentov nemusí byť priamo úmerný počtu naplánovaných úloh. Nastáva však komplikácia, keď viaceré komponenty Flume agenta začnú príjmať tie isté udalosti, tým sa udalosti **zduplikujú**, čiže takýto systém nie je vhodný pre bežné použitie. Avšak vyhovuje pre testovanie výkonu a stability Apache Spark komponenty.

Apache Spark ponúka webové rozhranie **Spark UI**, ktoré už výkonáva určité merania nad spusteným `Spark Streaming jobom`. Tieto merania nám pomáhajú ladiť aplikáciu a tiež zistiť výkonnosť a stabilitu systému. Keďže je aplikácia spustená v tzv. `cluster mode`, čiže `master` je prepojený s komponentou *YARN* 6.3.1, tak nie je presne definované na ktorom uzle v clustri Spark UI pobeží. V prípade spustenia na `locale`, Spark UI pobeží na `http://localnode:4040`. Táto informácia sa dá dohladať v logoch Sparku alebo v *ResourceManager UI* (http://masternode:8088). Spark UI beží len počas doby zapnutého Spark Streaming jobu a pri vytvorení viacerých `SparkContextov` sa port inkrementuje, v našom prípade bol však spustený iba 1 SparkContext 6.3.1.

Rozoberieme jednotlivé metriky, ktoré budeme sledovať:

- **Processing Time** - Čas potrebný na spracovanie danej dávky pre všetky operácie, od začiatku po koniec. V našom prípade to znamená úlohy vymenované v sekcii 6.3.1.
- **Scheduling Delay** - Čas potrebný pre plánovač Spark Streamingu na vykonanie všetkých operácií v jednej dávke. Počíta sa to nasledovne. Predpokladajme, že sa naša dávka odoberá zo zdroja (v našom prípade **Flume collector**) napríklad každé 3 sekundy. Teraz predpokladajme, že výpočet danej dávky trval 8 sekúnd. To znamená, že sme teraz 8 - 3 = 5 sekúnd pozadu, čo vytvorí oneskorenie plánovania 5 sekúnd.
- **Total Delay** - Toto je výpočet `Processing Time` + `Scheduling delay`. Podľa predchádzajúceho príkladu, ak máme 5 sekundové oneskorenie plánovania, a ďalšia dávka trvala ďalších 8 sekúnd, to znamená, že celkové oneskorenie je teraz 8 + 5 = 13 sekúnd.

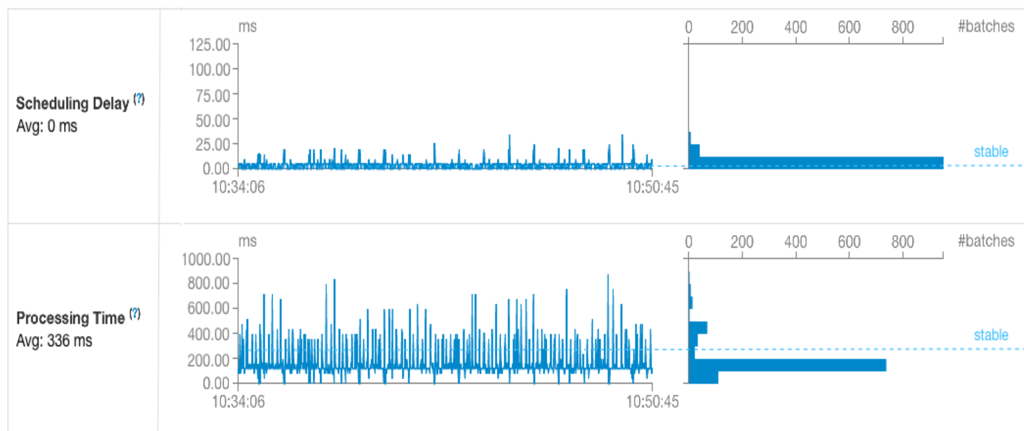
Zaťaženie nášho systému trvalo približne **2 hodiny**, pričom sme mali nastavené **spracovanie dávok po 1 sekunde** a podarilo sa nahrať do jednej dávky približne **100 udalostí**.

Batch Time	Records	Scheduling Delay (?)	Processing Time (?)	Total Delay (?)	Output Ops: Succeeded/Total
2019/05/08 11:57:40	87 records	0 ms	652 ms	652 ms	1/1
2019/05/08 11:57:39	95 records	2 ms	859 ms	861 ms	1/1
2019/05/08 11:57:38	94 records	4 ms	506 ms	992 ms	1/1
2019/05/08 11:57:37	82 records	30 ms	351 ms	361 ms	1/1
2019/05/08 11:57:36	101 records	0 ms	867 ms	867 ms	1/1
2019/05/08 11:57:35	89 records	1 ms	421 ms	422 ms	1/1

Obr. 7.5: Spracované dávky v Spark Streaming jobe za určitý časový úsek.

Na obrázku vyššie 7.5 je možné vidieť jednotlivé spracované časové dávky v každej sekunde. **Počet spracovaných udalostí** sa líši v každej dávke a takisto aj čas spracovania. Avšak každá dávka sa stihla spracovať danom časovom itervale 1 sekundy, takže nevznikali žiadne významné oneskorenia. Oneskorenia, ktoré vznikali sú zanedbateľné. Pri väčšom objeme dát - väčších dávkach za 1 sekundu, by sme už pravdepodobne potrebovali posilniť systém. Buď pridaním ďalšieho uzlu, alebo navýšením RAM pamäte.

Na ďalšom obrázku 7.6 je zobrazená štatistika Spark Streaming jobu za určitý časový úsek. Zo štatistík vidíme 2 grafy 2 premenných, ktoré nám určujú **stabilitu systému**.



Obr. 7.6: Štatistiky streamingu.

Priemerný čas spracovania je `336 ms` a priemerné oneskorenie plánovania je `0 ms`, resp. je také malé, že je zanedbateľné. Na pravej strane obrázku je možné vidieť počet spracovaných dávok v jednotlivých časových intervaloch. Následne je tam vyhodnotenie, z ktorého vidíme, že systém je stabilný. Ak by sa priemerná doba spracovania priblížila k veľkosti intervalu dávok alebo zväčšila, potom by streamingová aplikácia začala radiť do fronty udalosti na spracovanie, čo bude mať za následok **oneskorenie**, čo môže viesť k zlyhaniu Spark Streaming jobu.

7.4 Rýchlosť spracovania a priestorová náročnosť

Pri rôznych testovaniach a skúškach funkčnosti sme sledovali, zaznamenávali aj ďalšie vlastnosti systému. Jednou z nich bolo meranie času spracovania jednej udalosti v komponente Flume. Čas sa začínal merať zachytením prichádzajúcej udalosti v logu `/var/log/flume/flume.log` vo *Flume agent* a porovnal sa s časom, ktorý sa zachytil pred odoslaním udalosti na *Flume collector*. Vykonalo sa približne 20 meraní pri nezaťaženom systéme a spravil sa priemer hodnôt. Následne sa sledoval spotrebovaný úložný priestor v *HBase*. Keďže časové rady uložené v *OpenTSDB* sa v skutočnosti ukladajú v HBase v binárnom formáte (ukladanie časových rád z OpenTSDB v HBase je podrobne vysvetlené na nasledujúcom odkaze: HBase Schema), ako to už bolo spomenuté v teoretickej časti 4.4.2, sledoval sa spotrebovaný úložný priestor v HBase. Sledovať úložný priestor je možné po prihlásení sa cez webové rozhranie *HBase Master UI*, na ktorý je možné sa dostať cez monitorovací systém *Ambari* 4.4.12. V monitorovacom systéme HBase je možné sledovať aj ďalšie údaje, sledovať stav úložiska je možné na odkaze `http://<hbase-master-node>:16030/rs-statusbc_l1`. Sledoval sa aj čas kompletného spracovania jednej udalosti. Pri odoslaní udalosti z IQRF siete sa každá udalosť označila časovou známkou, ktorá sa porovnala s časovom známkou, ktorá bola použitá pri uložení dátového objektu do OpenTSDB. Zase sa vykonalo 20 meraní a spravil sa priemer. Obsah *HDFS* úložiska je možné sledovať tiež cez webové rozhranie Ambari. Ako to už bolo spomenuté aj v návrhu 5.3.2, pre možnosti neskoršieho spracovania sa ukladajú prijaté udalosti z IQRF siete do HDFS úložiska. Aktuálne je nastavené aby sa udalosti ukladali po prijatých 1000 udalostiach. Tieto udalosti sa nachádzajú v súborovom systéme HDFS v priečinku `/tmp/flume/events`. Spomínané hodnoty sú zaznamenané nižšie.

- **Spracovanie udalosti vo Flume** - priemerný čas 7ms
- **Spotrebovaný úložný priestor v HBase** - pri 500 000 OpenTSDB záznamoch bol spotrebovaný priestor 1.3MB.
- **Spracovanie udalosti od prijatia vo Flume až po uloženie do OpenTSDB** - priemerný čas 3 sekundy.
- **Spotrebovaný úložný priestor v HDFS** - 401KB/1000 udalostí, približne každé 3 hodiny sa vytvorí nový súbor.

7.5 Informačný systém

Testovanie informačného systému bolo rozdelené na 2 časti. Pre serverovú aplikáciu boli vytvorené automatizované testy a webová aplikácia bola testovaná ručne. Ako už bolo spomenuté v sekcii 6.4.1, serverová aplikácia vytvára REST API rozhranie, ktorá je zdokumentovaná pomocou nástroja Swagger. Tento nástroj čiastočne umožňuje aj prevolávanie jednotlivých endpointov, ale hlavne ich vystavuje pre webové aplikácie cez anotácie vo frameworku *Spring Boot* 6.4.1. Dokumentácia nášho systému sa nachádza v prílohe B.1. Lepší nástroj na skúšanie a testovanie endpointov sa volá Postman. Tento nástroj navyše umožňuje vytváranie kolekcií testov, kde test sa skladá z dotazu na endpoint a testovanie odpovede. Je možné vytvárať aj scenáriá, kde jeden test nasleduje ďalší, čiže je možné otestovať celý *Use-case*. Pomocou nástroja Newman sa dajú potom tieto testy spúšťať cez príkazový riadok a teda automatizovať. Takéto testy slúžia hlavne na odhalenie chýb v celom systéme pri zmenách pred zaverzovaním týchto nových zmien do spoločného repozitáru. Testy sú súčasťou repozitáru serverovej aplikácie.

Kapitola 8

Záver

Cieľom tohto projektu bolo zoznámenie sa s problematikou spracovávania *BigData* v prostredí IoT zariadení, aby sme následne dokázali vytvoriť vlastný systém, pomocou ktorého budeme vedieť spracovávať a ukladať dáta z viacerých *IoT* sietí. Keďže formát vstupných dát nebolo možné špecifikovať, tak bolo potrebné analyzovať možnosti obecného uloženia dát. Pred konečným uložením sa ale vyžadovalo záznamy analyzovať a vyhodnotiť v reálnom čase. Na základe analýz sa mal systém navrhnúť a realizovať. Pre odskúšanie systému sa mali použiť vstupné dáta z *IQRF* siete. Výsledný systém mal umožniť vizualizáciu spracovaných dát a predikcie, ktoré z nich mali byť vytvorené. Systém mal tiež upozorniť užívateľa v reálnom čase, pokiaľ boli hodnoty mimo jeho zvolený rozsah.

Výsledkom práce je analýza aspektov *BigData* a *IoT*, ich relácie, ukladania a spracovania. Podrobnejšie sa pojednávalo o možnostiach *Hadoop ekosystému*, keďže tvorí podstatnú časť tohto projektu. Pre systém, ktorý údaje príjma, spracuje a ukladá bola vybratá distribúcia Hadoopu spoločnosti *Hortoworks*. Táto distribúcia sa nainštalovala na cluster, ktorý tvorili 3 servery, ktorých konfigurácie boli škálované na minimálne možné hodnoty. Celý proces začína pripojením sa na *IQRF* sieť, kde bola vybratá možnosť pripojenia sa cez *WebSocket* protokol. Na nahrávanie dát v reálnom čase do nášho systému bola vybratá komponenta *Apache Flume*, ktorá je súčasťou zvolenej distribúcie. Keďže komponenta Flume nepodporovala protokol *WebSocket*, tak sa použil `plugin`, ktorý to umožnil. Z IQRF *gatewayu* sa príjmajú aj neúplné správy, prípadne nežiadúce správy a takisto sa predpokladalo, že vstupné dáta nemusia byť vždy v tom istom formáte v prípade rôznych IoT sietí. Tieto skutočnosti viedli k **predspracovaniu dát** vo Flume komponente použitím a integrovaním komponenty *Morphline*, ktorá ponúka široké možnosti pre predspracovanie udalostí. Po výskume sa zvolil jednotný formát dát *Avro*, ktorý slúži na zjednotenie vstupných udalostí v rôznych formátoch na jeden formát, s ktorým sa potom ďalej pracuje. Pomocou komponenty Morphline sa odfiltrujú nežiadúce správy a skonvertuje sa udalosť na Avro formát. Komponenta Flume sa v našej realizácii skladá z najmenej dvoch uzlov, kde jeden uzol `agent` zbiera udalosti, predspracuje a preposiela na druhý uzol, ktorým je `collector` a ten udalosti **uloží do súborového systému** *HDFS* pre neskoršie spracovanie alebo posunie ďalej komponente *Apache Spark* na **spracovanie v reálnom čase**. Pri štarte Spark aplikácie sa načítajú nastavenia používateľa z *informačného systému*. V Sparku sa dáta síce spracovávajú v reálnom čase, ale ten je rozdelený na menšie dávky aby bolo možné dáta zhlukovať v prípade potreby. V prípade hodnôt mimo rozsah, ktorý bol definovaný používateľom sa posiela notifikácia do informačného systému, inak sa udalosť **uloží do OpenTSDB** databáze, čo je nadstavba s podporou pre časové rady dát nad *HBase* databázou. OpenTSDB navyše podporuje integráciu so systémom *Grafana*, ktorá umožňuje

vizualizáciu hodnôt. Hadoop ekosystém a IoT ekosystém sú prístupné len z internej siete. **Informačný systém** sa skladá z webovej a serverovej aplikácie, kde serverová aplikácia je napísaná v jazyku *Java* s použitím frameworku *Spring Boot* a webová aplikácia je napísaná v jazyku *PHP* s použitím frameworku *Nette*. Informačný systém prepája používateľa s Hadoop ekosystémom, ktorý **notifikuje používateľa asynchrónne** v prípade hodnôt mimo rozsah využitím technológie *Firebase*, kde môže rozsah sám používateľ modifikovať a takisto sa v informačnom systéme overujú používateľove prístupy. Pre užívateľa je informačný systém prístupný z verenej siete cez protokol *HTTPS*. Výsledný systém bol na záver **otestovaný** aj výkonnostným testovaním, na základe ktorej sa určilo, že systém je stabilný pri určitom zaťažení. Pre serverovú aplikáciu sa vytvorili automatizované testy a webová aplikácia bola otestovaná ručne na základe prípadov použitia. Z výsledkov testov vyplývajú aj **splnené požiadavky** na systém.

Tento projekt by mal byť štartom k výslednému veľkému systému, ktorý bude schopný spracovávať údaje z rôznych IoT sietí a z veľkého počtu senzorov, ktoré budú produkovať rôzne dáta. Aby to tento systém zvládol budú nutné niektoré rozšírenia a riešenia potencionálnych problémov.

Jedným takýmto **potencionálnym problémom** v budúcnosti je nestabilita komponenty *Apache Flume* 4.4.8. Pri výkonnostnom testovaní sa nevyskytli žiadne problémy s komponentou *Flume*, avšak v rôznych recenziách sa spomína, že *Flume* môže byť v niektorých prípadoch nestabilný. Napríklad pri nastavení kapacity *Flume channel* na vysokú hodnotu môže mať za následok, že sa dáta stratia v prípade výpadku *Flume collector* komponenty. Vylepšením by mohla byť kombinácia s komponentou *Apache Kafka*. V porovnaní s Flume, Kafka ponúka lepšiu škálovateľnosť a trvalosť správ. Najlepšou kombináciou komponetov Kafka a Flume sú použitie *Flume source*, *Kafka channel*, *Flume sink*. Udalosti Flume, ktoré prechádzajú cez *Kafka channel*, sú uložené a replikované cez *Kafka brokery* pre elastickosť. Ďalšou zmenou by mohlo byť použitie novej verzie komponenty *Apache Ambari* 4.4.12. Nová verzia má vynovený dizajn, ponúka aj **lepšie monitorovanie** a ďalšie vymoženosti, ktoré pri práci s Hadoop ekosytémom významne pomáhajú, a ktoré staršia (aktuálna) verzia neobsahuje. Táto nová verzia Ambari už prichádza s novou verziou *HDP* 4.4, ktorej inštalácia by stála za zváženie.

Keďže sa tento systém bude chcieť využívať pre väčšie množstvo vstupných dát, tak bude nevyhnutné **horizontálne aj vertikálne škálovanie**. Bude potrebné pridať ďalší/ďalšie uzol/uzly a zvýšiť pamäť RAM, takisto zvýšiť kapacitu úložiska a navýšiť počet jadier procesora. V prípade žeby sa malo nasadiť centrálne úložisko pre produkované súbory a logy aplikácií, tak nie je nutné navýšiť kapacitu úložiska jednotlivých uzlov, avšak bude potrebný nový server kam sa tieto dáta budú môcť preposielať a ukladať. Taktiež bude potrebné nastaviť ukladanie a zálohovanie dát.

Riešenie všetkých týchto vylepšení systému bude mať zmysel v prípade rozšírenia systému o ďalší vstup. Či to už bude nový typ IoT siete, v ktorom bude odlišný typ IoT *gatewayu*, ktorý bude generovať odlišné typy dát cez odlišný protokol alebo to bude len väčšie množstvo senzorov, ktoré budú zbierať odlišné a väčšie množstvo dáta - vo všetkých prípadoch pôjde o napredovanie projektu a rozširovanie celého systému.

Literatúra

[1] Ahmed, E.; Yaqoob, I.; Hashem, I.; aj.: The role of big data analytics in Internet of Things. *Computer Networks*, 12 2017, doi:10.1016/j.comnet.2017.06.013.
URL https://www.researchgate.net/publication/317617290_The_role_of_big_data_analytics_in_Internet_of_Things

[2] Aljawarneh, S.; Radhakrishna, V.; Kumar, P. V.; aj.: *A similarity measure for temporal pattern discovery in time series data generated by IoT*. 2016, ISBN 978-1-5090-5579-1, 1–4 s.

[3] Apache Hadoop community: Hadoop. 2018.
URL http://hadoop.apache.org

[4] Apache HBase Team: Apache HBase Reference Guide.
URL https://hbase.apache.org/book.html

[5] Bashir, M. R.; Gill, A. Q.: *Towards an IoT Big Data Analytics Framework: Smart Buildings Systems*. Dec 2016, ISBN 978-1-5090-4297-5, 1325-1332 s., doi:10.1109/HPCC-SmartCity-DSS.2016.0188.

[6] Brewer, E. A.: Towards robust distributed systems. ročník 7, 2000, doi:10.1145/343477.343502.

[7] Cai, H.; Xu, B.; Jiang, L.; aj.: IoT-Based Big Data Storage Systems in Cloud Computing: Perspectives and Challenges. *IEEE Internet of Things Journal*, ročník 4, č. 1, Feb 2017: s. 75–87, ISSN 2327-4662, doi:10.1109/JIOT.2016.2619369.

[8] Chandio, A. A.; Tziritas, N.; Xu, C.-Z.: Big-data processing techniques and their challenges in transport domain. *ZTE Communications*, ročník 1, č. 010, 2015, doi:10.3969/j.issn.1673-5188.2015.01.007.
URL https://www.researchgate.net/profile/Aftab_Chandio/publication/275518375_Big-Data_Processing_Techniques_and_Their_Challenges_in_Transport_Domain/links/553e16930cf2fbfe509b8fc9/Big-Data-Processing-Techniques-and-Their-Challenges-in-Transport-Domain.pdf

[9] Firebase community: Firebase Cloud Messaging.
URL https://firebase.google.com/docs/cloud-messaging/

[10] framework Nette, Český: nette.org [online],[cit. 11.5. 2019].
URL http://nette.org/cs

[11] Gamage, T. A.: HTTP and Websockets: Understanding the capabilities of today's web communication technologies. 2017.
URL https://medium.com/platform-engineer/web-api-design-35df8167460

[12] Grafana community: Grafana Labs: Grafana Documentation.
URL https://grafana.com/docs/

[13] Holubová, I.; Kosek, J.; Minařík, K.; aj.: *Big Data a NoSQL databáze*. Grada, 2015, ISBN 978-80-247-5466-6.

[14] Hopkins, C.: *PHP okamžitě*. Computer Press, Albatros Media as, 2017, ISBN 978-80-251-4196-0.

[15] Hortonworks community: HDP Projects.
URL https://hortonworks.com/ecosystems/

[16] Jin, J.; Gubbi, J.; Luo, T.; aj.: Network architecture and QoS issues in the internet of things for a smart city. Oct 2012, doi:10.1109/ISCIT.2012.6381043.
URL https://ieeexplore.ieee.org/document/6381043

[17] Landset, S.; Khoshgoftaar, T. M.; Richter, A. N.; aj.: A survey of open source tools for machine learning with big data in the Hadoop ecosystem. *Journal of Big Data*, ročník 2, č. 1, 2015: str. 24, ISSN 2196-1115.

[18] Loughran, S.: JobTracker. 2010.
URL https://wiki.apache.org/hadoop/JobTracker

[19] Manoj, K.: Why Java Is The Future Of Big Data, And The IoT. 2018.
URL https://think360studio.com/why-java-is-the-future-of-big-data-and-the-iot/

[20] Marjani, M.; Nasaruddin, F.; Gani, A.; aj.: Big IoT data analytics: architecture, opportunities, and open research challenges. *IEEE Access*, ročník 5, 2017: s. 5247–5261, doi:10.1109/access.2017.2689040.

[21] Marr, B.: *Big Data: Using SMART big data, analytics and metrics to make better decisions and improve performance*. John Wiley & Sons, 2015, ISBN 978-1-118-96583-2.

[22] Prasad, S.; Avinash, S.: *Smart meter data analytics using OpenTSDB and Hadoop*. 2013, ISBN 978-1-4799-1347-3, 1–6 s.

[23] Redakcia: WiFi HaLow. 2018.
URL https://www.iot-portal.cz/2016/02/29/wi-fi-halow/

[24] Redakcia: ZigBee. 2018.
URL https://www.iot-portal.cz/2016/02/24/zigbee/

[25] Rouse, M.: Horizontal scalability (scaling out).
URL https://searchcio.techtarget.com/definition/horizontal-scalability

[26] Rouse, M.: Search Business Analytics, Hadoop cluster. 2013.
URL http://searchbusinessanalytics.techtarget.com/definition/Hadoop-cluster

[27] Salman, T.; Jain, R.: Networking protocols and standards for internet of things. 2015. URL https://www.cse.wustl.edu/~jain/cse570-15/ftp/iot_prot.pdf

[28] Schneider, R.: Hadoop Buyer's Guide. 2013.
URL
https://www.wwt.com/wp-content/uploads/2015/03/Hadoop-Buyers-Guide.pdf

[29] Shubham, S.: Hadoop Ecosystem: Hadoop Tools for Crunching Big Data. 2018. URL https://www.edureka.co/blog/hadoop-ecosystem

[30] Vývojári: IQRF GW Daemon. 2018.
URL https://www.iot-portal.cz/2016/02/29/wi-fi-halow/

[31] Webb, P.; Syer, D.; Long, J.; aj.: Spring boot reference guide. *Part IV. Spring Boot features*, 2018.
URL
https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/

Príloha A

IQRF DTO - dotaz a odpoveď


```
{
    "mType": "iqrfEmbedFrc_Send",
    "data": {
        "msgId": "testEmbedFrc",
        "rsp": {
            "nAdr": 0,
            "hwpId": 0,
            "rCode": 0,
            "dpaVal": 75,
            "result": {
                "status": 7,
                "frcData": [
                    0,27,26,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
                    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
                ]
            }
        },
        "raw": [
            {
                "request": "00.00.0d.00.ff.ff.80.00.00",
                "requestTs": "2019-03-27T22:44:51.261313",
                "confirmation": "",
                "confirmationTs": "",
                "response": "00.00.0d.80.00.00.00.4b.07.00.1b.1a.00.00.00.00.
                00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.
                00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.00.
                00.00.00.00.00.00.00.00",
                "responseTs": "2019-03-27T22:44:52.315759"
            }
        ],
        "insId": "iqrfgd2-default",
        "statusStr": "ok",
        "status": 0
    }
}
```


Obr. A.1: IQRF response DTO

```
{
	"mType": "mngScheduler_AddTask",
	"data": {
		"msgId": "testSchedulerAdd",
		"req": {
			"clientId": "SchedulerMessaging",
			"task": {
				"messaging": "WebsocketMessaging",
				"message": {
					"mType": "iqrfEmbedFrc_Send",
					"data": {
						"msgId": "testEmbedFrc",
						"req": {
							"nAdr": 0,
							"param": {
								"frcCommand": 128,
								"userData": [0,0]
							}
						},
						"returnVerbose": false
					}
				}
			},
			"timeSpec": {
				"cronTime": ["*/10","*","*","*","*","*","*"],
				"periodic": false,
				"period": 0,
				"exactTime": false,
				"startTime": ""
			},
			"persist": true
		},
		"returnVerbose": true
	}
}
```


Obr. A.2: IQRF request DTO

Príloha B

REST API Serverovej aplikácie

swagger
Select a spec default

Api Documentation 1.0
[Base URL: 89.185.246.84:8080/iot_bigdata_controller]
http://89.185.246.84:8080/iot_bigdata_controller/v2/api-docs

Api Documentation

Terms of service
Apache 2.0

Authorize

notification-controller Notification Controller

POST /v1/notification/status Send temperature notification to room.

prediction-controller Prediction Controller

POST /v1/prediction DO prediction.

room-controller Room Controller

PUT /v1/room/{roomId}/temperature/{temperature} Change temperature of room.
GET /v1/room/all Get rooms
POST /v1/room/subscribe Subscribe user to room notifications.
POST /v1/room/unsubscribe Unsubscribe user from room notifications.
GET /v1/room/users/all Get rooms with users

user-controller User Controller

POST /v1/user/change_password Change password of user
POST /v1/user/login Login user
POST /v1/user/register Register brand new user
GET /v1/user/user Get user or verify if user exists

Obr. B.1: REST API Serverovej aplikácie

Príloha C

Obsah CD

- Dokumentácia práce - zdrojové súbory a PDF (**dokumentacia**)
- Konfiguračné súbory komponenty Flume (**flume_conf**)
- Konfiguračné súbory komponenty Morphline (**morphline_conf**)
- Spark Streaming aplikácia (**spark_streaming**)
- Serverová aplikácia (**backend**)
- Webová aplikácia (**frontend**)