VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

CLOUD COMPUTING SYSTÉM PRO REAL-TIME ZPRACOVÁNÍ DAT

CLOUD COMPUTING SYSTEM FOR REAL-TIME DATA PROCESSING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. DAŠA NOSKOVÁ**
AUTHOR

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

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání diplomové práce

150725

Ústav:	Ústav informačních systémů (UIFS)
Studentka:	**Nosková Daša, Bc.**
Program:	Informační technologie a umělá inteligence
Specializace:	Softwarové inženýrství
Název:	**Cloud computing systém pro real-time zpracování dat**
Kategorie:	Softwarové inženýrství
Akademický rok:	2023/24

Zadání:

1. Seznamte se s prostředím cloud computing, typy aplikací pro toto prostředí a způsobem poskytování služeb. Prozkoumejte možnosti vývoje aplikací pro zpracování dat v prostředí cloud computing. Zaměřte se zejména na architekturu, proudové zpracování a na kvalitativní aspekty (např. rychlost zpracování, doba odezvy, využití vyrovnávací paměti).
2. Prozkoumejte možnosti ukládání a zpracování dat v reálném čase v prostředí cloud-computing. Prostudujte a porovnejte existující řešení pro ukládání a zpracování dat v reálném čase a jak se takové systémy navrhují i implementují v cloud-computing.
3. Navrhněte systém pro cloud-computing prostředí, který bude schopen uložit a zpracovat velký tok geo-lokalizovaných temporálních dat (např. polohy, data ze senzorů atd.) a v reálném čase je poskytovat klientům v různých časově omezených pohledech (tzn. data platná v určitý čas nebo historický interval). Zvolte vhodný typ databáze.
4. Po konzultaci s vedoucím implementujte funkční prototyp navrženého systému a pomocí testů ověřte jeho výkon. Např. jak rychle je systém schopen zpracovat tok 1 mil. ukládaných záznamů (ideálně za minutu) a přitom obsloužit 1000 klientů dotazující data v různých pohledech (ideálně se zpožděním do 5 sekund); jaké je urychlení v případě opakovaných dotazů (ideálně do 200 ms z původních 500 ms). Zvažte také implementaci geografických dotazů.
5. Výsledky zhodnoťte a navrhněte možná vylepšení a rozšíření.

Literatura:

- D. Comer: The cloud computing book: the future of computing explained. First edition. ISBN 978-0-367-70680-7
- Ch. Fehling, F. Leynman, F. Rette, W. Schupeck and P. Arbitter: Cloud Computing Patterns. Springer-Verlag Wien, 2014. ISBN 978-3-7091-1568-8
- S. R. Poojara, C.K. Dehury, P. Jakovits, S.N. Srirama: Serverless data pipeline approaches for IoT data in fog and cloud computing. Future Generation Computer Systems, Volume 130, 2022, pp. 91-105, ISSN 0167-739X, https://doi.org/10.1016/j.future.2021.12.012
- J. S. Hurwitz, D. Kirsch: Cloud Computing For Dummies. 2nd Edition. For Dummies, 2020, 320 p., ISBN 978-1119546658.

Při obhajobě semestrální části projektu je požadováno:
Body 1, 2 a 3.

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

Vedoucí práce:	**Rychlý Marek, RNDr., Ph.D.**
Konzultant:	Ing. Marek Řehulka
Vedoucí ústavu:	Kolář Dušan, doc. Dr. Ing.
Datum zadání:	1.11.2023
Termín pro odevzdání:	17.5.2024
Datum schválení:	30.10.2023

Abstrakt

Táto práca sa zaoberá návrhom systému pre spracovanie veľkého toku geo-lokalizovaných temporálnych dát v reálnom čase. Teoretická časť sa venuje konceptom a nástrojom pre spracovanie dát v reálnom čase a vlastnostiam systémov pracujúcich v reálnom čase. V práci je predstavený návrh a architektúra škálovateľného systému, ktorý využíva fronty správ. Ďalej sú priblížené hlavné koncepty implementácie navrhnutého systému, kde boli využité technológie kladúce dôraz na rýchlu odozvu. V rámci práce bol vytvorený škálovateľný systém, ktorý dokáže spracovať veľký tok dát.

Abstract

This thesis focuses on designing a system for processing a large stream of geo-localized temporal data in real-time. The theoretical part addresses concepts and tools for real-time data processing and characteristics of real-time systems. The thesis introduces the design and architecture of a scalable system that utilizes message queues. Furthermore, the main concepts of implementation of the proposed system are outlined, using technologies emphasizing fast response times. As part of the work, a scalable system capable of processing a large data stream was developed.

Kľúčové slová

spracovanie v reálnom čase, cloud computing, škálovanie, mikroslužby, prúdové spracovanie, spracovanie dát, fronta správ, škálovateľná architektúra, apache kafka, apache flink

Keywords

real-time processing, cloud computing, scaling, micro services, stream processing, data processing, message queue, scalable architecture, apache kafka, apache flink

Citácia

NOSKOVÁ, Daša. *Cloud computing systém pro real-time zpracování dat.* Brno, 2024. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce RNDr. Marek Rychlý, Ph.D.

Cloud computing systém pro real-time zpracování dat

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracovala samostatne pod vedením RNDr. Mareka Rychlého Ph.D. Ďalšie informácie mi poskytol Ing. Marek Řehulka. Uviedla som všetky literálne pramene, publikácie a ďalšie zdroje, z ktorých som čerpala.

.....
Daša Nosková
12. mája 2024

Poďakovanie

Rada by som poďakovala vedúcemu práce RNDr. Marekovi Rychlému Ph.D za poskytnuté rady a cenné odborné informácie pri vypracovávaní tejto diplomovej práce. Taktiež by som chcela poďakovať pánovi Ing. Marekovi Řehulkovi za odbornú pomoc a pozitívny a motivujúci prístup.

Obsah

Zoznam obrázkov

Kapitola 1

Úvod

S vývojom technológií sa posunuli aj možnosti toho, čo je dnes možné. Ešte pred dvadsiatimi rokmi boli počítačové aplikácie obmedzené malým objemom dát, a tak veľký objem dát, aký je produkovaný a prenášaný dnes, neexistoval. Každý „like“, komentár, nahratá fotka alebo odoslaná správa musí byť prenesená cez internet a táto informácia musí byť spracovaná. V dobe okamžitej komunikácie a sociálnych sietí sa jedná o milióny udalostí za minútu. V minulosti bola zároveň technológia limitovaná kapacitou pre ukladanie dát a dostupným výkonom. Kvôli týmto obmedzeniam nebolo možné uchovávať toľko dát, ako je možné dnes, a nízky výkon obmedzoval možnosti vykonávania komplexných analýz a transformácií dát. Postupne sa zvýšili požiadavky na objem dát potrebných k získaniu nových poznatkov a začali sa objavovať aplikácie, ktoré potrebovali pracovať s vysokým objemom dát v reálnom čase (ang. *real-time*). To znamená, že aplikácia by mala byť schopná okamžite reagovať na dáta, ktoré získala (napr. navigačný systém navrhne obchádzku dopravnej kolóny). Asi najbežnejším príkladom takejto aplikácie je instantná komunikácia. Ak vám niekto napíše správu, tak by ste ju mali dostať takmer okamžite a nie o 5 minút.

Ako reakcia na rastúce požiadavky týkajúce sa vyššieho objemu dát, časových požiadaviek na výpočet a kapacít dátových úložísk, vznikali nové technológie, ktoré umožňujú efektívne spracovanie veľkých objemov dát. Jeden z takýchto príkladov je cloud computing, ktorý umožňuje užívateľom tejto služby uchovávať, spracovávať a zdieľať dáta a aplikácie na vzdialených serveroch, pričom prístup k nim je dostupný odkiaľkoľvek a kedykoľvek cez internetové pripojenie. Cloud computing sa stal kľúčovým prvkom digitálnej transformácie, pretože zjednodušuje využívanie zdrojov a služieb na základe potreby. Vďaka tejto technológii sa môžeme dnes spoľahnúť na rýchle a spoľahlivé spracovanie dát a bez problémov využívať aplikácie pracujúce v reálnom čase. Cloud computing otvoril dvere novým inováciám a možnostiam, a preto stále viac firiem zvažuje migráciu svojich systémov na cloudovú platformu.

Cieľom tejto práce je práve demonštrovať efektívnosť a výhody real-time spracovania dát v cloud computing, konkrétne na aplikácii, ktorá zaznamenáva polohy zariadení, sleduje ich pohyb a zaznamenáva v akých oblastiach sa jednotlivé zariadenia nachádzajú. Práca je vytvorená v spolupráci s firmou GINA Software. Takýto typ aplikácie môže byť využívaný napr. bezpečnostnými zložkami alebo pri humanitárnych akciách.

Práca začína kapitolou 2, kde čitateľ nájde informácie o cloud computing, vrátane možností vývoja aplikácií v cloudovom prostredí, a prehľad aktuálnych poskytovateľov služieb. Práca pokračuje časťou 3, ktorá sa zaoberá real-time spracovaním dát. V tejto kapitole sa nachádza prehľad nástrojov a architektúr vhodných pre real-time spracovanie dát v cloud computing. Nasledujúca kapitola 4 približuje spôsoby ukladania dát v cloud computing

a rieši integráciu dátových úložísk so systémom a celkové nasadenie systému v cloud computing. Práca pokračuje kapitolou 5, kde sú na úvod v podkapitole 5.1 predstavené existujúce návrhy real-time aplikácií. Zvyšok kapitoly sa venuje návrhu prototypu real-time systému, ktorý je cieľom tejto práce. Realizácia systému na základe architektúry popísanej v podkapitole 5.3 je opísaná v kapitole 6. V tejto kapitole sú predstavené použité technológie, popis implementácie jednotlivých častí systému. Na záver kapitoly 6.6 je opísané nasadenie systému. Na záver práce 7 je uvedený popis testovania a jeho vyhodnotenie.

Kapitola 2

Čo je cloud computing

Cloud computing je výpočtový model založený na poskytovaní služieb, softvéru a výkonu cez sieť, a realizovaný cez vysoko škálovateľné a distribuované zdroje, čo znamená, že dáta a výpočtové operácie sú distribuované cez mnoho serverov a zariadení, ktoré sú navzájom prepojené sieťou [18]. Táto technológia zahŕňa mnoho aspektov, ako výpočtový výkon, infraštruktúru, aplikácie a obchodné procesy, ktoré sa dajú vnímať ako ponúkané služby a zákazníci si ich môžu upraviť podľa svojich potrieb. Jedným z kľúčových prvkov cloudového prostredia je pružnosť (ang. *elasticity*), čo umožňuje jednoduché rozširovanie alebo zmenšovanie využívaných zdrojov v súlade s aktuálnymi potrebami. Cloud sám o sebe je zložený z hardvéru, sietí, úložísk, rozhraní a služieb. Okrem pružnosti ponúka cloud aj rôzne aplikačné rozhrania (*skr. API*) a fakturáciu na základe využitia zdrojov. [20] Bližšie sa venuje charakteristike cloud computing kapitola 2.1.

Cloud computing sa dá klasifikovať podľa typu poskytovaných služieb, viac v podkapitole 2.3, a podľa operačného modelu. Podľa operačného modelu existujú rôzne kategórie cloudových prostredí, ako verejný cloud (ang. *public cloud*), súkromný cloud (ang. *private cloud*), hybridný cloud, komunitný cloud a priemyslový cloud. Verejný cloud je prístupný neobmedzenej verejnosti a väčšinou je vlastnený treťou stranou, čo znamená, že užívatelia nemajú kontrolu nad fyzickou infraštruktúrou a výpočtovými zdrojmi. Naopak, súkromný cloud patrí organizácii a umožňuje jej rozhodovať, ktorí užívatelia môžu cloud využívať. Komunitný cloud je vytvorený pre skupinu organizácií, ktoré spája nejaký vzájomný vzťah a všetci členovia tejto skupiny môžu cloud využívať. Hybridný cloud kombinuje prvky verejného a súkromného cloudu s on-premise infraštruktúrou, čo umožňuje užívateľom vlastniť časti cloudu a kontrolovať, ako tieto časti zdieľajú s ostatnými. Priemyslový cloud je určený pre potreby samotného cloudu a nie pre užívateľov ani vlastníka cloudu. [18] V rámci tejto práce je uvažované o verejnom cloude.

2.1 Charakteristika

Táto podkapitola je založená na zdrojoch [16] [20]. Medzi charakteristické vlastnosti cloud computing patria pružnosť a škálovateľnosť, samoposkytované služby na vyžiadanie, platba na základe využitia (pay-per-use), zdieľanie zdrojov a široký prístup k sieti, pretože všetky cloudové služby sa spoliehajú práve na pripojenie k sieti, hlavne na internetové pripojenie, ktoré poskytuje prístup k výpočtovým zdrojom kedykoľvek a kdekoľvek. Cloudoví poskytovatelia musia okrem toho zabezpečiť dostatočnú bezpečnosť, aby sa zabránilo neoprávnenému prístupu k informáciám klientov, využívajúcich ich služby, a poskytovať mo-

nitorovanie výkonu pre potreby optimalizácie. Zároveň musia mať cloudoví poskytovatelia štandarizované API, ktoré definujú pokyny pre komunikáciu medzi aplikáciami a dátovými zdrojmi.

Pružnosť a škálovateľnosť

Užívatelia cloudových platforiem môžu využívať služby, buď iba príležitostne, alebo pravidelne. Služba musí byť schopná prispôsobiť sa meniacim sa potrebám, v prípade zvýšenia aj zníženia spotreby, alebo podľa zmeny požiadaviek na aplikáciu. Pružnosť podporuje schopnosť škálovateľnosti, pretože ako je spomenuté nižšie, zdroje, ktoré momentálne jeden zákazník nepotrebuje, môžu byť využívané iným zákazníkom. Schopnosti pružnosti a škálovateľnosti musia byť prítomné aj pri hostovanej aplikácií.

Samoposkytované služby na vyžiadanie

Táto charakteristika funguje na princípe, kde zákazník podľa svojich špecifických požiadaviek reguluje využívanie výpočtových zdrojov, úložísk, softvéru a ďalších prostriedkov, ktoré cloudový poskytovateľ poskytuje.

Platba podľa využitia

Cloudové prostredie obsahuje zabudovanú službu, ktorá má za úlohu meranie využitia služieb a fakturáciu za poskytované služby. Takáto služba zabezpečuje transparentnosť pre poskytovateľa aj zákazníka. Platba podľa využitia poskytuje možnosť zákazníkovi platiť iba za čas, kedy je služba využívaná, a za množstvo spotrebovaných zdrojov. V časoch, keď zákazník tieto zdroje nevyužíva, môže ich poskytovateľ prideliť inému zákazníkovi.

Zdieľanie zdrojov

Aby poskytovatelia mohli poskytovať nevyužívané zdroje iným zákazníkom, sú používané zdieľané prostriedky, ktoré musia podporovať pružnosť, aby sa mohli dynamicky prideľovať zákazníkom na základe ich potrieb. Zákazník má možnosť spravovať tieto prostriedky manuálne alebo môžu byť automaticky spravované poskytovateľom.

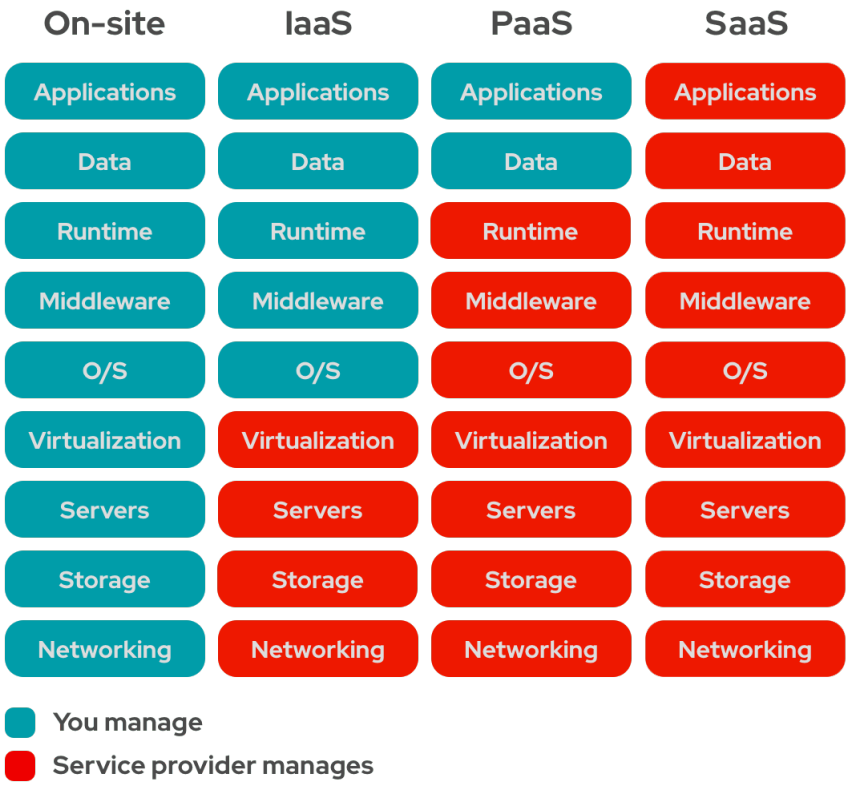
2.2 Porovnanie cloud vs on premise riešenia

Ako bolo spomenuté vyššie, pri cloudových službách zákazník nevlastní infraštruktúru, ale prenajíma si zdroje a služby od poskytovateľov cloudových služieb. Naopak, v prípade on premise riešení organizácia vlastní a riadi všetky hardvérové, softvérové a infraštruktúrne komponenty. Toto poskytuje úplnú kontrolu, ale vyžaduje si významnú počiatočnú investíciu a ďalšie náklady na údržbu a aktualizáciu hardvéru a softvéru, čo môže byť náročné z hľadiska času, personálu aj financií. Pri využívaní cloudových služieb sa o správu, bezpečnosť a aktualizáciu týchto komponentov stará poskytovateľ cloudovej služby. Výhodou cloudových služieb je aj rýchla škálovateľnosť, pretože pri náraste požiadaviek nie je potrebné investovať do nákupu a nasadenia nového hardvéru, čo je typické pre on-premise riešenia. Jedným z rozhodujúcich faktorov môže byť aj bezpečnosť. Aj keď poskytovatelia cloudových služieb musia dodržiavať bezpečnostné predpisy, on-premise riešenie poskytujú väčšiu kontrolu nad implementáciou bezpečnostných opatrení organizácie. Cloudové služby operujú

na princípe redundantných dátových centier a služieb, čo zabezpečuje vysokú úroveň spoľahlivosti služby. V prípade on-premise riešení miera spoľahlivosti závisí od infraštruktúry organizácie. Cenu on-premise riešení môžu ovplyvniť rôzne faktory, vrátane nákladov na akvizíciu hardvéru a softvérových licencií, frekvencie výmeny hardvéru, náklady na elektrickú energiu a chladenie hardvéru, náklady na údržbu hardvéru a infraštruktúry ako aj náklady na personál, zodpovedný za tieto úlohy. Pri riešení v cloud computing je dôležité zvážiť cenu za jednotlivé poskytované služby a vhodný spôsob využívania cloudových služieb [15].

2.3 Spôsoby poskytovania služieb

Cloudová služba zahŕňa široké spektrum služieb, a preto sú rôzne typy poskytovaných služieb kategorizované do troch hlavných skupín: IaaS, PaaS a SaaS. V tejto časti budú tieto kategórie služieb predstavené bližšie na základe kníh [13], [20].



Obr. 2.1: Porovnanie IaaS, PaaS, SaaS a on premise riešenia[1].

IaaS - Infrastructure as a Service

V prípade služby typu IaaS sú zákazníkovi poskytované fyzické hardvérové prostriedky, ako sú servery, sieťové zariadenia a základné úložiská dát. Okrem toho môžu byť k dispozícii ďalšie služby, ako je vyvažovanie záťaže (ang. *load balancing*), zálohovanie dát, zabezpečenie prenosu dát a pridelenie internetovej adresy. Zákazník si prenajíma výpočtovú

[1] Prevzaté z https://www.redhat.com/ko/topics/cloud-computing/iaas-vs-paas-vs-saas

techniku a nemusí investovať do nákupu a inštalácie hardvéru vo vlastnom dátovom centre. Zákazník má možnosť voľby operačného systému, aplikácií a kontrolu nad sieťovým prístupom (napr. prostredníctvom firewallu).

PaaS - Platform as a Service

PaaS poskytuje zákazníkovi vývojové prostredie, ktoré umožňuje vytváranie cloud aplikácií, pričom nevyžaduje spravovanie infraštruktúry a zariadení, ako sú servery, úložiská, operačné systémy, databázy a sieťové pripojenia. Nevýhodou tejto služby môže byť náročný proces migrácie systému na inú platformu, pretože softvér často závisí od špecifických technológií konkrétneho poskytovateľa tejto služby.

SaaS - Software as a Service

SaaS poskytuje zákazníkovi konkrétny softvér, za ktorý platí na pravidelnej mesačnej báze, namiesto jednorázového nákupu. Aplikácie typu SaaS bežia na serveroch v cloudovom centre, čo znamená, že nie je potrebná inštalácia na užívateľovom počítači a všetky dokumenty sú ukladané v cloudovom centre. Toto prináša výhody, ako napr. dostupnosť služby kedykoľvek z akéhokoľvek zariadenia, synchronizáciu dát medzi zariadeniami, poskytovanie vysokej dostupnosti a fakturáciu na základe skutočného využitia.

2.4 Vlastnosti aplikácií pre cloud computing

Cloud-native aplikácia je aplikácia, ktorá zahŕňa esenciálne vlastnosti cloudu (viď 2.1). Pre maximálne využitie služieb, ktoré ponúka cloud computing, by mala aplikácia, navrhnutá pre cloudové prostredie spĺňať, podľa zdroju [16], nasledujúce vlastnosti: izolovaný stav, distribuovanosť, pružnosť, automatická správa a minimálna previazanosť, čím po anglicky vzniká akronym IDEAL:

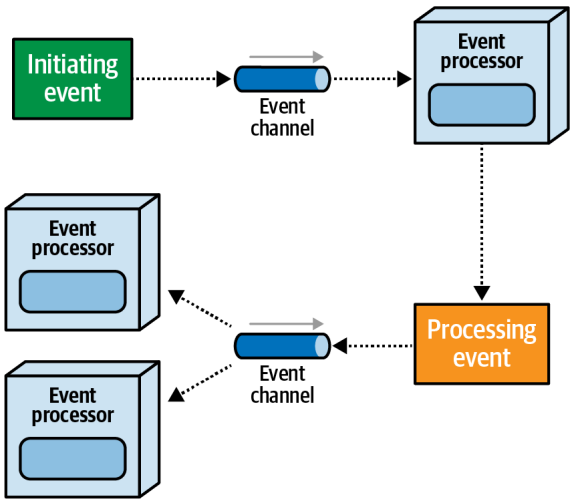
- **Distribuovanosť:** Táto vlastnosť sa týka distribuovanosti cloudového prostredia, ktoré pozostáva z viacerých zdrojov, a hovorí o tom, že aplikácie navrhnuté pre cloud by mali byť taktiež rozložené na viac častí, ktoré môžu byť distribuované medzi rôznymi zdrojmi.
- **Pružnosť:** Aplikácie by mali byť schopné realizovať horizontálne škálovanie, čo znamená, že v prípade potreby ďalších zdrojov, napr. viac výpočetného výkonu alebo zväčšenie úložiska, sa zvýši počet nezávislých IT zdrojov, ako napr. serverov. Aplikácia by mala byť navrhovaná tak, aby bola schopná fungovať na viacerých nezávislých zdrojoch. Okrem požiadavky horizontálnej škálovateľnosti, by mali byť aplikácie schopné aj dynamicky pridávať alebo odstraňovať zdroje na základe potreby. Táto schopnosť je nazývaná ako už spomenutá pružnosť aplikácie a je nutná pre využívanie pay-per-use modelu.
- **Izolovaný stav:** Táto charakteristika zdôrazňuje, že aplikácie by mali izolovať stav do malých častí, čo znamená, že by mali byť navrhované ako bezstavové (ang. *stateless*). Stav aktuálneho sedenia a rovnako aj stav aplikácie môžu mať veľký dopad na schopnosť škálovateľnosti, pretože bezstavové zdroje zjednodušujú ich pridávanie a odstraňovanie, pretože nie je potrebné synchronizovať ich stav medzi rôznymi zdrojmi.

- **Automatická správa:** Automatická správa znamená, že systém by mal automaticky spravovať pridávanie a odstraňovanie zdrojov na základe monitorovania aktuálnej záťaže.
- **Minimálna previazanosť:** Závislosti medzi jednotlivými komponentmi systému by mali byť minimálne, z dôvodu meniaceho sa počtu IT zdrojov. Jednotlivé časti aplikácie by nemali byť ovplyvnené zlyhaním iných častí aplikácie. Týmto sa zjednodušuje úloha pridávania a odstraňovania úloh a znižuje to dopad pri zlyhaniach niektorých častí aplikácie.

2.5 Architektúry pre aplikácie v cloud computing

Na základe IDEAL vlastností, popísaných v predchádzajúcej podkapitole 2.1, sú vhodnými architektúrami pre aplikácie v cloudovom prostredí architektúra mikroslužieb, architektúra riadená udalosťami (ang. *event-driven architecture*) a serverless computing, ktorým sa táto časť práce venuje bližšie. Okrem týchto architektúr patria medzi často využívané štýly architektúr v cloud computing aj N-tier a Web-Queue Worker. Medzi ďalšie relevantné princípy patria kontajnerizácia, vyvažovania záťaže, API gateway a vyrovnávacie pamäte. Tieto princípy poskytujú ďalšiu vrstvu abstrakcie, zabezpečenia a škálovateľnosti. Kontajnerizácia uľahčuje prenášanie a spúšťanie mikroslužieb, zatiaľ čo vyvažovanie záťaže a API gateway sú kľúčové pre riadenie a optimalizáciu komunikácie. Vyrovnávacie pamäte prispievajú k zníženiu latencie.

Event-driven architektúra



Obr. 2.2: Hlavné komponenty architektúry založenej na udalostiach [30].

Architektúra riadená udalosťami je založená na asynchrónnom spracovaní udalostí cez oddelené procesory udalostí, často nazývané aj služby, ktoré generujú a reagujú na asynchrónne udalosti. Proces spracovania začína primárnou udalosťou (ang. *initiating event*), ktorá väčšinou vzniká mimo systém. Táto primárna udalosť je následne vložená do kanálu udalostí (ang. *event channel*), často implementované pomocou frontu správ, odkiaľ si ju prevezme

príslušná služba. Keď sa stav služby zmení, služba vygeneruje procesnú udalosť (ang. *processing event* alebo *derived event*), aby informovala ostatné služby o zmene stavu. Táto architektúra je ideálna pre systémy vyžadujúce vysokú odolnosť voči chybám, škálovateľnosť a vysoký výkon, ktoré táto architektúra poskytuje práve kvôli jej asynchrónnej a oddelenej povahe. Architektúra riadená udalosťami je efektívna pre aplikácie, ktoré reagujú na udalosti a pri komplexných a nedeterministických pracovných tokoch, ktoré sú ťažko modelovateľné. Naopak, architektúra riadená udalosťami nie je vhodná, ak spracovanie vyžaduje synchrónne spracovanie, alebo ak väčšina operácií závisí od požiadaviek, ako napríklad v prípade systémov zamerané prevažne na CRUD operácie alebo na prácu s entitami systému. [30]

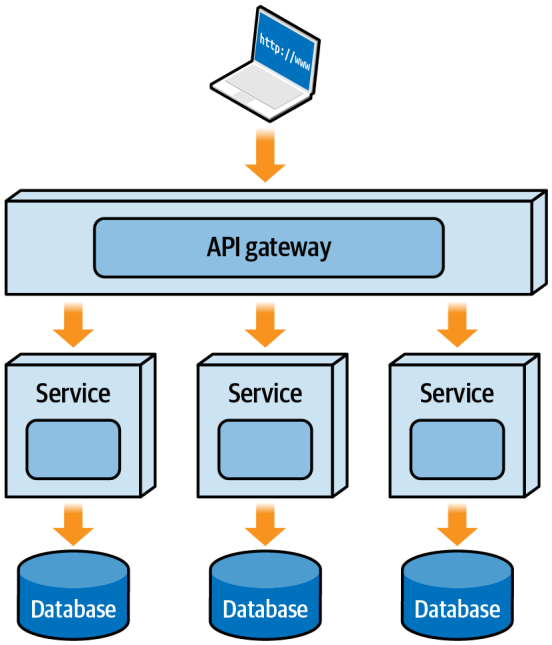
Serverless computing a FaaS

Serverless computing je spôsob navrhovania a implementácie aplikácií, ktorý umožňuje vývojárom písať kód a nasadzovať ho do cloudu bez nutnosti spravovať fyzický hardvér, virtuálne servery alebo kontajnery. Serverless computing často úzko súvisí s event-driven architektúrou, spomenutou vyššie, pretože serverless computing funguje na princípe, kedy funkcie alebo mikroslužby nemusia byť neustále aktívne a spúšťajú sa len v prípade, ak nastane konkrétna udalosť. Serverless computing sa zameriava na minimalizáciu správy a konfigurácie serverov a infraštruktúry a týka sa spôsobu, ako sú aplikácie implementované a nasadené v cloude, zatiaľ čo event-driven architektúra sa týka spôsobu, akým sú udalosti používané na riadenie toku dát a správy v aplikáciách. Serverless aplikácie môžu využívať event-driven architektúru, ale nie všetky event-driven aplikácie sú serverless. Pri serverless prístupe zákazník platí iba za čas, počas ktorého sú využívané výpočetné zdroje potrebné na vykonanie danej úlohy, funkcie alebo služby. Platba je za žiadosť o vykonanie funkcie alebo služby, preto to je vhodný spôsob pre prípad, ak sú žiadosti menej frekventované. Serverless computing vyžaduje, aby služby boli bezstavové, pretože môžu byť spustené na viacerých fyzických serveroch. V dôsledku toho musia byť všetky stavy ukladané v externej službe. Serverless computing je úzko spojený aj s konceptom FaaS — function-as-a-service, kde sa jedná iba o funkciu bez ďalších artefaktov, ale serverless computing nie je limitovaný iba na funkcie. V prípade FaaS sa samotný kód funkcie ľahko spravuje a automaticky škáluje. V prípade zlyhania funkcie, je funkcia automaticky obnovená. Funkcie sú bezstavové a úplne nezávislé, preto systém, ktorý je postavený na funkciách, je viac modulárnym a oddeleným. [12] [13] [18] [30]

Architektúra mikroslužieb

Architektúra mikroslužieb (ang. *microservice architecture*) spočíva v rozdelení aplikácie na skupinu menších služieb, kde je každá služba nezávislá, vykonáva konkrétnu úlohu a komunikuje s ostatnými službami cez aplikačné rozhranie (skr. API). Klient pristupuje k aplikácii cez službu API gateway, ktorá presmeruje jeho žiadosti na konkrétne služby. Úlohou služby API gateway je nielen skrývať umiestnenie a implementáciu služby, ale aj poskytovať bezpečnostné mechanizmy a niekedy zabezpečujú aj vyvažovanie záťaže. Každá služba môže, ale nemusí mať vlastnú databázu, ale každá služba pracuje so svojou množinou dát, ku ktorej pristupuje iba táto služba. Ak iné služby potrebujú prístup k týmto dátam, musia si ich vyžiadať cez službu, ktorá tieto dáta spravuje. Táto architektúra je často spájaná s kontajnerizáciou a orchestráciou služieb. Výhody tejto architektúry spočívajú v efektívnom využívaní výpočtových zdrojov, poskytovaní prirodzenej distribúcie a jednoduchšej implementácii zmien, pretože služby nie sú navzájom závislé a sú menej komplexné.

Táto architektúra je vhodná pre aplikácie, ktoré vyžadujú rýchlu reakciu na zmeny, vysokú škálovateľnosť a vysokú odolnosť voči zlyhaniam. Taktiež je dobrým výberom, ak sa plánuje budúca architektonická expanzia. [18] [30]



Obr. 2.3: Architektúra mikro služieb (prevzaté z [30]).

2.6 Prehľad poskytovateľov cloudových služieb

V tejto časti sú predstavení traja najväčší poskytovatelia cloudových služieb, ktorými sú aktuálne Amazon Web Services (skr. AWS), Microsoft Azure a Google Cloud Platform (skr. GCP). Medzi ďalších poskytovateľov cloudových služieb patria Oracle Cloud alebo IBM Cloud. Pri výbere poskytovateľa cloudových služieb je dôležité zvážiť prípady použitia systému, umiestnenie dátových centier, aby sa minimalizovala latencia a zároveň zohľadnili bezpečnostné obmedzenia v rôznych krajinách, v ktorých môžu byť dáta umiestnené. Je dôležité preskúmať, aké úložiská cloudový poskytovateľ poskytuje a aké služby podporuje. V poslednom rade, je pri výbere cloudového poskytovateľa a následnom návrhu aplikácie, dôležité dbať na tzv. vendor-lock-in, teda aby aplikácia nebola priveľmi viazaná na konkrétneho poskytovateľa a bola zachovaná určitá flexibilita aplikácie v prípade zmien ako je napríklad zvýšenie cien poskytovateľa služby.

Amazon Web Services

AWS poskytuje široký ekosystém služieb a umožňuje široké možnosti integrácie so službami tretích strán. K roku 2023 má najväčšie zastúpenie na trhu, s čím je spojená aj najväčšia a najaktívnejšia komunita užívateľov. AWS má postavenú infraštruktúru na bezpečnosti, a na dostupnosti aj pre kritické sektory. Výpočtové jednotky ponúka pod názvom EC2 s rôznymi druhmi procesorov a 400 Gbps ethernetovým pripojením. Dátové centrá sú geograficky rozdelené do regiónov, ktoré sú ďalej segmentované na Availability Zones (prel. zóny dostupnosti). Tieto zóny dostupnosti sú spojené redundantnými sieťami s ultra nízkou la-

tenciou, pričom každý región garantuje vysokú bezpečnosť. Pre ukladanie dát poskytuje možnosti využitia blokových úložísk EBS (viac v 4.2), ktoré ponúka vo viacerých verziách s rôznym prietokom dát a počtom IO operácií. Pre ukladanie dát sú dostupné aj ďalšie možnosti a ceny úložných kapacít sú stanovené na základe ich veľkosti a objemu prichádzajúcich a odchádzajúcich dát (s prevahou fakturácie len za egress). AWS poskytuje širokú paletu služieb i nástrojov a poskytuje robustnú infraštruktúru, preto je vhodným výberom pre širokú škálu aplikácií a prípadov použitia. [5]

Microsoft Azure

Microsoft Azure disponuje širokým spektrom služieb, pričom sa špecializuje na výpočet založený na produktoch Microsoft, čo ho robí dobre integrovateľným práve s týmito produktmi. Azure kladie vysoký dôraz na bezpečnosť dát, čím sa taktiež stáva vhodným kandidátom pre kritické odvetvia. Podobne ako AWS aj Azure štrukturuje svoje dátové centrá do regiónov, ktoré sú navzájom prepojené sieťou s nízkou latenciou, a v regiónoch tak vznikajú zóny dostupnosti, pričom Azure sa vyznačuje najširším pokrytím regiónov. Azure sa hodí pre firmy, ktoré uprednostňujú hybridné cloudové prostredie, keďže poskytuje široké možnosti využitia tejto kombinácie. Azure ponúka výpočtové zdroje pod názvom Virtual Machines (prel. *virtuálne stoje*), ktoré sú dostupné v rôznych verziách, a podporuje na virtuálnych strojoch rôzne operačné systémy. Pre ukladanie dát ponúka platforma Azure opäť viac možností a ceny sa líšia na základe vybraných jednotiek a počtu IO operácií. [8]

Google Cloud Platform

GCP poskytuje rozsiahle portfólio produktov so zameraním na umelú inteligenciu a spracovanie veľkých dát, vďaka čomu vyniká v oblastiach analytických riešení a strojového učenia. Výpočtové zdroje sú prístupné prostredníctvom služby Google Compute Engine, ktorá poskytuje flexibilné inštancie virtuálnych strojov. GCP ponúka blokové úložisko Google Persistent Disk a prenos dát je účtovaný na základe objemu prijatých a odoslaných dát. GCP vyniká svojou výkonnou globálnou sieťou, ktorá je využívaná aj inými produktmi Google, čím zabezpečuje rýchly prenos dát a nízku latenciu. Dátové centrá GCP sú distribuované po celom svete, pričom každé centrum je vybavené pokročilými bezpečnostnými funkciami. [7]

2.7 Zhrnutie

V tejto časti bol predstavený koncept cloud-computing a jeho charakteristické vlastnosti spolu s vlastnosťami, ktoré by mali spĺňať aplikácie nasadené do tohto prostredia. Ďalej boli predstavené tri najpoužívanejšie architektúry pre aplikácie v tomto prostredí. Architektúra založená na udalostiach, ktorá súvisí so serverless computing, a architektúra mikroslužieb. Všetky tri architektúry môžu byť používané aj vo vzájomnej kombinácii. V závere tejto kapitoly boli predstavení traja poskytovatelia cloudových služieb, pričom všetci traja ponúkajú vysokú bezpečnosť dát, a preto sú vhodné aj pre kritické odvetvia. Pre základné výpočty poskytujú všetci podobné nástroje, takže pri výbere môže byť rozhodujúcim faktorom napr. cena daného riešenia. AWS má pred ostatnými poskytovateľmi časový náskok, pretože prišiel na trh ako prvý, a preto ponúka širšiu ponuku výpočetných zdrojov a druhov úložísk, ale vyniká aj v rámci integrácie so službami tretích strán. AWS má taktiež najväčšiu komunitu používateľov. Azure je ideálny pre organizácie, ktoré preferujú integráciu

s produktmi Microsoft alebo by chceli využiť kombináciu cloudového a on-premise prostredia. GCP je vhodným kandidátom pri používaní strojového učenia alebo analytických riešení, ale vyniká aj pri využívaní open-source nástrojov. Dôležité je pri výbere zohľadniť približnú cenu zvoleného riešenia a geografickú polohu dátových centier.

Kapitola 3

Spracovanie real-time dát v cloud computing

„Systém, pracujúci v reálnom čase, je počítačový systém, kde správnosť chovania systému závisí nielen na logických výsledkoch výpočtov, ale tiež na fyzickom čase, kedy sú tieto výsledky produkované. Chovaním systému sa myslí postupnosť výstupov systému v priebehu času. [23]"

Systémy pracujúce v reálnom čase (ang. *real-time*) pracujú s neobmedzeným (ang. *unbounded*, inak nekonečným) tokom dát, ktorý postupne priteká do systému. Tieto systémy často súvisia s prúdmi dát, ktorými sa zaoberá podkapitola 3.2. Cieľom týchto systémov je spracovať a získať potrebné informácie, v čo najkratšom čase od prijatia dát. Vo svojej podstate sa nejedná vždy o systémy pracujúce úplne v reálnom čase, kvôli latencii spôsobenej prenosom dát a ich spracovaniu. V takomto prípade sú označované ako soft real-time systémy. Hard real-time systémy, naopak, musia produkovať výsledky v presne určenom okamihu, väčšinou v priebehu milisekúnd, a často sa jedná o vstavané systémy. Táto práca je zameraná na soft real-time systémy.

Systémy pracujúce v reálnom čase, sa odlišujú od dávkového spracovania (ang. *batch processing*) v dobe spracovania. Dávkové spracovanie spočíva v spracovaní dát po menších dávkach. Real-time systém sa snaží spracovať dáta čo najrýchlejšie po obdržaní udalosti alebo dát, na rozdiel od dávkového spracovania, kde sa dáta spracovávajú v časových intervaloch. Pri spracovaní po dávkach môže dochádzať k oneskoreniu voči fyzickému času medzi obdržaním dát a ich spracovaním, čo spôsobuje, že užívatelia nemôžu okamžite reagovať na udalosti. Aj keď v prípade menších časových intervalov (napr. sekunda, minúta) sa aj tieto systémy približujú k spracovaniu v reálnom čase. Výhody spracovania dát v reálnom čase zahŕňajú nízku latenciu, generovanie aktuálnych informácií a rýchlu možnosť reakcie na zmeny, a spracovanie dát v reálnom čase môže prispieť k lepšiemu využívaniu zdrojov. [23] [27] [31]

Nízka latencia predstavuje výhodu real-time systémov, ale zároveň jednu z hlavných výziev pri návrhu. Medzi ďalšie možné výzvy pri návrhu real-time systému môžu patriť problémy s dostupnosťou, synchronizáciou dát, spracovaním veľkého objemu dát a s tým spojená potreba škálovateľnosti systému. Práve cloud computing môže byť vhodnou voľbou pre realizáciu soft real-time systémov, pretože poskytuje nízku latenciu, škálovateľnosť v závislosti od aktuálneho objemu prúdu dát a adresuje problémy s dostupnosťou prostredníctvom distribúcie a redundancie zdrojov.

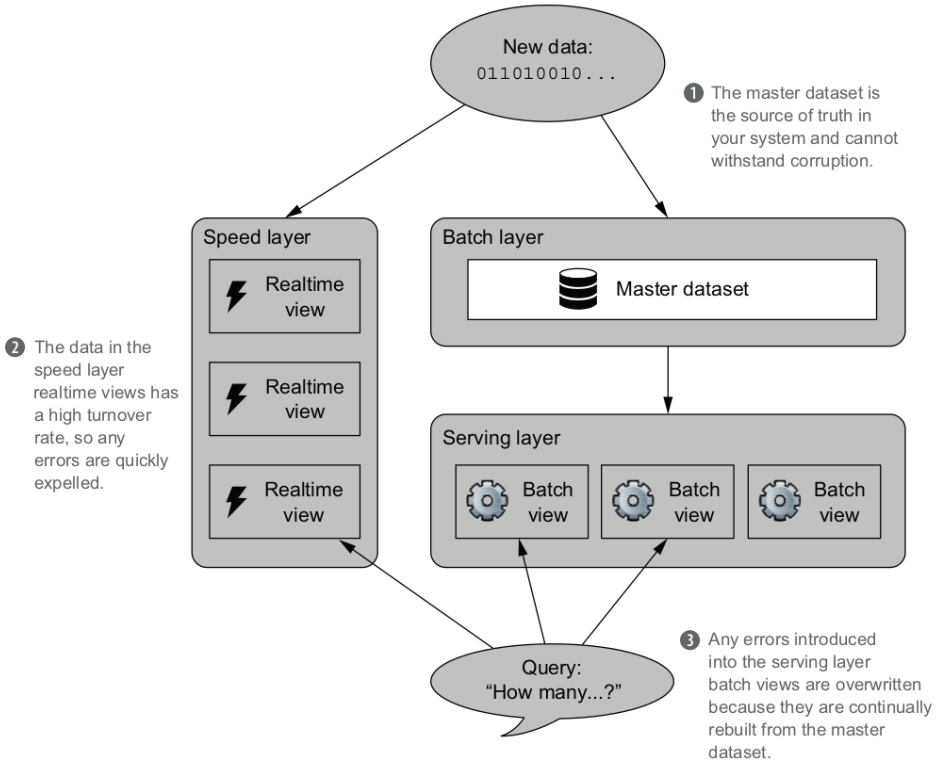
3.1 Architektúry spracovania dát v reálnom čase

Okrem architektúr, uvedených v 2.5, sú pre návrh real-time systému vhodné aj lambda architektúra, kappa architektúra a CEP architektúra. Tieto architektúry spolu s architektúrou mikro služieb, serverless computing a event-driven architektúrou môžu byť kombinované podľa konkrétnych požiadaviek na danú aplikáciu.

Lambda architektúra

Hlavná myšlienka lambda architektúry je rozdeliť systém do 3 vrstiev:

- **Batch layer** - vrstva pre dávkové spracovanie
- **Speed layer** - vrstva pre spracovanie prúdu dát
- **Serving layer** - optimalizovaná databáza pre žiadosti



Obr. 3.1: Lambda architektúra (prevzaté z [26]).

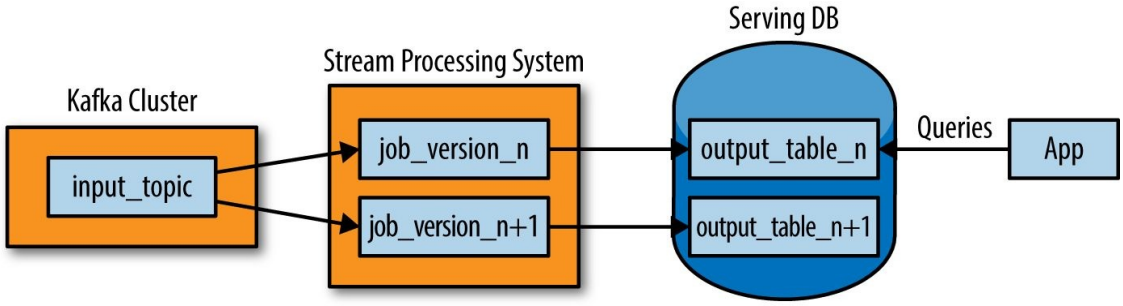
Ako je vidieť na obr. 3.1, prichádzajúce dáta sú spracovávané v speed vrstve, ktorá zahŕňa stream procesor[1] a zároveň sú dáta zapisované do hlavného datasetu (nazývané aj batch storage [19]). Tento dataset je následne pravidelne spracovávaný dávkovým procesorom (ang. *batch processor*), ktorý vytvára predpočítané dávkové pohľady (ang. *batch views*). Speed vrstva počíta približné výsledky skôr než vrstva pre dávkové spracovanie. Čo sa týka ďalšieho postupu, literárne zdroje poskytujú odlišné informácie. Niektoré zdroje uvádzajú,

[1] prekl. procesor spracovávajúci prúd dát (ďalej iba ako stream procesor, nakoľko to je zaužívaný pojem)

že dáta zo speed vrstvy aj batch vrstvy sú zapisované do serving vrstvy [27], zatiaľ čo iné uvádzajú, že speed layer má vlastnú databázu, do ktorej sa zapisujú dáta (viď 3.1) [26]. Bez ohľadu na konkrétny postup je konečná odpoveď na žiadosť kombináciou výsledkov speed vrstvy a batch vrstvy. Po vypočítaní presného výsledku batch procesorom sú nepresné výpočty z real-time pohľadu odstránené. Hlavnou výhodou tejto architektúry je uchovávanie vstupných dát v nezmenenej podobe, čo umožňuje ich opakované spracovanie iným spôsobom. Na druhú stranu táto architektúra vyžaduje dve sémanticky rovnaké implementácie aplikačnej logiky pre dva rôzne procesory a výsledky produkované speed vrstvou sú len približné, čo predstavuje dve hlavné nevýhody. Treťou nevýhodou je, že táto architektúra je náročná na implementáciu a údržbu. [19] [26] [27]

Kappa architektúra

Kappa architektúra bola vytvorená ako alternatíva k lambda architektúre. Na rozdiel od lambda architektúry kappa architektúra využíva iba stream procesor a front správ s možnosťou uloženia celého log dát určitý čas pre opätovné spracovanie. V prípade, že je potrebné dáta opätovne spracovať, spustí sa nová inštancia úlohy spracovania, ktorá začne spracovávať dáta od začiatku logu. Výsledky sa uložia do novej tabuľky. Keď sa nová inštancia dostane na aktuálnu úroveň spracovania dát, aplikácia začne čítať dáta z novej tabuľky a stará verzia úlohy sa zastaví, a stará tabuľka sa môže odstrániť. Opätovné spracovanie sa vykonáva iba v prípade, ak sa zmení logika spracovania a je potrebné prepočítať výsledky. Nevýhodou kappa architektúry je, že dočasne potrebuje dvojnásobné množstvo úložiska a vyžaduje, aby databáza podporovala vysoko objemné zápisy. Výhoda kappa architektúry spočíva v tom, že využíva iba jeden systém pre spracovanie, čo vývojárom umožňuje ľahšiu údržbu aj implementáciu. [24]



Obr. 3.2: Kappa architektúra, kde autor navrhuje ako technológiu pre front správ využiť Apache Kafka (prevzaté z [24]).

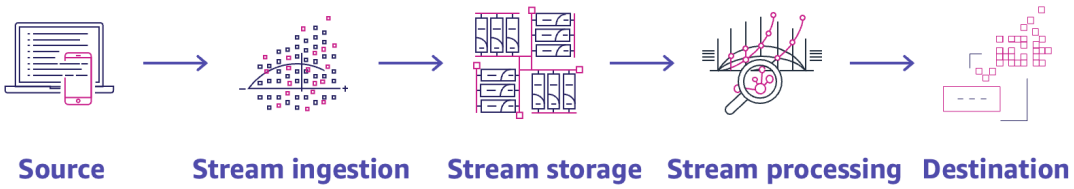
Komplexné spracovanie udalostí - CEP

Komplexné spracovanie udalostí (ang. *complex event processing*) je technika pre analyzovanie, spracovanie a monitorovanie dát v reálnom čase. V tejto architektúre sa ako prvé definujú vzorce v tzv. CEP engine a udalosti, prichádzajúce do tohto CEP engine, sú porovnávané s definovanými vzorcami. CEP kombinuje dáta z rôznych zdrojov a dokáže v nich identifikovať vzorce a komplexné vzťahy, čo umožňuje napríklad identifikáciu príležitostí a hrozieb. Najčastejšie je táto technika využívaná vo finančných, leteckých, zdravotníckych a telekomunikačných priemysloch, ale zároveň má potenciál aj v analýze veľkých dát. [21] [27]

Moderná architektúra prúdov dát

S rozvojom cloudových služieb sa okrem už spomenutých klasických architektúr pre spracovanie prúdov dát objavili nové možnosti architektúr. Všeobecný postup pre návrh takejto architektúry je prezentovaný v článku [32] od AWS, na ktorom je táto podsekcia založená.

Moderná architektúra dovoľuje príjem, spracovanie a analýzu vysokého objemu rýchlo pribúdajúcich dát v reálnom čase. Táto architektúra väčšinou pozostáva z piatich vrstiev, ako je zobrazené na obrázku 3.3.



Obr. 3.3: Rozdelenie architektúry do 5 vrstiev, počínajúc zdrojom a končiac v cieľovej destinácii (prevzaté z [32]).

- **Zdroj (Source)** - Zdroje dát môžu byť rôzne, napr. senzory, sociálne média, databázy.
- **Prijímanie dát (Stream Ingestion)** - Vrstva prijímania dát je zodpovedná za ukladanie dát z rôznych zdrojov do nasledujúcej vrstvy (úložisko prúdu dát). Môže byť vnímaná ako producent dát.
- **Úložisko prúdu dát (Stream Storage)** - Úlohou tejto vrstvy je efektívne a cenovo efektívne ukladať prúdy dát v poradí, v akom sa vyskytli tak, aby bolo možné ich prípadné opätovné prehratie. Táto vrstva je často reprezentovaná frontami správ.
- **Spracovanie prúdu (Stream Processing)** - Spracovanie prúdu dát zahŕňa rôznorodé operácie, vrátane prípravy dát pre ďalšie operácie, ale aj samotné transformácie dát a analytické operácie. V tejto vrstve sú využívané technológie, spracovávajúce prúdy dát (3.2), ako napr. Apache Flink alebo pri spracovaní dát založenom na udalostiach sú využívané aj služby FaaS.
- **Cieľová destinácia (Destination)** - Vrstva cieľovej destinácie závisí od samotnej aplikácie a môže zahŕňať rôznorodé úložiská dát alebo nasledujúcu aplikáciu.

3.2 Prúdové spracovanie dát

Prúdové spracovanie dát (ang. *stream processing*) predstavuje kontinuálne spracovanie neohraničeného toku informácií a zohráva kľúčovú úlohu v real-time spracovaní. Pri tomto type spracovania sú dáta vnímané ako udalosti a prenos je uskutočňovaný pomocou modelu publish/subscribe, v ktorom producenti generujú udalosti a konzumenti na ne reagujú. Bežný prístup pre model publish/subscribe je využívanie systému správ, s možnosťou využitia prostredníka alebo priamej komunikácie. V prípade priamej komunikácie môže dochádzať k strateniu správy. Tento problém sa dá vyriešiť práve s využitím prostredníka — mediátora správ (ang. *message broker*) alebo frontu správ (viď 4.1). V prípade nutnosti

uchovávania všetkých udalostí, sa dá na prenos správ využiť aj log. Prúd udalostí môže mať rôzne výstupy, ako je napríklad uloženie do databázy alebo do vyrovnávacej pamäte, udalosti môžu byť doručené klientovi, vizualizované v reálnom čase alebo môžu byť spájané s inými prúdmi udalostí až kým nedosiahnu jeden z už spomenutých výstupov. V porovnaní so spracovaním dát vo forme menších dávok, prúdové spracovanie prináša aktuálnejšie informácie, nižšiu latenciu a rovnomerne rozdeľuje pracovné zaťaženie, pretože udalosti sú spracovávané okamžite po ich doručení stream procesoru. Avšak, stream procesor nemá prístup k celému datasetu, a v prípade vykonávania komplexnejších operácií nad udalosťami, ako napr. agregácie, je nutné, aby si udržiaval stav a musí byť schopný ukladať a pristupovať k medzivýpočtom. [11] [22] [24]

Porovnanie nástrojov pre prúdové spracovanie dát

Táto časť je zameraná na stručné predstavenie štyroch nástrojov pre spracovanie prúdov dát a sústredí sa na princípy spracovania, doby odozvy, uchovávanie stavu a odolnosť voči chybám.

Apache Flink

Nasledujúci text je založený na oficiálnej dokumentácii [1]. Apache Flink je open-source nástroj navrhnutý pre efektívne spracovanie veľkých prúdov dát, pričom je založený na princípe kappa architektúry popísanej vyššie (obr. 3.2). Okrem prúdového spracovania poskytuje aj možnosti spracovania v pamäti a dávkového spracovania. Flink ponúka vysoký výkon, nízku latenciu, flexibilitu, kvôli podpore dávkového aj prúdového spracovania a má rozsiahlu komunitu užívateľov.

- **Spracovanie:** Aplikácie sú zložené z prúdov dátových tokov, ktoré sú transformované pomocou operácií definovaných užívateľom — operátory. Tieto dátové toky tvoria orientované grafy. Počas vykonávania má každý prúd dát jednu alebo viac prúdových vetiev (ang. *stream partitions*) a každý operátor má jednu, alebo viac podúloh operátora. Podúlohy sú navzájom nezávislé a sú vykonávané rôznymi vláknami, v rôznych kontajneroch alebo na rôznych strojoch. Počet podúloh operátora reprezentuje paralelizmus daného operátora, pričom rôzne operátory v jednom programe môžu mať odlišné úrovne paralelizmu.

- **Latencia:** Flink zaručuje nízku latenciu využívaním kontrolných bodov (ang. *checkpoints*)[2] a lokálnym ukladaním stavov. Podľa [14] má Flink spolu s Kafka Streams veľmi nízku latenciu, pričom mediánová hodnota je 0 ms pre čítanie 450 správ za sekundu bez transformácií. Pri parsovaní správ sa táto mediánová hodnota zvyšuje o 1 ms.

- **Ukladanie stavu:** Množina paralelných inštancií stavového operátora funguje na princípe fragmentovaného úložiska kľúč-hodnota, pričom každá paralelná inštancia zodpovedá za spracovanie udalostí pre konkrétnu skupinu kľúčov a udržiava stav pre tieto kľúče lokálne. K stavu sa vždy pristupuje lokálne, čo umožňuje aplikáciám Flink dosahovať nízku latenciu a zároveň vysokú priepustnosť dát. Ukladanie indexov kľúč-hodnota závisí od vybraného stavového backendu.

- **Odolnosť voči chybám:** Flink zabezpečuje odolnosť voči chybám pomocou pravidelných asynchrónnych snapshotov stavov a možnosti prehratia prúdu (ang. *stream*

[2] ďalej používaný anglický výraz

replay). Flink taktiež pravidelne ukladá stav všetkých operátorov a aktuálnu pozíciu spracovania v prúdoch dát. V prípade zlyhania programu je tok dát pozastavený, operátory sú reštartované, vrátane ich nastavení, na posledný úspešný checkpoint a vstupné prúdy dát sú obnovené na stav posledného snapshotu.

Apache Kafka Streams

Nasledujúci text je založený na oficiálnej dokumentácii [2]. Apache Kafka Streams je open-source knižnica pre spracovávanie prúdov dát a je úzko prepojená s Apache Kafka, ktorú využíva pre dátový paralelizmus, koordináciu a odolnosť voči chybám. Táto tesná integrácia s platformou Apache Kafka umožňuje ľahké spracovanie a analýzu udalostí, ktoré prúdia cez Apache Kafka. Okrem Apache Kafka nemá žiadne iné závislosti, čo umožňuje jednoduché nasadenie.

- **Spracovanie:** Procesná topológia Kafka Streams je organizovaná do viacerých úloh, čím je umožnené škálovanie. Počet úloh je pevne stanovený na základe rozdelenia vstupného prúdu dát, pričom každá úloha má priradený zoznam oddielov (ang. *partitions*[3]) z príslušných Kafka tém (ang. *topic*[3]). Priradenie partitions úlohám ostáva nemenné, čo zabezpečuje, že každá úloha predstavuje pevnú jednotku paralelizmu. Úlohy si udržiavajú vyrovnávacie pamäte pre každú priradenú partition a postupne spracovávajú správy z týchto pamätí. Tento prístup umožňuje nezávislé a paralelné spracovanie prúdu úloh bez manuálneho zásahu, ale zároveň je práve kvôli tomuto princípu paralelizmus aplikácie obmedzený maximálnym počtom partitions pre vstupné topiky. Partitions sú priradené k úlohám a úlohy sú priradené k všetkým vláknam na všetkých inštanciách. Medzi vláknami neexistuje nijaký zdieľaný stav, preto medzi vláknami nie je potrebná koordinácia.

- **Latencia:** Apache Kafka Streams využíva princíp spracovania one-record-at-a-time[4], čím dosahuje milisekundovú latenciu, pretože každá prichádzajúca udalosť je okamžite spracovaná, čo vedie k rýchlej odozve a minimalizuje oneskorenie. K zníženiu latencii prispieva aj možnosť paralelného spracovania, ktoré bolo popísané vyššie. Okrem toho, článok [17] odporúča v prípade potreby veľmi nízkej latencie uprednostniť nastavenie krátkeho intervalu potvrdenia (ang. *commit interval*) na úkor využívania väčšieho počtu výpočetných zdrojov.

- **Ukladanie stavu:** Každá úloha v Kafka Streams obsahuje stavové úložiská (ang. *state stores*), ktoré umožňujú ukladanie a dotazovanie sa na dáta. Tieto úložiská môžu byť buď trvalé úložiská typu kľúč-hodnota, in-memory hashmap, alebo iná dátová štruktúra.

- **Odolnosť voči chybám:** Kafka Streams zakladá odolnosť voči chybám na schopnostiach frontu správ Apache Kafka, ktorý zabezpečuje vysokú dostupnosť jednotlivých partitions a ich replikáciu. V prípade zlyhania inštancie aplikácie sú všetky jej úlohy automaticky reštartované na iných inštanciách a pokračujú konzumáciou správ z rovnakých partitions.

[3] Ďalej používaný zavedený anglický výraz.
[4] prel. záznam po zázname

Apache Spark

Nasledujúci text je založený na oficiálnej dokumentácii nástroja Apache Spark [3] [9]. Apache Spark je univerzálny open-source nástroj pre spracovanie veľkých dát. Primárne bol navrhnutý pre dávkové spracovanie, ale ponúka možnosť spracovania dát v reálnom čase cez rozšírenie Spark Streaming alebo cez novšiu verziu Structured Streaming, ktorej sa táto časť bude venovať. Prúdové spracovanie v Apache Spark je založené na princípe mikrodávkového spracovania. Structured Streaming je založený na Spark SQL engine. Výhodou Structured Streaming je integrácia so samotným nástrojom Spark, z čoho vyplýva, že jeden nástroj dokáže spracovávať aj vstupný prúd dát, aj historické dáta. Structured Streaming umožňuje písanie aplikácií s využitím jazyka SQL.

- **Spracovanie:** Hlavná myšlienka Structured Streaming je založená na vnímaní vstupného prúdu dát ako tabuľky, ktorá je nazývaná DataFrame. Táto tabuľka neustále narastá, pričom spracovanie prúdu dát je formulované rovnakým spôsobom ako v prípade dávkového spracovania. Spark SQL engine sa stará o inkrementálne a kontinuálne spúšťanie tohto prúdu dát a aktualizáciu výsledkov. Žiadosť na vstupe generuje tabuľku s výsledkom a každý nastavený interval sa k vstupnej tabuľke pripojí nový riadok, čo nakoniec aktualizuje tabuľku s výsledkom. Táto aktualizácia spôsobuje záznam výsledku do externého sinku.

- **Latencia:** Interné spracovanie žiadostí je realizované prostredníctvom mikrodávkového stroja (ang. *micro-batch engine*), ktorý spracováva prúdy dát vo forme malých úloh a dosahuje latenciu na úrovni 100 milisekúnd, čo potvrdzuje aj [14]. V uvedenej štúdii má Structured Streaming medián latencie pri konzumovaní 450 správ za sekundu bez transformácií s hodnotou približne 105 ms, a pri parsovaní správ okolo 148 ms. V porovnaní s Apache Flink a Apache Kafka Streams je latencia výrazne vyššia, čo je spôsobené práve dávkovým spracovaním. Od verzie Spark 2.3 bol zavedený model Continous Processing, ktorý môže dosahovať latenciu 1 ms pri garancii spracovania aspoň raz.

- **Ukladanie stavu:** Ukladanie stavu v rámci stavových operácií je založené na základnej myšlienke Structured Streaming, ktorá využíva tabuľky. Pri skupinových agregáciách sú agregované hodnoty udržiavané pre každú jedinečnú hodnotu v špecifikovanom stĺpci určenom pre agregáciu.

- **Odolnosť voči chybám:** Structured Streaming implementuje viacero mechanizmov na zabezpečenie odolnosti voči chybám. Prvým z nich je sledovanie priebehu spracovania prostredníctvom monitorovania offsetov v prúde dát, pričom tieto offsety sú zaznamenávané cez checkpointy a write-ahead-logy. Druhým bezpečnostným opatrením sú idempotentné prúdové ciele pre umožnenie opätovného spracovania. Tieto ciele môžu spracovať údaje opakovane a výsledok zostane rovnaký, čo pomáha k zabezpečeniu konzistencie výsledkov.

Apache Storm

Nasledujúci text je založený na oficiálnej dokumentácii [4]. Apache Storm je open-source distribuovaný nástroj navrhnutý pre rýchle spracovanie prúdov udalostí v reálnom čase.

- **Spracovanie:** Apache Storm využíva model programovania nazývaný „topológia", ktorý vytvára graf komponentov zložený z tzv. **sprouts** a **bolts**. Sprouts sú zodpo-

vedné za načítanie dátových prúdov do topológie a rozdeľujú prúdy dát na tzv. „tuples", teda malé jednotky dát. Bolts sú komponenty starajúce sa o spracovanie dát. Apache Storm sa špecializuje prevažne na paralelné spracovanie úloh, kde sú úlohy paralelne rozdelené medzi viacero procesorov alebo vlákien. Výsledky spracovania sú odosielané sprouts alebo ďalším bolts, ktoré sú zodpovedné za ďalšie úlohy, ako napríklad uloženie dát.

- **Latencia:** Storm je schopný rýchleho spracovania veľkého objemu dát v reálnom čase. Podľa [31] dokáže efektívne spracovať až 1 milión 100B správ za sekundu na jednom uzle. Znižovanie latencie v Apache Storm je zabezpečené paralelným spracovaním úloh a topológiou spomenutou v bode vyššie. Práve kvôli tejto topológii sa úlohy môžu spracovávať súčasne na rôznych uzloch. Storm okrem toho podporuje rozdelenie záťaže medzi uzly, čo vedie k optimálnemu využitia dostupných prostriedkov.
- **Ukladanie stavu:** Bolts môžu ukladať stavy operácií, buď v pamäti, alebo v externej databáze Redis.
- **Odolnosť voči chybám:** Storm rieši odolnosť voči chybám pomocou automatizovaného reštartovania zlyhaných procesov. V prípade, že niektorý proces zlyhá, jeho nadriadený proces ho automaticky reštartuje. Tento mechanizmus pomáha minimalizovať vplyv chýb na celkovú spoľahlivosť systému. Manažment stavov je zabezpečený pomocou nástroja ZooKeeper, ktorý prispieva k udržaniu konzistentného a spoľahlivého sledovania stavov v distribuovanom prostredí.

3.3 Zhrnutie

Lambda architektúra bola kedysi obľúbeným riešením hlavne z dôvodu, že stream procesory neprodukovali vždy presné výsledky, čo sa zmenilo príchodom nových technológií, ako sú napr. Apache Flink alebo Apache Kafka, a do popredia sa dostala kappa architektúra, ktorá odstránila potrebu duplikátnej logiky. Napriek tomu, lambda architektúra môže stále predstavovať vhodné riešenie v prípadoch, ktoré vyžadujú kombináciu analýzy historických a aktuálnych dát. Komplexné spracovanie udalostí funguje na odlišnom princípe ako kappa a lambda architektúry a v dnešnej dobe je tento princíp často využívaný pre rôzne účely, kde je potrebná analýza vzorov. Pre účely tejto práce tento princíp nie je potrebný a bol zmienený iba pre demonštráciu aj iného prístupu. Moderná architektúra prúdov dát je typická pre riešenia v cloudovom prostredí a návrh v kapitole 5 je primárne založený na tejto architektúre, s využitím princípov mikroslužieb, event-driven spracovania a serveless computing, ktoré boli spomenuté v predchádzajúcej kapitole 2.5. Čo sa týka nástrojov pre spracovanie prúdov dát, boli predstavené celkovo 4 nástroje, pričom podľa článku [14] dosahujú Apache Flink a Apache Kafka Streams lepšie výsledky ako Spark Streaming a Spark Structured Streaming. V tomto článku nebol porovnávaný Apache Storm. Apache Storm bol jeden z prvých nástrojov pre spracovanie prúdov dát, ale tento nástroj stráca na popularite oproti ostatným, pretože priamo nepodporuje stavové operácie a jeho používanie je oproti iným nástrojom menej priamočiare. Apache Kafka Streams je priamo prepojený s Apache Kafka, čím sa znižuje celková latencia spracovania, a je to ľahko nasaditeľný nástroj bez potreby konfigurácie clustra. Apache Flink poskytuje široké spektrum operácií a konektorov pre rôzne zdroje a úložiská dát, avšak spotreba zdrojov môže byť vyššia kvôli spracovaniu na clusteri.

Kapitola 4

Ukladanie real-time dát v cloud computing

Predchádzajúca kapitola 3 sa venovala spracovaniu dát v reálnom čase, avšak rovnako dôležitým aspektom je aj ukladanie dát. Jedným z kľúčových prvkov ukladania dát v reálnom čase je využitie distribuovaných databázových systémov, ktoré umožňujú efektívne a rýchle spracovanie dát na viacerých serveroch súčasne. Ich výhodou je schopnosť ukladať a aktualizovať dáta takmer okamžite, čo je kľúčové pre aplikácie, vyžadujúce rýchle a spoľahlivé informácie. Cloud computing ponúka širokú škálu služieb na ukladanie dát, vrátane blokových úložísk, databázových služieb, súborových systémov a nástrojov na správu dát. Tieto služby často zahŕňajú automatizované zálohovanie a replikáciu dát, vrátane zrkadlenia dát medzi geograficky oddelenými dátovými centrami, čo zabezpečuje dostupnosť dát aj v prípade výpadku v jednom centre. Ukladanie dát v cloude môže prebiehať prostredníctvom blokových úložísk, ktoré je možné pripojiť k virtuálnym strojom pre spracovanie dát alebo súborových úložísk. Alternatívou je využitie už dostupných databázových systémov, ako sú napr. Amazon Dynamo, Azure Cosmos DB, Cloud FireStore ale aj mnoho ďalších. V predchádzajúcej kapitole, konkrétne v rámci modernej architektúry (obr. 3.3), sa nachádzajú dve vrstvy, zaoberajúce sa ukladaním dát. Jednou z nich je vrstva medzi zdrojmi dát 4.1 a spracovaním dát a druhou vrstvou je cieľové úložisko 4.2. Táto kapitola je zameraná na bližšie preskúmanie oboch týchto vrstiev, obsahuje informácie o prepojení spracovania dát s úložiskom v cloud computing 4.3 a informácie o celkovom nasadení systému do prostredia cloud computing.

4.1 Front správ

Nasledujúci text je založený na knihe [22]. Front správ (ang. *message queue*) predstavuje systém, umožňujúci asynchrónne spracovanie a výmenu správ medzi rôznymi službami. Tento systém pomáha k nízkej zviazanosti (ang. *low coupling*) systému, pretože rôzne služby komunikujú cez tento prostriedok. Front správ operuje ako server, ku ktorému sa producenti a konzumenti pripájajú ako klienti, čo pomáha k schopnosti systému tolerovať rôzne pripájanie a odpájanie klientov. Implementácia frontu správ sa líši v závislosti od použitej technológie a tieto technológie možno rozdeliť do dvoch kategórií: tradičné a založené na logoch. Obidva tieto princípy umožňujú konzumentom prihlásiť sa na spracovanie určitej podmnožiny topikov, čím je poskytnutá flexibilita vo výbere spracovávaného obsahu.

Tradičné fronty správ

Tradičné fronty správ predstavujú kvázi optimalizovanú databázu pre prúdy správ. Medzi tradičné fronty správ sú radené, napr. technológie RabbitMQ, Azure Service Bus a Google Cloud Pub/Sub. Niektoré z týchto technológií udržiavajú správy iba v pamäti, zatiaľ čo iné ich ukladajú aj na disk, aby sa v prípade chyby nestratili. Typicky sú správy odstránené z frontu, až keď boli úspešne doručené všetkým pripojeným konzumentom. Pri pripojení viacerých konzumentov na rovnaký topik existujú dva prístupy: vyvažovanie záťaže a fan-out. V prípade prístupu vyvažovania záťaže je každá správa doručená jednému konzumentovi, čo umožňuje paralelné spracovanie viacerých správ. Pri prístupe fan-out je, naopak, správa doručená všetkým konzumentom. Tieto prístupy sa môžu kombinovať. Pre riešenie odolnosti voči chybám sa používa potvrdenie klienta o úspešnom spracovaní správy pred jej odstránením z frontu. Nevýhodou tohto prístupu je, že správa je odstránená po prijatí potvrdenia od klienta, a v prípade, že konzument nie je schopný správu úspešne spracovať, nie je garantovaná záruka o opakovateľnosti procesu. Ďalšou nevýhodou je, že nový konzument, pridelený do systému, dostane iba nové správy od okamihu jeho registrácie, pričom nemá prístup k predchádzajúcim správam.

Fronty správ založené na logoch

Front správ založený na logoch funguje na princípe append-only, kde je správa od producenta pripojená na koniec logu a konzument číta správy sekvenčne. Pre potreby škálovania z dôvodu zvýšenia priepustnosti dát, môže byť log rozdelený na samostatné partitions, pričom topik je definovaný ako skupina týchto partitions. V každej partition sú správam pridelené sekvenčné čísla (offset), čím sú správy úplne usporiadané v rámci jednej partition. Tým sa zjednodušuje sledovanie spracovania správ, keďže systém frontu správ nemusí udžiavať informácie o potvrdeniach. Tieto fronty sú založené na jednoduchom princípe, kde každá správa s vyšším offsetom, ako má aktuálny konzument, je chápaná ako nespracovaná, čo umožňuje jednoduché zotavenie po výpadku uzla. V prípade výpadku prevezme iný uzol partitions od zlyhaného uzlu a začne ich spracovávať od posledného zaznamenaného offsetu. Offset, teda umožňuje správy opätovne spracovávať. Správy môžu byť čítané nezávisle viacerými konzumentami a celé partitions môžu byť priradené uzlom v skupine konzumentov. Každý klient potom konzumuje všetky správy patriace do jeho skupiny partitions. Ako bolo už spomenuté v kontexte Kafka Streams v časti 3.2, počet uzlov, zdieľajúcich prácu pri konzumácii z jedného topiku, je obmedzený počtom partitions logu pre daný topik, pretože správy patriace do rovnakej partition sú doručené na rovnaký uzol. To zabezpečuje, že ak je spracovanie správ pomalé, nezdržiava to spracovanie nasledujúcich správ v iných partitions. Medzi príklady frontov založených na logoch patria Apache Kafka, Amazon Kinesis Stream a Twitter Distributed Log.

Tradičný prístup je preferovaný v prípade, ak sú správy náročné na spracovanie a poradie, v ktorom sú správy spracovávané, nie je kľúčové. Naopak, prístup založený na logoch je preferovaný v situáciách, kedy je dôležité udržiavať poradie správ a musí byť rýchlo spracovávaný veľký prietok správ.

4.2 Perzistentné úložiská dát

V real-time systémoch je dôležité mať perzistentné úložisko, ktoré umožňuje rýchly prístup k dátam a minimalizuje oneskorenie pri ukladaní informácií. Rozdiel medzi systémom, pra-

cujúcim v reálnom čase, a inými systémami spočíva v tom, že real-time systémy veľakrát pracujú s temporálnymi dátami, ktoré s časom strácajú svoju relevanciu, a to aj v prípade historických dát, preto je vhodné historické dáta ukladať do databáz, ktoré podporujú automatické odstraňovanie dát po uplynutí určitého intervalu. Výber perzistentného úložiska dát v cloud computing závisí od typu a konkrétnych prípadov použitia aplikácie. Možnosťami sú využitia súborových úložísk, objektových úložísk alebo blokových úložísk či využitie databázy ako služby. Dôležité je dodať, že aplikácia nemusí byť fixovaná na jeden spôsob ukladania dát.

Cloudové úložiská dát

Poskytovatelia cloudových služieb ponúkajú rôzne možnosti ukladania dát, aby uspokojili rozmanité požiadavky. Tieto možnosti zahŕňajú nasledujúce kategórie:

- **Blokové úložiská:** Blokové úložiská v rámci cloud computing predstavujú fyzický hardvér, na ktorom sú uložené dáta. Pri tomto type úložiska užívateľ riadi svoju databázu bežiacu napr. na VM alebo v kontajneri a dáta sú ukladané na blokové úložisko, pričom tieto dáta môžu byť opäť využívané inými službami na virtuálnych strojoch alebo kontajneroch. Blokové úložiská umožňujú zákazníkovi kontrolu nad svojimi dátami. Počet blokových úložísk je možné meniť podľa potreby. Tieto úložiská je možné registrovať väčšinou priamo iba k jednej inštancii virtuálneho stroja, v ktorom sa následne zobrazuje ako lokálny hard disk. Blokové úložiská síce nemôžu byť často priamo registrované u viacerých inštancií, ale väčšinou to je umožnené skrz inú službu. Napríklad Microsoft Azure neumožňuje, aby bol Azure Disk zdieľaný viacerými inštanciami VM, avšak poskytuje pre tieto účely Azure Shared Disk. V prípade blokových úložísk AWS je registrácia jedného blokového úložiska pre viac EC2 umožnená iba v prípade určitých typoch blokového úložiska.

- **Objektové úložiská:** Objektové úložiská predstavujú riešenie pre ukladanie a správu veľkých objemov dát a sú špeciálne navrhnuté na uchovávanie objektových dát rôznych typov, vrátane videí a obrázkov. Dáta sú organizované do hierarchie podobne ako lokálny súborový systém a každému objektu je priradený jedinečný identifikátor. Príkladmi objektových úložísk sú Amazon S3, Google Cloud Storage a Azure Blob Storage. [16]

- **Súborové úložiská:** Súborové úložiská sú navrhnuté s cieľom centralizovane ukladať a zdieľať súbory medzi viacerými inštanciami aplikácie alebo používateľmi, keď je dôležitá flexibilita, škálovateľnosť a prístupnosť k dátam. Príklady súborových úložísk v cloud computing zahŕňajú Google Cloud FileStore, Azure File Storage a Amazon Elastic File System.

  K súborovým úložiskám patria aj archivačné úložiská, ktoré sa vyznačujú nízkou frekvenciou prístupu k dátam. Príkladom je napr. Amazon Glacier alebo Azure Archive Storage.

- **Databáza ako služba:** Databáza ako služba (skr. DBaaS) poskytuje zákazníkovi prístup k databáze bez potreby databázu nasadzovať, spravovať infraštruktúru a starať sa o veľkosť úložiska. Tieto služby umožňujú užívateľom využívať databázy bez potreby širších vedomostí o správe databázového systému. DBaaS sú automaticky škálované, zálohované a sú vysoko dostupné, čo dáva zákazníkovi viac priestoru sústrediť sa

na samotnú aplikáciu. Poskytovatelia cloudových služieb ponúkajú databázy rôznych typov, vrátane spomenutých v nasledujúcej časti.

Databázové systémy

Systémy pracujúce v reálnom čase vyžadujú nízku latenciu, vysoký prietok dát a rýchlu reakciu na prijaté zmeny, preto sú často využívané distribuované databázy, ktoré sú ale obmedzené CAP teoréom. Typy možných databázových systémov využitých pre real-time aplikácie sú rôzne a sú popísané nižšie.

Relačné databázy

Relačné databázy sú známe a široko využívané. Dáta sú organizované do relácií známych ako tabuľky. V kontexte real-time systémov nie sú zvyčajne prvou voľbou a často sa využívajú len pre ukladanie dát, ktoré si potrebujú zachovávať konzistenciu, pretože sú založené na transakčnom spracovaní a princípoch ACID, kde je kladený práve vysoký dôraz na konzistenciu, čo však môže mať vplyv na rýchlosť vykonania transakcií. Okrem toho relačné databázy vyžadujú fixnú štruktúru dát, čo pri real-time dátach, ktor majú častokrát rôznorodú podstatu, nie je veľmi efektívne. Kvôli týmto dôvodom sú oveľa vhodnejšie databázy popísanie nižšie.

NoSQL databázy

NoSQL databázy sa od tradičných relačných databázových systémov líšia v spôsobe ukladania dát. Zatiaľ čo relačné databázy vyžadujú dopredu preddefinovanú schému dát, NoSQL databázy ponúkajú vyššiu flexibilitu v schéme ukladania a dokážu ukladať rôznorodé typy dát. Tieto databázy sú väčšinou navrhované pre ukladanie veľkého objemu dát a pre poskytovanie dobrej škálovateľnosti, ale často uprednostňujú dostupnosť a čiastočnú odolnosť voči chybám pred konzistenciou (AP z CAP). Pojem NoSQL zahŕňa databázy viacerých kategórii:

- **Databázy typu kľúč-hodnota:** Pri tomto type databáz je hodnota priradená konkrétnemu kľúču, ktorý slúži ako index v tabuľke. V prípade, že príde nová hodnota s rovnakým kľúčom, hodnota priradená ku kľúču sa iba aktualizuje. Vyhľadávanie je možné iba podľa kľúča a nie podľa hodnoty. Vďaka jednoduchému princípu sú tieto databázy vysoko výkonné, škálovateľné a poskytujú nízku latenciu, ale nie sú vhodné v prípade zložitejších dotazov a agregácií [29]. Tieto databázy sú kvôli ich rýchlosti, častokrát využívané aj ako vyrovnávacie pamäte. Populárne databázy kľúč-hodnota sú napr. Redis, Memcached, Voldemort a Oracle KV store.
- **Dokumentové databázy:** V dokumentových databázach sú dáta ukladané ako hodnota, ktorá je spojená s jedinečným identifikátorom. Sú nazývané dokumentové databázy, pretože ukladajú štrukturované alebo semištrukturované dáta, ktoré sa nazývajú dokumenty. Dokumenty môžu byť zanorované a odkazovať sa na iné dokumenty. Dokumenty sú ukladané v rôznych formátoch, ako sú formáty XML, JSON, BSON, čo tieto dokumenty robí jednoducho prenositeľnými, hlavne cez webové technológie. Ponúkajú horizontálne škálovanie a vysokú flexibilitu, preto dokážu ukladať veľké množstvo dát rôznej variácie a umožňujú vyhľadávanie a získavanie dát na základe hodnôt. Tieto databázy ale nie sú vhodné v prípade komplikovaných many-to-many

vzťahoch, pretože mnohokrát poskytujú obmedzené funkcie typu join [22]. MongoDB a CouchDB zastupujú tento typ databáz.

- **Stĺpcové databázy:** Stĺpcové databázy pripomínajú relačné databázy, ale v tomto type databáz sú dáta z jedného stĺpca uchovávané spolu, čo robí pridávanie nových stĺpcov nenákladnou operáciou. Každý riadok môže mať rôzny počet stĺpcov, čo umožňuje, aby tabuľky boli riedke, bez pridanej náročnosti. [29] Tieto databázy sú efektívne pri čítaní veľkého množstva dát a umožňujú jednoduchý prístup k dátam z vybraných stĺpcov. Kvôli svojej stĺpcovej povahe sú ideálne pri agregregáciách a analýzach a mnohokrát sú využívané aj pri spracovaní časových radov. Tieto databázy ale nemusia byť najvhodnejšie pre real-time spracovanie, ktoré vyžaduje okamžitú dostupnosť najnovších dát. [28] Medzi populárne stĺpcové databázy patria HBase, Cassandra a ClickHouse.

- **Grafové databázy:** Tento typ databáz využíva pre ukladanie dát grafovú štruktúru s vrcholmi, hranami a vlastnosťami, ktoré popisujú hrany. Hrany s vlastnosťami tvoria vzťahy, a preto grafové databázy nevyžadujú žiadnu preddefinovanú schému, pretože tá je budovaná na základe pridávaných dát. Hrany môžu byť aj orientované, čo umožňuje efektívne prehľadávanie grafu. Ku grafovým databázam patria Neo4J a Polyglot.

Databázy časových radov

Niektoré databázy časových radov môžu byť radené medzi NoSQL databázy, ale nie všetky, a preto majú svoj vlastný odstavec.

Ako vyplýva zo samotného názvu, databázy časových radov (ang. *time-series databases*) pracujú s časom a každý záznam obsahuje časové razítko. Sú optimalizované pre prácu s veľkým objemom časových radov, pričom sa pozerajú na čas ako na hlavný prvok. Tieto databázy slúžia k časovým analýzam a agregáciám, ktoré sú založené na časových intervaloch. Time-series dáta sú podmnožinou temporálnych dát a tento typ databáz často umožňuje nastavenie automatického odstraňovania dát po ich „expirácii", čo umožňuje efektívnu správu miesta na disku. Zmeny v uložených záznamoch (t.j. aktualizácie) nie sú pre tento typ databáz bežné. Sú využívané hlavne pri IoT aplikáciách, finančných službách, analytických službách a pri monitoringu systémov. Príklady databáz časových radov sú napr. InfluxDB, TimeScaleDB, Prometheus.

4.3 Nasadenie aplikácie a integrácia spracovania dát s úložiskom

Ako bolo už spomenuté, pre výpočty v cloud computing sa využívajú virtuálne stroje a pre ukladanie dát možnosti spomenuté v časti 4.2. Spracovanie dát často neprebieha iba prostredníctvom jednej služby, ale cez mnoho spolu súvisiacich vrstiev a spracovanie zvyčajne končí uložením. Všetky tieto služby musia medzi sebou komunikovať a prípadne ukladať dáta do databázy. Služby spolu komunikujú cez API, HTTP protokol, cez fronty správ (4.1) alebo cez zbernice. Technológie zamerané na spracovanie prúdov dát sú veľakrát vybavené adaptérmi pre priame uloženie dát do databázy. Čo sa týka prepojenia blokových úložísk s virtuálnym strojom, táto integrácia je možná pomocou pár kliknutí. Prepojené blokové úložisko sa zobrazuje vo virtuálnom stroji ako klasický blok v počítačovom systéme,

napr. `/dev/sdg`. Medzi spôsob ako nasadiť služby do cloudového prostredia, okrem už spomenutého serverless prístupu alebo využitia SaaS, patria virtuálne stroje a kontajnerizácia. Táto časť sa sústredí na tieto dva princípy a ich integráciu s úložiskom dát.

Virtuálne stroje

Virtuálne stoje (skr. VM) sú virtuálne inštancie počítača v cloudovom prostredí, pričom zákazník má možnosť výberu operačného systému. VM bežia na fyzickom stroji a získavajú výpočtové zdroje od softvéru nazývaného hypervisor, ktorý riadi prideľovanie zdrojov rôznym VM na základe potreby. Pre zaistenie bezstavosti VM sú blokové úložiská pripájané k VM ako lokálny hard drive, ale väčšinou môže byť blokové úložisko pripojené iba k jednému virtuálnemu stroju, preto sa často využívajú skôr zdieľané súborové systémy. Neukladanie dát priamo na VM prináša jednoduchú škálovateľnosť ako pre výpočet tak aj pre úložisko. Pri takejto separácii zostávajú dáta perzistentne uložené aj po vypnutí alebo odpojení VM. Aplikácie môžu byť na VM nasadené v kontajneroch alebo môžu byť spustené priamo po inštalácií všetkých potrebných závislostí. Škálovateľnosť je zaistená pridávaním ďalších VM inštancií, kde každá inštancia VM obsahuje kópiu aplikácie. V prípade viacerých inštancií hrá dôležitú úlohu aj vyvažovanie záťaže, vyberajúce, na ktorú inštanciu budú aktuálne dáta odoslané. [13] [16]

Kontajnerizácia

Kontajnerizácia je koncept, v rámci ktorého sú aplikácie organizované do kontajnerov, pričom tieto kontajnery poskytujú pre aplikácie izolované prostredie s operačným systémom a zabezpečujú rovnaké podmienky pre rozdielne prostredie, čím sa zjednodušuje prenositeľnosť. Kontajnery môžu byť spustené na lokálnom počítači, na VM, fyzických serveroch alebo v cloudovom prostredí bez toho, aby bola potrebná ich modifikácia pre dané prostredie. Okrem toho kontajnery zjednodušujú škálovateľnosť, pretože je možné jednoducho spustiť viac inštancií kontajneru, keďže sú izolované. V prípade potreby sú kontajnery prepojené sieťou, cez ktorú dokážu navzájom komunikovať. Pre uchovanie dát aj po ukončení alebo reštarte kontajneru je nutné prepojiť kontajner s perzistentným úložiskom. Medzi najznámejšiu službu pre kontajnerizáciu patrí technológia Docker, ktorá dovoľuje vytvoriť tzv. docker volume na ukladanie a zdieľanie dát medzi kontajnermi. [13] [18]

Poskytovatelia cloudových služieb ponúkajú nasadenie kontajnerov aj prostredníctvom serverless spôsobu, kedy sú kontajnery spúšťané na VM, ale o právu VM sa stará poskytovateľ služby. AWS pre tieto účely ponúka službu AWS Fargate, Microsoft Azure poskytuje službu Azure Container Instances a GCP službu Google Cloud Run. Azure Container Instances neumožňuje škálovanie kontajnerov a v prípade potreby viacerých inštancií je nutné využiť API na automatizáciu a vytváranie nových požiadaviek. Automatické škálovanie kontajnerov na AWS Fargate je možné pomocou služby Elastic Container Service. Google Cloud Run škáluje kontajnery automaticky. Serverless kontajnerizované služby zvyčajne poskytujú určitú kapacitu dočasného úložiska počas behu kontajneru zdarma, ale v prípade nutnosti uchovávania informácií je možné pripojiť perzistentné úložisko súborových systémov alebo manažovaných databáz.

Orchestrácia

Systém orchestrácie koordinuje všetky potrebné podsystémy pre prevádzkovanie služby, vrátane procesov nasadzovania kontajnerov a konfigurácie siete a úložiska. Okrem toho

orchestrácia zabezpečuje dynamické škálovanie služieb, koordináciu medzi viacerými servermi a automatickú obnovu kontajnerov v prípade výpadku. Medzi najznámejšiu službu pre orchestráciu kontajnerov patrí technológia Kubernetes. Táto technológia je založená na modeli clustra, ktorý v sebe spája viac kontajnerov. V prípade architektúry mikroslužieb je aplikácia zložená z viacerých kontajnerov, ktoré sú navzájom prepojené a pri replikácii týchto kontajnerov by mali byť spúšťané spolu. Kubernetes používa na označenie aplikácie, zloženej z jedného alebo viacerých kontajnerov, ktoré sú spúšťané spolu ako celok, pojem *pod*. [13]

Pre nasadenie kontajnerov do prostredia cloud computing ponúkajú poskytovatelia cloudových služieb možnosť využitia manažovaných služieb pre orchestráciu kontajnerov. Tieto služby automatizujú vytváranie inštancií pods, zabezpečujú pružnosť aplikácie a spravujú dostupnosť a škálovateľnosť uzlov ovládacieho plánu Kubernetes. Zákazník dodá svoje služby vo forme kontajnerov a definuje zdroje, ktoré aplikácia vyžaduje. V takomto prípade sa o správu infraštruktúry stará poskytovateľ cloudovej služby. Čo sa týka úložiska pre kontajnerizované služby pre jeden pod, je možné využitie blokového úložiska, ale pre viac pods, ktoré potrebujú zdieľať úložisko, je nutné využiť zdieľané súborové úložisko. AWS poskytuje takúto službu pod názvom EKS, Azure ako AKS a GCP pod názvom GKE.

4.4 Zhrnutie

V tejto kapitole boli predstavené dva typy frontu správ, pričom obidva typy predstavujú dôležitý prvok systému, ktorý zjednodušuje celkovú architektúru a umožňuje výmenu správ medzi rôznymi komponentami v systéme. V rámci tejto práce je front založený na logoch vhodnejším riešením, pretože pre navrhovaný systém je dôležité poradie správ, hlavne v kontexte vstupov a výstupov z polygonov, a zároveň je pre systém nutná schopnosť rýchleho spracovania veľkého objemu dát. Ďalej boli predstavené možnosti ukladania dát v rámci cloud computing, rôzne druhy databázových systémov a možnosti nasadenia systému v prostredí cloud computing. Vzhľadom na štruktúru dát uvedenú v časti 5.6, ide o dáta rôznorodé a nie vždy sú dáta vyžadované všetkými službami, navrhované riešenie bude využívať viac druhov databáz, opísanými v tejto kapitole. Presné typy je možné nájsť v časti 5.6. Čo sa týka cloudových úložísk dát, ich výber záleží od špecifického prípadu použitia. Blokové úložiská majú využitie ako úložiská pre databázy, pre virtuálne stroje a ako perzistentné úložiská pre docker kontajnery. Nevýhodou blokových úložísk je, že často ich nie je možné pripojiť k viac inštanciám VM a ani k viacerým kontajnerom. V prípade nutnosti zdieľať úložisko medzi viacerými inštanciami VM alebo medzi kontajnermi sú vhodnejšie súborové úložiská. Objektové úložiská sú vhodné pre neštrukturované dáta a sú využívané pre ukladanie rôznych objektov ako sú videá, obrázky, dokumenty, ale aj rôzne logy alebo veľké datasety. Databáza ako služba prináša jednoduché a rýchle databázové riešenie, ale v niektorých prípadoch môže byť menej cenovo výhodné ako využitie iného typu úložiska a správy vlastnej databázy. Na druhej strane správa a údržba vlastnej databázy môže byť taktiež nákladná operácia. V závere boli predstavené možnosti nasadenia služieb do prostredia cloud computing, pričom VM a kontajnery spolu úzko súvisia. Aplikácia môže byť nasadená na VM priamo alebo pomocou kontajnerov pre lepšiu škálovateľnosť, prípadne môže byť aplikácia nasadená pomocou SaaS služieb pre správu a nasadenie kontajnerov.

Kapitola 5

Návrh real-time systému v cloud computing

Táto kapitola sa zaoberá návrhom systému, pracujúcom v reálnom čase. Začína prieskumom existujúcich real-time systémov, navrhnutých pre prostredie cloud-computing, a pokračuje samotným návrhom systému pre spracovanie veľkého toku geo-temporálnych dát.

5.1 Existujúce riešenia

Pri navrhovaní a implementácii nového systému v oblasti cloud computingu je dôležité preskúmať existujúce riešenia v tejto doméne pre lepšie pochopenie aktuálnych postupov pri návrhu systému. Väčšina moderných architektúr v prostredí cloud computing využíva základ architektúry podobný ako na obr. 3.3. Okrem toho poskytovatelia cloudových služieb ponúkajú mnoho ukážkových príkladov architektúr pre rôzne prípady použitia, ktoré môžu pomôcť k lepšiemu pochopeniu, ako vhodne navrhnúť architektúru systému pre prostredie cloud computing. Keďže táto práca je zameraná na spracovanie geo-temporálnych dát v reálnom čase, porovnávané riešenia budú systémy založené na princípe spracovania tohto typu dát v reálnom čase.

Prvé riešenie, ktoré bude bližšie preskúmané je architektúra služby Uber. Druhým riešením pre demonštráciu, ako môže byť takýto systém nasadený a navrhnutý v rámci prostredia cloud computing, bude systém, spracovávajúci prúd dát v cloudovom prostredí od Azure.

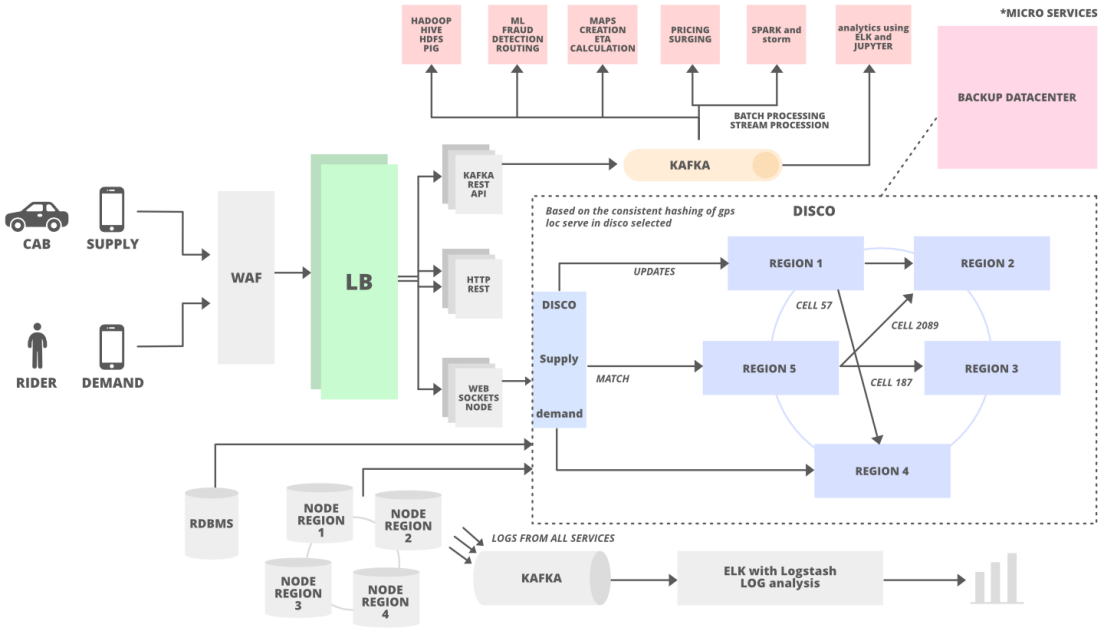
Architektúra služby Uber

Aplikácia Uber začala ako služba, ktorá spája vodičov áut so zákazníkmi, potrebujúcimi odvoz. Postupne bola táto služba rozšírená aj o ponuku doručovanie jedla a prepravu tovaru. Momentálne je pre túto aplikáciu využívaná architektúra orientovaná na služby. V tomto krátkom prehľade bude predstavená hlavná myšlienka systému a základné služby, ktoré spájajú šoférov so zákazníkmi.

Služba ponuky Služba zodpovedná za sledovanie dostupných áut (služba ponuky) musí pravidelne zaznamenávať polohu áut. Autá periodicky posielajú svoje GPS súradnice na webový server prostredníctvom aplikačného firewallu a cez službu vyvažovania záťaže. Táto poloha v podobe GPS súradníc je následne odoslaná do frontu správ Apache Kafka cez REST API tohto frontu správ. Následne je poloha z frontu správ ukladaná do databázy a do dispečingového systému.

Služba dopytu Služba dopytu spracováva žiadosti o jazdu. Žiadosť je prijatá cez protokol WebSocket spolu s polohou zákazníka a rôznymi špecifickými požiadavkami zákazníka. Služba priradí polohe zákazníka identifikátor a na základe tohto identifikátora vytvorí žiadosť o jazdu.

Dispečingový systém Aplikácia musí efektívnym spôsobom vybrať najbližšie autá vyhovujúce vytvorenej žiadosti o jazdu. Uber používa pre tento problém dispečingový systém (na obr. 5.1 ako DISCO), ktorý priraďuje dostupné vozidlá k požiadavkám zákazníkov na základe GPS súradníc. Po spracovaní žiadosti o novú jazdu je zákazníkova poloha odoslaná na server, kde sú vyhľadané dostupné autá v rámci definovanej vzdialenosti. Zoradený zoznam dostupných áut podľa vzdialenosti je odoslaný naspäť službe zodpovednej za dopyt. Takýto systém musí zvládať obrovské množstvo zápisov a čítanie množstva polôh za krátky časový úsek a pre tento účel je využívaná knižnica Google S2 a Geohash.



Obr. 5.1: Architektúra služby Uber [10].

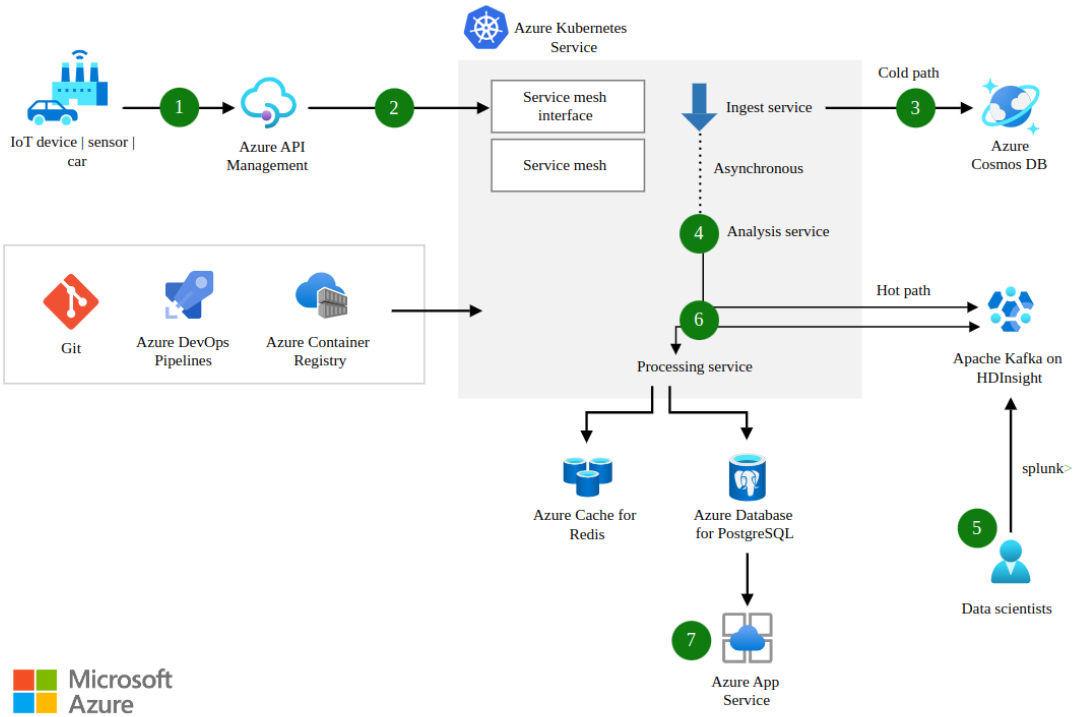
Čo sa týka ostatných využívaných technológií, Uber využíva Web Application Firewall, ktorý slúži pre bezpečnostné účely a využíva vyvažovanie záťaže cez viac vrstiev. Systém Uber implementuje rôzne iné služby, ako služby pre analytické účely alebo strojové učenie a preto využíva široké množstvo technológií, ako sú Apache Spark, Hive, HDFS, Apache Samza a ďalšie. Uber využíva aj technológiu Apache Flink, ktorá spracováva prúdy dát pre modely strojového učenia, ktoré rozhodujú napríklad o meniacich sa cenách[1].

Uber využíva pre ukladanie dát viac databáz pre rôzne účely. Pre historické ukladanie dát využíva databázu Schemaless, zatiaľ čo pre dosiahnutie nízkej latencie a vysokého stupňa dostupnosti využíva stĺpcovú databázu Cassandra a databázu typu kľúč-hodnota Riak. Ďalšia databáza typu kľúč-hodnota Redis je využívaná ako vyrovnávacia pamäť. Pred expirá-

[1]pre viac info. viď https://www.uber.com/en-PL/blog/building-scalable-streaming-pipelines/

ciou dát z frontu správ Kafka sú dáta ukladané do služby Hadoop alebo do súborového systému. [25] [10]

IoT systém pre spracovanie dát v reálnom čase



Obr. 5.2: Architektúra a nasadenie real-time IoT systému v prostredí Microsoft Azure (prevzaté z [6]).

Návrh aplikácie na obr. 5.2 prezentuje systém vhodný pre spracovanie veľkého objemu dát v rozsahu miliónov záznamov, ktorý prúdi zo senzorov, IoT zariadení alebo dopravných prostriedkov. Hlavnú rolu v spracovaní takého veľkého objemu dát hrajú kontajnerizované služby nasadené cez službu AKS, v ktorej sú služby dynamicky škálované podľa potreby. Dáta, prichádzajúce do systému, sú riadené službou Azure API Management, ktorá umožňuje riadiť a sprístupňovať rôzne API v rámci cloudového prostredia. AKS cluster spúšťa mikroslužby, nasadené ako kontajnery, s využitím služby Service Mesh, ktorá reprezentuje vrstvu, zabezpečujúcu komunikáciu medzi službami. Služba AKS bola popísaná už v kapitole 4.3 v časti orchestrácie. Tieto kontajnery sú uložené v registri kontajnerov Azure Container Registry, čo uľahčuje nasadenie kontajnerov v rámci služby AKS. Služba zodpovedná za prijímanie dát, ukladá tieto dáta do databázy a súčasne sú dáta odosielané do analytickej služby, ktorá odosiela prijaté dáta do služieb Apache Kafka a Azure HDInsight, ktoré zabezpečujú analytické výpočty, s ktorými ďalej pracujú dátoví analytici. Služba Azure HDInsight je služba typu SaaS, ktorá umožňuje spracovanie veľkých objemov dát pomocou nástrojov ako sú Apache Spark alebo Apache Hadoop. Následne služba zodpovedná za spracovanie dát, dáta po spracovaní uloží do relačnej databázy a do vyrovnávacej pamäte Redis. Na-

koniec môžu byť dáta vizualizované pomocou webovej aplikácie spustenej pomocou Azure App Service. [6]

Zhrnutie návrhu IoT systémov

Architektúra riešenia Uber ponúka komplexnejší dátový tok ako druhé predstavené riešenie, avšak obe preskúmané riešenia využívajú princípy určitého prostredníka, do ktorého prúdia vstupné dáta, ktoré sú následne ďalej spracovávané inými službami. Tieto riešenia demonštrujú, že je bežné používať viac typov databáz. Druhé riešenie ukazuje možnosť nasadenia aplikácie pomocou manažovanej služby pre Kubernetes. Ďalšie možnosti ako môžu byť takéto aplikácie nasadzované v prostredí cloud computing už boli popísané v časti 4.3.

5.2 Analýza požiadaviek

Systém má slúžiť ako prototyp pre demonštráciu zvládnutia efektívneho spracovania veľkého toku geo-temporálnych dát v prostredí cloud computing. Systém pracuje hlavne s entitami bodov a polygonov. Bodom sa rozumie zariadenie alebo satelitný lokátor, pričom v rámci systému platí bijekcia, kde jeden užívateľ má priradené jedno zariadenie. Systém má za úlohu zaznamenávať polohu týchto bodov. Polygon reprezentuje kategorizované ohraničené územie. Bod a polygon sú statické entity, ktoré sú menené zodpovedným užívateľom, a majú priradené statické atribúty.

Hlavné prípady použitia:

Do systému prúdi tok vstupných dát v objeme jedného milióna záznamov za minútu. Tento tok dát obsahuje údaje s identifikátorom zariadenia, časové razítko a GPS súradnice. 1000 klientov číta informácie o aktuálnej polohe rôznych bodov v intervale 5 sekúnd. Okrem toho klient môže požiadať o informácie popísané v zozname nižšie.

Prípady použitia sa týkajú získavania informácií o nasledujúcich prípadoch:

- Aktuálna poloha užívateľov
- Zoznam užívateľov
- Zoznam polygonov
- Aktuálne prebiehajúce kolízie
- Kolízie v časovom intervale
- Historické polohy užívateľov v určitom časovom intervale

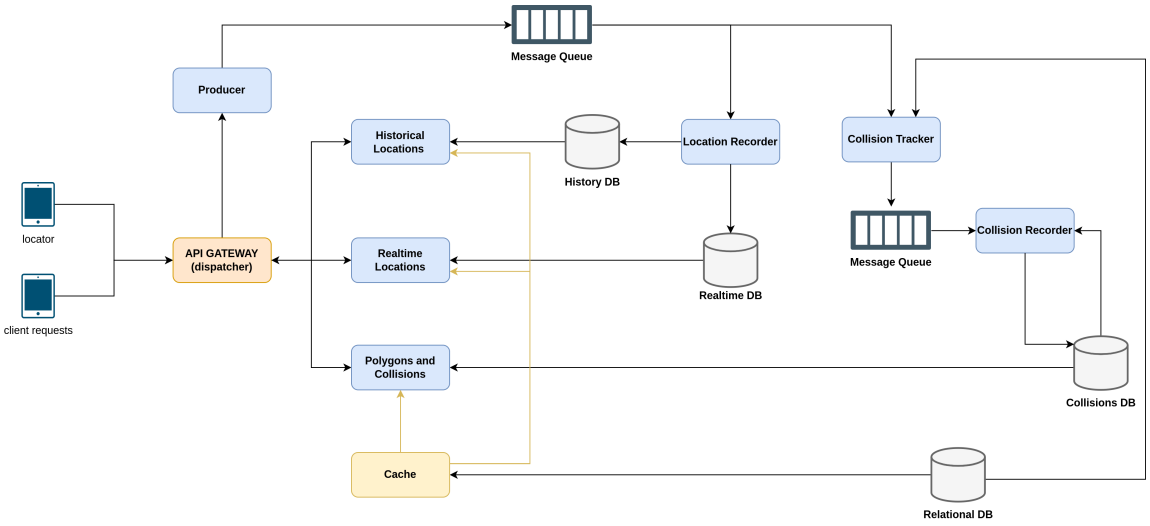
Ako je vidieť zo zoznamu prípadov použitia, ide o prípady použitia v prítomnom i minulom čase. Pre systém je veľmi dôležitá rýchla projekcia vstupných dát do odpovedí na žiadosti týkajúcich sa aktuálnych informácií.

5.3 Architektúra systému

Navrhované riešenie využíva architektúru riadenú udalosťami a rozdeľuje celú architektúru do viacerých mikroslužieb. Rozdelenie do mikroslužieb umožňuje horizontálne škálovanie

a zachováva princíp nízkej zviazanosti medzi službami. Riešenie je navrhnuté s ohľadom na to, aby bolo možné aplikáciu nasadiť aj on-premise spôsobom a zároveň aj v cloud computing prostredí bez fixácie na jedného poskytovateľa.

Na základe analýzy požiadaviek z kapitoly 5.2 je systém rozdelený do viacerých modulov. Systém bude využívať štyri databázy podľa príkladu existujúcich riešení z predchádzajúcej časti 5.1. Pre lepšie pochopenie celkovej architektúry na obrázku 5.3 sa dá architektúra zhrnúť do 4 častí: prijímanie dát, spracovanie dát, ukladanie dát a obsluha žiadostí od klientov.



Obr. 5.3: Rozdelenie architektúry do mikroslužieb. Zobrazené fronty správ predstavujú tú istú službu, ale ide o rozdielne topiky, a preto pre lepšie vyjadrenie prúdu dát je znázornená ako dve služby.

Vstupné polohy i žiadosti o dáta sú odosielané na API gateway, ktoré ich následne odošle na príslušnú službu. Vstupné dáta API gateway presmeruje na službu Producer, ktorá ich vloží do frontu správ, odkiaľ si ich prevezmú dve služby. Prvá z týchto služieb je služba Location Recorder, ktorá je zodpovedná za uloženie prijatej polohy z frontu správ do databázy aktuálnych polôh bodov, a druhej databázy, ktorá zaznamenáva historické záznamy. Druhou službou, ktorá si prevezme vstupné dáta, je služba Collision Tracker zodpovedná za výpočty, týkajúce sa prekročenia hraníc polygonov. V prípade prekročenia hraníc vygeneruje novú udalosť, ktorá je uložená do iného topiku vo fronte správ. Z tohto topiku si vygenerované udalosti preberá služba Collision Recorder, ktorej úloha je tieto udalosti uložiť do databázy. Bližšie sa týmto službám venujú kapitoly nižšie. Pri dopytovaní sa na dáta, sú žiadosti rozdelené do troch služieb na základe toho či sa jedná o historické alebo aktuálne žiadosti, alebo o žiadosti, týkajúce sa prekročenie hraníc. Tieto tri služby vyplývajú z vyššie popísaných prípadov použitia.

5.4 Príjmanie dát

Služby, zodpovedajúce za prijatie dát, sú API gateway, služba Producer a front správ. Vstupným bodom systému je služba API gateway, ktorá poskytuje klientovi jednotný prístupový bod, cez ktorý komunikuje so systémom. API gateway sa stará o presmerovanie žiadostí k správnym službám. V tomto prípade ide o prijatie nových polôh, ktoré API gateway pre-

smeruje na službu producenta — Producer. Služba Producer slúži ako WebSocket server a producent vstupných polôh do frontu správ. Front správ slúži ako prostredník pre zníženie závislostí medzi službami, čím je podporované aj nezávislé a paralelné spracovanie. Front správ bude typ založený na logu, pretože je dôležité udržať správne poradie správ pre ďalšie spracovanie a systém musí byť schopný rýchlo spracovať veľký tok správ. V navrhovanom riešení má tento topik dvoch konzumentov, ktorí spracovávajú dáta. Front správ zároveň zjednodušuje budúce rozšírenie systému, pretože umožňuje jednoduché pripojenie ďalších konzumentov, t.j. služieb.

5.5 Spracovanie dát

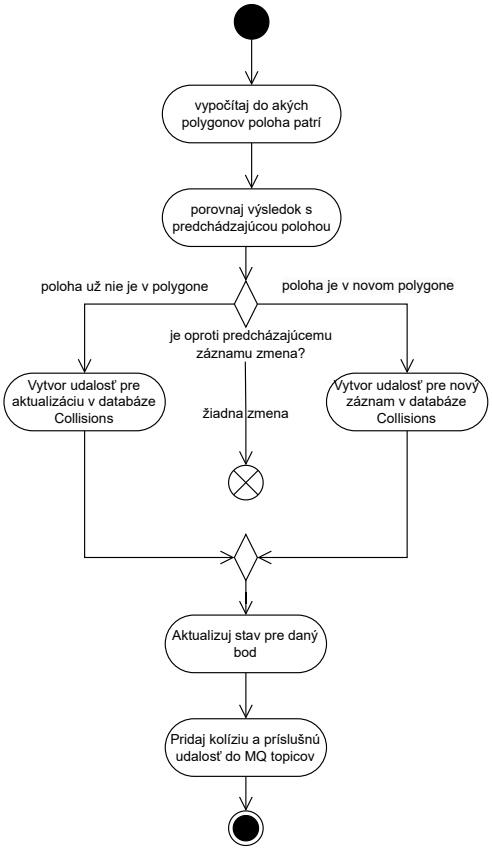
Za spracovanie dát sú zodpovedné služby Location Recorder, Collision Tracker a Collision Recorder, ktoré fungujú ako konzumenti správ z frontu správ. Location Recorder, je služba zodpovedajúca za ukladanie vstupných dát, služba Collision Tracker je zodpovedná za výpočty nad prúdom vstupných dát a vzhľadom na spracovanie veľkého toku dát bude táto služba využívať stream procesor. Tretia služba, Collision Recorder, je zodpovedná za prípadné ukladanie vygenerovaných udalostí službou Collision Tracker.

Location Recorder

Ako bolo už spomenuté, služba Location Recorder získava dáta z frontu správ a je zodpovedná za uloženie prijatých polôh do dvoch databáz, keďže je pre systém dôležité poskytovať pohľad na aktuálne polohy aj pohľady v historických intervaloch, pričom prvý spomínaný prípad je dôležitejší z pohľadu latencie, a preto je vybraný typ databázy poskytujúci najnižšiu latenciu (viac v 5.6).

Collision Tracker a Collision Recorder

Služba Collision Tracker konzumuje správy z topiku, obsahujúceho nové polohy, a zároveň pracuje s polygonmi definovanými v priestore. Prvý krok výpočtu kolízií sa týka toho, či sa aktuálny záznam polohy nachádza v niektorom z polygonov. Následne musí toto zistenie porovnať s predchádzajúcim stavom pre rovnaké zariadenie, t.j. či došlo k prekročeniu hraníc polygonu (kolízii). Prekročením hraníc polygonu je myslený vstup do polygonu alebo výstup z polygonu. Táto služba je stavová, pretože si musí pamätať, v akých polygonoch sa dané zariadenie nachádza. Keďže sa polygony môžu prekrývať, zariadenie sa môže nachádzať aj vo viacerých polygonoch naraz. Služba porovnáva aktuálnu polohu s predchádzajúcim stavom. Ak došlo k novej kolízii (t.j. predchádzajúca poloha sa nachádzala v polygone a aktuálna sa nenachádza alebo predchádzajúca poloha sa nenachádzala v polygone a aktuálna sa nachádza v polygone), interný stav procesoru je pre dané zariadenie aktualizovaný, kolízia je zapísaná do príslušného topiku frontu správ, reprezentujúceho nové udalosti kolízií. V prípade, že došlo k výstupu z polygonu, záznam, ktorý zaznamenával vstup do polygonu, musí byť aktualizovaný, a ak došlo k novému vstupu do polygonu, tento záznam musí byť v databáze kolízií vytvorený. Pre odľahčenie práce stream procesoru a udržanie služby Collision Tracker, sústredenej na jednu zodpovednosť, nebude služba Collision Tracker vykonávať zmeny v databáze, ale iba vygeneruje udalosť do topiku, z ktorého si ich následne prevezme služba Collision Recorder, ktorá tieto udalosti uloží do databázy Collisions DB. V prípade, že aktuálny stav je identický s predchádzajúcim, výpočet končí bez zmeny. Pre lepšie pochopenie viď diagram aktivít 5.4.



Obr. 5.4: Diagram aktivít zobrazujúci tok akcií pre službu Collision Tracker.

5.6 Ukladanie dát

Vstupné dáta systému sú dvoch typov. Prvé sú dáta statické, v podobe polygonov a užívateľov. Tieto dáta budú menené iba zriedka. Druhým typom je vstupný prúd aktuálnych polôh zariadení. Z tohto vstupného prúdu dát vytvára služba Collision Tracker nové dáta o kolíziách. Vstupné dáta systému sú geo-temporálneho typu, čo znamená, že obsahujú dve rôzne dimenzie: časovú a priestorovú. Typy ukladaných dát a ich atribúty sú popísané v tejto časti. Podkapitola pokračuje databázami návrhu.

Ukladané dáta

Polohy

Zaznamenávané polohy sú ukladané do dvoch databáz a obsahujú nasledujúce atribúty:

- `device_id:Int` - identifikátor zariadenia
- `timestamp:DateTime` - časové razítko kedy došlo k odoslaniu polohy
- `Point(longtitude, latitude)` - poloha bodu

V prípade real-time databázy, ktorá je typu kľúč-hodnota, je ako kľúč použitý identifikátor zariadenia `device_id` a časové razítko spolu s bodom musia byť obalené do objektu, ktorý bude reprezentovať hodnotu.

Užívatelia

Užívatelia majú priradené konkrétne id zariadenia.

- `id:Int` - identifikátor užívateľa
- `device_id:Int` - identifikátor zariadenia

Polygony

Polygony sú statické údaje, ktoré môžu byť menené priamo v databáze. Polygony môžu byť neskôr označené ako neplatné, preto je pridaný atribút `valid:Boolean`, aby sa pri počítaní či bod patrí do polygonu, počítalo iba s aktuálne platnými polygony.

- `id:Int` - identifikátor polygonu
- `category:Int` - kategória polygonu
- `creation:DateTime` - dátum vytvorenia polygonu
- `fence:Geometry` - ohraničenie polygonu
- `valid:Boolean` - či je polygon, aktuálne platný

Kategórie

- `id:Int` - identifikátor kategórie
- `name:String` - popis kategórie

Kolízie

Kolízie sú ukladané do databázy kolízií s nasledujúcimi atribútmi, pričom pri vytvorení záznamu sú atribúty `collision_date_out:DateTime`, `collision_point_out:Point` nastavené ako null, a atribút `in:Boolean` je nastavený na hodnotu true. Atribút `in` je zmenený na hodnotu false pri výstupe z polygonu a zároveň sú doplnené nevyplnené atribúty.

- `collision_date_in:DateTime` - časové razítko dátumu kedy došlo ku kolízii vstupu do polygonu
- `in:Boolean` - tento atribút hovororí o tom, či je bod stále v polygone (t.j. či je kolízia aktívna)
- `polygon_id:Int` - identifikátor polygonu, pri ktorom došlo ku kolízii
- `device_id:Int` - identifikátor bodu, ktorý spôsobil kolíziu (t.j. prekročil polygon)
- `collision_point_in:Point` - poloha, pri ktorej došlo ku kolízii pri vstupe do polygonu
- `collision_date_out:DateTime` - časové razítko dátumu kedy došlo ku kolízii výstupu z polygonu
- `collision_point_out:Point` - poloha, pri ktorej došlo ku kolízii pri výstupe z polygonu

Databázy

Na základe ukladaných dát, popísaných vyššie, a prípadov použitia, ktoré sa vzťahujú k prítomnosti a minulosti a aj rozdelenia systému do služieb, sú pre tento systém navrhnuté štyri databázy. Zvolenými typmi a zdôvodnením sa zaoberá táto časť.

- **Relačná databáza:** Ako vyplýva z názvu, prvou databázou bude databáza relačná, ktorá bude použitá pre uchovávanie záznamov o definovaných polygonoch, ich kategóriách a o informáciách o užívateľoch. Táto databáza bola zvolená pre tento typ dát, pretože pri týchto dátach je dôležitá konzistencia záznamov, a relačné databázy podporujú aj zaznamenávanie geometrických objektov, čo je dôležité pre definovanie polygonov.
- **Databáza aktuálnych polôh:** Systém potrebuje v prvom rade poskytovať dáta takmer v reálnom čase a preto je kľúčové, aby zápis aktuálnych polôh a ich následné čítanie prebiehali s minimálnou latenciou. Pri žiadostiach na historické polohy, zápis aktuálnej polohy nie je až taký naliehavý. Z tohto dôvodu je ukladanie aktuálnych polôh rozdelené do dvoch databáz. Pre ukladanie aktuálnych polôh bola zvolená databáza typu kľúč-hodnota, kvôli efektívnemu ukladaniu dát a rýchlemu čítaniu aktuálnych hodnôt. Výhodou tohto typu databáz je, že pri novej polohe sa stará poloha iba aktualizuje.
- **Historická databáza:** Pre ukladanie historických hodnôt bola vybratá databáza časových radov, pretože sa jedná o záznamy polôh s časovou známkou a databázy časových radov sú optimalizované pre dotazovanie sa podľa konkrétneho času alebo v rámci časových okien. Zároveň temporálne dáta strácajú po čase svoj význam a tento typ databáz automaticky premazáva dáta po uplynutí nastavenej doby.
- **Databáza kolízií:** Táto databáza má za úlohu zaznamenávať históriu kolízií zariadení s hranicami polygonov, pričom charakteristika dát bola už opísaná vyššie. Keďže je nutné v prípade udalosti, týkajúcej sa výstupu z polygonu, aktualizovať už existujúci záznam, databáza by mala byť schopná rýchlych aktualizácií záznamov. Vhodná databáza by bola databáza dokumentová ako MongoDB, ktorá podporuje zaznamenávanie geografických polôh a objektov cez GeoJSON alebo je možné aj využitie databázy relačnej, pretože ako bolo opísané v časti 5.5, zmeny budú produkované do frontu správ, čo je vhodné v prípade budúceho rozšírenia systému, o službu, ktorá by informovala o vzniku kolízie formou notifikácií. V takomto prípade, by rýchlosť uloženia záznamu kolízie do databázy nebola časovo kritická.

5.7 Obsluha žiadostí od klientov

Cez API gateway prichádzajú okrem zaznamenávaných polôh aj žiadosti o dáta od klientov. API gateway presmeruje žiadosť na základe typu na jednu z troch služieb: Historical Locations, Realtime Locations a Polygons and Collisions. Tieto služby komunikujú s príslušnými databázami (zobrazené v diagrame 5.3) a každá služba potrebuje komunikovať s databázou relačnou, pretože napríklad pri obsluhe žiadostí pre aktuálne polohy užívateľov sa z databázy vrátia polohy s identifikátorom zariadenia, ale odpoveď pracuje s identifikátorom užívateľa. Pre zníženie doby odozvy z databázy je tu navrhnutá vyrovnávacia pamäť, ktorá bude obsahovať údaje `user_id - device_id`. Pre zníženie latencie bude využitý taktiež princíp vyrovnávacej pamäte v rámci API gateway, pre obsluhu opakujúcich sa žiadostí

o rovnaké dáta. Pre lepšie pochopenie úloh týchto služieb sú službám priradené prípady použitia z časti 5.2.

Historical Locations Služba Historical Locations obsluhuje žiadosti, vzťahujúce sa k minulým polohám bodov, a komunikuje hlavne s historickou databázou.

- `/history/locations/` - Historické polohy užívateľov v určitom časovom intervale
- `/history/locations/{user}` - Historick0 polohy (trajektória) špecifikovanej jednotky

Realtime Locations Úloha API služby Realtime Locations je rýchlo reagovať na žiadosti týkajúce sa aktuálnych polôh užívateľov. Komunikuje prevažne s databázou aktuálnych polôh.

- `/rt/locations/` - Aktuálna poloha užívateľov (jednotiek)
- `/rt/users/` - Zoznam užívateľov (jednotiek) v databáze

Polygons and Collisions Služba Polygons and Collisions má za úlohu obsluhovať žiadosti, týkajúce sa kolízií a polygonov. Komunikuje s databázou kolízií, odkiaľ získava informácie o aktuálnych zariadeniach v polygonoch, ale aj historické informácie o tom, kto sa nachádzal v polygone. Táto služba obsluhuje aj žiadosti, týkajúce sa priamo polygonov, z dát získaných z databázy relačnej.

- `/polygons/` - Zoznam polygonov v databáze
- `/collisions/current/` - Aktuálne prebiehajúce kolízie
- `/collisions/history/` - Kolízie v časovom intervale (jednotky nachádzajúce sa v polygonoch)

Kapitola 6

Implementácia real-time systému

Nasledujúca kapitola sa sústredí na koncepty implementácie jednotlivých častí navrhnutých v predchádzajúcej kapitole 5. Pre implementáciu jednotlivých služieb boli využité programovacie jazyky Python a Java, viac sa použitým technológiám venuje časť 6.1. Vstupný prúd dát je simulovaný s využitím generátora dát opísaného v časti 6.2. Prijímanie vstupných dát službou Producer je následne opísané v časti 6.3. Implementácii, týkajúcej sa spracovania dát, sa venuje podkapitola 6.4. Implementácia API, API gateway a vyrovnávacích pamätí je popísaná v časti 6.5. V závere tejto kapitoly sa nachádza časť o nasadení systému 6.6.

6.1 Použité technológie

Ako bolo už spomenuté, boli použité programovacie jazyky Python a Java. Konkrétne bol programovací jazyk Python využitý pre implementáciu služieb Producer a služieb API (Historical Locations, Realtime Locations a Polygons and Collisions). Jazyk Python bol vybratý pre tieto služby pre jednoduchosť a efektivitu v implementácii. Služby Location Recorder, Collision Tracker a Collision Recorder sú implementované v jazyku Java, pretože ponúka širšiu dostupnosť potrebných knižníc, ale v prípade budúcej potreby je možné služby Location Recorder a Collision Recorder implementovať aj v jazyku Python. Pre komunikáciu medzi systémom a generátorom dát bol zvolený protokol WebSocket, ktorý je vhodný pre real-time systémy, pretože ide o trvalé spojenie medzi serverom a klientom, čo umožňuje odosielanie a prijímanie správ bez potreby tvorby nového spojenia pri každej správe, čím sa znižuje latencia komunikácie. Pre front správ bola zvolená technológia Apache Kafka, pretože je navrhnutá pre distribuované spracovanie prúdu dát, čo je kľúčové pre spracovanie veľkých objemov dát. Okrem toho pri zvýšení počtu partitions, je zabezpečené, že správy s rovnakým kľúčom budú vždy priradené rovnakej partition, čo je potrebná vlastnosť v prípade škáľovania služby Collision Tracker, ktorá ako bolo už spomenuté v časti návrhu, pracuje so stavom. Pre výpočet kolízií zo vstupného prúdu dát bol zvolený stream procesor Apache Flink (3.2) s využitím technológie Apache Sedona[1], ktorá poskytuje funkcie pre priestorové výpočty, ako je napr. výpočet či sa bod nachádza v polygone. API služby využívajú komunikačný protokol HTTP a pre ich implementáciu bol zvolený framework FastAPI[2], ktorý poskytuje vysokú výkonnosť, umožňuje jednoduchý vývoj a podporuje dátovú validáciu. Pre API gateway bola zvolená technológia

[1] https://sedona.apache.org/
[2] https://fastapi.tiangolo.com/

NGINX[3], ktorá obsahuje aj vyvažovanie záťaže (ang. *loadbalancer*) a podporuje možnosť ukladania odpovedí na žiadosti do vyrovnávacej pamäti. Pre real-time databázu a zároveň aj pamäť cache pre uchovávanie bijekcie užívateľ–zariadenie bola zvolená databáza Redis[4], pretože ide o vysoko výkonnú databázu a dá sa využiť práve aj ako pamäť cache. Pre relačnú databázu bol použitý open source databázový systém PostgreSQL s priestorovým rozšírením PostGIS[5]. QuestDB[6] je výkonná databáza časových radov, ktorá podporuje WAL[7] tabuľky, čím sa znižuje latencia zápisu a zvyšuje priepustnosť a v rámci návrhu systému v tejto práci tvorí databázu historickú. Poslednou databázou, pre ukladanie kolízií, je dokumentovo-orientovaná databáza MongoDB[8], ktorá sa vyznačuje relatívne vysokou efektivitou pri aktualizáciách záznamov. Okrem toho ponúka možnosti pre ukladanie časových radov, ponúka možnosť ukladania dát aj vo formáte GeoJSON a disponuje priestorovými funkciami, čo predstavuje potenciálnu výhodu pre budúce rozšírenia systému. Jednotlivé služby sú nasadené pomocou Docker[9] kontajnerov pre umožnenie škálovateľnosti, jednoduchej prenositeľnosti na iné systémy a zjednodušenie spustenia systému.

6.2 Generátor polôh

Generátor polôh slúži pre simuláciu vstupného prúdu dát do systému. Ide o WebSocket klienta, ktorý má za úlohu odosielať nové polohy na WebSocket server. Generátor prijíma ako vstupné argumenty počet zariadení, rozsah súradníc pre zemepisnú dĺžku a šírku, URI WebSocket serveru a limit pre počet správ. Tieto argumenty nie sú povinné, v prípade nezadania sa použijú predvolené hodnoty.

Generátor najprv vytvorí prvotné spojenie s WebSocket serverom a pokračuje generovaním správ. Správy sú generované náhodne a kontinuálne, ak nie je vstupným argumentom služby definovaný limit vygenerovaných správ. Generátor vyberie náhodné id zariadenia z určeného počtu zariadení, vygeneruje súradnice x a y (reprezentujú zemepisnú dĺžku a šírku) zo zvoleného intervalu a priradí správe aktuálne časové razítko. Takto vygenerované správy následne odosiela na WebSocket server. Štruktúra správy je vidieť na kóde 6.1.

```
{
    "id": id zariadenia,
    "point": {
           "x": random(range_x),
           "y": random(range_y)
       },
    "timestamp": timestamp
}
```

Kód 6.1: Štruktúra odoslanej správy; x reprezentuje zemepisnú dĺžku (longtitude) a y reprezentuje zemepisnú šírku (latitude). Funkcia `random(range)` slúži pre vyjadrenie, že tieto body sú generované náhodne z definovaného intervalu.

[3] https://www.nginx.com/
[4] https://redis.io/
[5] https://postgis.net/
[6] https://questdb.io/
[7] skr. write-ahead log
[8] https://www.mongodb.com/
[9] https://www.docker.com/

6.3 Služba Producer

Služba Producer slúži ako WebSocket server a zároveň producent správ do frontu správ. Služba prijíma správy od klientov a ukladá ich do topiku (t.j. `new_locations`) vo fronte správ. Ako kľúč pre uloženie je použité id zariadenia získaného zo správy. Celá prijatá správa (pre formát správy viď 6.1) je predaná frontu správ ako hodnota. Štruktúru konfiguračného súboru je možné nájsť v časti príloh B.1.

6.4 Služby pre spracovanie dát

Po predaní vstupných polôh do Apache Kafka začína ich spracovanie. Ako bolo už opísané v časti 5.5, za spracovanie sú zodpovedné tri služby, ktorých implementácii sa venujú nasledujúce sekcie. Tieto tri služby predstavujú konzumentov správ (Collision Tracker je zároveň aj producent správ) z frontu správ. Pri implementácii konzumentov je dôležité nastaviť ich s určitými vlastnosťami. Jednou z kľúčových vlastností je `group.id`, ktoré jednoznačne identifikuje skupinu konzumentov. Toto umožňuje paralelné spracovanie správ pri vyššom počte partitions topiku, kde jednotliví konzumenti, patriaci do tejto skupiny, spracovávajú správy každý zo svojej priradenej partition. Do každej partition prúdia správy s rovnakým kľúčom, čo je užitočné pri stavovom spracovaní. Zároveň počet partition určuje hornú hranicu paralelného spracovania, keďže každá partition musí mať jednoznačne priradeného jedného konzumenta správ z danej skupiny konzumentov. Ďalšia kľúčová vlastnosť sú offsety, ktoré hrajú dôležitú úlohu pri konzumovaní správ z topiku, pretože určujú bod, po ktorý boli správy úspešne spracované. Medzi dôležité vlastnosti, nad ktorými je vhodné sa zamyslieť patria `auto.offset.reset` a `enable.auto.commit`. Vlastnosť `enable.auto.commit` určuje, či je povolené automatické zapisovanie offsetu prečítaných správ. Ak je táto vlastnosť nastavená na hodnotu `true`, konzument automaticky zapisuje offsety v intervaloch špecifikovaných nastavením `auto.commit.interval.ms`. Automatické zapisovanie offsetov je priamočiare, ale môže viesť k situáciám, kedy sú správy označené za spracované, aj keď ich spracovanie ešte nebolo dokončené, a preto je to vhodné nastavenie v situáciách, kde nevadí opätovné spracovanie rovnakých správ (tzv. at-least-once spracovanie). V prípade vypnutia automatického zapisovania offsetov (t.j. `enable.auto.commit=false`) je nutné offsety zapísať manuálne po spracovaní správ. Druhá spomenutá vlastnosť `auto.offset.reset` určuje od akej správy začne nový alebo konzument po reštarte spracovávať správy. Dve hlavné možnosti nastavenia tejto vlastnosti sú `earliest` a `latest`. Pre nastavenie `earliest` začína konzument spracovávať správy od najstaršej správy, ktorá ešte nebola označená za spracovanú[10] pre danú partition. Z tohto vyplýva, že konzument nevynechá pri spracovaní žiadne správy. V druhom prípade, t.j. `auto.offset.reset=latest`, začína konzument spracovávať správy od posledného dostupného offsetu, teda offsetu nachádzajúceho sa vedľa poslednej správy (t.j. offset poslednej správy plus 1). Z toho vyplýva, že konzument bude spracovávať iba správy, ktoré boli publikované do topiku po jeho štarte. Správy, ktoré boli publikované do topiku pred jeho štartom, bude ignorovať. Pre konzumentov opísaných v tejto časti je vlastnosť `auto.offset.reset` nastavená na hodnotu `earliest`, keďže je potrebné spracovávať aj správy, ktoré boli publikované do topiku, aj keď konzument nepracoval. Nastavenie `enable.auto.commit`, ale už nie je pre všetkých konzumentov rovnaké. Nastavenie tejto vlastnosti a vysvetlenie voľby pre každého konzumenta je popísané v nasledujúcich sekciách.

[10] pozn. najmenší zapísaný offset

Pri implementácií služieb Location Recorder a Collision Recorder bol pre kostru Apache Kafka konzumenta využitý kód zo stránky Confluent[11].

Location Recorder

Služba Location Recorder má podľa pôvodného návrhu uložiť nové polohy do dvoch databáz. Pri konkrétnej realizácii, ale nastala zmena v tom, že služba Location Recorder ukladá nové polohy iba do real-time databázy (t.j. Redis) a to z dôvodu toho, že pri vybratej databáze pre historické polohy (QuestDB) bol zvolený priamy spôsob ukladania dát z topiku Apache Kafka cez konektor[12], ktorý správy z topiku uloží priamo do definovanej tabuľky v konfigurácii tohto konektoru a na základe definovaného konvertoru hodnôt vytvorí príslušné stĺpce v tabuľke, do ktorých budú ukladané hodnoty záznamov. Pre aktivovanie konektoru je nutné pri prvom štarte Apache Kafka poslať REST žiadosť s konfiguráciou (viď 6.2) na adresu, na ktorých server Apache Kafka prijíma nové konektory[13]. Pre zjednodušenie registrácie konektoru je táto žiadosť odoslaná skriptom `create_kafka_questdb_connector.py` pri spustení docker compose. Riešenie cez konektor bolo zvolené na základe poskytnutých informácií v dokumentácii QuestDB[14], pretože QuestDB konektor pre Apache Kafka, poskytuje lepšiu výkonnosť ako využitie konzumenta správ z topiku v spojení s ukladaním dát do QuestDB cez konektor JDBC.

```
{
   "name": "QuestDB",
   "config": {
        "connector.class": "io.questdb.kafka.QuestDBSinkConnector",
        "topics": "new_locations",
        "host": "questdb:9009",
        "name": "QuestDB",
        "value.converter.schemas.enable": "false",
        "value.converter": "org.apache.kafka.connect.json.JsonConverter",
        "timestamp.field.name": "timestamp",
        "include.key": "false",
        "key.converter": "org.apache.kafka.connect.storage.StringConverter",
        "table": "locations_table"
    }
}
```

Kód 6.2: Konfigurácia pre vytvorenie QuestDB konektoru v Apache Kafka. Host je nastavený na adresu docker kontajneru QuestDB, cieľová tabuľka v databáze má názov `locations_table` a časové razítko pre indexovanie v tejto databáze je nastavené na parameter `timestamp` v správach, nachádzajúcich sa v topiku `new_locations`. Konvertor hodnôt je nastavený na JsonConverter, keďže správy v tomto topiku majú formát JSON. Bližšie vysvetlenie konfiguračných možností je možné nájsť na stránke QuestDB[12].

Samotnej službe Location Recorder zostala zodpovednosť uloženia dát do databázy Redis. Ide o klasického konzumenta správ z Apache Kafka implementovaného v jazyku

[11] https://developer.confluent.io/get-started/java/#build-consumer

[12] https://questdb.io/docs/third-party-tools/kafka/questdb-kafka/

[13] v prípade internej siete v rámci docker compose ide o adresu `http://connect:8083`, v prípade, že sa žiadosť neposiela cez internú sieť docker kontajnerov, je to adresa `http://localhost:8083/connectors`

[14] https://questdb.io/docs/ingestion-primer/

Java, ktorý ako kľúč pre vloženie záznamu do databázy použije kľúč správy a hodnotu správy z Apache Kafka použije ako hodnotu aj v databáze. V prípade tohto konzumenta je automatické zapisovanie offsetov, ktoré bolo spomenuté v úvode tejto podkapitoly, povolené keďže ide o ukladanie dát do databázy, v ktorej sa hodnoty pre kľúče menia pomerne rýchlo a často a opakované spracovanie správ nevadí, pretože staršia hodnota správy bude vždy prepísaná najnovšou. Nastavenie intervalu zápisu je ponechané na predvolenú hodnotu 5 sekúnd. Konfiguračný súbor pre službu Location Recorder sa nachádza v prílohe B.2.

Collision Tracker

Hlavná úloha tejto služby je výpočet kolízií medzi polygonmi a vstupným prúdom polôh zariadení. Apache Flink ponúka vlastné konektory pre Apache Kafka pre konzumovanie a produkovanie správ. Pre konzumovanie správ ide o triedu `KafkaSource` a pre produkovanie správ sa jedná o triedu `KafkaSink`. Okrem procesoru Apache Flink je využívané aj priestorové rozšírenie Apache Sedona, ktoré umožňuje nad prúdom dát robiť priestorové výpočty, týkajúce sa priestorových objektov ako sú body a polygony. Apache Sedona poskytuje funkcie pre výpočet, napr. či polygon obsahuje bod alebo výpočet vzdialenosti, z čoho sa dá dodatočnými operáciami získať KNN daného objektu[15]. Toto rozšírenie pracuje nad tabuľkami a nie nad samotným prúdom dát, preto je potrebné vstupný prúd dát zmeniť na tabuľku. Okrem vstupného prúdu dát, pracuje táto služba aj s polygonmi z databázy, ktoré musia byť taktiež prevedené do formátu tabuľky Apache Flink spolu s geometrickými stĺpcami, ktoré využíva Apache Sedona. Keďže ide o rôzne prevedenie dvoch druhov dát, ale rovnaký cieľ, bol využitý návrhový vzor Factory pre vytvorenie jednotného rozhrania (viď 6.3) pre tvorbu tabuľky pre polygony aj pre vstupný prúd polôh.

```
public interface TableFactory<T, U> {
/** Creates new table in stream environment with given colNames.
    Types of columns are specified inside concrete functions **/
Table createTable(StreamTableEnvironment tableEnv, T data, String[] colNames);
/** Creates Geometry column for given object and selects needed columns */
Table createGeometryTable(String[] colNames, Table sourceTable);
/** Creates a row for flink table */
Row createWKTGeometry(U object);
}
```

Kód 6.3: Rozhranie factory pre tvorbu tabuľky s tromi metódami.

Pre konštrukciu tabuľky je volaná metóda `createTable`, ktorá z prúdu správ z Apache Kafka vytvorí tabuľku a hodnotu obsahujúcu polohu namapuje na WKT formát pomocou metódy `createWKTGeometry()`. Pre tvorbu tabuľky pre polohy je funkcia `createTable()` založená na základe funkcie `createTextTable()` licencovanej pod Apache Licence 2.0 z oficiálneho GitHub repozitára Apache Sedona[16]. Pre polygony metóda `createTable()` ako prvé získa dáta z databázy a stĺpec obsahujúci polygony namapuje na formát WKT a vytvorí tabuľku. Po vytvorení oboch tabuliek sa musí vytvoriť ešte tabuľka geometrických objektov, aby sa dali aplikovať funkcie nástroju Apache Sedona. Túto zodpovednosť má metóda `createGeometryTable()`, ktorá vyberie potrebné stĺpce pre ďalšie spracovanie a zo spome-

[15] https://sedona.apache.org/1.5.1/tutorial/flink/sql/

[16] https://github.com/apache/sedona/blob/master/examples/flink-sql/src/main/java/Utils.java#L100/

nutých stĺpcov formátu WKT vytvorí geometrický stĺpec typu `POINT` pre polohy a `POLYGON` pre polygony. Z vytvoreného objektu tabuľky sa pred využitím funkcií poskytovaných nástrojom Apache Sedona musí zavolať interná funkcia `createTemporaryView()`, ktorá vytvorí z objektu príslušnej tabuľky pomenovanú reprezentáciu, nad ktorou sa už dá pracovať pomocou jazyka SQL, ako je možné vidieť na ukážke 6.4, v ktorej sú `locationTable` a `polygonTable` pomenované reprezentácie objektu príslušnej tabuľky, vytvorené pomocou už spomenutej funkcie `createTemporaryView()`.

```
Table joined = sedona.sqlQuery(
       "SELECT *, ST_Contains(geom_polygon, geom_point)
       AS is_in_polygon FROM locationTable, polygonTable");
```

Kód 6.4: Kartézske spojenie tabuliek pre polohy a polygony a pridanie stĺpca `is_in_polygon`, ktorý obsahuje boolean informáciu, či sa poloha pre daný riadok nachádza v polygone.

Po získaní informácie o tom, či sa bod nachádza v polygone, sa táto tabuľka opäť zmení na prúd dát, avšak ide o prúd riadkov tabuľky. Nad týmto prúdom je následne vykonaná postupná aplikácia transformácií, ktorá je zobrazená na ukážke kódu 6.5.

```
DataStream<String> collisionsEvents =
           resultStream.keyBy(r -> (Integer) r.getField(1))
           .flatMap(new PolygonMatchingFlatMap())
           .map(e -> e.toString());
```

Kód 6.5: Postupná aplikácia funkcií pre výpočet nových kolízií, kde `r.getField(1)` vyjadruje stĺpec z pôvodnej tabuľky, obsahujúci id zariadenia.

Prvou akciou je zoskupenie prúdu dát podľa id zariadenia, na ktorý sa následne aplikuje funkcia `flatMap()` triedy `PolygonMatchingFlatMap`. Táto trieda rozširuje abstraktnú triedu procesoru Apache Flink `RichFlatMapFunction`. Triedy typu `RichFlatMapFunction` sa líšia od klasickej FlatMap funkcie v tom, že umožňujú uchovávať aj stav operácií, čo je pre správny výpočet kolízií podstatná vlastnosť. Zoskupenie prúdu dát podľa kľúča (id zariadenia) zabezpečuje, že stav je uchovávaný pre každé id zariadenia samostatne. Uchovávaný stav je zoznam identifikátorov polygonov, v ktorých sa dané zariadenie nachádza. Z každého riadka tabuľky (keďže ide o prúd riadkov) sa získajú jednotlivé hodnoty stĺpcov a na základe zoznamu polygonov, v ktorých sa zariadenia aktuálne nachádza a hodnoty stĺpca `is_in_polygon`, získanej vyššie v ukážke (6.4), sa vypočíta, či nastala nová udalosť kolízie pre zariadenie, t.j. udalosť vstupu alebo výstupu z polygonu. Priebeh výpočtu je zobrazený na diagrame aktivít 5.4. Zoznam polygonov, v ktorých sa zariadenie nachádza, sa aktualizuje a v prípade novej kolízie je vygenerovaná udalosť jednej z podtried abstraktnej triedy `PolygonOutputEvent`, reprezentujúc druh kolízie, ktorý nastal. Táto udalosť je následne zmenená na textový reťazec pomocou metódy `toString()` objektov patriacich pod triedu `PolygonOutputEvent`. Z vytvoreného objektu je vytvorená textová reprezentácia formátu JSON pre následné uloženie kolízie do frontu správ (topik s názvom `collisions`). Výsledný reťazec, vytvorený volaním funkcie `toString()` pre udalosť značiacu vstup do polygonu, je možné vidieť na ukážke 6.6. Pre udalosť výstupu z polygonu sa táto ukážka líši v posledných dvoch atribútoch, ktoré sú zmenené na `collision_point_out` a `collision_date_out`.

```
{
    "event_type": typ udalosti
    "polygon": id polygonu
    "device": id zariadenia
    "in": boolean hodnota, ci sa zariadenie nachadza v polygone
    "collision_point_in": suradnice vzniku kolizie
    "collision_date_in": datum vzniku kolizie
}
```

Kód 6.6: Štruktúra hodnoty odosielanej do topiku pre triedu `PolygonEnterEvent`. Pre triedu `PolygonExitEvent` sa štruktúra líši v názve posledných dvoch atribútov. Atribút `event_type` bol pridaný pre jednoznačnejšie pochopenie, o aký typ udalosti sa jedná. Je možné ho odstrániť a pracovať iba s atribútom `in`.

Pre udalosti kolízií je dôležité, aby bola vždy do frontu správ pridaná naozaj udalosť iba jedenkrát, pretože duplicitné správy by následne pri ukladaní do databázy vytvárali nekonzistenciu v tom, v akých polygonoch sa zariadenie aktuálne nachádza. Z tohto dôvodu je pre tvorbu kľúča použitý hašovací algoritmus MD5, ktorý z rovnakej hodnoty správy vždy vytvorí rovnaký kľúč. Rovnaký kľúč vo fronte správ Apache Kafka avšak sám nezabezpečí neduplikovanie správ. Pre získanie tejto vlastnosti je nutné nastaviť vlastnosť `cleanup.policy` topiku `collisions` na hodnotu `compact` alebo `compact, delete`, čo zabezpečí, že pre rovnaký kľúč sa v topiku vždy ponechá iba posledná správa. Pre produkciu správ do topiku je využitý konektor od Apache Flink, už spomenutý `KafkaSink`, pričom hodnota je textová reprezentácia udalosti triedy `PolygonOutputEvent` a kľúč je haš tejto hodnoty.

Keďže pri tejto službe nie je využitý klasický Kafka konzument správ, ale je využitý konzument pre Flink, vlastnosti `enable.auto.commit` a `auto.offset.reset` sa správajú mierne inak. Offsety sú v aplikácii Flink zapisované automaticky pri pravidelne vykonávaných checkpointoch, ktoré Apache Flink robí. Vlastnosť `auto.offset.reset` sa pre triedu `KafkaSource` nastavuje funkciou `setStartingOffsets()` pri jej inštanciácii, pričom okrem možností `earliest` a `latest` je možné nastaviť počiatočný offset aj na iné hodnoty[17]. Pre túto službu bolo zvolené nastavenie:

```
OffsetsInitializer.committedOffsets(OffsetResetStrategy.EARLIEST)
```

Pri tomto nastavení konzument začína spracovávať správy od svojho posledného zapísaného offsetu. V prípade, že zapísaný offset neexistuje, tak je využitá stratégia spracovávania správ od najstaršej.

Apache Flink aplikácie môžu byť spustené na clustri Apache Flink alebo lokálne ako skompilovaný JAR súbor, obsahujúci knižnice Flink. Pre účely tejto práce bol zvolený druhý spôsob, keďže je táto možnosť jednoduchšia na nasadenie a tento systém slúži na ilustráciu konceptov a určitých vlastností, a preto sa nejedná o produkčnú verziu, kde by bola potrebná vyššia dostupnosť a vyššie nároky na paralelizáciu spracovania.

Štruktúru konfiguračného súboru pre túto službu je možné nájsť v prílohe B.3.

[17] https://nightlies.apache.org/flink/flink-docs-release-1.19/docs/connectors/datastream/kafka/#starting-offset

Collision Recorder

Zodpovednosť služby Collision Recorder je perzistentne zaznamenať udalosti vygenerované službou Collision Tracker do databázy. Podobne ako pri službe Location Recorder je implementovaný konzument správ, ktorý spracováva správy z topiku `collisions`. Na základe hodnoty atribútu `event_type` sa služba rozhodne, či ide o vytvorenie nového záznamu (`event_type=enter`) alebo o aktualizáciu existujúceho (`event_type=exit`) záznamu. Pôvodne bolo vyskúšané využiť databázu QuestDB aj na ukladanie kolízií, pre zjednodušenie celkového systému, a využitia iba troch databáz. Toto riešenie sa ale ukázalo ako neefektívne riešenie vzhľadom na vysoký počet operácií, ktoré vyžadujú aktualizáciu existujúcich záznamov, a QuestDB nie je pre takéto operácie vhodná, pretože pri návrhu tejto databázy sa predpokladalo, že dáta nebudú musieť byť často aktualizované, a preto využíva append-only zápis do súborov. Využitie dokumentovo-objektovej databázy MongoDB sa ukázalo ako vhodnejšie riešenie, ktoré bolo aj pôvodne navrhnuté v návrhu 5.6.

Pre prácu s databázou MongoDB bol využitý driver Reactive Streams, ktorý umožňuje asynchrónnu neblokujúcu komunikáciu s databázou, čím sa aplikácia stáva jednoduchšie škálovateľnou, zlepšuje sa jej výkonnosť a zväčšuje sa objem dát, ktorý dokáže spracovať. Pre využitie funkcií, ponúkaných touto knižnicou, je nutné implementovať novú triedu, ktorá implementuje rozhranie `Subscriber`[18]. Táto trieda konzumuje dáta získané od triedy `Publisher` a dokončuje operáciu zápisu. Trieda `ObservableSubscriber`, patriaca k službe Collision Recorder, implementuje rozhranie `Subscriber`. Pre využitie implementovanej triedy je nutné definovať typ operácie, ktorá bude vykonávaná, ako je vidieť na kóde 6.7. Pre vloženie nového záznamu (dokumentu) do databázy (konkrétne do kolekcie dokumentov) je potrebný objekt triedy `MongoCollection`, reprezentujúci kolekciu dokumentov, a pripravený dokument pre vloženie (trieda `Document`). Volaním samostatnej metódy triedy `MongoCollection` sa stav databázy nezmení. Takéto volanie má návratnú hodnotu triedy `Publisher`, nad ktorou je potrebné zavolať metódu `subscribe()`, ktorá očakáva ako parameter objekt triedy `ObservableSubscriber` s dátovým typom odpovedajúcim typu operácie (viď 6.7). Následné použitie týchto objektov v službe Collision Recorder a vzťah medzi nimi je možné vidieť na ukážke 6.8.

```
ObservableSubscriber<UpdateResult> subscriberUpdate
ObservableSubscriber<InsertOneResult> subscriberInsert
```

Kód 6.7: Pre aktualizáciu je vytvorená inštancia, pracujúca s dátovým typom `UpdateResult`, a pre vloženie nového záznamu je vytvorený objekt pracujúci s dátovým typom `InsertOneResult`.

Pri zvýšenom objeme požiadaviek na spracovanie nastával pri asynchrónnom spracovaní stav, kedy nebola databázová operácia vykonaná, pretože nastávalo preťaženie databázového serveru, preto bol čas čakania na databázové pripojenie (ang. *connection timeout*) zvýšený na 5 minút, čím bol tento problém odstránený. Ďalším možným riešením by bolo navýšenie počtu možných pripojení (tzv. connection pool) alebo distribúcia databázového systému na viac uzlov (tzv. sharding).

Pre databázu sú vytvorené indexy pre urýchlenie čítania a aktualizácií záznamov. Index je vytvorený pre zariadenie a okrem toho je vytvorený zložený index (ang. *compound index*)

[18] https://mongodb.github.io/mongo-java-driver/5.0/driver-reactive/getting-started/

```
collection.insertOne(document).subscribe(subscriberInsert);
collection.updateOne(filter, update).subscribe(subscriberUpdate);
```

Kód 6.8: Objekt `collection` je inštancia triedy `MongoCollection` a poskytuje rôzne operácie. V tejto ukážke je vidieť operácie vkladania a aktualizácie jedného dokumentu. Objekty `document`, `update` a `filter` sú inštanciami triedy `Document`, pričom dokument `filter` špecifikuje kritéria pre vyhľadanie už existujúceho dokumentu, ktorý má byť aktualizovaný podľa štruktúry, ktoré definuje dokument `update`. Nakoniec je pripojený objekt triedy `ObservableSubscriber`, ktorý umožní dokončenie operácie zápisu.

pre zariadenie a polygon práve kvôli aktualizáciám, ktoré zahŕňajú vyhľadanie príslušného dokumentu, ktorý má byť aktualizovaný.

6.5 API služby

Komunikácia medzi systémom a klientom prebieha cez API gateway, ktoré zabezpečuje jednotný prístupový bod pre API[19]. Všetky API služby využívajú protokol HTTP. Ako bolo už spomenuté v časti, venujúcej sa technológiám 6.1, API gateway je realizované cez webový server NGINX a je využívané ukladanie odpovedí na požiadavky do pamäte cache. Pre službu Realtime Locations je nastavená platnosť pamäte cache na 30 sekúnd z dôvodu, že pri tejto službe je dôležitá aktuálnosť dát. Pre historické polohy a polygony je doba platnosti pamäte cache 10 minút, keďže sa neočakávajú žiadne zásadné zmeny pri týchto požiadavkách. Pre kolízie je zvolená doba uchovávania odpovedí 1 minúta, keďže je dôležitá určitá aktuálnosť dát. Pre WebSocket sa zo zásady pamäť cache nevyužíva. Celková konfigurácia serveru NGINX sa nachádza v súbore `nginx.conf`.

Aj keď API gateway ponúka jednotné rozhranie z pohľadu klienta, v pozadí obsluhujú jednotlivé požiadavky celkovo tri REST služby (popísané v 5.7, parametre a formát dát, ktorý služby vracajú sa nachádza v prílohe C). Tri služby boli zvolené pre zníženie záťaže na jednotlivé služby, ale aj kvôli používaniu rôznych databáz pre rôzne typy pohľadov na dáta. Zároveň bolo týmto myslené na budúce rozšírenia jednotlivých služieb. Všetky tri služby pracujú s mapovaním user id – device id, pretože interne sú všetky dáta uložené v databázach s informáciou id zariadenia, zatiaľ čo sa klienti pýtajú na konkrétnych užívateľov a nie na zariadenia. Keďže by sa pri každej obsluhe požiadavky museli služby pripájať do relačnej databázy Postgres a získať bijektívne zobrazenie user id – device id, bola využitá vyrovnávacia pamäť Redis (vyrovnávacia pamäť je db=1; pre realtime DB ide o db=0), kde sa tieto vzťahy uchovávajú. Časová invalidácia tejto pamäte cache nebola riešená, ale jej životný cyklus je riadený cez metódu FastAPI `lifespan`. Implementácia takejto funkcie umožňuje pri štarte FastAPI služieb vykonať tzv. startup, v tomto prípade vykonať inicializáciu pamäte cache na základe získania dát z relačnej databázy. Pri ukončení služby umožňuje tzv. teardown, v tomto prípade ide o vyprázdnenie danej databázy v službe Redis. Mapovanie užívateľa na zariadenie potom prebieha v hlavnej myšlienke nasledovne:

1. Z pamäte cache sa získa zoznam obsahujúci dvojice (user_id, device_id)
2. Vytvorí sa mapovanie v podobe slovníku (`dict`), kde je kľúčom device_id a hodnotou je user_id.

[19] v rámci tejto práce je to adresa: `localhost:8088`; port špecifikovaný cez docker-compose

3. Ak požiadavka obsahuje špecifikáciu týkajúcu sa konkrétnych užívateľov, tak sa z výsledku z kroku 1. získa zoznam s device_id odpovedajúci zadaným užívateľom.

4. Získajú sa záznamy z databázy obsahujúce device_id.

5. Využije sa vytvorený slovník z kroku 2. pre nahradenie device_id vo výsledku z kroku 4. Prípadne sa v odpovedi ponechá aj device_id a pridá sa user_id.

6.6 Nasadenie docker kontajnerov

Vzhľadom na zložitosť celkového systému a početné množstvo rôznych služieb bol pre nasadenie zvolený prístup kontajnerizácie pomocou docker kontajnerov a dostupného nástroju docker-compose. Súbor `docker-compose.yaml` obsahuje celkovo 18 služieb, ktoré sú v skratke popísané nižšie. Všetky služby sú prepojené internou sieťou `internal_network`, pre umožnenie komunikácie medzi službami. Databázy a služba `nginx` majú vytvorené `volumes` pre perzistenciu dát aj po vypnutí služieb. Prístupové body k jednotlivým službám sú opísané v časti príloh D.

Popis služieb:

- **questdb:** Databáza časových radov pre historické dáta. Poskytuje webové rozhranie na adrese `localhost:9000`.
- **mongodb:** Dokumentová databáza pre kolízie.
- **redis:** Databáza typu kľúč–hodnota pre ukladanie aktuálnych polôh. Použitá aj ako pamäť cache.
- **postgres:** Relačná databáza pre statické dáta. Pri spustení je inicializovaná skriptom `docker/init-postgres.sh` z obrazu predpripravenej databázy, nachádzajúcej sa v súbore `database.sql`.
- **kafka-ui:** UI rozhranie pre monitorovanie Apache Kafka. Pomocou `environment` atribútov sú špecifikované parametre, týkajúce sa Kafka clustra. Pri škálovaní je dôležitá hlavne premenná `KAFKA_CLUSTERS_0_BOOTSTRAPSERVERS`, ktorá by mala vždy obsahovať adresy všetkých kafka brokerov. Webové rozhranie je prístupné na adrese `localhost:8080/ui`.
- **zookeeper:** Centralizovaná služba pre správu stavu clustra Apache Kafka.
- **kafka1:** Apache Kafka broker. Atribút `KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR` určuje replikáciu jednotlivých topikov. Je nastavený cez docker compose `environment`.
- **kafka-init:** Pomocná služba pre automatické nastavenie vlastností topikov, ako je nastavenie počtu partitions a nastavenie `cleanup.policy` pri topiku `collisions`. Využíva skript `docker/kafka_init.sh`
- **connector-init:** Pomocná služba pre automatické vytvorenie QuestDB konektoru pre Apache Kafka topik `new_locations`. Kontajner je vytvorený cez Dockerfile, kde je využitý pripravený skript `create_kafka_questdb_connect.py`.

- **connect:** Služba pre connector QuestDB–Kafka. Opäť je pri škálovaní dôležité pridať ďalších brokerov do `environment` atribútu `CONNECT_BOOTSTRAP_SERVERS`.
- **locationrecorder:** Služba Location Recorder 6.4.
- **collisionrecorder:** Služba Collision Recorder 6.4.
- **collisiontracker:** Služba Collision Tracker 6.4.
- **producer:** Služba Producer 6.3.
- **api_collisions:** Služba API pre kolízie. Z vonkajšieho prostredia nie je táto ani ďalšie dve API služby dostupné a sú prístupné iba v rámci docker siete `internal_network`.
- **api_historical:** Služba API pre historické polohy.
- **api_realtime:** Služba API pre aktuálne polohy.
- **nginx:** API gateway využíva konfigurácia v súbore `nginx.conf`. Dostupná z lokálneho počítača na adrese `localhost:8088`.

Kapitola 7

Vyhodnotenie a testovanie

Táto kapitola sa venuje testovaniu a zhodnoteniu implementovaného systému. Testovanie bolo rozdelené do 3 častí. Prvá časť 7.1 sa sústredí na testovanie produkcie vstupných dát. Druhá časť testovania 7.2 je zameraná na výkonnosť spracovania dát jednotlivými konzumentmi správ z Apache Kafka. Posledná časť 7.3 sa zaoberá testovaním latencie obsluhy požiadaviek od klientov. Vyhodnotenie prebiehalo na školskom serveri `grace2.fit.vutbr.cz` so špecifikáciou:

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

7.1 Latencia produkcie vstupných dát

Táto podkapitola sa zaoberá testovaním rýchlosti generovania 1 milióna správ cez generátor 6.2 a následným trvaním prenosu týchto správ cez WebSocket protokol a ich uložením do frontu správ službou Producer 6.3. Pre účely tohto testovania neboli potrebné služby, týkajúce sa spracovania správ, t.j. Location Recorder, Collision Tracker, Collision Recorder. Zároveň nebol spustený ani konektor do databázy QuestDB. Vygenerované dáta prechádzali do systému cez API gateway. Produkcia vstupných dát bola testovaná pre nasledujúce prípady:

- 1 producent - 1 generátor
- 1 producent - 2 generátory
- 1 producent - 4 generátory
- 2 producenti - 2 generátory
- 2 producenti - 4 generátory

Počet vygenerovaných správ pomocou generátorov bol vždy v súhrne 1 milión správ (t.j. v prípade 2 generátorov generoval každý 500 tisíc správ; pre 4 generátory generoval každý 250 tisíc správ) a generovanie 1 milióna správ bolo opakované celkovo trikrát. To znamená, že na konci testovania sa v topiku Apache Kafka nachádzalo celkovo 3 milióny správ. Na získanie meraných hodnôt sa použil mechanizmus logovania metrík z Kafka producenta pridaním atribútu `stats_cb` do konfigurácie, ktorý odkazuje na callback funkciu, ktorá vypisuje potrebné hodnoty[1].

[1] https://github.com/confluentinc/librdkafka/blob/master/STATISTICS.md

Samotný generátor dokázal vygenerovať 1 milión správ bez prenosu cez WebSocket protokol v priemernej dobe 45 sekúnd.

Výsledky generovania a odoslania správ do frontu správ

Pri spustení viacerých generátorov sa doba celkového spracovania správ výrazne skrátila. Ako je vidieť v tabuľke 7.1, je to spôsobené hlavne spustením paralelne pracujúcich generátorov. Hodnoty v tabuľke naznačujú, že s nárastom počtu generátorov sa doba generovania správ priamo úmerným spôsobom skracuje. Aj keď celkový čas pri viacerých generátoroch zodpovedá času, za ktorý jeden generátor vygeneruje rovnaký počet správ, celková rýchlosť generovania sa skrátila o 50 % pri 2 generátoroch a približne o 66 % pri 4 generátoroch, pretože viac generátorov generovalo správy paralelne. Na grafoch 7.1, 7.2, E.1 je zreteľné, že správy sú do topiku zapisované takmer okamžite v čase, v ktorom sú aj produkované, a počet správ, ktoré čakajú na odoslanie vo fronte producenta[2], sa stáva rovnomernejším z nárastom počtu generátorov.

	1 generátor			**2 generátory**			**4 generátory**		
iterácia:	*1.*	*2.*	*3.*	*1.*	*2.*	*3.*	*1.*	*2.*	*3.*
1. generátor	291	297	296	155	148	151	99	99	99
2. generátor	X	X	X	155	148	151	99	99	99
3. generátor	X	X	X	X	X	X	99	98	99
4. generátor	X	X	X	X	X	X	99	100	99

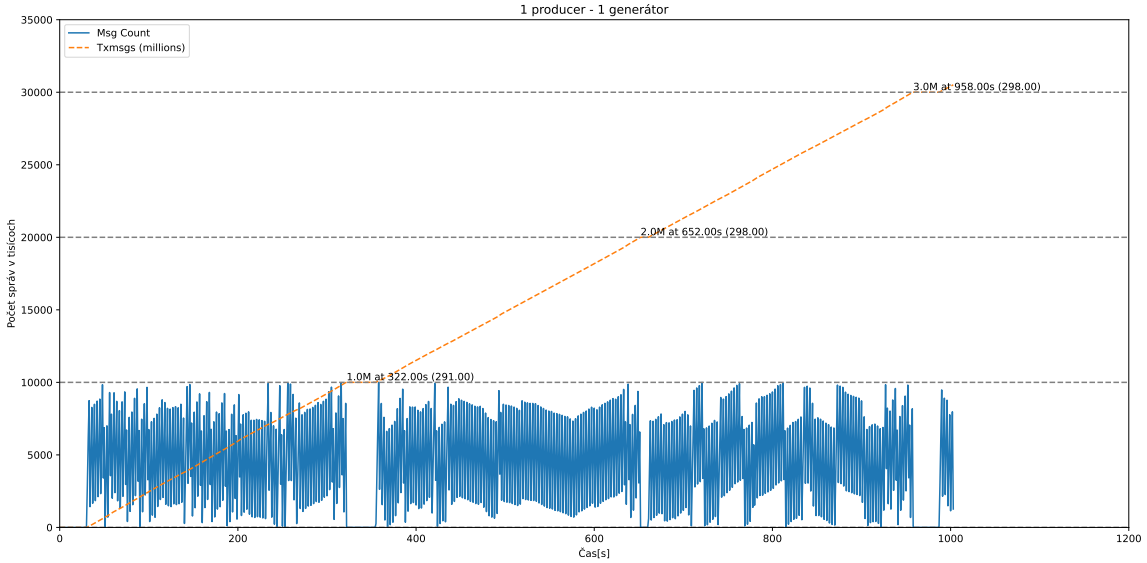
Tabuľka 7.1: Čas trvania vygenerovania správ generátormi pri jednej inštancii služby Producer.

	2 generátory			**4 generátory**		
iterácia	*1.*	*2.*	*3.*	*1.*	*2.*	*3.*
1. generátor	143	146	148	78	75	81
2. generátor	142	145	148	77	75	80
3. generátor	X	X	X	78	75	81
4. generátor	X	X	X	77	75	80

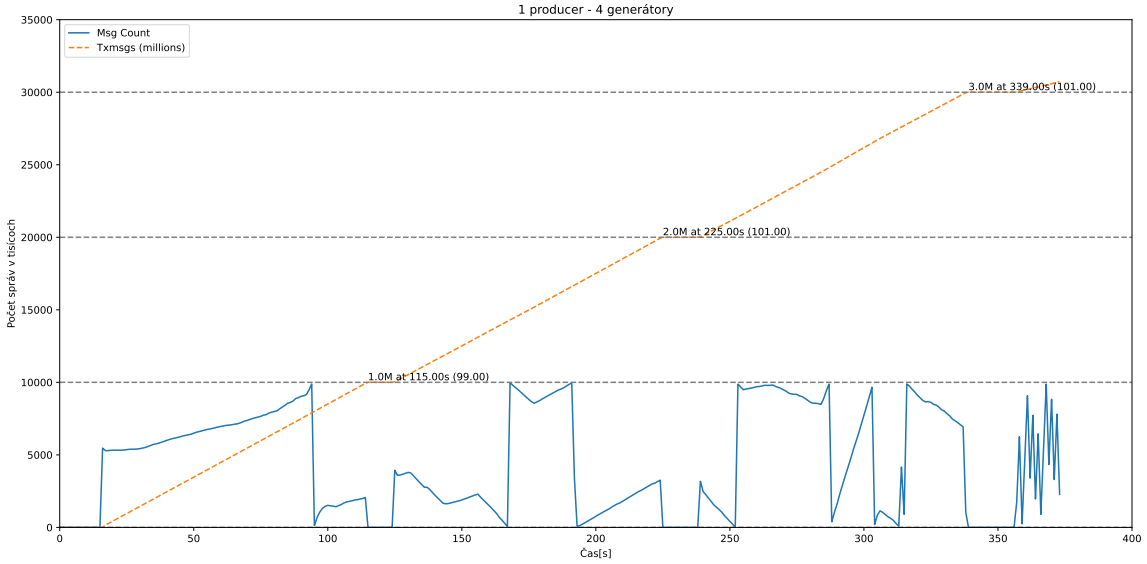
Tabuľka 7.2: Čas trvania vygenerovania správ generátormi pri dvoch inštanciách služby Producer.

Pri použití dvoch inštancií služby Producer bolo vynechané testovanie s jedným generátorom, pretože vzhľadom na duplexnú povahu spojenia WebSocket by tento generátor aj tak obsluhovala iba jedna službu. Po pridaní ďalšieho producenta bola záťaž rovnomerne rozdelená pomocou API gateway, ale každý generátor zostal už naviazaný na producenta, s ktorým nadviazal prvotné spojenie. Každá inštancia služby Producer mala teda pridelených vždy 50 % generátorov, od ktorých prijímala žiadosti. Pri porovnaní tabuliek 7.1 a 7.2 je vidieť, že rýchlosť generovania celkového počtu správ sa znížila o niekoľko sekúnd. Na grafoch E.2 a 7.3 je zretelné, že obaja producenti produkovali správy do topiku v rovnakom čase bez výrazných odchýlok a taktiež, že správy, čakajúce na odoslanie, boli rovnomerne rozdelené. Pri 4 generátoroch je opäť vidieť znižujúca sa intenzita oscilovania, ktorá značí, že správy vo fronte čakajúcich správ na odoslanie sa zaplnili pre daný limit skôr.

[2] v grafoch označené ako msg count



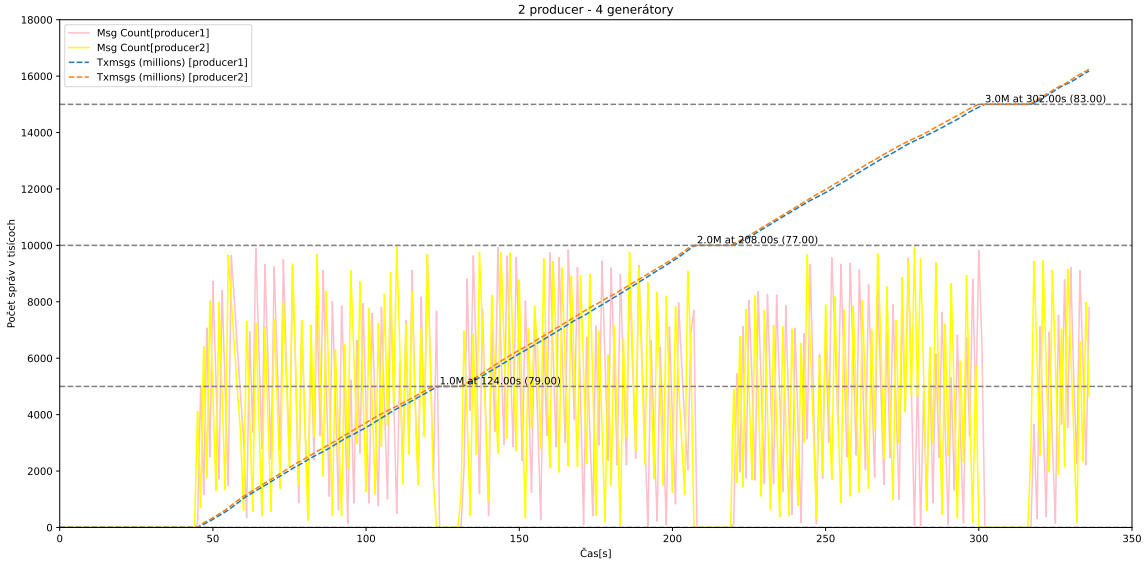
Obr. 7.1: Celkový počet odoslaných správ do Apache Kafka (txmsgs) a počet správ, čakajúcich na odoslanie vo fronte producenta (msg count) pre 1 inštanciu producenta a 1 inštanciu generátora. Milión správ je odoslaný do topiku v rozmedzí 291–298 sekúnd.



Obr. 7.2: Celkový počet odoslaných správ do Apache Kafka (txmsgs) a počet správ, čakajúcich na odoslanie vo fronte producenta(msg count) pre 1 inštanciu producenta a 4 inštancie generátora. Milión správ je odoslaný do topiku v rozmedzí 99–101 sekúnd, pričom je vidieť, že oproti predchádzajúcemu grafu front čakajúcich správ osciluje menej, čo znamená, že býva rovnomernejšie zaplnený.

Záver

Loadbalancer v rámci API Gateway rovnomerne rozdeľuje nové pripojenia na WebSocket server. Pri pridaní paralelne pracujúcich generátorov sa obsluha požiadaviek urýchľuje, pretože pracovná záťaž je rozdelená medzi viac generátorov. Služba Producer neprodukuje



Obr. 7.3: Celkový počet odoslaných správ do Apache Kafka (txmsgs) a počet správ, čakajúcich na odoslanie vo fronte producenta (msg count) pre 2 inštancie producenta a 4 inštancie generátora. Milión správ je odoslaný do topiku v rozmedzí 77–83 sekúnd, pričom je vidieť, že fronty čakajúcich správ oboch producentov oscilujú podobne a taktiež celkový počet odoslaných správ je takmer rovnaký pre oboch producentov.

výrazné odchýlky pri odosielaní správ do Apache Kafka, ale vzhľadom na nastavený flush interval, ktorý je nastavený na 10 tisíc správ, pri nižšom počte generátorov dochádza k väčším osciláciám vo fronte čakajúcich správ na odoslanie. Zaplnenie frontu čakajúcich správ nemá zásadný vplyv na celkovú dobu spracovania, keďže je vidieť, že celkový počet odoslaných správ rastie lineárne vo všetkých grafoch, a celková doba spracovania zodpovedá hodnotám vygenerovaných správ samotnými generátormi uvedenými v tabuľkách 7.1, 7.2.

7.2 Testovanie spracovania dát

Pre účely testovania a zabezpečenia konštantných podmienok pre testy bolo generátorom opísaným v časti 6.2 vygenerovaných 1 milión správ, ktoré boli zapísané do súboru. Vygenerovanie tohto počtu správ trvalo 45 sekúnd.

Testovanie rýchlosti spracovania dát prebiehalo prostredníctvom upravenej služby Producer, ktorá postupne odosiela predpripravené dáta zo súboru `test_data.gz` do Apache Kafka bez použitia WebSocket spojenia pre zaistenie rýchlejšieho naplnenia frontu správ. Pri testovaní boli nasadené 2 brokery služby Apache Kafka pre zvýšenie dostupnosti a zníženie rizika výpadku frontu správ. Spracovanie dát jednotlivými konzumentami z topiku `new_locations` (t.j. LocationRecorder, CollisionRecorder a konzument pre QuestDB konektor) bolo logované každé 3 sekundy do logového súboru zavolaním príkazu:

```
kafka-consumer-groups -bootstrap-server $BOOTSTRAP_SERVER -describe -all-groups
```

Počas prvotného spustenia pre jednu partition a jednu repliku bolo zistené, že najväčší bottleneck (prekl. úzke hrdlo) je služba CollisionTracker, testovanie sa ďalej sústredilo práve na túto službu. Celkovo bolo otestované spracovanie pri využití 1, 2, 4 a 8 replík a nastavenie počtu partition na topiku bolo testované s hodnotami 8, 16, 32, 64 a 128. Nastavenie po-

čtu partition na topikoch je dôležitým faktorom, pretože určuje hornú hranicu paralelného spracovania.

Pri testovaní bolo zistené, že je nutné nakonfigurovať zapisovanie kontrolných bodov (ang. *checkpoints*) do súboru pomocou `FileSystemCheckpointStorage`, keďže pri využití predvolenej konfigurácie `JobManagerCheckpointStorage` dochádzalo pri použití stavového backendu `HashMapStateBackend` k problému, kedy aplikácia Flink po určitom čase prestala zapisovať offsety spracovaných správ. Tento problém spôsoboval, že konzument Collision Tracker spracovával opakovane správy iba od začiatku logu frontu správ. Iné riešenie je využitie stavového backendu `EmbeddedRocksDBStateBackend`, ale toto riešenie môže spomaľovať celkovú dobu spracovania na úkor škálovateľnosti [1]. Pri využití RocksDB ako backendu závisí spracovanie na nastavení parametru `managed_memory_size`, čo dokázali aj výsledky zobrazené na grafoch 7.5 a 7.6.

Spracovanie CollisionTracker - HashMapStateBackend

Na grafe 7.4 je vidieť, že pre niektoré konfigurácie bolo zaplnenie a následný začiatok spracovania správ oneskorené. Najvýraznejšie sa to prejavilo pri spracovaní jednou replikou pre 128 partitions, zároveň ale táto konfigurácia patrí medzi tie, ktoré najrýchlejšie spracovávajú správy. Ďalej je na grafe možné pozorovať, ako sa so zvyšujúcim počtom replík zväčšujú skoky spracovania správ. Zatiaľ čo pri menšom počte partitions sú správy spracovávané pomerne konštantne, pri vyššom počte partitions dochádza k značným skokom v spracovaní, čo môže byť spôsobené dlhším trvaním registrácie offsetov spracovaných správ pre vyšší počet partitions. Pri spracovaní 8 replikami je vidieť pre počet partitions 64 a 32 určitá oscilácia, čo bolo pravdepodobne zapríčinené zlyhaním niektorej z replík alebo oneskorenými informáciami naprieč replikami. Väčšina testovaných konfigurácií spracováva správy pomerne podobne rýchlo. Pri prechádzaní logových súborov bolo zistené, že pri vyššom počte partitions občas dochádza k oneskoreniu registrácie spracovaných správ.

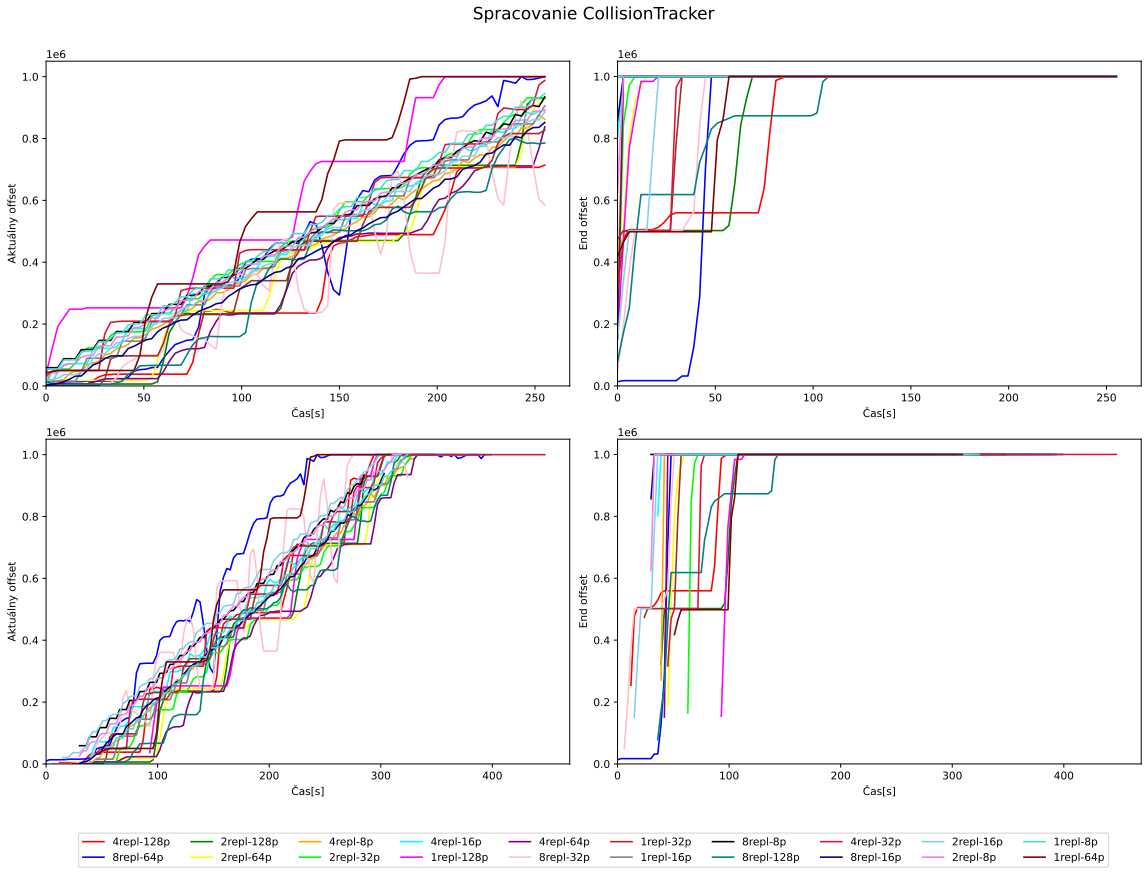
Ako je vidieť v tabuľke 7.3, po jednej minúte spracovávania mala najviac spracovaných správ jedna replika s konfiguráciou topiku pre 8, 64 a 128 partitions. Najmenej spracovaných správ mali štyri repliky s 64 a 128 partitions pre topik.

Z tabuľky 7.4 je zrejmé, že po 4 minútach spracovala najviac záznamov konfigurácia s jednou replikou a 64 a 128 partitions. Tretie miesto v spracovaní má konfigurácia s 8 replikami a 64 partitions. Najpomalšie správy spracovávali 4 repliky s 128 partitions, ale podobne pomalé výsledky mali aj 4 repliky s 64 partitions alebo 2 repliky s 64 a 128 partitions.

Zaujímavé je taktiež spracovanie správ 8 replikami pre 64 a 128 partitions, pri ktorých spracovanie po 1 minúte patrilo k jedným z najnižších, ale po 4 minútach bol offset spracovaných správ značne dobehnutý.

repliky\partitions	**8**	**16**	**32**	**64**	**128**
1	258 820	207 322	137 303	329 881	252 344
2	242 147	218 221	234 187	152 372	79 348
4	198 931	237 016	209 285	45 946	38 070
8	237 722	187 873	114 808	89 944	66 921

Tabuľka 7.3: Počet spracovaných záznamov službou CollisionTracker po 1 minúte.



Obr. 7.4: Spracovanie záznamov službou CollisionTracker pre nastavenie backendu `HashMapStateBackend` a zapisovaní checkpointov do súboru. V hornej časti sa nachádza spracovanie od momentu, odkedy Apache Kafka zaznamenala prvýkrát túto službu vo svojich konzumentoch. V dolnej časti sa nachádza reálny čas spracovania, t.j. od spustenia docker-compose. V ľavom stĺpci sa nachádzajú posledné zaregistrované offsety (spracované správy). V pravom stĺpci sa nachádzajú koncové offsety (celkový počet správ v topiku).

repliky\partitions	**8**	**16**	**32**	**64**	**128**
1	901 057	847 238	816 014	1 000 000	1 000 000
2	858 131	842 196	878 067	708 859	713 346
4	803 452	833 418	893 563	711 420	706 746
8	877 914	806 008	870 264	978 680	799 853

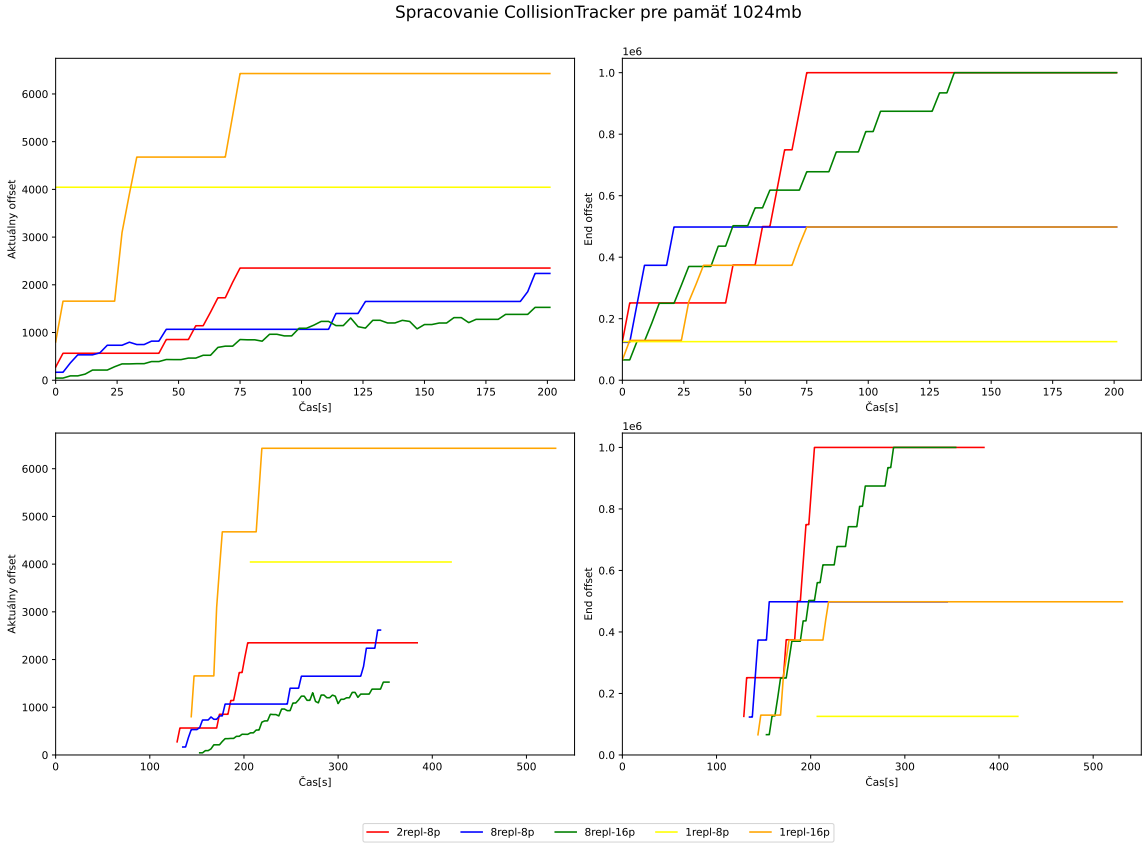
Tabuľka 7.4: Počet spracovaných záznamov službou CollisionTracker po 4 minútach.

Spracovanie CollisionTracker - EmbeddedRocksDBStateBackend

Pre stavový backend `EmbeddedRocksDBStateBackend` bolo testované spracovanie správ pri nastavení parametru `managed_memory_size` na hodnoty: 1024mb, 2048mb, 4096mb, 8192mb a 16g a boli využité inkrementálne checkpointy. Pri niektorých konfiguráciách sa ukázalo, že služba Collision Tracker nebola schopná správy ani začať spracovávať. Najhoršie výsledky mal konzument pri nastavení `managed_memory_size` na hodnotu 1024mb, čo je možné pozorovať aj na grafe 7.5, kde po 200s bol najvyšší spracovaný offset 6428. Pre túto

konfiguráciu bolo testované spracovanie 1,2,4 a 8 replikami a 8, 16 a 32 partitions, ale hodnoty produkovali iba konfigurácie vykreslené na grafe. Ostatné logové súbory túto skupinu konzumentov ani nezaregistrovali.

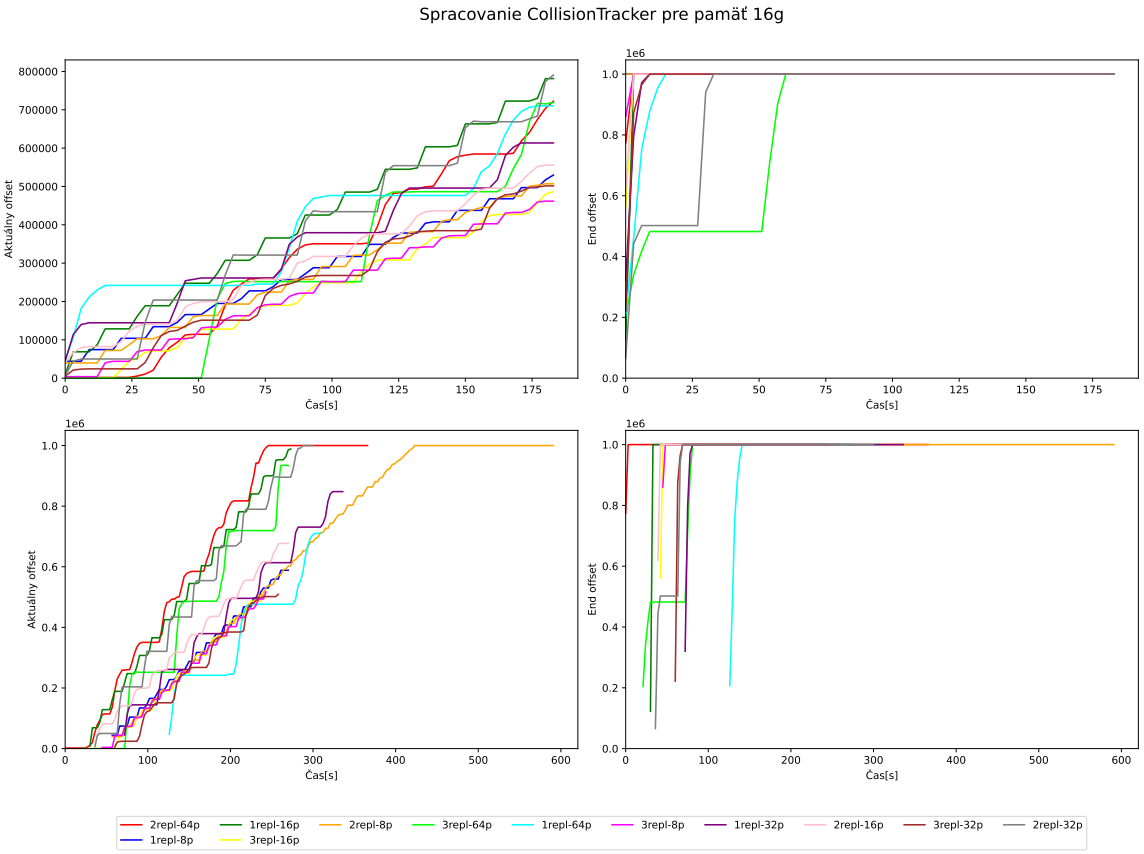
Na grafe je zároveň možné pozorovať problém zaseknutia offsetov (grafy začali produkovať konštatné hodnoty), ktoré boli opísané v tejto časti v úvode testovania. Na grafe je tiež možné pozorovať tvorenie schodovitého tvaru pri zmenách počtu partitions, opísanom aj v predchádzajúcej časti.



Obr. 7.5: Spracovanie správ službou CollisionTracker pre nastavenie backendu `EmbeddedRocksDBStateBackend` a nastavením memory size 1024MB. Význam jednotlivých grafov je rovnaký ako pri 7.4.

Najlepšie výsledky boli dosiahnuté pri 16GB hodnote (graf 7.6) `managed_memory_size`, kedy bol pri použití 2 replík a 64 partitions spracovaný milión správ za 246 sekúnd. Pre túto konfiguráciu bolo testované spracovanie 1, 2 a 3 replikami a 8, 16, 32 a 64 partitions. Počet replík bol zvolený, berúc do úvahy veľkosť pamäte RAM testovaného stroja.

Grafy pre ostatné testované hodnoty nastavenia pamäte je možné nájsť v časti príloh E. V tabuľke 7.5 je jednoznačne vidieť, že najviac správ za minútu pre túto konfiguráciu dokázala spracovať 1 replika s 16 partitions, ale z grafu 7.6 je vidieť, že konfigurácia pre 2 repliky a 64 partitions ju o pár sekúnd nakoniec predbehla.



Obr. 7.6: Spracovanie správ službou CollisionTracker pre nastavenie backendu `EmbeddedRocksDBStateBackend` a nastavením memory size 16GB. Význam jednotlivých grafov je rovnaký ako pri 7.4.

repliky\partitions	**8**	**16**	**32**	**64**
1	194 941	307 376	261 264	241 910
2	193 720	199 386	269 196	189 883
3	153 200	127 928	151 190	246 920

Tabuľka 7.5: Počet spracovaných záznamov po 1 minúte s využitím `EmbeddedRocksDBStateBackend` a nastavením memory size 16GB.

Spracovanie CollisionTracker v cloud computing

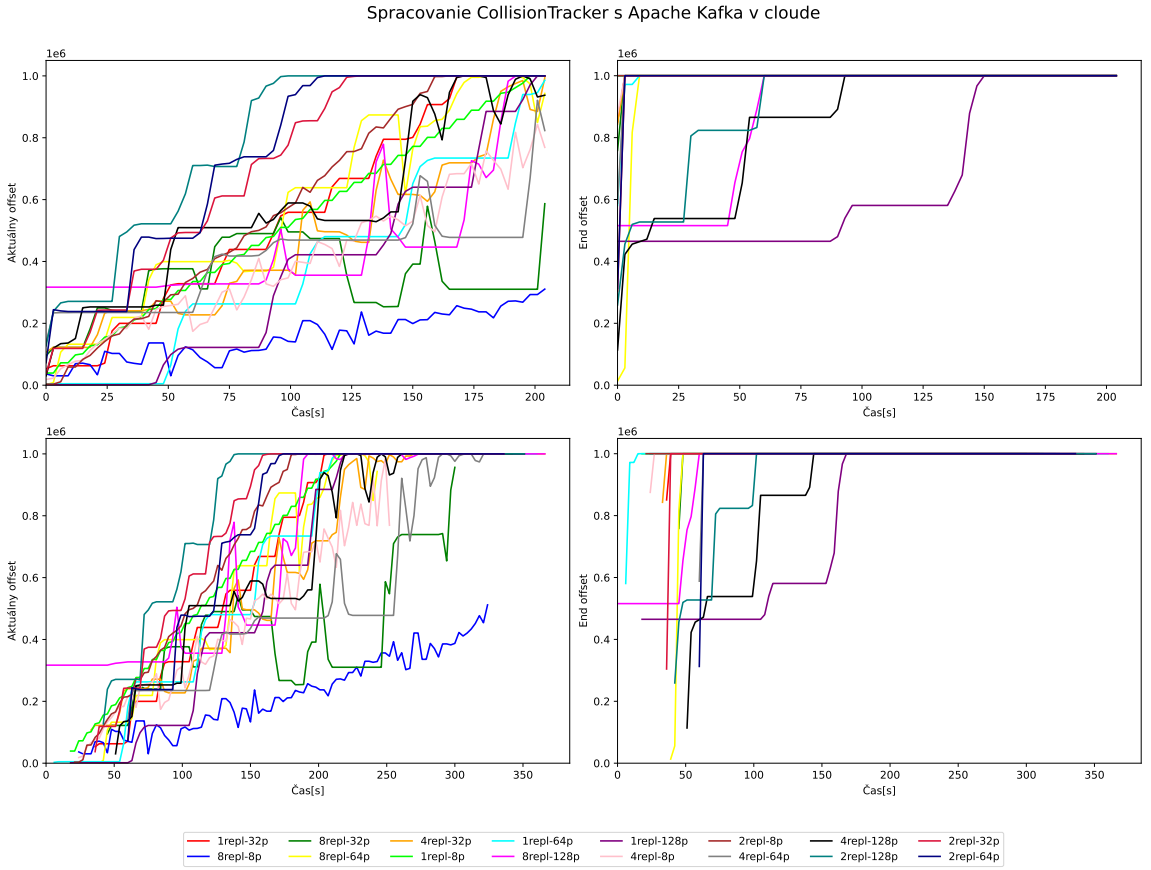
Bolo vykonané aj testovanie pre službu CollisionTracker s použitím nastavenia backendu `HashMapStateBackend`, a využitím služby Apache Kafka v cloud computing prostredí. Postup nasadenia sa nachádza v časti príloh A.1. Boli testované konfigurácie s počtom replík: 1, 2, 4, 8 a počtom partitions: 8, 32, 64, 128.

Ako je značné z tabuľky 7.7, najlepšie výsledky mali konfigurácie využívajúce 2 repliky. Najrýchlejšie spracovala milión záznamov konfigurácia s 2 replikami a 128 partitions a to za čas 99 s. Ako je vidieť na grafe 7.7, najhoršie výsledky mala konfigurácia pre 8 replík s 8 partitions, ktorá značne zaostávala, a preto jej hodnota nie je uvedená v tabuľke 7.7. Konfigurácia pre 8 replík a 32 partitions taktiež nie je uvedená v tabuľke 7.7, pretože v rámci testovania nestihla dosiahnuť tento limit, avšak dosiahla spracovania 956 455 správ za 255

s. Konfigurácia pre 4 repliky a 8 partitions značne oscilovala a priblížila sa k spracovaniu 983 905 záznamov v čase 225 s, ale pri ďalšej hodnote sa počet spracovaných správ zmenil na 769 297.

Pri porovnaní tabuliek výsledkov spracovania po 1 minúte pri lokálnom nasadení Apache Kafka 7.3 a pri využití Apache Kafka v prostredí cloud computing 7.6, je viditeľné značné zrýchlenie spracovania záznamov pre niektoré konfigurácie, avšak zrýchlenie sa prejavilo hlavne na tom, že väčšina konfigurácií zvládla do 4 minút spracovať celý milión správ (viď 7.7), čo pri lokálnom nasadení Apache Kafka zvládli iba konfigurácie s jednou replikou s 64 a 128 partitions (7.4).

Na grafe 7.7 je možné pozorovať, že konfigurácie pre 4 a 8 replík mali značné výkyvy v počte spracovaných správ.



Obr. 7.7: Spracovanie správ službou CollisionTracker s využitím `HashMapStateBackend` a frontom správ Apache Kafka nasadením v cloud computing prostredí.

Zároveň bolo otestované aj spracovanie 1 milióna záznamov pri nasadení služby CollisionTracker na Flink clustri v cloudovom prostredí (manuál pre nasadenie je možné nájsť v časti príloh A.1). Testovanie prebehlo na VM `e2-highmen-16` so špecifikáciou 16 vCPU, 8 jadier, 128GB pamäťou. Lepšiu špecifikáciu nebolo možné zvoliť vzhľadom na obmedzenia free účtu na GCP.

Bolo zistené, že rýchlosť spracovania nezávisí od počtu partition topiku v Apache Kafka, pretože pre rôzny počet partition, spracovávala služba CollisionTracker 1 milión záznamov

repliky\partitions	**8**	**32**	**64**	**128**
1	335 384	328 108	262 809	122 136
2	345 049	493 700	475 166	710 209
4	174 105	227 376	235 556	509 341
8	114 459	376 466	399 452	327 504

Tabuľka 7.6: Počet spracovaných záznamov službou CollisionTracker po 1 minúte pri využití Apache Kafka v cloudovom prostredí.

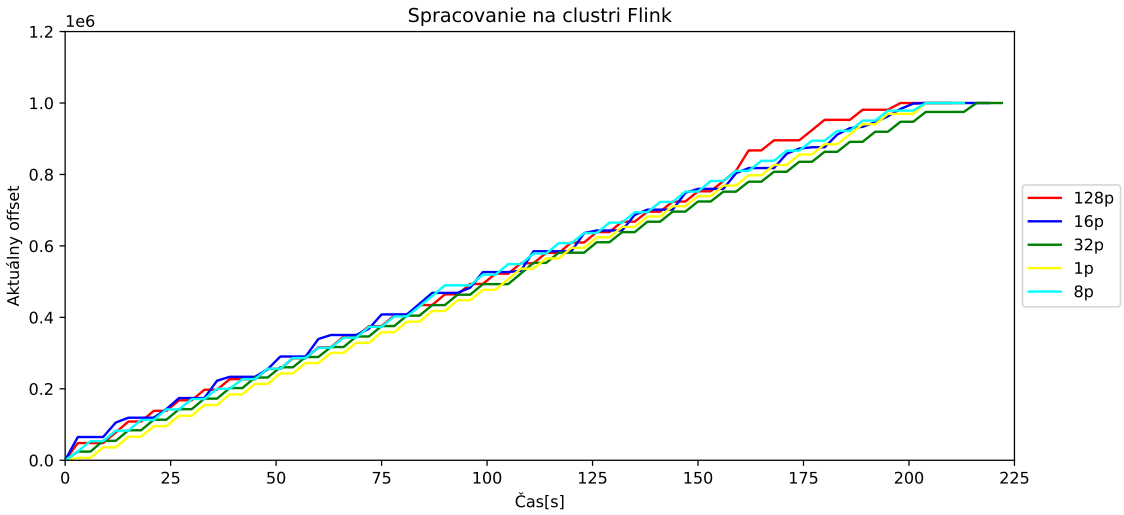
repliky\partitions	**8**	**32**	**64**	**128**
1	198 s	171 s	207 s	201 s
2	165 s	129 s	114 s	99 s
4	X	237 s	231s	171s
8	X	X	180 s	204 s

Tabuľka 7.7: Čas spracovania 1 milióna záznamov službou Collision Tracker pri využití Apache Kafka v cloud computing prostredí.

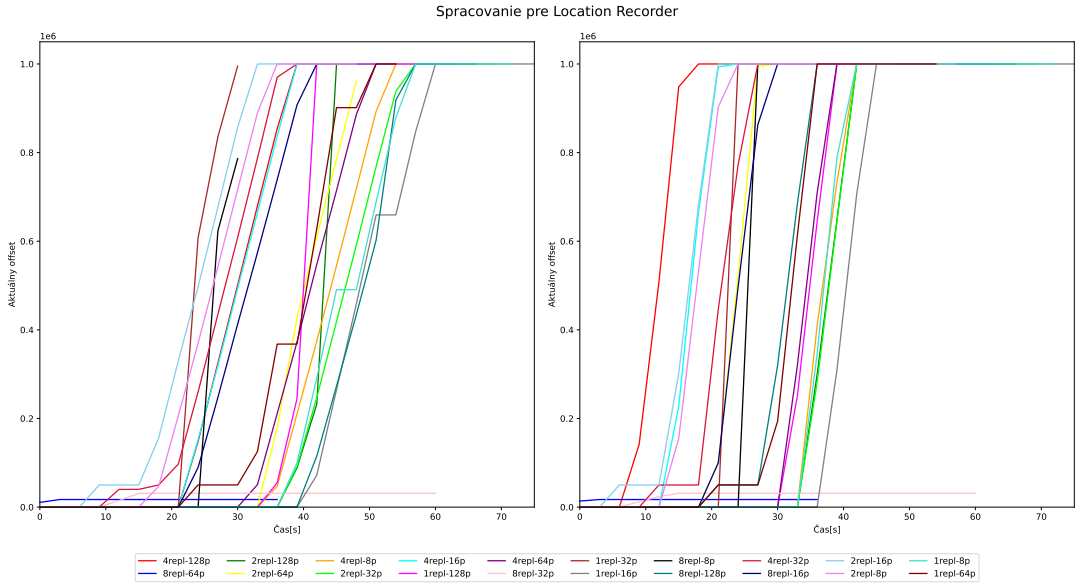
na clustri Flink za približne rovnakú dobu 200 sekúnd. Ako je vidieť na grafe 7.8 spracovávanie malo lineárny priebeh a prebehlo bez výkyvov.

Spracovanie konzumentov Connect QuestDB a Location Recorder pri rôznych zaťaženiach konzumenta Collision Tracker

Ako už bolo spomenuté, ostatní konzumenti neboli testovaní s rôznymi konfiguráciami, pretože ako je zrejmé z grafov 7.9 a 7.10, títo konzumenti boli schopní spracovať daný objem správ do jednej minúty. Na týchto grafoch je možné tiež pozorovať, ako sa títo konzumenti správali pri rôznych konfiguráciách služby Collision Tracker pri využití nastavenia `HashMapStateBackend`. Služba Location Recorder nezvládala správy spracovávať na úkor 8 replík služby Collision Tracker (8repl-64p, 8repl-32p, 8repl-8p). Služba Location Recorder aj konektor Connect QuestDB, ale spracovávajú správy lineárne bez významných skokov. Konzument Location Recorder spracoval milión správ väčšinou v rozsahu 6-30 sekúnd, avšak pri 8 replikách CollisionTracker mala táto služba problém. Pri Connect QuestDB je priemerná doba spracovania 3 sekundy.



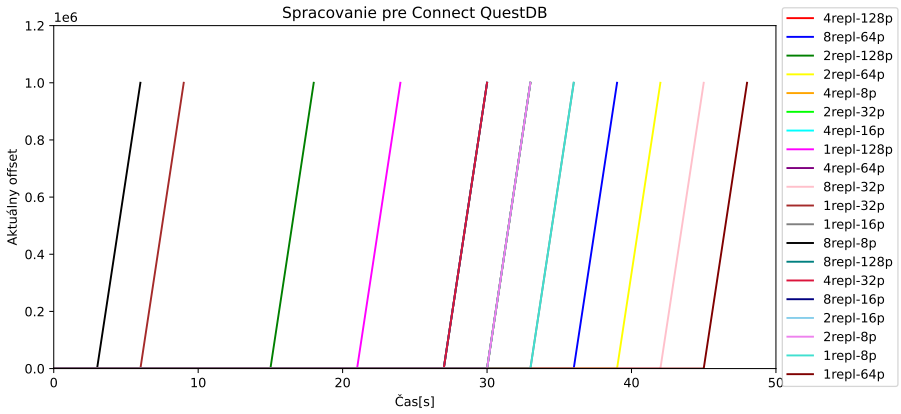
Obr. 7.8: Spracovanie 1 milióna záznamov službou CollisionTracker na flink clustri na VM v cloud computing.



Obr. 7.9: Spracovanie správ službou Location Recorder pri rôznych konfiguráciách služby Collision Tracker.

Zhrnutie

Celkovo sa zdá, že konfigurácie s využitím backendu `HashMapStateBackend` s jednou replikou dosahovali najlepšie výsledky, ale ani ostatné výsledky významne nezaostávali. Pri niektorých konfiguráciách je pozorovateľný offset začiatku spracovania, vrátane zapísania vstupných správ do topiku `new_locations`. Spracovanie 1 milióna záznamov trvalo väčšine konfigurácií približne 5 minút. Konzumenti Location Recorder a konektor pre QuestDB spracovávajú správy do 1 minúty, ale sú mierne ovplyvnené konfiguráciou služby Collision Tracker, preto môže byť vhodným riešením prenesenie služby Collision Tracker a frontu



Obr. 7.10: Spracovanie správ konektorom QuestDB pri rôznych konfiguráciách služby Collision Tracker.

správy Apache Kafka do cloudového prostredia, zatiaľ čo ostatné služby by mohli zostať nasadené on-premise. Pri využití backendu `EmbeddedRocksDBStateBackend` bolo ukázané, že pri lokálnom spracovaní záleží na nastavení parametru `managed_memory_size` a pri 16GB hodnote tohto parametru dosahuje spracovanie podobné výsledky ako pri využití `HashMapStateBackend`, ale `HashMapStateBackend` sa ukazuje ako rýchlejšia voľba, pretože pri využití `EmbeddedRocksDBStateBackend` je podľa Flink dokumentácie [1] nutná serializácia a deserializácia pri každom prístupe a zmene stavu a je nutné aj čítanie stavu z disku. Pri nasadení služby Apache Kafka v cloud computing a spustení ostatných služieb na lokálnom serveri došlo k výraznému zrýchleniu spracovania. Pri nasadení služby CollisionTracker na cluster Flink, ktorý bol spustený na VM v cloud computing, malo spracovanie viac lineárny priebeh, ale rýchlosť spracovania niektorých konfigurácií nepredbehla výsledky pri nasadení Apache Kafka v cloud computing prostredí a služby CollisionTracker lokálne. Je to pravdepodobne spôsobené lepšou technickou špecifikáciou školského serveru ako využitej VM v cloude. Zjavne spracovanie závisí od technickej špecifikácie stroju, a v prípade využitia lepšej VM v cloud computing prostredí by sa rýchlosť spracovania správ mohla zlepšiť.

7.3 Testovanie obsluhy žiadostí od klientov

Testovanie obsluhy žiadostí od klientov bolo realizované na rovnakom datasete s veľkosťou 1 milióna záznamov ako v predchádzajúcej časti a bolo rozdelené do dvoch hlavných častí:

- Rýchlosť obsluhy API požiadavky pre 1 klienta vs porovnanie rýchlosti obsluhy požiadavky s využitím cachovania odpovedí
- Rýchlosť obsluhy požiadaviek v prípade opakovaných dotazov od 1000 klientov.

Testovanie tejto časti ale prebehlo na inom stroji ako predchádzajúce dve časti testovania, a to z dôvodu zjednodušenia analýzy výsledkov testovaní. Testovanie prebehlo na stroji so špecifikáciou:

- 8x Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz
- 32GiB RAM

Obsluha žiadostí jedného klienta

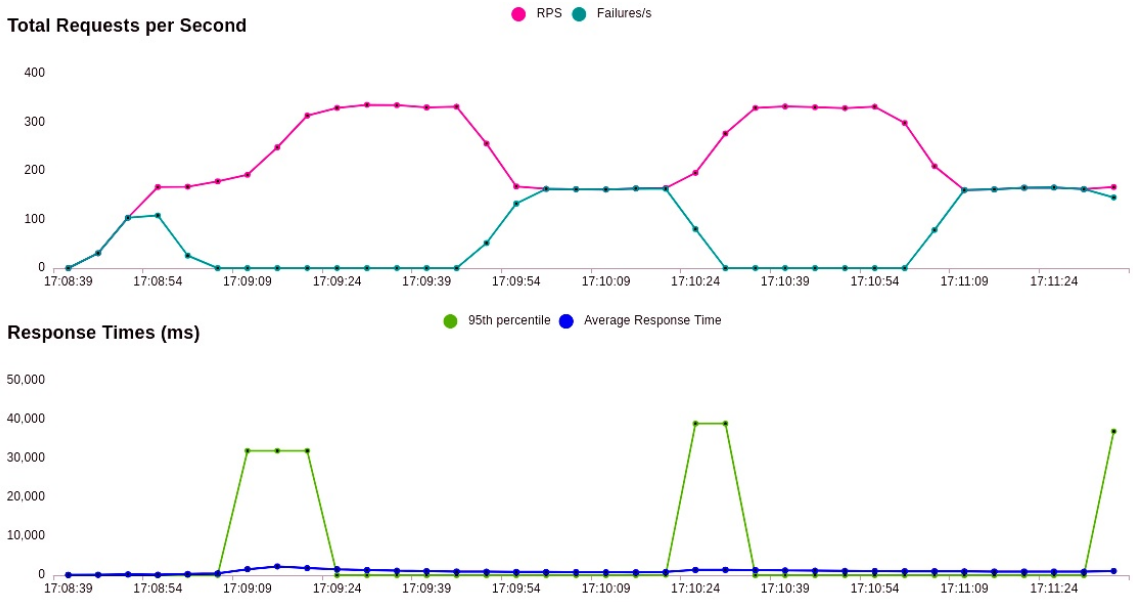
Testovanie bolo vykonané pre otestovanie obsluhy požiadaviek všetkými tromi API službami cez tri endpointy a bolo opakované 50-krát. Na meranie času odozvy bol využitý skript napísaný v jazyku shell, ktorý opakovane volal príkaz `curl`.
V tabuľke 7.8 je vidieť použitie vyrovnávacej pamäte v službe Nginx, ktorá výrazne zrýchľuje reakciu na požiadavky od klienta. Pre požiadavky týkajúcich sa aktuálnych polôh všetkých užívateľov (`/rt/locations/`), dochádza k zrýchleniu o takmer 300 milisekúnd vďaka využitiu pamäti cache. Pri požiadavkách, pýtajúcich sa na historické polohy užívateľa, sa zrýchlenie pri využití pamäte cache pohybuje okolo 75 milisekúnd a pri požiadavkách, pýtajúcich sa na užívateľov nachádzajúcich sa aktuálne v polygonoch, nastalo zrýchlenie až o 1420 milisekúnd.

	s cache	**bez cache**
/rt/locations/	2.7	299.1
/history/locations/?time=2024&user=500	3.3	78.1
/collisions/current/	1.9	1421.9

Tabuľka 7.8: Priemerná odozva spracovania dotazu pre jedného klienta v milisekundách.

Obsluha konkurentných žiadostí od 1000 klientov

Testovanie konkuretných požiadaviek prebiehalo cez službu `Locust` pričom klienti posielali žiadosti v intervaloch 1 až 5 sekúnd po dobu troch minút. Pre účely tohto testovania boli nasadené dve repliky služby RealTime API a bolo zapnuté ukladanie odpovedí na požiadavky do vyrovnávacej pamäte.



Obr. 7.11: Vizualizácia testovania obsluhy konkurentných žiadostí službou Locust.

Na vizualizácii zo služby Locust 7.11 je možné na spodnom grafe vidieť, že po každej minúte sa zvyšuje čas odpovede, pričom tieto intervaly tvoria 95% percentilové hodnoty. Je

to spôsobené uchovávaním odpovedí vo vyrovnávacej pamäti po dobu 30 sekúnd. Po 30 sekundách sa pamäť cache vymaže a odpoveď musí byť znovu načítaná z databázy. Tento jav sa prejavuje aj na hornom grafe, kedy po 30 sekundách dochádza k zlyhaniam pri obsluhe žiadostí. Priemerná doba odpovede v prípade, kedy bolo nutné najprv čítať z databázy, sa pohybovala medzi 1350 a 2251 ms. Celková priemerná doba odpovede bola 1120 ms a hodnota mediánu bola 3 ms. Detailné percentilové rozdelenie je zobrazené v tabuľke 7.9.

Testovanie zároveň ukázalo, že v niektorých prípadoch dochádza k zlyhaniam pri obsluhe žiadostí od klientov a dochádza k výpadku služby API. Celkovo pre 40820 žiadostí došlo k 12208 zlyhaniam, čo predstavuje 30% mieru zlyhania. Počas zlyhaní sa zobrazovala správa: `RemoteDisconnected('Remote end closed connection without response')`, čo naznačuje, že služby neboli schopné zvládnuť danú záťaž. Testovanie bolo tiež vykonané pri nasadení 4 replík API služby. Miera zlyhania sa znížila na 22 %, medián zostal na hodnote 3 ms, a priemerná doba odozvy sa zmenila na 921.88 ms, čomu zodpovedá aj rozdiel v 99% a 100% percentiloch v tabuľke 7.10 oproti tabuľke pre dve repliky 7.9.

50%il	**60%il**	**70%il**	**80%il**	**90%il**	**95%il**	**99%il**	**100%il**
3	4	4	6	10	34	38000	40000

Tabuľka 7.9: Percentilové rozdelenie trvania odpovede na požiadavky od klientov v milisekundách pri 2 replikách API služby.

50%il	**60%il**	**70%il**	**80%il**	**90%il**	**95%il**	**99%il**	**100%il**
3	4	5	8	14	61	31000	34000

Tabuľka 7.10: Percentilové rozdelenie trvania odpovede na požiadavky od klientov v milisekundách pri 4 replikách API služby.

Zhrnutie

Testovaním bolo zistené, že použitie vyrovnávacej pamäte v službe Nginx, ktorá slúži ako API gateway, prináša výrazné zrýchlenie pri odpovediach na požiadavky klientov. Konkurentné testovanie s 1000 klientmi ukázalo, že systém trpí pri zvýšenom počte požiadaviek vysokou mierou zlyhania v prípade nutnosti načítania dát z databázy, avšak miera zlyhania bola znížená pri nasadení štyroch replík API služby. Pri nasadení viacerých replík bola taktiež znížená aj priemerná doba obsluhy požiadaviek. Mediánová hodnota (3 ms) zostáva pri využití pamäte cache rovnaká pre dve aj štyri repliky a zároveň táto hodnota odpovedá priemernej hodnote odpovede na rovnakú požiadavku pri testovaní obsluhy požiadavky od jedného klienta z tabuľky 7.8.

Kapitola 8

Záver

Cieľom tejto práce bolo navrhnúť a implementovať škálovateľný systém schopný spracovať veľký prúd dát. V práci boli preskúmané architektúry a princípy vhodné pre systémy navrhované pre prostredie cloud computing a systémy, zameriavajúce sa na spracovanie prúdu dát v reálnom čase. Na základe prieskumu vhodných riešení bola navrhnutá škálovateľná architektúra systému, pracujúcom v reálnom čase, a vzhľadom na rôznorodosť dát boli zvolené viaceré databázy. Navrhnutý systém bol implementovaný s využitím princípov mikroslužieb, ktoré využívajú front správ Apache Kafka, čím je zaistená nezávislosť služieb a umožnené nasadenie viac replík služieb a prípadné rozšírenie systému. Najväčší problém pri spracovaní mala služba Collision Tracker, ktorá dokázala spracovať 1 milión záznamov v najnižšej dobe za 4 minúty. Doba spracovania sa mierne zlepšila pri využití Apache Kafka v cloud computing. Bolo otestované aj nasadenie služby CollisionTracker na cluster Flink, ktorý bol spustený na virtuálnom stroji v cloud computing, ale rýchlosť spracovávania pravdepodobne záleží od vybratého stroju pre VM. Iným riešením by mohlo byť využitie technológie Kubernetes v cloudovom prostredí. Zvyšní konzumenti dokázali správy spracovávať pod 1 minútu, a preto by ostatné služby mohli zostať nasadené spôsobom on-premise pre zníženie nákladov za cloudové služby. Hlavne konektor pre QuestDB sa ukázal ako veľmi rýchle a stále spracovanie správ (1 mil/3s), a preto by v budúcnosti mohlo byť zvážené aj využitie konektoru Redis pre Apache Kafka. Pri testovaní obsluhy požiadaviek bolo zistené, že pri nižšej záťaži ako obsluhe požiadaviek od jedného klienta dokáže systém bez využitia cache obslúžiť požiadavok do 300 milisekúnd. Pri zvýšenej záťaži s využitím cache dokázal systém obslúžiť 95 % požiadaviek do 61 ms, so zachovaním mediánovej hodnoty 3 ms, čo zodpovedá hodnote pri obsluhe požiadavky s využitím cache. Celkovo sa ukázala pamäť cache pre ukladanie odpovedí ako dobrá voľba pre urýchlenie obsluhy požiadaviek. Ak by som na práci pracovala znova, tak by som pre zložitosť nasadenia prostredia Apache Flink uprednostnila jednoduchšie riešenie cez knižnicu Kafka Streams s využitím priestorovej knižnice ako napr. spatial4j. Budúcim rozšírením systému by mohli byť upozornenia v prípade prekročenia hraníc polygonu napojením na topik `collisions` vo fronte správ alebo využitie priestorových dátových typov v rámci databáz Redis a MongoDB pre výpočet priestorových operácií. Zdrojové kódy sú verejne dostupné[1].

[1] https://github.com/dnosko/RTStreamProcessing

Literatúra

[1] *Apache Flink Documentation* [online]. Apache Software [cit. 2023-11-15]. Dostupné z: https://nightlies.apache.org/flink/flink-docs-release-1.18/.

[2] *Apache Kafka Streams* [online]. Apache Software [cit. 2023-11-14]. Dostupné z: https://kafka.apache.org/documentation/streams/.

[3] *Apache Spark Streaming Documentation* [online]. Apache Software [cit. 2023-11-16]. Dostupné z: https://spark.apache.org/docs/latest/streaming-programming-guide.html.

[4] *Apache Storm Documentation* [online]. Apache Software [cit. 2023-11-17]. Dostupné z: https://storm.apache.org/.

[5] *AWS* [online]. Amazon Web Services [cit. 2023-11-25]. Dostupné z: https://aws.amazon.com/.

[6] *Data streaming with AKS* [online]. Microsoft [cit. 2023-12-04]. Dostupné z: https://learn.microsoft.com/en-us/azure/architecture/solution-ideas/articles/data-streaming-scenario.

[7] *Google Cloud* [online]. Google [cit. 2023-11-28]. Dostupné z: https://cloud.google.com/.

[8] *Microsoft Azure* [online]. Microsoft [cit. 2023-11-25]. Dostupné z: https://azure.microsoft.com/.

[9] *Structured Streaming Programming Guide* [online]. Apache Software [cit. 2023-11-24]. Dostupné z: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html.

[10] *System Design of Uber App – Uber System Architecture* [online]. [cit. 2023-12-04]. Dostupné z: https://www.geeksforgeeks.org/system-design-of-uber-app-uber-system-architecture/.

[11] Akidau, T., Chernyak, S. a Lax, R. *Streaming Systems: The what, where, when, and how of large-scale data processing.* 1. vyd. Sebastopol, CA: O'Reilly, 2018. ISBN 978-1-491-98387-4.

[12] Burns, B. *Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services.* 1. vyd. Sebastopol, CA: O'Reilly, 2018. ISBN 978-1-491-98364-5.

[13] Comer, D. E. *The Cloud Computing Book: The Future Of Computing Explained.* 1. vyd. Boca Raton: CRC Press, 2021. ISBN 978-1-003-14750-3.

[14] Dongen, G. van a Poel, D. Van den. Evaluation of Stream Processing Frameworks. *IEEE Transactions on Parallel and Distributed Systems*. 2020, zv. 31, č. 8, s. 1845–1858. DOI: 10.1109/TPDS.2020.2978480.

[15] Executech. *The Cloud vs. On-Premise Cost: Which One is Cheaper?* [online]. [cit. 2023-11-05]. Dostupné z: https://www.executech.com/insights/the-cloud-vs-on-premise-cost-comparison/.

[16] Fehling, C., Leymann, F., Retter, R., Schupeck, W. a Arbitter, P. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Wien: Springer-Verlag, 2014. ISBN 978-3-7091-1568-8.

[17] Henning, S. a Hasselbring, W. Theodolite: Scalability Benchmarking of Distributed Stream Processing Engines in Microservice Architectures. *Big Data Res.* NLD: Elsevier Science Publishers B. V. jul 2021, zv. 25, C. DOI: 10.1016/j.bdr.2021.100209. ISSN 2214-5796.

[18] Huawei Technologies Co., L. *Cloud Computing Technology*. 1. vyd. Singapore: Springer, 2023. ISBN 978-981-19-3026-3. Dostupné z: https://doi.org/10.1007/978-981-19-3026-3.

[19] Hueske, F. a Kalavri, V. *Stream Processing with Apache Flink: Fundamentals, Implementation and Operation of Streaming Applications*. 1. vyd. Sebastopol, CA: O'Reilly, 2019. ISBN 978-1-491-97429-2.

[20] Hurwitz, J., Bloor, R., Kaufman, M. a Halper, F. *Cloud Computing For Dummies*. 1. vyd. Indianapolis, Indiana: Wiley Publishing, 2010. ISBN 978-0-470-48470-8.

[21] Jha, S., Jha, M., O'Brien, L. a Singh, P. K. Architecture for Complex Event Processing Using Open Source Technologies. In: *2016 3rd Asia-Pacific World Congress on Computer Science and Engineering (APWC on CSE)*. 2016, s. 218–225. DOI: 10.1109/APWC-on-CSE.2016.044.

[22] Kleppmann, M. *Designing Data-Intensive Applications: The big ideas behind reliable, scalable, and maintainable systems*. 1. vyd. Sebastopol, CA: O'Reilly, 2017. ISBN 978-1-449-37332-0.

[23] Kopetz, H. a Steiner, W. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. 3. vyd. Cham, Switzerland: Springer, 2022. ISBN 978-3-031-11992-7.

[24] Kreps, J. *I Heart Logs: Event Data, Stream Processing, and Data Integration*. 1. vyd. Sebastopol, CA: O'Reilly, 2015. ISBN 978-1-491-90938-6.

[25] Lozinski, L. *The Uber Engineering Tech Stack, Part II: The Edge and Beyond* [online]. [cit. 2023-12-04]. Dostupné z: https://www.uber.com/en-CZ/blog/uber-tech-stack-part-two/.

[26] Marz, N. a Warren, J. *Big Data: Principles and best practices of scalable real-time data systems*. 1. vyd. Shelter Island, NY: Manning, 2015. ISBN 978-1-617-29034-3.

[27] Needham, M. *Building Real-Time Analytics Systems: From Events to Insights with Apache Kafka and Apache Pinot*. 1. vyd. Sebastopol, CA: O'Reilly, 2023. ISBN 978-1-098-13879-0.

[28] Prakash, A. *Columnar Storage Formats: Benefits and Use Cases in Data Engineering* [online]. [cit. 2023-11-28]. Dostupné z: https://airbyte.com/data-engineering-resources/columnar-storage.

[29] Redmond, E. a Wilson, J. R. *Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement*. 1. vyd. Dallas, Texas: The Pragmatic Bookshelf, 2012. ISBN 978-1-93435-692-0.

[30] Richards, M. *Software Architecture Patterns: Understanding Common Architectural Styles and When to Use Them*. 2. vyd. Sebastopol, CA: O'Reilly, 2022. ISBN 978-1-098-13427-3.

[31] Saxena, S. a Gupta, S. *Practical Real-Time Data Processing and Analytics: Distributed Computing and Event Processing using Apache Spark, Storm, and Kafka*. 1. vyd. Birmingham, UK: Packt Publishing, 2017. ISBN 978-1-78728-120-2.

[32] Sodabathina, R., Gong, C., Upadhyay, N. a Konka, S. *Build Modern Data Streaming Architectures on AWS: AWS Whitepaper* [online]. Amazon Web Services, máj 2022, 2022-08-25 [cit. 2023-11-18]. Dostupné z: https://docs.aws.amazon.com/whitepapers/latest/build-modern-data-streaming-analytics-architectures/.

Príloha A

Manuál pre spustenie

Spustenie:

```
docker compose build
docker compose up
```

Spustenie generovania správ:

```
python3 generator/main.py [--num\_dev] [--limit\_x] [--limit\_y]
[--ws] [--limit] [--file]
```

Upratanie:

```
docker compose down // zmazanie kontajnerov
```

Kód A.1: Vhodné je prípadne zmazať aj volumes a vytvorené obrazy – images. Ak nie sú pri generovaní správ špecifikované argumenty, použijú sa implicitné hodnoty.

A.1 Nasadenie Apache Kafka, služby CollisionTracker a relačnej databázy na GCP

Po nasadení je nutné odstrániť tieto služby z docker-compose súboru a je potrebné nahradiť služby, ktorých sa zmeny týkajú súbormi zo zložky `src/cloud`.

Nasadenie Apache Kafka

Postup nasadenia:

1. Vyhladať na GCP službu Apache Kafka® & Apache Flink® on Confluent Cloud™.
2. Po aktivovaní služby je prehliadač presmerovaný na stránku platformy Confluent.
3. Je potrebné vytvoriť cluster.
4. Po vytvorení clustra je nutné vytvoriť topiky new_locations, ktorý môže byť ponechaný s predvolenou konfiguráciou, a topik collisions, pri ktorom je potrebné zvoliť `cleanup.policy` ako compact, delete.
5. Následne kliknúť na `API Keys` v menu.

6. Vytvoriť Global Access Key.

7. Pre správne fungovanie služieb komunikujúcich s Apache Kafka, musia byť okrem adresy bootstrap serveru uvedené aj údaje z vygenerovaného API kľúču a dodatočné bezpečnostné atribúty `security.protocol=SASL_SSL`, `sasl.mechanism=PLAIN` a `sasl.jaas.config`. Pre zjednodušenie nasadenia, sa v zložke `src/cloud/kafka` nachádzajú zmenené súbory služieb Producer, Collision Tracker, Collision Recorder a Location Recorder, ktorými je možné nahradiť súbory s rovnakým názvom v zložke `src`.

Nasadenie PostgreSQL

Postup nasadenia:

1. Ako prvé je potrebné nahrať obraz databázy `database.sql` do služby Buckets pre umožnenie importovania databázy.

2. Vyhľadať službu SQL na GCP.

3. Kliknúť na `Create Instance`.

4. Zvoliť PostgreSQL.

5. Nakonfigurovať databázu a zvoliť heslo. Pre účely testovania bola zvolená verzia PostgresSQL 15. IP connections boli zvolené na Public IP.

6. Kliknúť na `Create Instance`.

7. Kliknúť na `Databases` na pravom menu. Kliknúť na `Create Database` a vytvoriť databázu s názvom **data**.

8. Po vytvorení databázy, sa vrátiť cez menu na `Overview` a kliknúť na `Import`.

9. Zvoliť súbor database.sql z bucketu, nastaviť import na SQL a destination na vytvorenú databázu **data**. Kliknúť na `Import`. Ukážka je zobrazená na obr. A.1.

10. Ďalej je nutné povoliť IP adresy. Kliknutím na tlačidlo `Edit configuration` nachádzajúcim sa pod názvom **Configuration**.

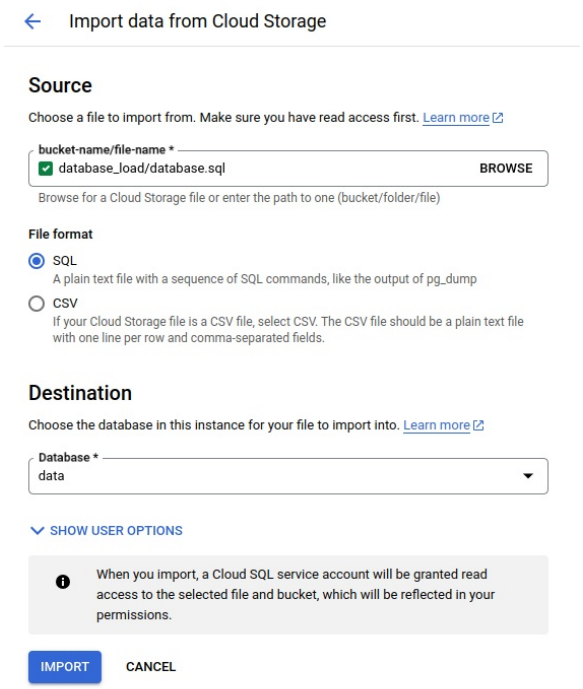
    (a) Zvoliť `Connections`.

    (b) Kliknúť na `Add a Network`.

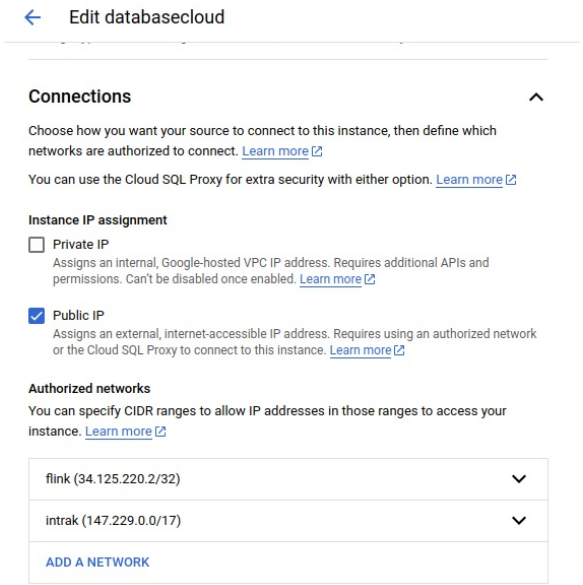
    (c) Zadať názov a povolený rozsah IP adries (viď A.2).

    (d) Uložiť tlačidlom `Save`.

11. Connection String pre pripojenie k databáze pre aplikácie napísané v jazyku Java je nasledujúceho formátu:
    `jdbc:postgresql://{Public IP address}:5432/data?cloudSqlInstance={Connection name}`. Public IP adress databázy a Connection name je možné nájsť v sekcii `Overview` pod nadpisom **Connect to this instance**. Pre triedy napísané v jazyku Python to je formát:
    `postgresql://postgres:{password}@Public IP address:5432/data`.



Obr. A.1: Importovanie databázy z predpripraveného obrazu uloženého cez cloud úložisko Buckets.

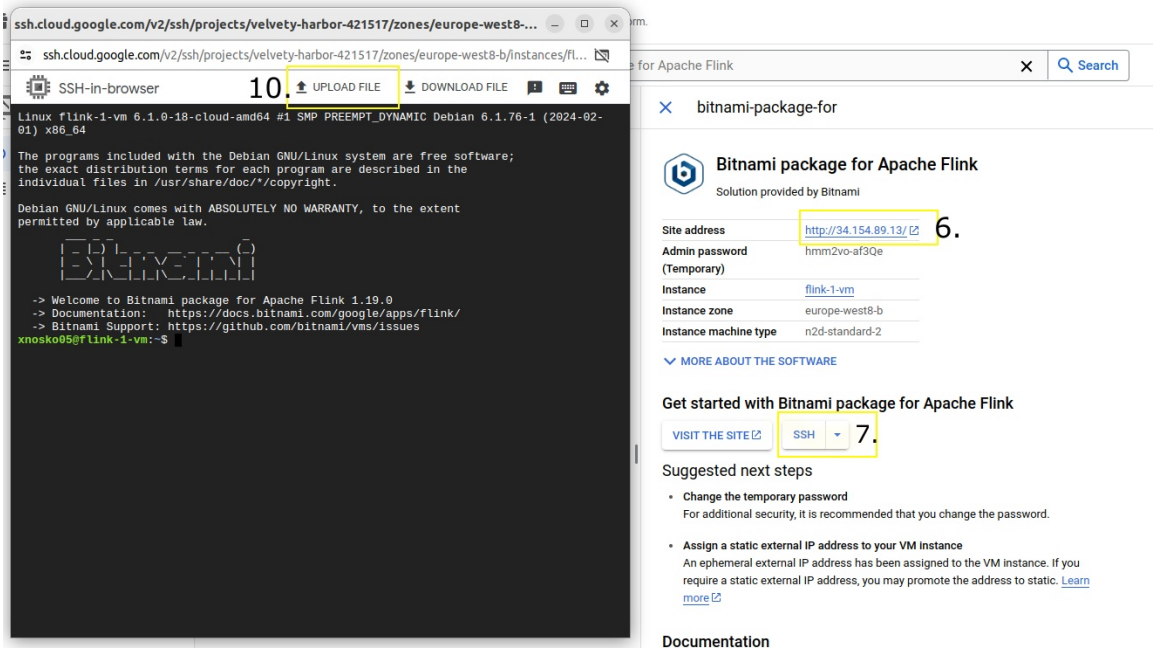


Obr. A.2: Ukážka konfigurácie povolených IP adries pre pripojenie k databáze.

Nasadenie CollisionTracker

Postup nasadenia:

1. Ako prvé je potrebné nasadenie databázy popísané v časti vyššie.



Obr. A.3: Označené kroky 6., 7. a 10. pre nasadenie služby CollisionTracker na GCP.

2. Vyhľadať službu `Bitnami package for Apache Flink` na GCP.
3. Kliknúť na `Launch`.
4. Zvoliť špecifikáciu stroju pre VM.
5. Kliknúť na `Deploy`.
6. Potom ako je VM nasadené a má priradenú IP adresu (viď 6. na A.3), je nutné túto IP adresu pridať do povolených IP adries databázy (viď A.2).
7. Zvoliť ssh pripojenie na VM (viď 7. na A.3).
8. Zmeniť súbor gradle.build, app.config a CollisionTracker.java za súbory s rovnakým menom nachádzajúce sa v zložke `src/cloud/flink`.
9. Skopilovať aplikáciu CollisionTracker príkazom `./gradlew shadowJar`.
10. Nahrať súbor app.config a jar súbor na VM (viď 10. na A.3).
11. Vytvoriť na VM zložku pre checkpointy: `sudo mkdir /flink/checkpoints/`
12. Povolenie vykonávať zmeny v zložke: `sudo chmod -R 777 /flink/checkpoints/`
13. Spustenie služby:
    `flink run CollisionTracker-1.0-SNAPSHOT-all.jar app.config`
14. Zrušenie úlohy: `flink cancel {JOB ID}`

Príloha B

Konfiguračné súbory

V prípade využitia viac brokerov Apache Kafka sú oddelované čiarkou v každom konfiguračnom súbore.

Konfiguračný súbor pre službu Producer

```
{
  "host": "0.0.0.0",
  "port": 8001,
  "kafka_topic": "new_locations",
  "bootstrap_servers": "kafka1:19092",
  "flush_interval": 10000
}
```

Kód B.1: Konfiguračný súbor pre službu Producer. Host a port, vyjadrujú na akom porte a adrese by mal websocket server fungovať. Kafka_topic je názov topiku, do ktorého sú ukladané prijímané polohy. Bootstrap_servers sú adresy kafka brokerov.

Konfiguračný súbor pre službu Location Recorder

```
{
  redis_host=redis
  redis_port=6379
  kafka_server=kafka1:19092
  topic_name=new_locations
  group_id_config=kafka-location-recorder
}
```

Kód B.2: Redis_host a redis_port je adresa, na ktorej je spustená databáza Redis. V tomto prípade ide o adresu v rámci docker siete, kde redis je návov kontajneru. Kafka_topic je názov topiku, z ktorého sú príjmané polohy (rovnaké ako pre službu Producer). kafka_servers sú adresy kafka brokerov.

Konfiguračný súbor pre službu Collision Tracker

```
{
  kafka_server=kafka1:19092
  group_id=collision-tracker
  postgres_connection_string=jdbc:postgresql://postgis:5432/data
  postgres_username=postgres
  postgres_password=password
}
```

Kód B.3: Podobne ako v predchádzajúcich ukážkach.

Konfiguračný súbor pre službu Collision Recorder

```
{
  kafka_server=kafka1:19092
  topic_name=collisions
  group_id_config=kafka-collisions-recorder
  mongodb=mongodb://user:pass@mongodb:27017
}
```

Kód B.4: Podobne ako v predchádzajúcich ukážkach.

Príloha C

Dostupné API endpointy

Realtime Locations

- `/rt/locations/`

  Štruktúra odpovede:

  ```
  [{
      user_id: int,
      device_id: int,
      point: {
          x: float,
          y: float
      },
      timestamp: int
  }]
  ```

  Parametre:

  – user: List[int]

- `/rt/users/`

  Štruktúra odpovede:

  ```
  [{
      user_id: int,
      device_id: int
  }]
  ```

Historical Locations

- `/history/locations/{user}/`

  Štruktúra odpovede:

  ```
  {
      id_user: int,
      id_device: int,
      collisions: List[
          id_polygon: int
          enter_date: datetime
      ]
  }
  ```

  Parametre:

  - user: int
  - time: str
    - formát je voľnejší,
      je možné špecifikovať okno napr.
      `2018-01;-3d`[1]

- `/history/locations/`

  Štruktúra odpovede:

  ```
  {
      id_user: int,
      id_device: int,
      collisions: List[
          id_polygon: int
          enter_date: datetime
      ]
  }
  ```

  Parametre:

  - time: str (pre formát viď predchádzajúci endpoint)
  - user: Optional[List[int]]

Polygons and Collisions

- `/polygons/` - polygony v databáze

  Štruktúra odpovede:

  ```
  [{
      id: int,
      creation: datetime,
      valid: bool,
      category: str,
      fence: str
  }]
  ```

  Parametre:

  - valid: Optional[bool]
  - category: Optional[int]

[1] https://questdb.io/docs/reference/sql/where/#timestamp-and-date

- `/collisions/history/` - kolízie v časovom intervale (jednotky, nachádzajúce sa v polygonoch)

Štruktúra odpovede:

```
{
    time: str,
    collisions: List[
        id_user: int,
        id_device: int,
        collisions: List[
                id_polygon: int,
                inside: bool,
                enter_date: datetime,
                exit_date: Optional[
                    datetime],
                enter_point: List[float
                    ],
                exit_point: Optional[
                    List[float]],
        ]
    ]
}
```

Parametre:

- time: str
  - (formát `%Y-%m-%dT%H:%M:%S`, pre špecifikovanie intervalu (od dátumu, do dátumu) je nutné oddeliť časy bodkočiarkou ; pričom prvá hodnota je starší dátum a druhá hodnota je aktuálnejší)
- polygons: Optional[List[int]]
- user: Optional[List[int]]

- `/collisions/current/` - aktuálne kolízie (jednotka sa nachádza v polygone)

Štruktúra odpovede:

```
[{
    id_user: int,
    id_device: int,
    collisions: List[
        id_polygon: int
        enter_date: datetime
    ]
}]
```

Parametre:

- polygons: Optional[List[int]]
- user: Optional[List[int]]

Príloha D

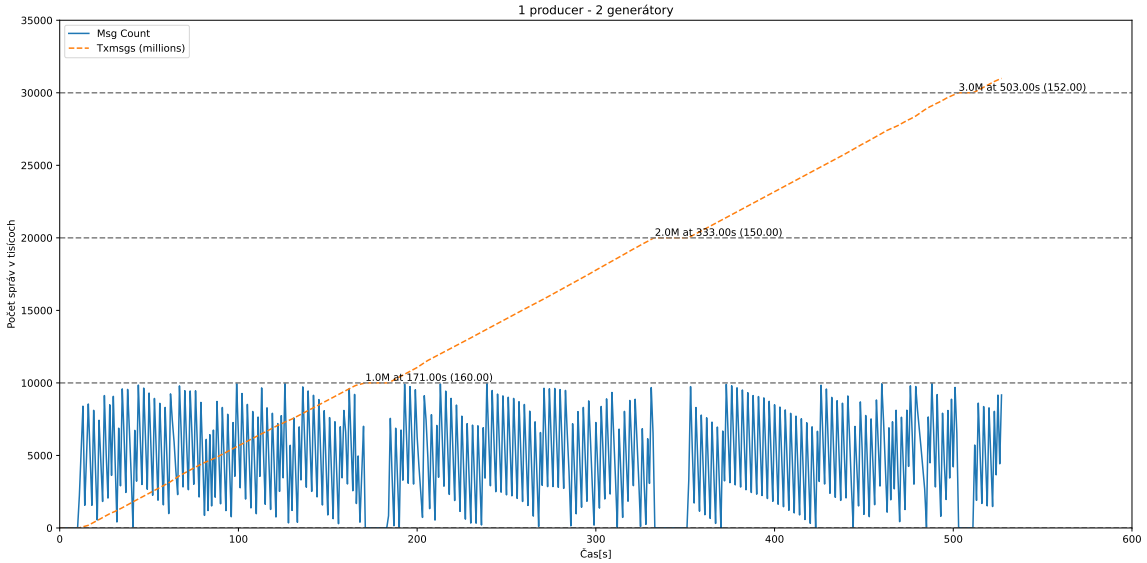
Prístupové body pre služby

Prvý stĺpec značí prístupový bod pre lokálne pripojenie, druhý stĺpec pripojenie z docker prostredia.

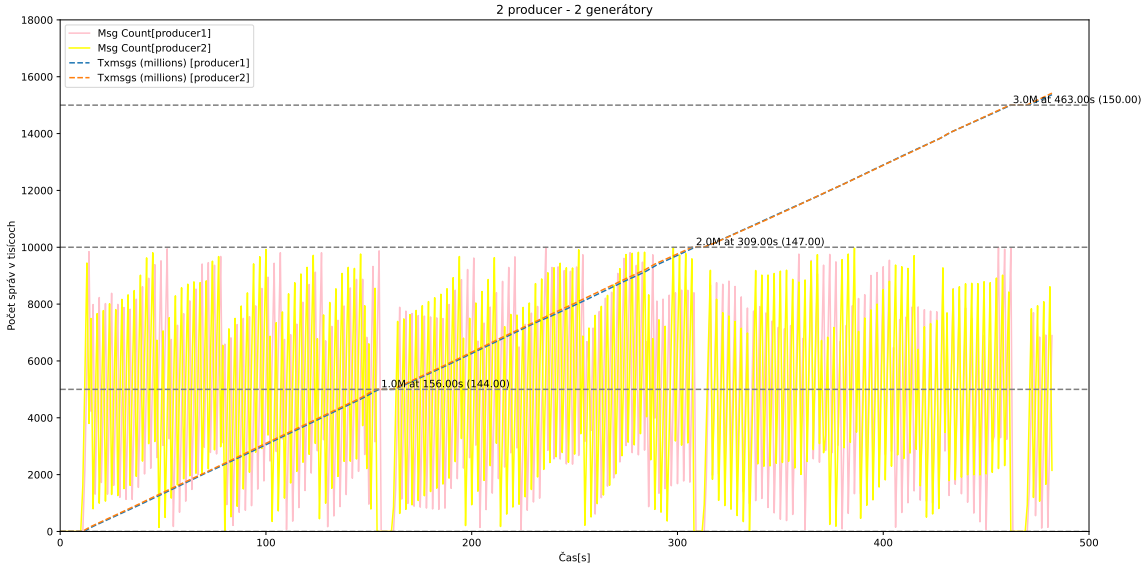
- **questdb:** QuestDB ponúka viac možností pripojenia cez rôzne porty.
  - **REST API a webové rozhranie:** localhost:9000 – questdb:9000
  - **Postgres wire protocol:** localhost:8812 – questdb:8812
    - user=admin
    - password=quest
    - dbname=qdb
  - **InfluxDB line protocol:** localhost:9009 – questdb:9009 (pozn. využitý pri konektore Apache Kafka)
- **mongodb:** localhost:7017 – mongodb:27017
  - username = user
  - password = pass
- **redis:** localhost:6379 – redis:6379
- **postgres:** localhost:25432 – postgres:5432
  - databáza = data
  - user = postgres
  - password = password
- **kafka-ui:** localhost:8080 – kafka-ui:8080
- **zookeeper:** localhost:2181 – kafka-ui:2181
- **kafka1:** localhost:9092 – kafka1:19092
- **connect:** localhost:8083 – connect:8083
- **api_collisions:** none – api_collisions:8100
- **api_historical:** none – api_historical:8101
- **api_realtime:** none – api_realtime:8102
- **nginx:** localhost:8088 – nginx:80

Príloha E

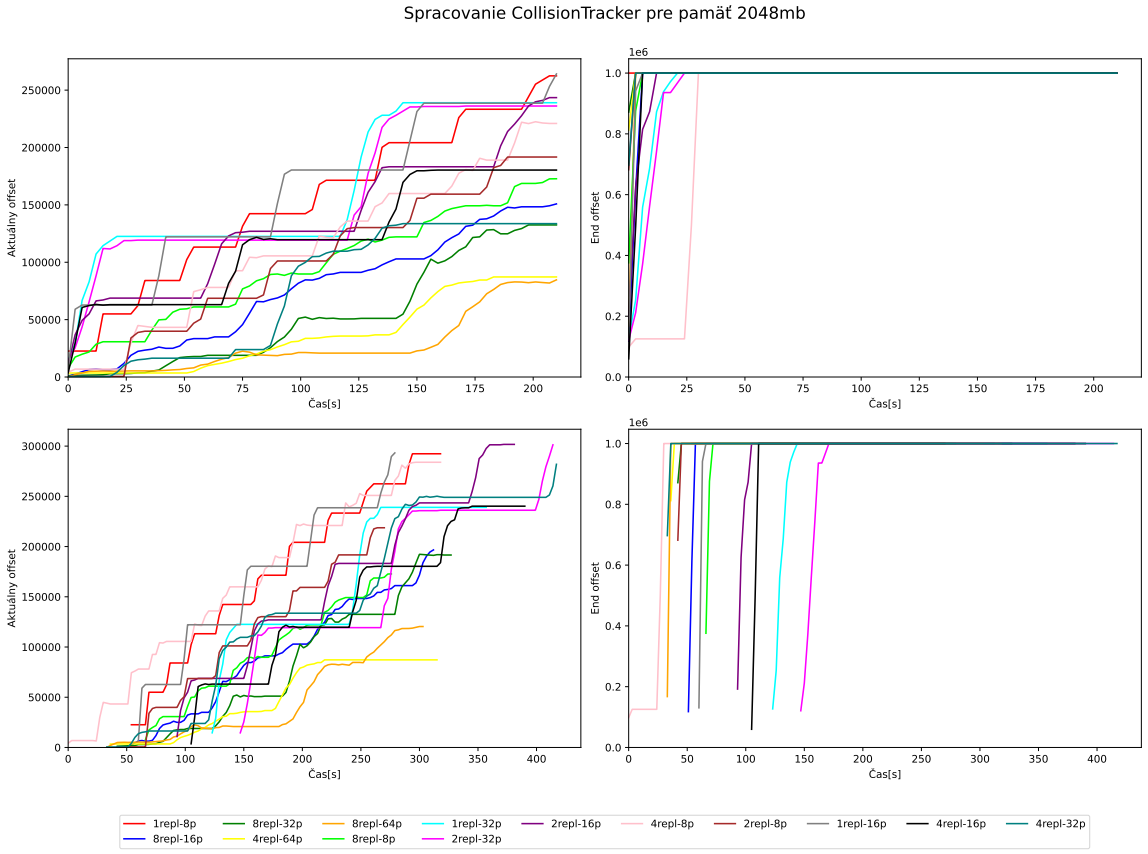
Dodatočné grafy



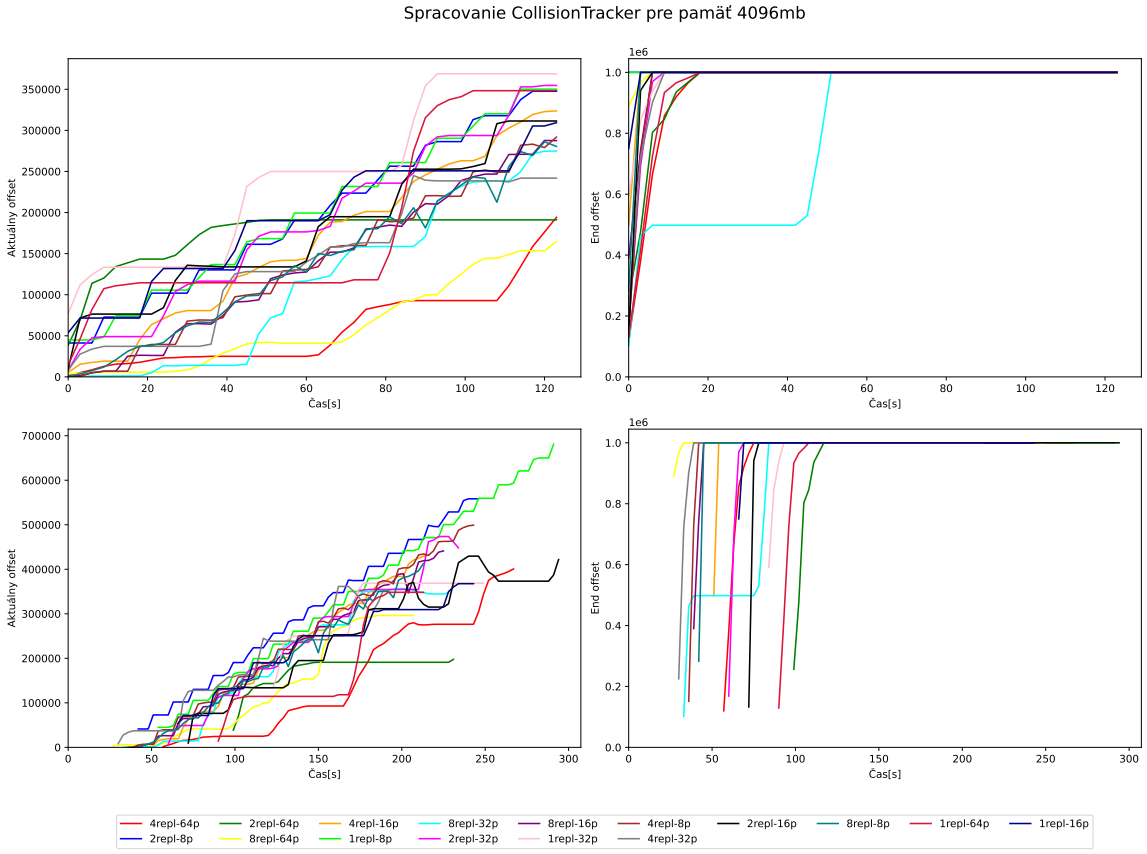
Obr. E.1: Ukladanie vygenerovaných správ do Apache Kafka jednou službou Producer pri generovaní správ dvomi generátormi.



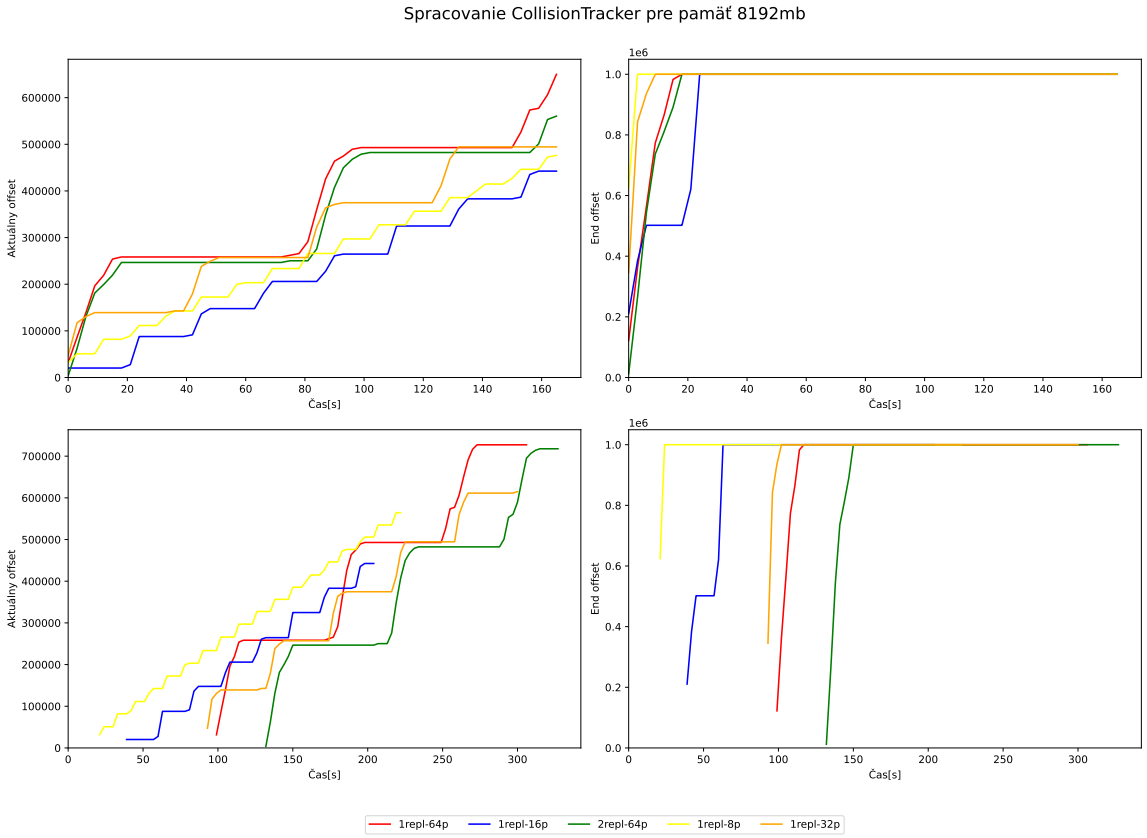
Obr. E.2: Ukladanie vygenerovaných správ do Apache Kafka dvomi službami Producer pri generovaní správ dvomi generátormi.



Obr. E.3: Spracovanie správ CollisionTracker pre backend EmbeddedRocksDBStateBackend `managed_memory_size` 2048MB.



Obr. E.4: Spracovanie správ CollisionTracker pre backend EmbeddedRocksDBStateBackend `managed_memory_size` 4096MB.



Obr. E.5: Spracovanie správ CollisionTracker pre backend EmbeddedRocksDBStateBackend `managed_memory_size` 8192MB.

Príloha F

Obsah priloženého pamäťového média

- `docs/` - technická správa vo formáte pdf a zdrojové kódy technickej správy (LaTeX)
- `src/` - zdrojové kódy programu
  - `src/` - zdrojové kódy pre lokálne spustenie
  - `src/cloud/` - zmenené zdrojové kódy pre nasadenie v cloud computing prostredí. Nasadenie opísané v A.1.

Príloha G

Hodnotenie spolupráce v rámci diplomovej práce z pohľadu GINA Software

Tato príloha poskytuje prehľad o procese spolupráce a spätnú väzbu na formu a efektivitu tejto spolupráce.

Forma spolupráce

Spolupráce byla zahájena definováním vzájemných očekávání a analýzou požadavků a případů užití s doplněním průmyslového kontextu. Následně probíhaly pravidelné konzultace o pokroku a řešení případných problémů a otázek.

Hodnocení spolupráce

Komunikace se studentkou byla pravidelná a produktivní. Studentka byla velmi proaktivní ve vyhledávání vhodných řešení a rychle reagovala na naše podněty. Velmi si vážíme analytického a koncepčního přístupu k řešení problémů ze strany studentky. Ačkoliv se vzhledem ke komplikované konfiguraci zvolené Apache Flink technologie nepodařilo limity výsledného řešení plně otestovat, jsou dosažené výsledky velmi přesvědčivé a uspokojivé.

Naše technická podpora a poskytnuté zdroje byly efektivně využívány k dosažení cílů práce. Spolupráce s námi poskytla studentce praktické znalosti a obohatila její práci o průmyslový kontext, čímž zvýšila její hodnotu a relevanci.

Závěr

Z pohledu GINA Software byla spolupráce se studentkou úspěšná a vedla k výsledkům, které splnily naše očekávání. Realizované řešení je z architektonického pohledu velmi zdařilé. Oceňujeme její snahu, odbornost a angažovanost při realizaci diplomové práce. Výsledky práce potvrdily vhodnost cloudových technologií i pro naše budoucí projekty.

Marek Řehulka
CTO
rehulka@ginasystem.com
GINA Software s. r. o. | www.ginasystem.com