

**SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE  
FAKULTA ELEKTROTECHNIKY A INFORMATIKY**

Evidenčné číslo: FEI-5384-92224

**VYUŽITIE APACHE KAFKA PRE SPRACOVANIE  
ÚDAJOV A DETEKCIU BEZPEČNOSTNÝCH  
INCIDENTOV NA ZÁKLADE ARCHITEKTÚRY  
RIADENEJ UDALOSŤAMI**

**DIPLOMOVÁ PRÁCA**

**2022**

**Bc. Vasyl' Klevťanyk**

**SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE  
FAKULTA ELEKTROTECHNIKY A INFORMATIKY**

Evidenčné číslo: FEI-5384-92224

**VYUŽITIE APACHE KAFKA PRE SPRACOVANIE  
ÚDAJOV A DETEKCIU BEZPEČNOSTNÝCH  
INCIDENTOV NA ZÁKLADE ARCHITEKTÚRY  
RIADENEJ UDALOSŤAMI  
DIPLOMOVÁ PRÁCA**

Študijný program:	Aplikovaná informatika
Študijný odbor:	Informatika
Školiace pracovisko:	Ústav informatiky a matematiky
Vedúci záverečnej práce:	Ing. Štefan Balogh, PhD.

**2022**

**Bc. Vasyl' Klevťanyk**



## ZADANIE DIPLOMOVEJ PRÁCE

Autor práce: Bc. Vasyl' Klevľanyk  
Študijný program: aplikovaná informatika  
Študijný odbor: informatika  
Evidenčné číslo: FEI-5384-92224  
ID študenta: 92224  
Vedúci práce: Ing. Štefan Balogh, PhD.  
Vedúci pracoviska: Dr. rer. nat. Martin Drozda  
Miesto vypracovania: Ústav informatiky a matematiky

Názov práce: **Využitie Apache Kafka pre spracovanie údajov a detekciu bezpečnostných incidentov na základe architektúry riadenej udalosťami**

Jazyk, v ktorom sa práca slovenský jazyk  
vypracuje:

Špecifikácia zadania: Detekcia incidentov je veľmi potrebná a aktuálna oblasť výskumu. Pri detektii je potrebné sledovať dátá z mnohých zdrojov, potrebných pre detekciu rôznych typov útokov. To prináša potrebu streamového spracovania dát a následne ich paralelne spracovanie pre detekciu rôznych vektorov útoku. Cieľom práce je otestovať možnosti využitia frameworku Apache Kafka a jej integráciu do procesu detektie.

### Úlohy:

1. Analyzujte existujúce systémy a software vhodné pre detekciu bezpečnostných incidentov na základe architektúry riadenej udalosťami.
2. Implementujte vybrané riešenia s využitím frameworku Apache Kafka a streamoveho paralelného spracovania dát pre detekciu rôznych vektorov útoku.
3. Otestujte možnosti a úspešnosť detektie útokov s údajov z rôznych zdrojov.

### Literatúra:

1. Ouhssini, M., Afdel, K., Idhammad, M., & Agherrabi, E. (2021, October). Distributed intrusion detection system in the cloud environment based on Apache Kafka and Apache Spark. In 2021 Fifth International Conference On Intelligent Computing in Data Sciences (ICDS) (pp. 1-6). IEEE.
2. Tidjon, L. N., Frappier, M., & Mammar, A. (2020, April). Intrusion detection using astds. In International Conference on Advanced Information Networking and Applications (pp. 1397-1411). Springer, Cham.

Dátum schválenia zadania 25. 04. 2022  
práce:

Dátum odovzdania: 13. 05. 2022

**Bc. Vasyl' Klevčanyk**  
študent

**Dr. rer. nat. Martin Drozda**  
vedúci pracoviska

**prof. RNDr. Gabriel Juhás, PhD.**  
garant študijného programu

# SÚHRN

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE  
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Študijný program :

Aplikovaná informatika

Diplomová práca:

Využitie Apache Kafka pre spracovanie údajov a detekciu bezpečnostných incidentov na základe architektúry riadenej udalosťami

Autor:

Bc. Vasyl' Klevl'anyk

Vedúci diplomovej práce:

Ing. Štefan Balogh, PhD.

Miesto a rok predloženia práce:

Bratislava 2022

Cieľom diplomovej práce je zoznámenie sa s problematikou spracovania veľkého množstva dát a s existujúcimi architektúrami vytvorenia softvérových produktov a použitie najvhodnejšej architektúry pre návrh a implementovanie systému detekcie a spracovania bezpečnostných incidentov. Diplomová práca sa skladá z dvoch častí. Prvá časť je venovaná podrobnému rozboru existujúcich architektúr vytvorenia softvéru, ich výhodám a nevýhodám a posudku vhodnosti ich použitia pre návrh nášho systému. V druhej časti sa popisujú technológie, ktoré sú použité v našom systéme. Tretia časť sa venuje návrhu, implementovaniu a popisu nášho systému detekcie a spracovania bezpečnostných incidentov. V závere práce sú zhodnotené výsledky a jej užitočnosť.

Kľúčové slová: Apache Kafka, Architektúra riadená udalosťami, Python backend

# ABSTRACT

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA  
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION  
TECHNOLOGY

Study Programme:	Applied Informatics
Diploma Thesis:	Using Apache Kafka for data processing and detection security incidents based on the event-driven architecture
Author	Bc. Vasyl' Klevl'anyk
Supervisor:	Ing. Štefan Balogh, PhD.
Place and year of submission:	Bratislava 2022

The aim of this thesis is to familiarize with the issues of processing large amounts of data and with existing architectures for creating software products and the use of the most appropriate architecture for the design and implementation of a system for the detection and processing of security incidents. The diploma thesis consists of two parts. The first part is devoted to a detailed analysis of existing software architectures, their advantages and disadvantages and an assessment of its suitability for the design of our system. The second part of the thesis describes the technologies, which are used in our system. The third part is devoted to the design, implementation and description of our security incident detection and processing system. The thesis concludes with an evaluation of the results and its usefulness.

Key words: Apache Kafka, Event-driven architecture, Python backend

# Vyhľásenie autora

Podpísaný Bc. Vasyl' Klevľanyk čestne vyhlasujem, že som Diplomovú prácu Využitie Apache Kafka pre spracovanie údajov a detekciu bezpečnostných incidentov na základe architektúry riadenej udalosťami vypracoval na základe poznatkov získaných počas štúdia a informácií z dostupnej literatúry uvedenej v práci.

Uvedenú prácu som vypracoval pod vedením Ing. Štefana Balogha, PhD.

V Bratislave dňa 13.05.2022

.....  
podpis autora

# **Pod'akovanie**

Týmto by som sa chcel pod'akovať pánovi prof. Ing. Štefanovi Baloghovi, PhD. za odbornú pomoc pri písaní tejto práce, taktiež za cenné rady a informácie pri vypracovaní tejto práce. Zároveň by som sa chcel pod'akovať svojim rodičom za možnosť študovať na Slovensku a Terezii Hagarovej za podporu.

# Obsah

<b>Úvod</b>	14
<b>1 Architektúry softvérových systémov</b>	15
1.1 Monolitická architektúra.....	15
1.2 Architektúra mikroslužieb .....	18
1.3 Architektúra riadená modelom .....	21
1.4 Architektúra orientovaná na služby.....	24
1.5 Architektúra riadená udalosťami.....	27
1.5.1 Realizácia pomocou návrhového vzoru sprostredkovateľ.....	29
1.5.2 Realizácia pomocou návrhového vzoru sprostredkovateľ správ .....	31
<b>2 Technológie použité v práci</b>	34
2.1 Apache Kafka.....	34
2.2 Docker .....	37
2.3 ASTD.....	37
<b>3 Systém pre detekciu a spracovanie bezpečnostných incidentov</b>	39
3.1 Implementácia systému .....	40
3.1.1 Implementácia klienta .....	40
3.1.2 Implementácia manažéra partícií .....	42
3.1.3 Implementácia sprostredkovateľa správ .....	44
3.1.4 Implementácia serverovej služby .....	45
3.1.5 Implementácia monitorovania pomocou databázy .....	50
3.2 Testovanie systému .....	52
3.2.1 Popis testovania.....	52
3.2.2 Výsledok testovania sekvenčného spracovania udalostí .....	54
3.2.3 Výsledok testovania paralelného spracovania udalostí bez MP .....	57

3.2.4	Výsledok testovania paralelného spracovania udalostí pomocou MP .....	59
3.2.5	Zhrnutie výsledkov testovania .....	63
<b>Záver</b>		<b>65</b>
<b>Zoznam použitej literatúry</b>		<b>67</b>
<b>Prílohy</b>		<b>I</b>
<b>A Nastavenie sprostredkovateľa správ</b>		<b>II</b>
<b>B Spustenie systému spracovania incidentov</b>		<b>VI</b>
<b>C Používateľská príručka</b>		<b>VII</b>

# Zoznam obrázkov a tabuliek

Obrázok 1 – Diagram monolitickej architektúry s jedným procesom[2].	16
Obrázok 2 – Diagram modulárnej monolitickej architektúry[2].	16
Obrázok 3 – Diagram architektúry mikroslužieb[2].	19
Obrázok 4 – Diagram tradičného procesu vývoja aplikácie[1].	22
Obrázok 5 – Diagram vývoja aplikácie podľa architektúry riadenej modelom[1].	23
Obrázok 6 – Diagram architektúry orientovanej na služby[2].	26
Obrázok 7 – Diagram architektúry riadenej udalosťami[5].	28
Obrázok 8 – Diagram sprostredkovateľskej topológie[5].	29
Obrázok 9 – Diagram topológie sprostredkovateľa správ[5].	31
Obrázok 10 – Diagram fungovania paternu SAGA[1].	33
Obrázok 11 – Návrh systému pre detekciu a spracovanie útokov.	39
Obrázok 12 – Ukážka Sysmon žurnálovej správy v XML formáte.	41
Obrázok 13 – Ukážka niekoľkých ASTD udalostí.	41
Obrázok 14 – Ukážka kódu kontroléra manažéra partícií.	43
Obrázok 15 – Diagram aktivít pre serverovú službu.	48
Obrázok 16 – Sekvenčný diagram serverovej aplikácie.	49
Obrázok 17 – Zobrazenie dát z tabuľky <b>local_security_incident</b> pomocou pgAdmin4.	51
Obrázok 18 – Zobrazenie dát z tabuľky <b>scanning_result</b> pomocou pgAdmin4.	51
Obrázok 19 – Konzola s výpisom testovacej aplikácie kolegu.	54
Obrázok 20 – Moja konzola s výpisom testovacej aplikácie.	55
Obrázok 21 – Dáta zo vstupnej témy zobrazené pomocou pgAdmin4.	56
Obrázok 22 – Dáta z výstupnej témy zobrazené pomocou pgAdmin4.	56
Obrázok 23 – Výpis zo serverovej konzoly spracovania udalostí niekoľkými službami.	57
Obrázok 24 – Pokračovanie výpisu zo serverovej konzoly.	58
Obrázok 25 – Výpis z konzoly spracovania udalostí niekoľkými službami.	60
Obrázok 26 – Pokračovanie výpisu z konzoly.	61
Obrázok 27 – Výpis z konzoly prvého kolegu.	61
Obrázok 28 – Výpis z našej konzoly.	61
Obrázok 29 – Výpis zo serverovej konzoly.	62

Obrázok 30 – Výsledný výpis z konzoly druhého kolegu.	62
Obrázok A.1 – Dostupný Apache Kafka klaster v Control Center.	II
Obrázok A.2 – Prehľad vytvorených tém v Kafka Control Center.	III
Obrázok A.3 – Zobrazenie novovytvorených tém pre sprostredkovateľa správ.	IV
Obrázok A.4 – Menu prepojovačov.	IV
Obrázok A.5 – Nastavenie prepojovača.	V
Obrázok A.6 – Okno s jedným spusteným prepojovačom.	V
Tabuľka 1 – Porovnanie výsledkov testovania rôznych typov spracovania udalostí.	63

# Zoznam skratiek a značiek

API – Application Programming Interface.

GUI – Grafical User Interface.

IT – Informačné Technológie.

NoSQL – Non relational SQL.

REST – Representational State Transfer.

RPC – Representational State Transfer.

SOAP – Simple Object Access Protocol.

RNI – Raw Native Interface.

JNI – Java Native Interface.

SQL – Structured Query Language.

XML – Extensible Markup Language.

# Úvod

V posledných rokoch rástlo množstvo dát, ktoré sa zbiera od používateľov pomocou rôznych aplikácií a servisov, približne o 20% ročne. Na konci roka 2021 obsahovalo množstvo dát okolo 79 zettabytov<sup>1</sup>. Pri takejto obrovskej kapacite dát rastie, samozrejme, aj záťaž pri ich preposielaní a spracovaní. Okrem toho novodobé biznis požiadavky potrebujú spracovanie dát v reálnom alebo k nemu približnom čase. Z toho vyplýva, že sa každú sekundu cez internetové prepojenie preposielajú enormné množstvo dát, ktoré má byť bezchybne doručené medzi rôznymi aplikáciami na spracovanie. Zároveň je v poslednej dobe v IT priemysle veľmi populárna architektúra pre vytvorenie aplikácií – Architektúra mikroslužieb (z angl. Microservice Architecture). Táto architektúra využíva na rozdiel od štandardných monolitných programov princíp rozdelenia aplikácie na viaceré menšie časti a princíp preposielania dát medzi nimi, čo ešte viac zvyšuje internetový beh dát. Pre vyššiu rýchlosť preposielania dát, spoľahlivosť ich doručenia a ich zjednotenie na jednom mieste pre vkladanie a vyberanie sa používa architektúra riadená udalosťami pomocou sprostredkovateľa správ (z angl. Message Broker).

Sprostredkovateľ správ prijíma dátá, stará sa o ich zálohovanie a dostupnosť pomocou náhradných partícií a dáva možnosť iným aplikáciám vyberať, spracovať a uložiť dátá späť do sprostredkovateľa na ďalšie spracovanie inými aplikáciami alebo preoslať dátá na trvalé uchovávanie do databáz. V našej práci budeme používať Apache Kafka ako implementáciu sprostredkovateľa správ, ktorá zohráva dôležitú úlohu pri spracovaní veľkého množstva dát.

Daná práca sa skladá z troch častí. Teoretická časť sa zaoberá architektúrami vytvorenia aplikácií od štandardnej monolitickej architektúry až po architektúru riadenú udalosťami. Ďalej rozoberá technológie, ktoré sú nevyhnutnou súčasťou nášho systému. Úvod do praktickej časti predstavuje popis a návrh systému pre spracovanie údajov a detekciu bezpečnostných incidentov na základe architektúry riadenej udalosťami. Neskôr je predstavená implementácia vyššie spomenutého systému. Praktická časť je ukončená testovaním aplikácie za pomoci simulovanej záťaže. Práca končí záverom, v ktorom sú zhrnuté výsledky práce.

---

<sup>1</sup>Zdroj: <https://firstsiteguide.com/big-data-stats>

# 1 Architektúry softvérových systémov

Prvá kapitola má za úlohu predstaviť moderné architektúry pre vývoj softvéru. Okrem popisu a porovnania architektúr posúdime pomocou ich výhod a nevýhod aj ich použiteľnosť pre spracovanie veľkého množstva dát v reálnom čase. Ako prvú rozoberieme klasickú monolitickú architektúru, ktorú pozná každý vývojár. Ďalej sa bližšie pozrieme na architektúru riadenú modelom, architektúru orientovanú na služby a architektúru mikroslužieb, ktorá má pri správnom implementovaní v kombinácii s inými architektúrami najväčší prínos pre čo najrýchlejšie spracovanie dát. Nakoniec rozoberieme architektúru riadenú udalosťami, ktorú napokon použijeme v našej implementácii systému pre spracovanie údajov a detekciu bezpečnostných incidentov.

## 1.1 Monolitická architektúra

V softvérovom inžinierstve sa monolitická architektúra označuje ako jedna nedeliteľná entita. Koncepcia monolitického softvéru spočíva v tom, že rôzne komponenty aplikácie sú spojené do jedného programu na jednej platforme. Monolitická aplikácia sa zvyčajne skladá z databázy, klientskeho používateľského rozhrania a serverovej aplikácie[1]. Všetky časti softvéru sú zjednotené a všetky jeho funkcie sa spravujú na jednom mieste. Monolitická architektúra je klasický a najjednoduchší spôsob vývoja softvéru[2]. V monolitickej aplikácii sú všetky komponenty vytvorené ako jedna kódová báza a nasadené ako jeden súbor. Ak je potrebné aktualizovať alebo zmeniť nejakú časť kódu, tieto zmeny nie je možné nasadiť samostatne. Vývojár musí použiť rovnakú kódovú bázu, vykonať potrebné zmeny kódu a potom aktualizovaný kód znova nasadiť. Týmto prístupom aj v prípade potreby najmenšej zmeny bude celá aplikácia ovplyvnená a preto sa celá musí rozmiestniť na server alebo oblačné platformy. Samotná monolitická architektúra je rozdelená na dva podtypy: na monolitickú architektúru s jedným procesom a modulárnu monolitickú architektúru[1].

Monolitická architektúra s jedným procesom, ktorá je zobrazená na obrázku č. 1, predstavuje spôsob vytvárania softvérových produktov, podľa ktorého má aplikácia jeden hlavný proces, ktorý zahŕňa celú funkcionality produktu[1]. Väčšina klasických aplikácií v priemysle informačných technológií je založená práve na tejto architektúre.

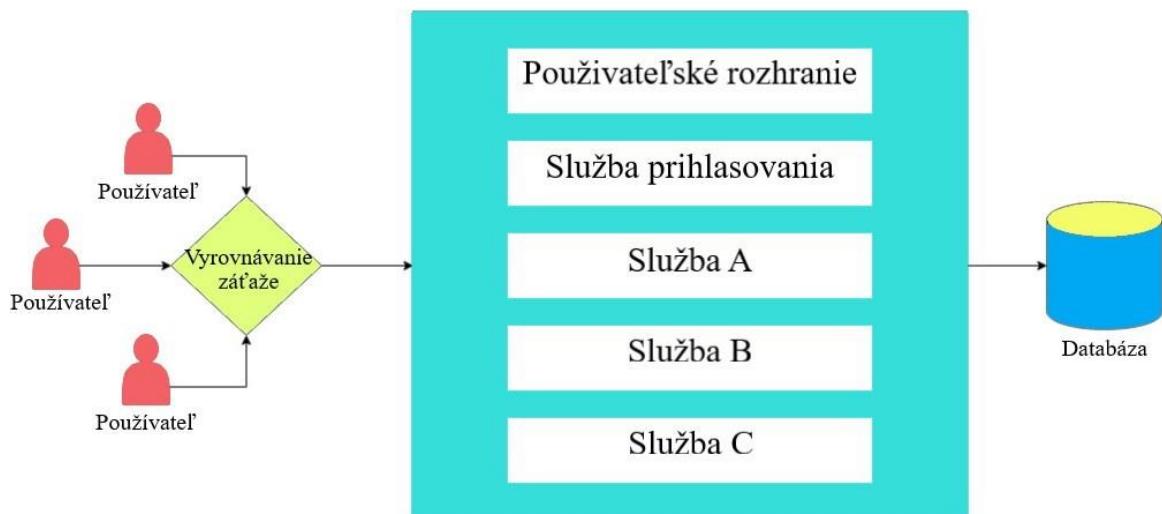
## Monolitická architektúra s jedným procesom



Obrázok 1 – Diagram monolitickej architektúry s jedným procesom[2].

Modulárna monolitická architektúra, ktorá je zobrazená na obrázku č. 2, je architektúrou, v ktorej je hlavný monolitický proces rozdelený do niekoľkých modulov alebo služieb. Každý z týchto modulov alebo služieb môže pracovať nezávisle. Moduly majú rozhrania a prostredníctvom nich môžu navzájom komunikovať. Všetky moduly používajú rovnakú databázu pre svoje operácie. Napriek tomu je potrebné spojiť všetky moduly do jedného súboru za účelom rozmiestnenia na server[1].

## Modulárna monolitická architektúra



Obrázok 2 – Diagram modulárnej monolitickej architektúry[2].

Hlavné výhody monolitickej architektúry sú:

- Jednoduchý vývoj a spustenie programu a produktu.

Vďaka tomu, že celý vývoj je sústredený na jednom mieste, je jednoduchšie integrovať vývojové nástroje a používať spoločné triedy, moduly alebo knižnice. Pri zmene správania programu nie sú potrebné zmeny na rôznych miestach – všetko sa vykonáva z jediného miesta. Tak isto prebieha iba na jednom mieste aj spustenie programu, vďaka čomu nie je potrebné medzi sebou skoordinovať niekoľko aplikácií alebo služieb[2].

- Krížové vývojové problémy prakticky nie sú.

Veľký počet aplikácií, modulov alebo ich inštancií je závislý od úloh, ktoré sa vykonávajú medzi programovými komponentmi: logovanie, riešenie komunikácie a preposielanie dát medzi aplikáciami, rozdelenie transakcií a sietové oneskorenia. Pri monolitickej architektúre tieto problémy prakticky nie sú, pretože sa všetko sústreďuje v jednom kóde a všetko beží v jednej aplikácii[2].

- Väčší výkon.

Tá istá aplikácia, ktorá bola implementovaná podľa monolitickej architektúry, bude mať väčší výkon ako taká istá aplikácia, ktorá bola implementovaná podľa architektúry mikroslužieb. To je zabezpečené jednotným kódom programu a jeho vykonaním z jedného miesta bez sietových oneskorení. Takéto sietové oneskorenie veľmi ovplyvňuje výkon aplikácií a služieb vytvorených podľa architektúry mikroslužieb[2].

Ako nevýhody monolitickej architektúry sa prezentujú:

- Veľké množstvo kódu.

Ak je vyvíjaný produkt pomerne veľký a neustále škálovateľný, jeho kód časom prerastá do obrovských rozmerov. To stáže jeho pochopenie a údržbu. Navyše môže nastáť situácia, kedy sa kód stane "preťaženým" a stratí na kvalite, tým, že každá zmena bude vplývať na celý program negatívne[2].

- Náročnosť pridávania nových funkcionálít alebo zmien.

Častokrát je v priebehu údržby softvérového produktu potrebné do neho pridať nejakú novú funkciu. V prípade monolitickej architektúry existuje veľké množstvo obmedzení pri realizovaní danej činnosti vo veľkých aplikáciách. V niektorých prípadoch je pridanie nejakej funkcionality alebo zmeny realizovateľné iba pomocou kompletného prepísania aplikácie, čo je pre firmy časovo a finančne príliš náročné[1].

- Obmedzené opäťovné použitie.

Z predchádzajúceho bodu vyplýva, že pridanie niečoho nového do aplikácie je od určitého bodu veľmi náročné. Z tohto dôvodu je pri vytváraní nových aplikácií použitie predchádzajúceho podobného monolitického riešenia časovo a finančne takisto náročné, ako aj vytvorenie ďalšieho produktu od nuly. Preto sa nové monolitické aplikácie častejšie vytvárajú nanovo a nie sú použiteľné pre nasledujúce projekty, ako napríklad moduly mikroslužieb, ktoré sa dajú jednoducho opäťovne použiť[1].

- Silná závislosť a viazanosť medzi komponentmi.

Pri veľkej závislosti medzi komponentmi sú ovplyvnené ostatné časti programu každou novou zmenou, v dôsledku čoho vznikajú ľahko odhaliteľné chyby v softvérovom produkte[1].

- Malá odolnosť voči poruchám.

Najväčší problém monolitických aplikácií je, že pri vzniknutí hocijakej nepredvídateľnej chyby v ľubovoľnom mieste zlyháva celý softvérový produkt. Aplikácia sa väčšinou nachádza na jednom serveri, čo znamená, že aj pri fyzickom zlyhaní serveru budú aplikácia a databáza mimo prevádzky[1].

- Slabá škálovateľnosť.

Jedinou možnosťou reagovania pri zvýšení záťaže na monolitickú aplikáciu je zvyšovanie hardvérových kapacít, čo je vertikálna škálovateľnosť. Avšak vertikálna škálovateľnosť nemôže rást donekonečna a veľmi rýchlo sa stretne s fyzickými obmedzeniami serveru[1]. Horizontálna škálovateľnosť (vytváranie niekoľkých inštancií aplikácie), ktorá je výkonnejšia ako vertikálna, je pre monolitickú aplikáciu nedostupná.

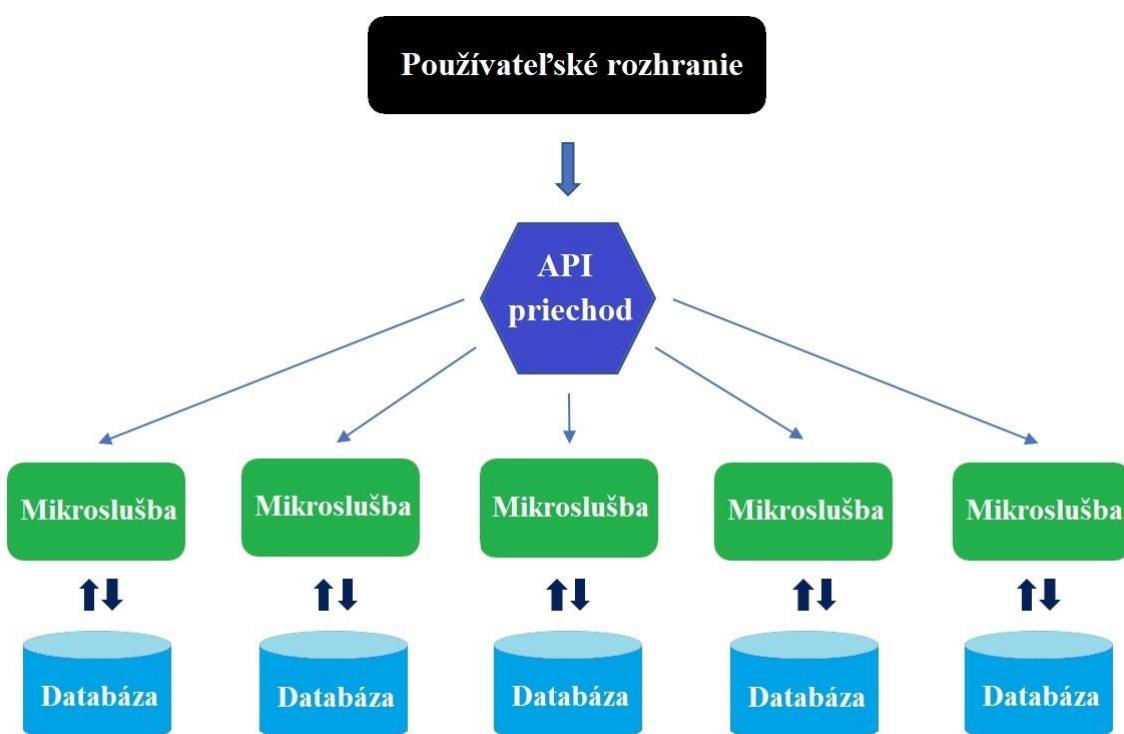
Táto architektúra je ideálna pre *start-up*, ktorý je finančne, časovo a technicky obmedzení a potrebuje čo najskôr vytvoriť hotový softvérový produkt a uviesť ho do prevádzky. Takisto sa hodí aj pre malé aplikácie, ktoré nemajú rozsiahlu biznis logiku a v blízkej budúcnosti nebudú potrebovať veľkú škálovateľnosť. Avšak pre našu aplikáciu je daná architektúra z dôvodu slabej škálovateľnosti, reakcií na zväčšenú záťaž a nepostačujúcej odolnosti voči poruchám nepoužiteľná[1].

## 1.2 Architektúra mikroslužieb

V dnešnej dobe pozná architektúru mikroslužieb každý programátor, ktorý pracuje v IT oblasti aspoň pár mesiacov. Ide o moderný trend, ktorý sice vznikol ešte v roku 2000, ale svoju najväčšiu popularitu začal získavať v rokoch 2012 až 2014, kedy obrovské firmy

ako Amazon, Netflix a Twitter začali prezentovať svoje aplikácie, ktoré sú postavené podľa tejto architektúry, vytvárať rôzne konferencie ohľadom mikroslužieb a vydávať nove knihy (napr. „Building Microservices“ od spisovateľa Sama Newmana, „Microservices Patterns“ od spisovateľa Chrisa Richardsona)[1].

Architektúra mikroslužieb predstavuje prístup k vytváraniu softvérových produktov, ktorý odmieta jedinú monolitickú štruktúru pre aplikáciu. To znamená, že namiesto toho, aby softvérový produkt pozostával z jednej aplikácie, pozostáva z viacerých menších aplikácií, ktoré sú medzi sebou prepojené nejakým rozhraním, ako napríklad REST, gRPC, GraphQL alebo sietovým volaním http, a pracujú ako jeden celok[2]. Každá mikroslužba musí byť najmenším logickým celkom, ktorý vykonáva jednu určitú funkciu alebo je zodpovedná za jednotlivú funkcionalitu systému. Na obrázku č. 3 je znázornený diagram architektúry mikroslužieb.



Obrázok 3 – Diagram architektúry mikroslužieb[2].

Takýto prístup k tvorbe softvérových produktov rieši niekoľko problémov monolitných produktov:

- Flexibilnosť softvérového produktu.

Mikroslužbu sa dá ľahko nahradí novšou verziou a osamostatniť vývoj a verziovanie od iných mikroslužieb vtedy, keď každá malá zmena môže v monolitnej aplikácii vytvoriť problémy a veľmi ovplyvniť ostatné časti softvérového produktu[2].

- Jednoduchosť samostatnej mikroslužby.

Každá mikroslužba je samostatný celok, ktorý obsahuje oveľa menej kódu ako monolitná aplikácia. Preto je aj pochopenie jej funkcionality, opravovanie chýb a vylepšenie danej mikroslužby oveľa rýchlejšie a jednoduchšie ako podobné operácie v monolitnej aplikácii[2].

- Vysoká horizontálna aj vertikálna škálovateľnosť.

Pomocou vytvárania alebo vymazávania niekoľkých inštancií jednej mikroslužby dokáže systém okamžite reagovať na zmenu záťaže. Vďaka tomu dokáže používať horizontálnu škálovateľnosť, zatiaľ čo jeho monolitný konkurent môže ovplyvniť iba vertikálne škálovanie[2].

- Odolnosť voči poruchám.

Ak jedna inštancia mikroslužby padne alebo prestane komunikovať, systém dokáže pracovať pomocou iných inštancií, čo robí systém mikroslužieb v porovnaní s monolitným systémom odolný voči poruchám. Zároveň sa jednotlivé mikroslužby môžu nachádzať na rôznych serveroch po celom svete vtedy, keď celá monolitná aplikácia beží iba na jednom serveri a pri jeho vypadnutí prestane fungovať[1].

Samozrejme, architektúra mikroslužieb má okrem výhod aj veľa nevýhod:

- Komunikácia medzi mikroslužbami.

Komunikácia medzi mikroslužbami je zložitý proces. Okrem preposielania informácií sa majú mikroslužby ešte aj starať o jednotné označenie týchto informácií v celom systéme za účelom logovania, hľadania chýb a spracovania distribuovaných transakcií. Pri zväčšení množstva mikroslužieb rastie aj zdržiavanie pri preposielaní informácií cez všetky potrebné mikroslužby[1].

- Testovanie mikroslužieb je náročnejšie ako testovanie monolitu.

Pri integračnom testovaní mikroslužieb je potrebné vytvárať namiesto reálnych mikroslužieb veľké množstvo *mock* objektov, aby sme každú mikroslužbu správne otestovali. Pri zvýšenom množstve mikroslužieb rastie aj komplexnosť systémového testovania[1].

- Architektúra mikroslužieb je príliš náročná pre mále projekty.

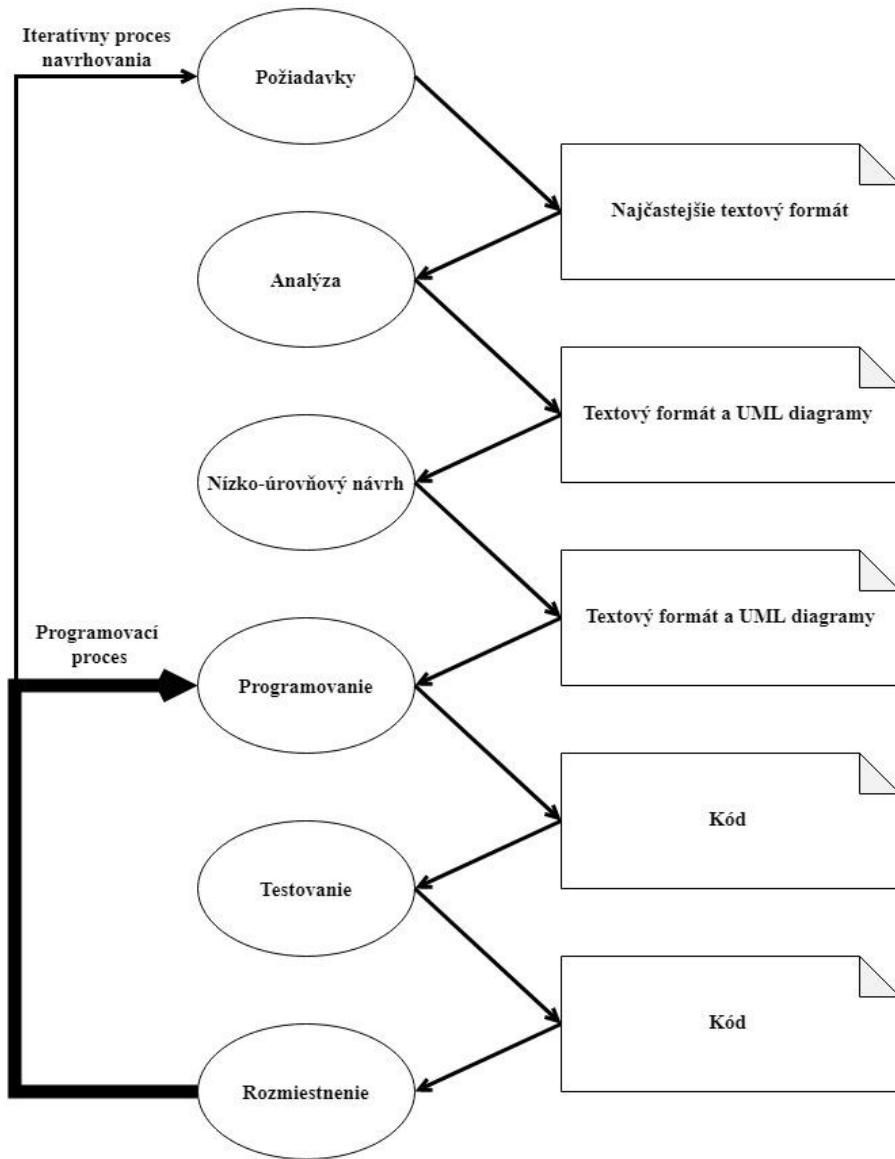
Pre vytvorenie menšieho alebo vnútorného softvérového projektu je oveľa rýchlejšie a jednoduchšie navrhnúť a implementovať monolitný systém, než pracovať na niekoľkých mikroslužbách a navyše aj na ich komunikácii, distribuovanom logovaní, orchestrácií a monitorovaní rôznych inštancií každej mikroslužby[1].

Avšak architektúra mikroslužieb je v poslednom desaťročí veľmi podporovaná zo strany vývoja otvoreného softvéru (z angl. Open-Source), ktorý sa snaží zjednodušiť jej implementovanie a vyriešiť niektoré vyššie uvedené problémy. Ide predovšetkým o softvér Docker, ktorý bude v práci neskôr podrobne rozoberaný. Taktiež si daná architektúra zaslúžila svoju popularitu aj riešením hlavného problému pri vytváraní softvérových produktov – riešením častej zmeny technických požiadaviek od zákazníkov[1]. Čím je mikroslužba menšia, tým sú v nej zmeny implementácie jednoduchšie a rýchlejšie. Zároveň pri vopred stanovenej API mikroslužby, zmena nejakej funkcionality neovplyvňuje prácu ostatných mikroslužieb, čo sa často stáva v monolitných aplikáciach[1]. Samotné mikroslužby vedia pomôcť s vyššie spomenutými problémami. Pre maximálnu efektivitu sa však architektúra mikroslužieb využíva aj pri ďalších architektúrach, ktoré popisujú, ako majú byť mikroslužby navrhnuté, ako majú komunikovať medzi sebou a načo sa treba zameriavať pri navrhovaní softvérového produktu[2].

### 1.3 Architektúra riadená modelom

Architektúra riadená modelom (z angl. Model-Driven Architecture – MDA) je architektúrny prístup k vytváaniu multikomponentného softvéru, ktorý je založený na vývoji reprezentácií (modelu) systému nezávislej od platformy a programovacieho jazyka s nasledujúcim prechodom na zdrojový kód systému prostredníctvom automatizovaného generovanie kódu. Vývoj tejto architektúry sa začal v roku 2000[1]. Hlavným cieľom architektúry riadenej modelom bolo vytvoriť prístup k vytvoreniu softvérových produktov, ktorý by znížil riziko spôsobené rozdielmi medzi jednotlivými technológiami a platformami, na ktorých by sa softvérový produkt používal. Pre riešenie tohto problému sa v danej architektúre zaviedol pojem model nezávislý od platformy (z angl. Platform Independent Model – PIM) a model závislý od platformy (z angl. Platform Specific Model – PSM)[1].

## Tradičný proces vývoja aplikácie

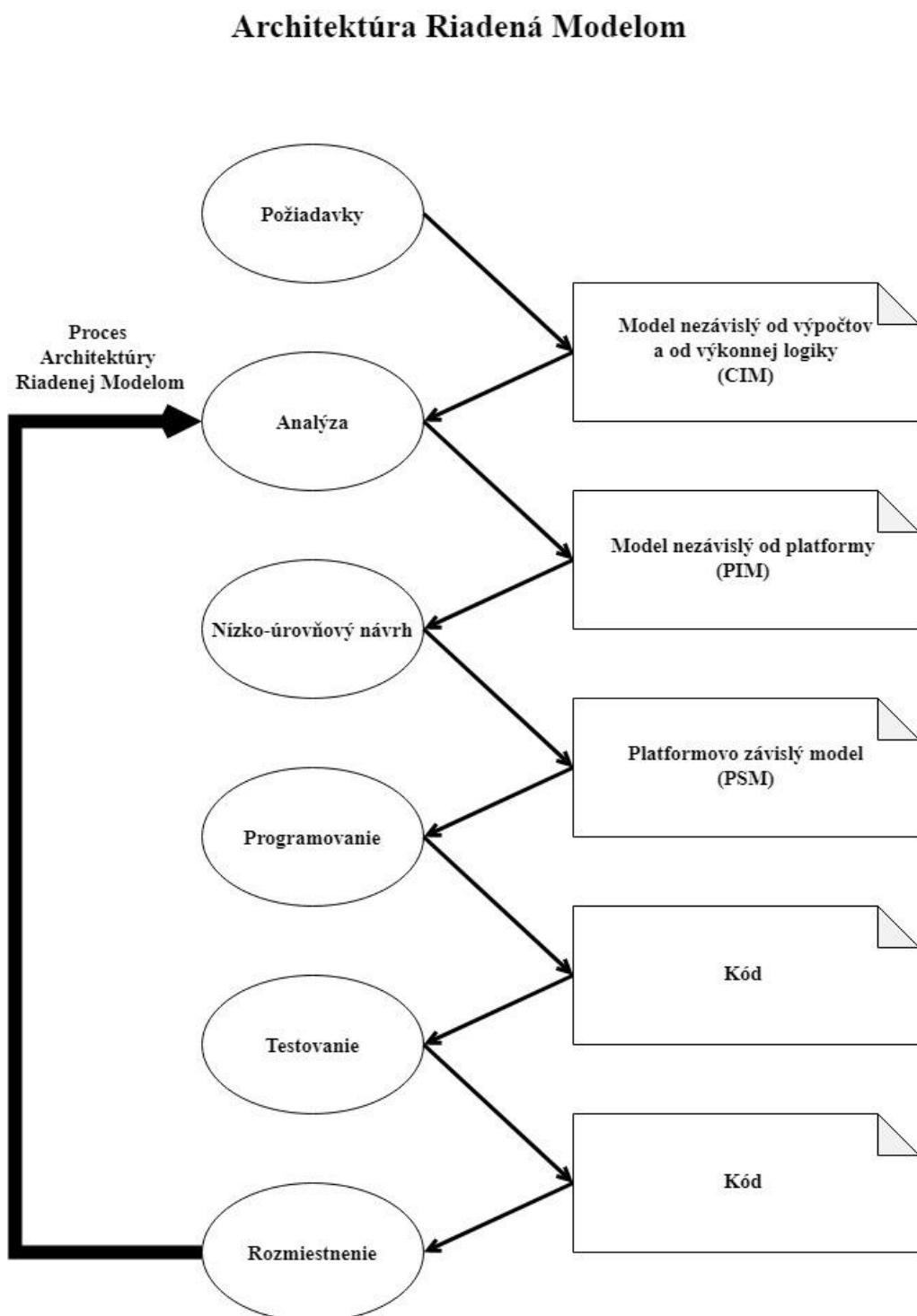


Obrázok 4 – Diagram tradičného procesu vývoja aplikácie[1].

Obrázok č. 4 zobrazuje tradičný proces vývoja aplikácie, ktorého prvou etapou je špecifikovanie požiadaviek, ktoré sú zadefinované pomocou textu. Ďalej sú tieto požiadavky až do etapy programovania spracované pomocou textu a UML diagramov[1].

Pri vytvorení aplikácie pomocou architektúry riedenej modelom, ktorej diagram je zobrazený na obrázku č. 5, sú požiadavky po ukončení etapy špecifikovania požiadaviek zadefinované pomocou modelu nezávislého od výpočtov a výkonnej logiky (z angl.

Computation Independent Model – CIM), čo sú obyčajne popisy domény. Tento model je ďalej transformovaný do ďalších modelov.



Obrázok 5 – Diagram vývoja aplikácie podľa architektúry riadenej modelom[1].

Nasledujúci model po vykonaní etapy analýzy je PIM, ktorý popisuje systém a jeho domény nezávisle od platformy a programovacích technológií. To znamená, že daný model používa iba elementy, ktoré sú podporované akoukoľvek platformou. Na základe platformovo nezávislého modelu sa vytvárajú všetky nasledujúce platformovo závislé modely, ktoré špecifikujú model pre zvolené technológie a pridávajú prvky, ktoré sú potrebné pre zvolenú platformu[1]. Poslednou časťou vývoja aplikácie je generovanie kódu v zvolenom programovacom jazyku pre špecifikovanú platformu pomocou platformovo závislého kódu[1].

Ako výhody danej architektúry sa dajú uviesť automatizované generovanie kódu, jednotný štýl vygenerovaného kódu, generovanie kódu pre rôzne platformy a technológie, zrýchlené vytváranie systémov a ľahšie pridanie zmien do systému pomocou uvedenia zmien do modelov[1]. Daná architektúra má však aj nevýhody. Kvôli zložitosti vytvorenia modelov, rôznym obmedzeniam aplikácií, ktoré generujú kód pomocou modelov, kvôli nepriehľadnosti a zložitosti generovaného kódu a zložitosti pridania nových podporovaných platořiem a technológií sa architektúra riadená modelom vo väčšine prípadov nepoužíva na spracovanie veľkých dát v reálnom čase[1].

## 1.4 Architektúra orientovaná na služby

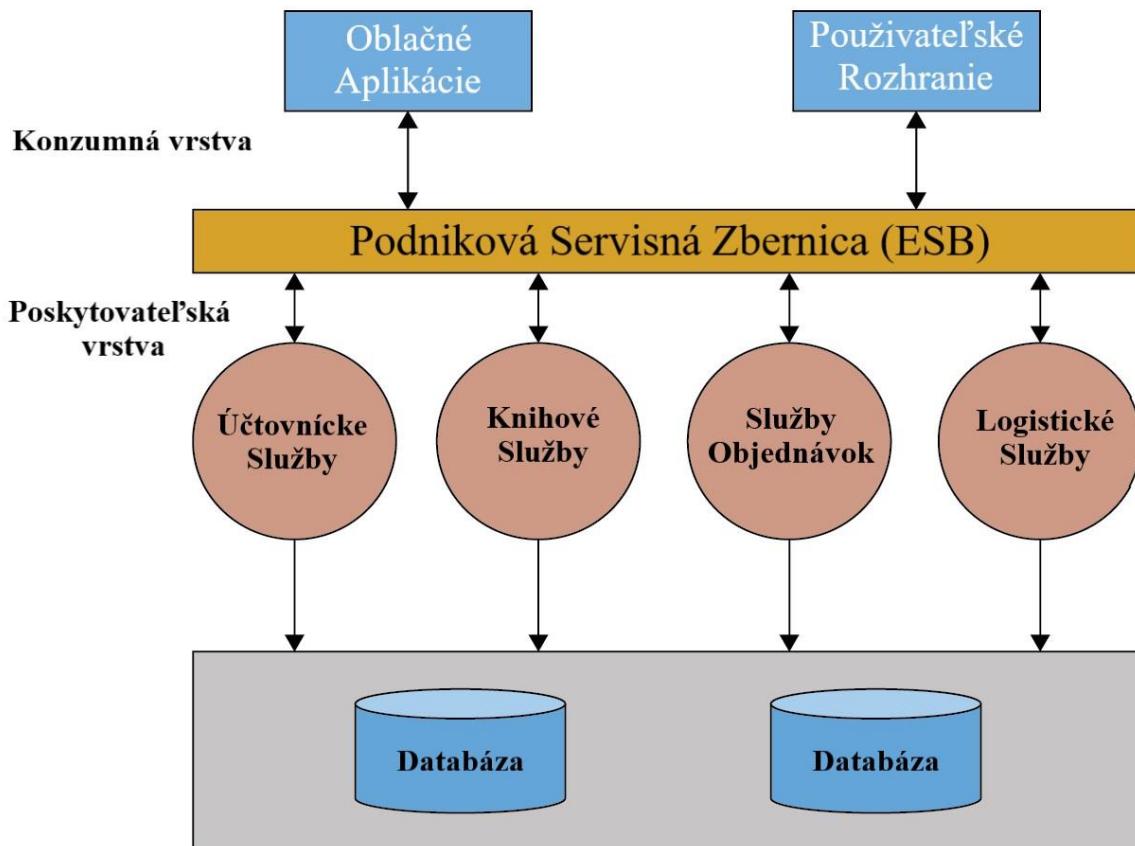
Architektúra orientovaná na služby (z angl. Service Oriented Architecture – SOA) je architektúrny prístup, ktorý je založený na pojme „služba“ ako základnej jednotke návrhu systému[1]. Daná architektúra ponúka spôsob opakovaného použitia softvérových komponentov prostredníctvom rozhrania služieb[1]. Tieto rozhrania využívajú spoločné komunikačné štandardy, čo je dôvodom, prečo ich je možné rýchlo opäťovne použiť v nových aplikáciách bez toho, aby sa museli zakaždým dôkladne integrovať. Každá služba v tejto architektúre obsahuje kód a integráciu údajov potrebných na vykonanie konkrétnej obchodnej funkcie, ako je napríklad kontrola prihlásovacích údajov, výpočet zľavy alebo spracovanie objednávky. Služby sú zostavované tak, aby ich vývojári mohli rýchlo nájsť a opäťovne použiť pri vytváraní nových aplikácií[4]. Táto architektúra nie je viazaná na konkrétnu technológiu. Môže byť implementovaná pomocou širokej škály technológií vrátane REST, RPC, SOAP, RNI, JNI alebo webových služieb. SOA môže byť implementovaná pomocou niektorého z týchto protokolov, ich kombináciou a môže napríklad dodatočne používať mechanizmus súborového systému pre výmenu údajov[4].

Klúčovou vecou, ktorou sa SOA odlišuje, je používanie nezávislých služieb s presne definovanými rozhraniami, ktoré možno zavolať určitým štandardným spôsobom, aby vykonali svoju úlohu za predpokladu, že služby nevedia nič o aplikácii, ktorá ich zavolá, a aplikácia nevie, ako presne služby svoje úlohy vykonávajú[3]. SOA predpokladá, že softvérový produkt sa bude používať ako poskytovateľ služieb alebo spotrebiteľ služby. Poskytovateľ implementuje určité rozhranie a spotrebiteľ prostredníctvom neho využíva možnosti služieb daného softvéru[1].

Podniková Servisná Zbernice (z angl. Enterprise Service Bus – ESB) je úzko spojená s architektúrou SOA. ESB je šablóna, v ktorej centralizovaný komponent vykonáva integrácie so základnými systémami a potom tieto integrácie sprístupňuje ako rozhrania služieb. Poskytuje konverziu dátového modelu, úzke prepojenie, smerovanie a dokonca aj vytváranie viacnásobných dotazov, pričom tieto funkcie spája do jedného rozhrania služby, ktoré je možné opäťovne použiť v nových aplikáciach[1]. Šablóna ESB sa zvyčajne implementuje pomocou špeciálne navrhnutého prostredia na vykonávanie integrácie a nástrojov, ktoré sú vhodné na čo najefektívnejšie vykonávanie uvedených funkcií[3]. Softvérové produkty založené na architektúre orientovanej na služby tak môžu byť nezávislé od vývojových technológií a platforiem, ako sú napríklad Java a .NET. Napríklad služby napísané v jazyku C# bežiace na platformách .Net a služby napísané v jazyku Java bežiace na platformách Java podnikovej edície Java EE (z angl. Java Enterprise Edition – Java EE) môžu byť rovnako úspešne volané cez podnikovú servisnú zbernicu. Aplikácie bežiace na jednej platforme môžu volať služby bežiace na iných platformách, čo uľahčuje opakované použitie komponentov[2].

Teoreticky by sa architektúra orientovaná na služby dala implementovať bez ESB, ale vývojári by museli nájsť jednotlivé spôsoby, ako zabezpečiť prístup k rozhraniam – čo je časovo veľmi náročná úloha (dokonca aj pri opakovane použiteľných rozhraniach), ktorá by taktiež veľmi stážila budúcu údržbu. V skutočnosti sa ESB nakoniec začala považovať za neoddeliteľný prvok každej implementácie SOA a tieto pojmy sa často používajú ako synonymá[2].

## Architektúra orientovaná na služby



Obrázok 6 – Diagram architektúry orientovanej na služby[2].

Na obrázku č. 6 je znázornený diagram architektúry SOA, ktorý sa realizuje pomocou použitia ESB.

Veľmi často architektúru SOA porovnávajú a si mylia s architektúrou mikroslužieb. Tieto architektúry majú veľa spoločných prvkov, ale zároveň sa aj značne od seba odlišujú. Prvým dôležitým rozdielom je použitie služieb[4]. Podľa SOA architektúry sa služby snažia byť univerzálne, aby sa dali opäťovne použiť aj pri iných aplikáciach. Podľa architektúry mikroslužieb je duplicita niektorých prvkov lepšou možnosťou, pretože každý deň môže prísť zmena požiadaviek alebo realizácie týchto prvkov[4].

Hlavné výhody danej architektúry sú:

- Opäťovné použitie služieb, ktoré veľmi urýchľuje vytváranie nových softvérových produktov[4].

- Jednoduchosť údržby produktu vďaka tomu, že každá softvérová služba je nezávislá entita, ktorú sa dá ľahko aktualizovať, opravovať a vylepšovať bez toho, aby ovplyvnila ostatné služby[4].
- Paralelný vývoj, ktorý je zabezpečený tým, že služby sú nezávislé entity a môžu byť vytvárané alebo testované súčasne[4].

Ako nevýhody danej architektúry sa dajú uviesť:

- Komplexnosť riadenia behu aplikácie, ktorá vzniká v dôsledku toho, že veľa služieb môže paralelne odosielat' veľké množstvo dát. Počet týchto odoslaných dát môže presiahnuť milión za sekundu, čo sťaže riadenie všetkých služieb[4].
- Vysoké investičné náklady, ktoré vznikajú v dôsledku zložitosti danej architektúry. Vývoj softvérového produktu si podľa SOA vyžaduje značné počiatočné investície pre dôkladný návrh a realizáciu tohto produktu[4].
- Dodatočné zaťaženie pri spracovaní dát. Podľa architektúry SOA sa všetky vstupné dáta kontrolujú predtým, ako jedna služba bude komunikovať s inou službou. Ak sa používa viacero služieb, predĺžuje sa tým čas odozvy a znižuje sa celkový výkon softvérového produktu[4].

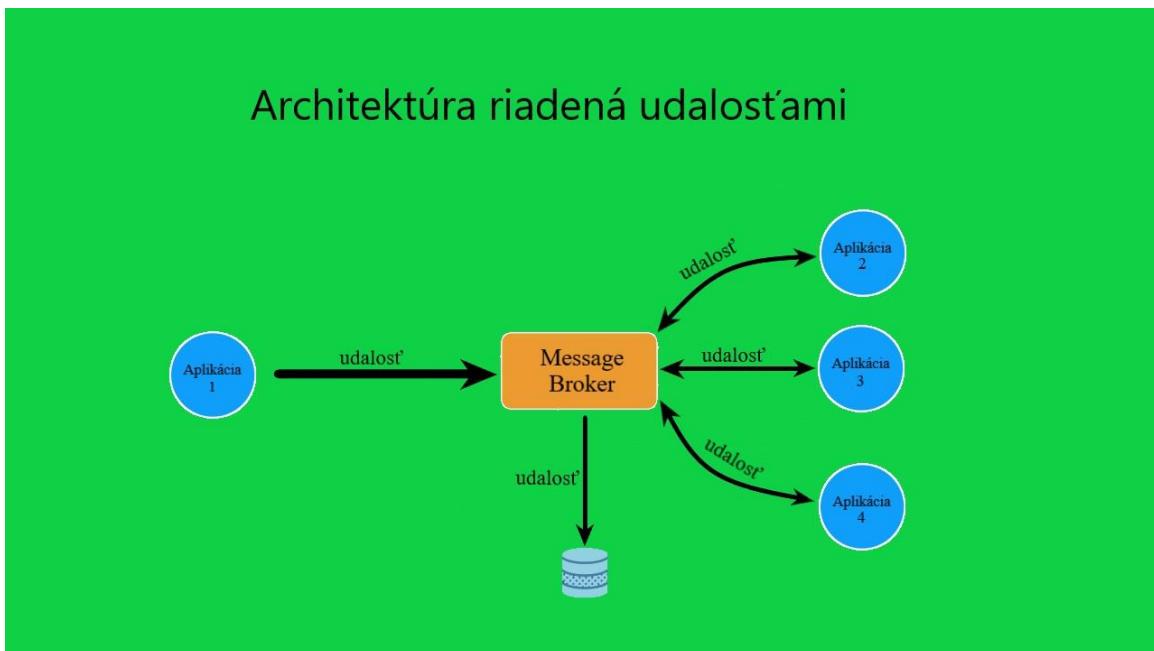
Táto architektúra je najvhodnejšia pre komplexné podnikové systémy, ako sú napríklad bankové systémy. Bankový systém je veľmi ťažké rozdeliť na mikroslužby. Monolitický prístup je však nevhodný aj pre bankový systém, pretože jedna časť môže negatívne ovplyvniť celú aplikáciu. Najlepším riešením v takomto prípade je architektúra SOA a organizácia komplexných aplikácií do izolovaných, nezávislých služieb. Danú architektúru však kvôli spracovaniu veľkého množstva dát v reálnom čase, komplexnosti a dodatočnému zaťaženiu pri spracovaní dát nebudeme používať.

## 1.5 Architektúra riadená udalosťami

Architektúra riadená udalosťami (z angl. Event-Driven Architekture – EDA) je paradigma softvérovej architektúry, ktorá umožňuje generovanie, detekciu, spotrebú a reakciu na udalosti[5]. Táto architektúra zvyčajne integruje niekoľko rôznych systémov alebo aplikácií pomocou radu správ. V klasických monolitných aplikáciách je jadrom systému databáza. V architektúre riadenej udalosťami sa pozornosť presúva na udalosti a ich tok v systéme. Udalosť je akcia, ktorá iniciuje bud' nejaké oznámenie alebo zmenu

stavu aplikácie, ako napríklad vytvorenie objednávky zákazníkom, registráciu užívateľa alebo pridanie nových dát do systému[5].

Udalosti sa vyskytujú ako výsledok akcie, čo znamená, že pre nich v danej architektúre neexistuje cieľový systém. Z obrázku č. 7 sa nedá s určitosťou odčítať, či aplikácia 1 posielala udalosti práve aplikácii 2. S jednoznačnosťou sa dá však povedať, že aplikácia 2 odoberá udalosti generované aplikáciou 1. Zároveň na tie isté udalosti môže v danom prípade reagovať aj aplikácia 3 a aplikácia 4.



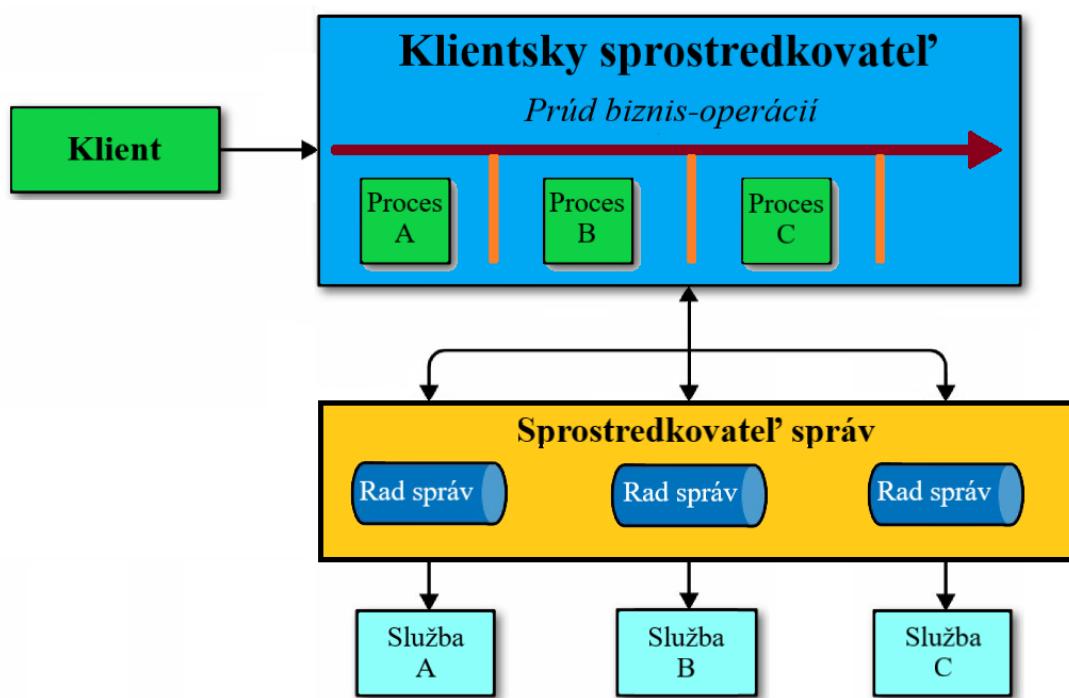
Obrázok 7 – Diagram architektúry riadenej udalosťami[5].

Tok udalostí je jednosmerný: od výrobcu udalosti k spotrebiteľovi. Pri porovnaní takejto architektúry s obyčajným volaním REST medzi mikroslužbami sa hned objavujú jej výhody. Producent udalosti neočakáva žiadnu odozvu od konzumenta, zatiaľ čo v prípade volania REST sa vždy očakáva spätná odpoveď. Žiadna odozva znamená, že nie je potrebné blokovať vykonávanie kódu, kým sa nestane niečo iné. Udalosti sa tým pádom vykonávajú asynchronne, čím sa úplne eliminuje riziko veľkého oneskorenia systému[1].

Existujú dve rôzne realizácie danej architektúry: návrhový vzor sprostredkovateľ správ, ktorý bol zobrazený na obrázku č. 7 a návrhový vzor sprostredkovateľ (z angl. Mediator Pattern)[1].

### 1.5.1 Realizácia pomocou návrhového vzoru sprostredkovateľ

Realizácia architektúry riadená udalosťami pomocou vzoru sprostredkovateľ alebo sprostredkovateľskej topológie je určená pre riadenie prúdu udalostí, ktorý sa skladá z viacerých krokov a má komplexnú biznis logiku[5]. Počas behu softvérového produktu môžu byť udalosti kombinované medzi sebou, transformované do iných udalostí alebo sa na nich môžu vykonávať nejaké operácie. Klientsky sprostredkovateľ určuje poradie spúšťaných procesov, sekvenčný alebo paralelný spôsob vykonávania procesov a určuje, ktorý rad správ sa má použiť ako vstupný a ktorý ako výstupný[5].



Obrázok 8 –Diagram sprostredkovateľskej topológie[5].

Ako vidno na obrázku č. 8, architektúra zvyčajne zahŕňa štyri typy komponentov: sprostredkovateľa správ, rad správ, klientskeho sprostredkovateľa a služby, ktoré spracovávajú udalosti[5]. Klient vytvorí udalosť a odošle ju do klientskeho sprostredkovateľa, ktorý na základe biznis logiky určí, do ktorého radu správ sprostredkovateľa správ má danú udalosť odoslať. Z radu správ bude udalosť prečítaná vhodnej službou, ktorá po spracovaní odošle novú alebo zmenenú udalosť do výstupného radu správ, odkiaľ si ju spisťočne prečíta klientsky sprostredkovateľ[5]. Klientsky sprostredkovateľ vykonáva v danej architektúre podobnú úlohu ako aj podniková servisná

zbernice v architektúre orientovanej na služby. Akékoľvek spracovanie udalostí, ktoré potrebuje komunikovať s iným procesom, sa vykonáva prostredníctvom klientskeho sprostredkovateľa. Aj keď klientsky sprostredkovateľ kontroluje koordináciu procesov, táto architektúra stále umožňuje, aby sa väčšina spracovania udalostí vykonávala súčasne[6].

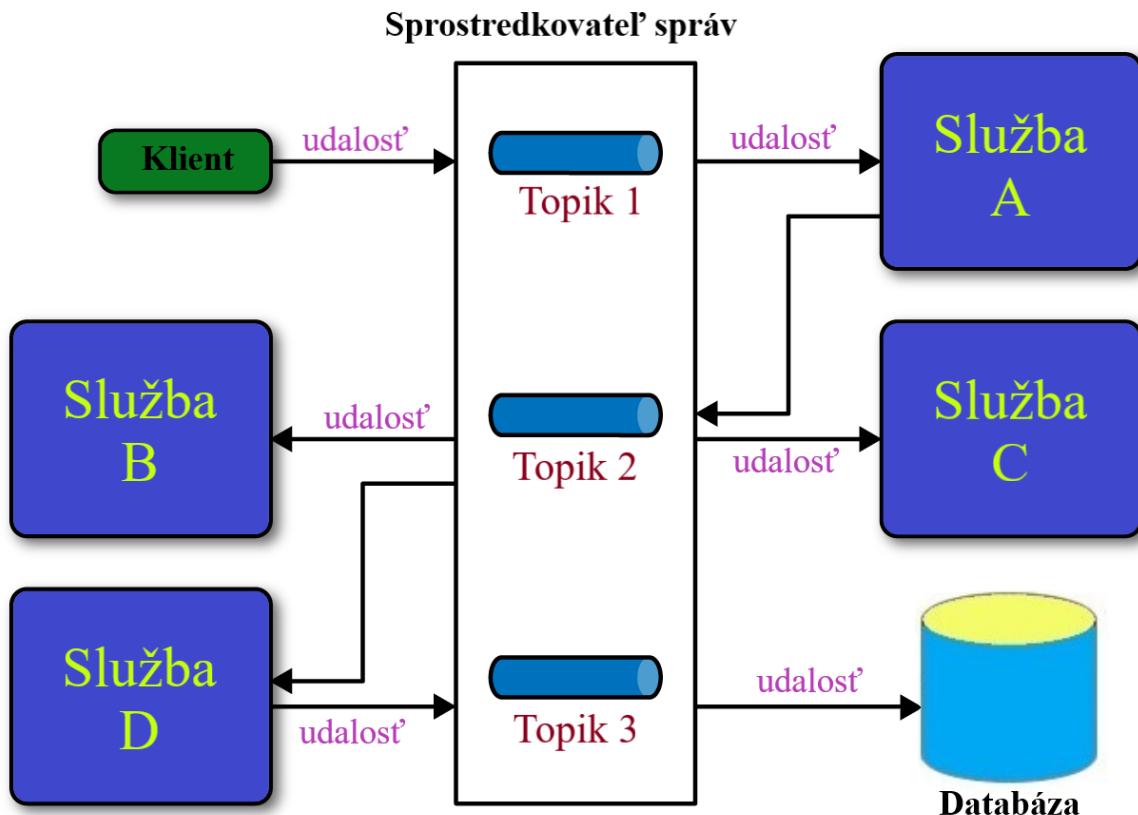
Schopnosť koordinovať transakcie je hlavnou výhodou sprostredkovateľskej topológie. Sprostredkovateľ môže kontrolovať stav vykonania prúdu biznis operácie a pri evidovaní chyby priamo zruší celú transakciu spracovania udalostí. V prípade architektúry realizovanej pomocou sprostredkovateľa správ je tento typ koordinácie zložitejší[6]. Ďalšia výhoda danej realizácie je zjednodušenie riadenia procesmi, ktoré je pomocou klientskeho sprostredkovateľa oveľa flexibilnejšie a medzi spustením daných procesov môže byť jednoducho pridaná ďalšia biznis logika, logovanie, monitorovanie stavu vykonania prúdu biznis operácií, zálohovanie nejakých dát alebo špeciálne šifrovanie. Rýchla odozva a možnosť asynchronného a distribuovaného spracovania veľkého množstva udalostí aj pomocou využitia niekoľkých inštancií služieb je tak, ako v architektúre mikroslužieb, ďalšou výhodou danej realizácie[6].

Hlavnou nevýhodou realizácie topológie sprostredkovateľa je jej komplexnosť a vysoká viazanosť klientskeho sprostredkovateľa. Pri veľkých, zložitých a zle navrhnutých softvérových produktoch sa stáva klientsky sprostredkovateľ monolitickou časťou, v ktorej môže každá ďalšia zmena viest k novým problémom a defektom. Zároveň sa klientsky sprostredkovateľ pri veľmi vysokej záťaži môže stať úzkym miestom aplikácie[6]. Ďalšia veľká nevýhoda danej realizácie je, že je slabo odolná voči poruchám. Ak sa ešte dajú jednotlivé služby nahradíť inými inštanciami, tak sa softvérový produkt pri vzniknutí poruchy vnútri klientskeho sprostredkovateľa môže stať nepoužiteľným.

Dána realizácia vyhovuje veľkým korporačným softvérovým projektom so zložitou vnútornou logikou a potrebou orchestrácie prúdu biznis operácií[6]. Pre malé projekty a pri malých technických úlohách je daná realizácia príliš zložitá. Čo sa týka odolnosti voči poruchám, topológia sprostredkovateľa sa pri kriticky dôležitých aplikáciách, ako napríklad v oblasti medicíny alebo riadenia nejakých technických procesov, takmer nepoužíva[6].

### 1.5.2 Realizácia pomocou návrhového vzoru sprostredkovateľ správ

Najpopulárnejšou realizáciou architektúry riadenej udalosťami je realizácia na základe sprostredkovateľa správ alebo topológie sprostredkovateľa správ[1].



Obrázok 9 – Diagram topológie sprostredkovateľa správ[5].

Topológia sprostredkovateľa správ sa od topológie sprostredkovateľa lísi tým, že ako prvá neobsahuje žiadny centrálny spracovateľ udalostí. Prúd udalostí je reťazovo rozdelený medzi službami spracovania udalostí prostredníctvom nejakého sprostredkovateľa správ (napr. ActiveMQ, RabbitMQ, Apache Kafka atď.). Táto topológia je veľmi užitočná, keď je úlohou vytvorenie relatívne jednoduchého toku spracovania udalostí bez centrálnej orchestrácie udalostí[1].

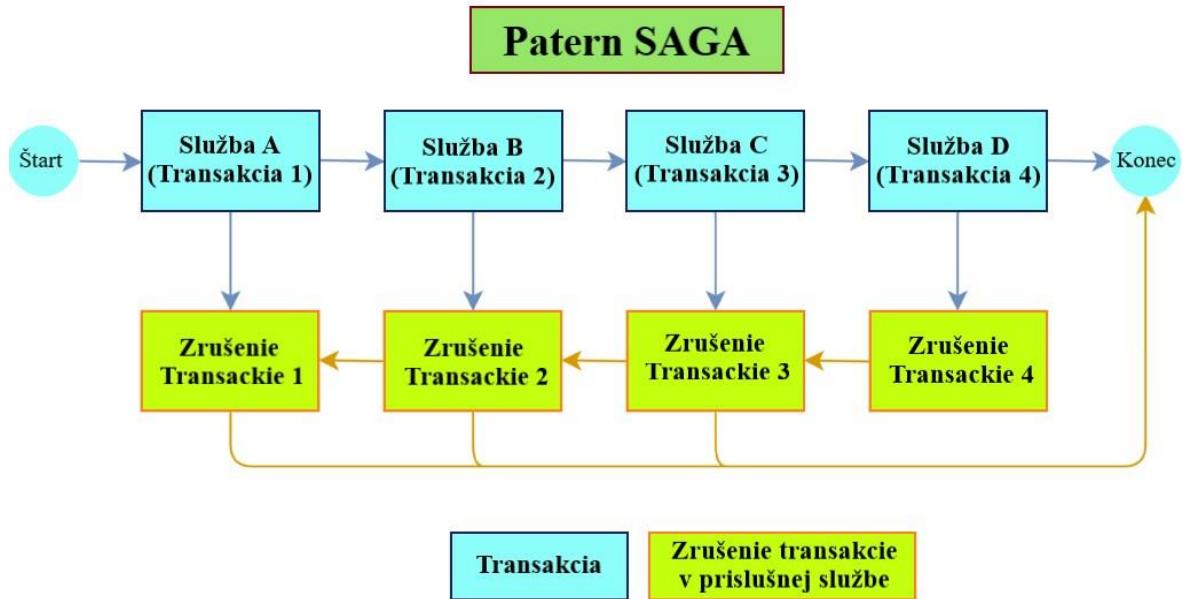
Ako je zobrazené na obrázku č. 9, v rámci architektúry existujú dva hlavné typy komponentov topológie sprostredkovateľa správ: samotný sprostredkovateľ správ a služby spracovania udalostí. Sprostredkovateľ správ môže byť centralizovaný alebo kombinovaný a obsahuje všetky kanály správ, ktoré sa používajú v rámci toku udalostí[5]. Kanály správ, ktoré sa nachádzajú vnútri sprostredkovateľa správ, môžu byť rady správ (z angl. Message Queues), témy správ (z angl. Message Topics) alebo ich kombinácia. V danej

realizácií je každá služba zodpovedná za spracovanie udalostí a publikovanie novej udalosti označujúcej akciu, ktorá bola práve vykonaná. Reakciou na novú udalosť je jej spracovanie ďalšou službou alebo spracovanie kombináciou služieb. Udalosti sa v rámci spracovania môžu rozdeľovať buď do niekoľkých kanálov správ alebo sa spájať do jednej udalosti[5].

Výhodu danej realizácie architektúry predstavuje veľmi vysoká škálovateľnosť softvérového produktu. Na rozdiel od realizácie pomocou sprostredkovateľa môže daná topológia v každom mieste zväčšiť svoju prieplustnosť pomocou vytvorenia nových inštancií. Ak sprostredkovateľ správ nebude zvládať množstvo vstupných udalostí, tak existuje možnosť spustenia ďalších sprostredkovateľov správ a spojenia ich do krastra sprostredkovateľov správ. Klaster obsahuje nejaké množstvo inštancií rovnakej aplikácie, ktoré si najčastejšie navzájom rozdeľujú vstupnú alebo výpočtovú zaťaž podľa metódy dookola (z angl. Round-Robin Load Balancing)[6]. Tým pádom môže niekoľko sprostredkovateľov správ zvládnúť oveľa väčšiu zaťaž, ktorú by iba jedna inštancia nezvládla. Okrem toho si môžu aj samotné služby rozdeľovať zaťaž pomedzi svoje inštancie[6]. Ďalšia veľká výhoda, ktorá spravila túto realizáciu veľmi populárnu na trhu softvérových produktov, je veľká odolnosť voči chýbam a poruchám. Kvôli tomu, že si vieme rozdeliť každý element softvérového produktu na určité množstvo inštancií, vieme tieto inštancie spúšťať na rôznych serveroch z rôznych miest a spájať ich do jedného produktu[6]. Tým pádom, ak v jednej časti sveta nastane celkový výpadok siete, aplikácia bude stále dostupná pomocou iných inštancií, ktoré bežia na iných funkčných serveroch. Okrem toho sa dá pomocou sprostredkovateľa správ predísť chybovému stavu aplikácie pri pokuse spracovania dát nesprávneho formátu. Pomocou zavedenia kontroly formátu pre vstupné dáta sa dajú dáta filtrovať. Tie, ktoré neprejdú kontrolou, vieme ukladať v špeciálnom rade pre ich nasledujúce ručné spracovanie. Služby v danej architektúre sú úplne nezávislé a majú v porovnaní s topológiou sprostredkovateľa menšiu viazanosť[6]. Vďaka tomu sa môžu vyvíjať, meniť, alebo sa paralelne vylepšovať s ostatnými službami. Okrem toho sa služby dajú pomocou svojej nezávislosti bez potreby spustenia ďalších služieb ľahko a dôkladne testovať[5].

Hlavnou nevýhodou danej realizácie je spracovanie globálnych transakcií medzi niekoľkými službami. Komunikácia medzi službami môže prebiehať iba pomocou odosielania a prijímania udalostí, čo spomaľuje tento proces[6]. Pre riešenie distribuovanej

transakcie sa používa patern SAGA, ktorý je zobrazený na obrázku č. 10. V porovnaní s topológiou sprostredkovateľa má však zložitejšiu realizáciu[1]. Ďalšia nevýhoda sa prejavuje pri zložitej biznis logike. Pomocou sprostredkovateľa správ sa nedá uskutočniť orchestrácia služieb, ktorá je dostupná pri topológii sprostredkovateľa[1].



Obrázok 10 – Diagram fungovania paternu SAGA[1].

Táto realizácia architektúry riadenej udalosťami sa používa v projektoch, ktoré nepotrebuju špecifické riadenie služieb a globálne transakcie[1]. Práve táto topológia je najvhodnejšia pre nás systém detekcie a spracovania bezpečnostných incidentov.

## 2 Technológie použité v práci

Táto kapitola sa zaobrá technológiami, ktoré sú nevyhnutnou súčasťou nášho systému. Na začiatku predstavíme sprostredkovateľa správ Apache Kafka, ďalej popíšeme Docker – softvérový produkt pre izoláciu aplikácií do kontajnerov a nakoniec spomenieme systém pre spracovanie bezpečnostných incidentov ASTD.

### 2.1 Apache Kafka

Moderné serverové aplikácie sú komplexné, viacúrovňové a obsahujú veľké množstvo komponentov a služieb. Tieto distribuované komponenty a služby však potrebujú nejakým spôsobom prenášať medzi sebou údaje, inými slovami – komunikovať. Pre riešenie problematiky komunikácie služieb alebo celých komponentov v rámci softvérového produktu bol vytvorený systém pre sprostredkovanie správ. Ide o rozvetvený „potrubný“ systém, do ktorého sa na jednom konci dá vložiť určité množstvo správ a na druhom konci si nejaká služba tieto správy od sprostredkovateľa správ vyberie alebo prijme a prečíta. Sprostredkovateľ správ, ak je správne implementovaný, umožňuje komponentom a službám si navzájom pridelovať úlohy, oznamovať zmeny v systéme a informovať kontrolujúcu časť aplikácie o svojich stavoch. Jedna z implementácií takého systému je Apache Kafka.

Sprostredkovateľ správ Kafka je distribuovaný systém[7]. Jeho servery sa dajú zoskupiť do veľkého klastru. Správy sa ukladajú a posielajú paralelne na rôzne servery, čo zvyšuje spoľahlivosť a odolnosť voči chybám. Aj keď zlyhá niekoľko sprostredkovateľov správ v klastru, správy sa budú stále prijímať a preposielat. Kafka je jednoducho horizontálne škálovateľná[7]. Pre zvýšenie kapacity stačí k Apache Kafka pripojiť ďalšie inštancie sprostredkovateľov správ. Je to najjednoduchší spôsob škálovania systémov, ale nie je vhodný pre všetky systémy, ako napríklad databázy. Kafka je od začiatku navrhnutá na expozívny rast výkonu[7].

Ďalšou výhodou je konzistentnosť údajov. Záznamy v Apache Kafka sa ukladajú ako protokoly revízií. Vyzerá to ako rad správ, do ktorého sa dajú pridať položky, ale sa nedajú upravovať. Tento prístup poskytuje veľkú spoľahlivosť a jednoduchosť pri zmene akýchkoľvek stavov – vždy je jasné, čo, ako a v akom poradí bolo zmenené. Pri používaní Kafka klastra každá inštancia sprostredkovateľa správ používa replikácie iných inštancií

pre prípad, že by jedna z inštancií prestala fungovať[7]. Replikácie slúžia na to, aby dátá boli konzistentné a zálohované.

Ako bolo vyššie spomenuté, Kafka pracuje s dvoma skupinami aplikácií: producentmi – s tými, čo posielajú správy do sprostredkovateľa správ a konzumentmi – s tými, čo tieto správy čítajú a spracovávajú. Vnútri obsahuje Kafka určité množstvo radov správ alebo tém správ (z angl. Topic)[7]. Do týchto tém vkladajú producenti správy pre spracovanie jedným alebo niekoľkými skupinami konzumentov. Pre odoslanie správy do Kafka je potrebné zadať adresu servera a porta, na ktorom je sprostredkovateľ správ spustený, samotnú správu a tému, do ktorej sa správa alebo určité množstvo správ odošle a typ serializácie a deserializácie správy, pretože Kafka si všetko ukladá ako pole bajtov a zvolený typ sa potom použije pre správne čítanie odoslanej správy konzumentom[7]. Okrem správy sa môže zadať kľúč, ktorý je potrebný pre identifikovanie a rozdelenie správ do skupín vo vnútri Kafka – do tzv. partícii (z angl. Partition). Tieto partície slúžia na paralelné spracovanie správ skupinou konzumentov (z angl. Consumers Group) zo zvolenej témy[7]. Skupina konzumentov sa pripája k vybranej téme a každému konzumentovi v skupine sa prideluje približne rovnaké číslo partícií, z ktorých budú daní konzumenti spracovávať správy. Ak počas behu softvérového systému niektorý konzument zo skupiny prestane so spracovateľom správ komunikovať, tak sa partície, ktoré mal spracovať, rozdelia medzi ostatných dostupných konzumentov v rámci skupiny[7]. Tým pádom bude môcť softvér pokračovať v práci. Kafka pracuje ako rad správ, ale každá nová skupina konzumentov si číta správy od začiatku a nie od momentu, kedy sa pripojila k sprostredkovateľovi správ. Kontrolu poradia a množstva prečítaných správ vykonáva Kafka vďaka metadátam offset[7]. Offset je číslo, ktoré je jedinečné pre každú správu. Napríklad v nejakej Kafka téme je uložených 10 správ. Ak jedna určitá konzumná skupina prečíta iba 7 správ, offset bude pre ňu 7. Ako náhle sa však aspoň jeden konzument od danej skupiny ku Kafka pripojí, tak dostane zvyšné správy, ktoré v rade správ začínajú v poradí od 8. miesta až po koniec. Takýmto spôsobom Kafka zabezpečuje jedinečné doručenie správ, čo znamená, že tá istá správa by sa na spracovanie nemala odoslať do tej istej konzumnej skupiny dvakrát[7]. Samotné správy, ako aj ich metadáta a kľúče, sa zapisujú na disk a ukladajú sa tam podľa konfigurácie, ktorú možno nastaviť pre každú tému zvlášť[7]. To umožňuje v závislosti od potrieb konzumentov ukladanie radov správ na rôzne časové obdobie. Dlhodobé ukladanie dát taktiež nepredstavuje nebezpečenstvo

straty údajov, aj v prípade, keby konzument kvôli pomalému spracovaniu alebo prudkému nárastu správ či spomaleniu internetového spojenia nové správy nestíhal čítať. Zároveň to znamená, že aj pri odpojení konzumenta od sprostredkovateľa správ alebo pri odpojení internetovej siete a po opäťovnom pripojení konzument dostane podľa offsetu všetky nové dátu zo zvolenej témy správ[7]. Vďaka tomu, že Kafka dáta po prečítaní nevymazáva a umožňuje nastaviť dobu uchovávania dát na nekonečno, používajú niektorí používatelia tento sprostredkovateľ ako databázu, do ktorej si ukladajú a zálohujú dátu[7].

Ako producenti a konzumenti môžu vystupovať nie len aplikácie, ale aj relačné a nerelačné databázy, rôzne dátové súbory (JSON, CSV, textové súbory), oblačné objektové úložiská (Amazon S3, Azure Blob Storage, Google Cloud Storage) alebo iní sprostredkovatelia správ (ActiveMQ, IBM MQ, RabbitMQ)[7]. Na prepojenie sprostredkovateľa Kafka s týmito zdrojovými alebo cieľovými službami slúži komponent Apache Kafka produktov – Kafka Connect. Kafka Connect je konektor, ktorý dokáže na základe konfiguračného súboru a príslušného ovládača spojiť sprostredkovateľa a potrebnú službu[7]. Pre náš systém spracovania incidentov sme použili relačnú databázu PostgreSQL, ktorá prijíma dátu z Apache Kafka pomocou JDBC prepojenia. Vo vnútri daného prepojovača sú 2 typy odosielania – *Source* pre odosielanie z databázy do témy sprostredkovateľa správ a *Sink* pre odosielanie z témy sprostredkovateľa do databázy[7]. V našej práci používame Kafka Connect s JDBC Sink konektorom pre monitorovanie, načítanie a ukladanie výsledkov a spracovanie bezpečnostných incidentov z témy správ do relačnej databázy PostgreSQL. Taktiež používame pre zobrazenie dát z databázy PostgreSQL aplikáciu PgAdmin4. Podrobný popis konfigurovania použitých vstupného a výstupného konektorov sa nachádza v prílohe A.

Spomedzi všetkých sprostredkovateľov správ sme zvolili Apache Kafka, pretože umožňuje dlhodobé ukladanie dát pre spracovanie detegovaných incidentov rôznymi aplikáciami a poskytuje množstvo hotovej funkcionality pre vývoj a monitorovanie, ako sú Kafka Control Center a Kafka Connect. Taktiež sme ho vybrali z dôvodu vysokej výkonnosti sprostredkovateľa správ a vysokej škálovateľnosti pre budúce vylepšenie softvérového produktu.

## 2.2 Docker

Docker je softvérový produkt, ktorého cieľom je poskytnúť jednotné rozhranie pre izoláciu aplikácií do kontajnerov[8]. Kontajnery predstavujú spôsob balenia aplikácie a všetkých jej závislostí do jedného obrazu. Tento obraz beží v izolovanom prostredí, ktoré neovplyvňuje základný operačný systém. Kontajnery umožňujú oddeliť aplikáciu od infraštruktúry[8]. Pri vývoji nemusíme premýšľať nad tým, v akom prostredí sa aplikácia bude spúšťať, a ani, či tam budú správne nastavenia a potrebné závislosti. Stačí vytvoriť aplikáciu a všetky závislosti a nastavenia zabaliť do jedného obrazu. Tento obraz je potom možné rovnako korektne spustiť na každom systéme, ktorý Docker podporuje.

Docker je platforma pre vývoj, poskytovanie a spúšťanie kontajnerových aplikácií[8]. Docker umožňuje vytvárať kontajnery, automatizovať ich spúšťanie a nasadzovanie a spravovať ich životný cyklus. Vďaka Docker sa da spustiť viacero kontajnerov v jednom systéme. Zároveň poskytuje možnosť spustenia a monitorovania viacerých inštancií jednej aplikácie[8].

Vo všeobecnosti kontajnerovanie zjednodušuje prácu programátorom aj správcom, ktorí také aplikácie nasádzajú na server a spracovávajú. Príchod kontajnerizácie zväčšíl popularitu architektúr mikroslužieb, ktoré pomocou Docker alebo iných aplikácií kontajnerovania umožňujú rýchlu a jednoduchú škálovateľnosť, nasadenie a spracovanie veľkého množstva inštancií nejakej aplikácie[8].

V praktickej časti sú sprostredkovateľ správ a ďalšie komponenty Apache Kafka, ktoré budú vymenované v tretej časti, nasadené pomocou Docker Compose inštrumentu, ktorý vytvára kontajnery pre komponenty Apache Kafka a spája ich vnútornou sieťou. Taktiež je kontajnerovaná aj serverová aplikácia pre detekciu a spracovanie bezpečnostných incidentov pomocou Docker a Docker Compose a je vytvorená niekoľkými inštanciami pre paralelné spracovanie dát. Pomocou Docker je zabezpečené jednoduché a automatizované spustenie našej aplikácie na ľubovoľnom operačnom systéme, ktorý Docker podporuje.

## 2.3 ASTD

Systém ASTD (z angl. Algebraic State-Transition Diagrams) je rozšírením bežných automatov a stavových tabuliek, ktoré je možné kombinovať s algebraickými operátormi

ako postupnosť, výber a kvantifikovaná synchronizácia[8]. Tento systém sa používa pre detekciu a kvalifikovanie útokov[8]. Pre detekciu kybernetických útokov pomocou ASTD je potrebné použiť špecifikácie, ktoré na základe množstva prepojených predchádzajúcich stavov vedia posúdiť, či ide o špecifikovaný útok alebo nie. Každá špecifikácia špecifikuje jeden typ útoku[8].

V našej práci používame systém ASTD ako jadro našej serverovej aplikácie pre detekciu a špecifikáciu útokov. Samotnú aplikáciu ASTD ako aj vytvorené špecifikácie pre naše detektie incidentov: *Lateral Movement, Ping, Port Scan, Ransomware a RAT* používame z diplomovej práce<sup>2</sup>.

---

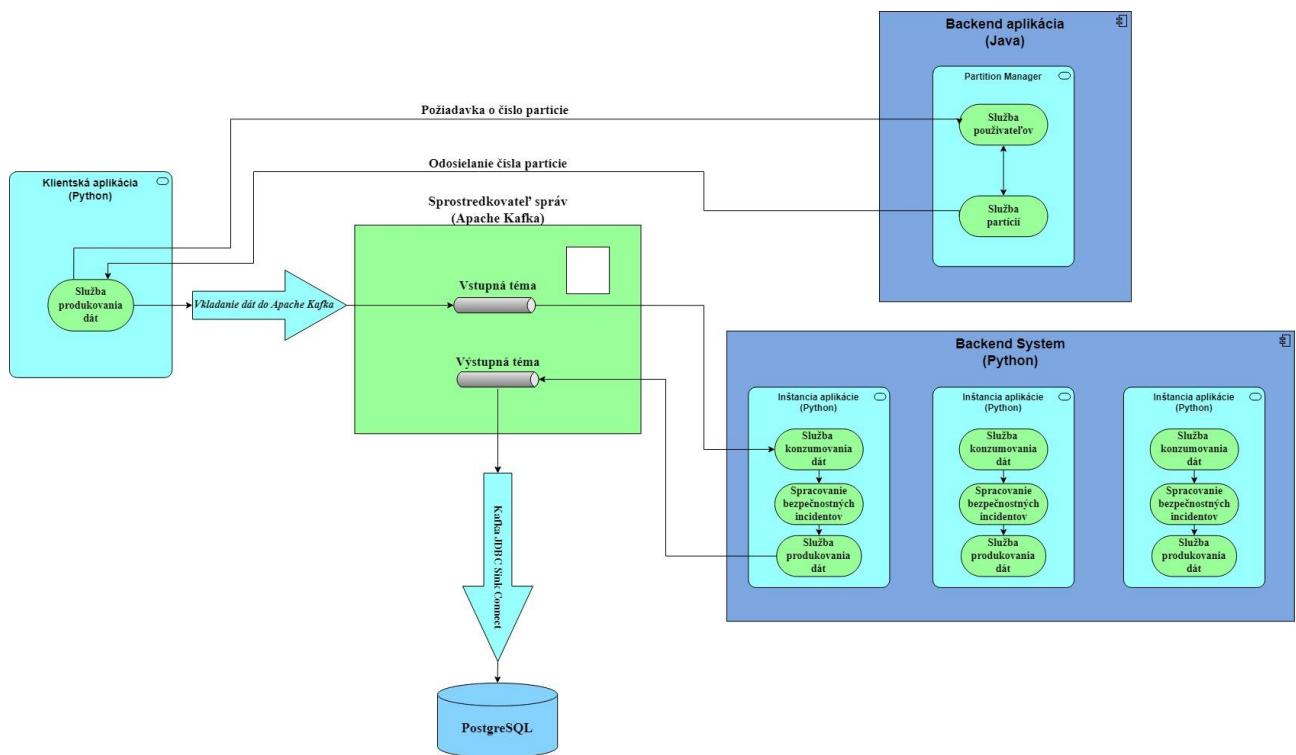
<sup>2</sup> Maslen, Martin. *Zber informácií o bezpečnostných incidentoch*. Bratislava: FEI STU, 2021. 80 s. Diplomová práca.

# 3 Systém pre detekciu a spracovanie bezpečnostných incidentov

V tejto časti sa rozoberá návrh, implementácia a testovanie systému pre spracovanie a detekciu bezpečnostných incidentov. Daná časť končí simulovaným záťažovým testovaním nášho systému.

## 3.1 Návrh systému

Po dôkladnej analýze a porovnaní architektúr pre vytvorenie systému detektie a spracovania bezpečnostných udalostí alebo bezpečnostných incidentov sme sa rozhodli použiť architektúru riadených udalosťami. To znamená, že systém je rozdelený na 3 časti: na klienta, ktorý posielá udalosti do nášho sprostredkovateľa správ, na samotného sprostredkovateľa správ, ktorý tieto udalosti prijíma a umožňuje ich vyberanie, a služby, ktoré tie udalosti vyberajú, spracovávajú a ktoré zároveň vkladajú nové, výsledné udalosti do iného radu správ sprostredkovateľa správ. V našom prípade patrí k službám aj manažér partícii, ktorý prideluje klientovi číslo partície, a tým riadi spracovanie dát.



Obrázok 11 – Návrh systému pre detekciu a spracovanie útokov.

Na obrázku č. 11 je zobrazený návrh nášho systému. V tomto systéme sa používa iba jedna služba s niekoľkými inštanciami pre detekciu a spracovanie bezpečnostných incidentov, avšak nami vybraná architektúra umožňuje jednoduché pridanie ďalších služieb spracovania udalostí pre nejakú novú funkcia.

Okrem týchto troch častí náš systém obsahuje aj relačnú databázu PostgreSQL, do ktorej posielame pomocou Kafka JDBC Sink Connect všetky potrebné dátá zo sprostredkovateľa správ na ukladanie a ručnú analýzu.

## 3.1 Implementácia systému

V danej časti rozoberieme funkcia a implementáciu nášho systému. Prejdeme klientskou aplikáciou, potom manažérom partícií, použitím sprostredkovateľa správ, serverovou službou spracovania incidentov a nakoniec monitorovaním udalostí pomocou databázy.

### 3.1.1 Implementácia klienta

V našom systéme je klient jednoduchá aplikácia, ktorá je vytvorená v programovacom jazyku Python verzie 3.8. Po spustení klientskej aplikácie, ktorá je popísaná v prílohe C, sa nainštalujú potrebné závislosti pre danú aplikáciu, ktorá sa následne spustí. Pri štarte této aplikácia prečíta adresu riadenia prístupu k médiu (z angl. Media Access Control – MAC address) používateľa nato, aby bol v systéme jedinečne identifikovateľný, odošle požiadavku o číslo partície do manažéra partícií a po odpovedi spustí nekonečný cyklus čítania Sysmon žurnálových správ (z angl. Log Messages) alebo ASTD udalostí. Po úspešnom prečítaní aspoň jedného z týchto typov dát ich pošle do sprostredkovateľa správ Apache Kafka s MAC adresou na miesto klíča správy do pridelenej partície vstupnej témy. V prípade, že súbor so vstupnými dátami bude obsahovať dátá iného formátu než sa očakáva, bude tento súbor vynechaný pri načítaní a používateľ dostane do konzoly hlášku: „*Bad format of events in file: názov\_súboru*“. Žurnálové správy zo Sysmon sa načítavajú z priečinka */sysmon\_logs/*, kam ich používateľ môže vkladať pred alebo počas behu aplikácie vo formáte XML. Ukážka takejto žurnálovej správy je zobrazená na obrázku č. 12.

```
1 <?xml version="1.0" encoding="utf-8" standalone="yes'?>
2 <Events>
3   <Event xmlns="http://schemas.microsoft.com/win/2004/08/events/event">
4     <System>
5       <Provider Name='Microsoft-Windows-Sysmon'
6           Guid='{5770385f-c22a-43e0-bf4c-06f5698ffbd9}' />
7       <EventID>1</EventID>
8       <Version>5</Version>
9       <Level>4</Level>
10      <Task>1</Task>
11      <Opcode>0</Opcode>
12      <Keywords>0x8000000000000000</Keywords>
13      <TimeCreated SystemTime='2022-04-17T13:24:51.1944318Z' />
14      <EventRecordID>148</EventRecordID>
15      <Correlation/>
16      <Execution ProcessID='11132' ThreadID='9816' />
17      <Channel>Microsoft-Windows-Sysmon/Operational</Channel>
18      <Computer>N6</Computer>
19      <Security UserID='S-1-5-18' />
20    </System>
21    <EventData>
22      <Data Name='RuleName'></Data>
23      <Data Name='UtcTime'>2022-04-17 13:24:51.193</Data>
24      <Data Name='ProcessGuid'>{f03562c7-1523-625c-b715-00000004e00}</Data>
25      <Data Name='ProcessId'>14820</Data>
26      <Data Name='Image'>C:\Windows\System32\SearchFilterHost.exe</Data>
27      <Data Name='FileVersion'>7.0.19041.1620 (WinBuild.160101.0800)</Data>
28      <Data Name='Description'>Microsoft Windows Search Filter Host</Data>
29      <Data Name='Product'>Windows® Search</Data>
30      <Data Name='Company'>Microsoft Corporation</Data>
31      <Data Name='OriginalFileName'>SearchFilterHost.exe</Data>
32      <Data Name='CommandLine'>"C:\WINDOWS\system32\SearchFilterHost.exe" 0 808 812 820 8192
33          816 792
34      </Data>
35      <Data Name='CurrentDirectory'>C:\WINDOWS\system32\</Data>
36      <Data Name='User'>NT AUTHORITY\SYSTEM</Data>
37      <Data Name='LogonGuid'>{f03562c7-5a35-6256-e703-000000000000}</Data>
38      <Data Name='Logonid'>0x3e</Data>
39      <Data Name='TerminalSessionId'></Data>
40      <Data Name='IntegrityLevel'>Medium</Data>
41      <Data Name='Hashes'>
42        SHA256={43488605B12495FE90A43856AA9A63030955936A828BE395578CBB6F2864F4
43      </Data>
44      <Data Name='ParentProcessGuid'>{f03562c7-d62d-6259-cc11-00000004e00}</Data>
45      <Data Name='ParentProcessId'>6948</Data>
46      <Data Name='ParentImage'>C:\Windows\System32\SearchIndexer.exe</Data>
47      <Data Name='ParentCommandLine'>"C:\WINDOWS\system32\SearchIndexer.exe /Embedding</Data>
```

Obrázok 12 – Ukážka Sysmon žurnálovej správy v XML formáte.

Následne je každá Sysmon žurnálová správa konvertovaná pomocou klienta do ASTD udalostí, ktorých ukážka je zobrazená na obrázku č. 13.

Obrázok 13 – Ukážka niekolkých ASTD udalostí.

Po prečítaní a konvertovaní všetkých dostupných Sysmon žurnálových správ ich klientska aplikácia odosielá do *local\_security\_incident* témy správ Apache Kafka, odkiaľ

následne budú službou spracovania udalostí vybrané a spracované. Každá udalosť je pred odosielaním serializovaná pomocou Avro schémy, ktorá pozostáva z troch textových premenných, ktoré odosielame do Kafka v rámci jednej udalosti. Prvá premenná sa volá *data* a obsahuje ASTD udalosť, ktorá bude vyhodnotená službou spracovania udalostí. Nasledujúca premenná je *session\_id*, ktorá je náhodne vygenerovaná pri spustení aplikácie nato, aby sme vedeli sledovať a rozlišovať skupiny udalostí pre jedného a toho istého používateľa, ktoré sa odosielajú počas niekoľkých spustení klientskej aplikácie. Posledná textová premenná v rámci udalosti je to isté ako aj kľúč udalosti – *user\_id*, ktorý používame pre jednoznačné identifikovanie používateľa v systéme. Potreba ukladat kľúč udalosti aj do vnútra samotnej udalosti vznikla kvôli problému ukladania dát do databázy pomocou Kafka JDBC Sink Connect. Tento prepojovač umožňuje ukladať kľúč do databázy iba v rámci primárneho kľúča, čo nám nevyhovuje, pretože primárny kľúč musí byť jedinečný a nemôže sa opakovať a v našom systéme môže mať jeden používateľ veľké množstvo udalostí. Tým, že sa kľúč udalostí nedá uložiť do databázy iným spôsobom ako primárnym kľúčom a pre nás to je dôležitá informácia, ktorú v databáze potrebujeme uchovávať, rozhodli sme sa ho pridať aj do vnútra odoslanej udalosti. Samotný kľúč udalosti je serializovaný pomocou textového serializovania (z angl. String Serialization). Po tejto činnosti si naša aplikácia prečíta z priečinku *astd\_events/* všetky vložené ASTD udalosti. Ak také sú, odošle ich s tým istým kľúčom do tej istej partície a témy, ako aj konvertované Sysmon žurnálové správy. Ďalej aplikácia pokračuje v behu toho istého cyklu, počas ktorého používateľ môže vložiť dátu, ktoré sa aplikácia pokúsi prečítať a odoslať do sprostredkovateľa správ. Aby sa predišlo čítaniu tých istých správ a udalostí v ďalšom kole cyklu, klientska aplikácia si ukladá názov už prečítaného súboru žurnálových správ alebo ASTD udalostí, čiže, ak bude používateľ počas behu klientskej aplikácie ukladať nové správy do už prečítaného súboru, tak tie dátu budú klientom ignorované. Pre pridanie nových dát počas behu aplikácie je potrebné do vyššie uvedených priečinkov vložiť vhodný nový súbor s novými dátami.

### 3.1.2 Implementácia manažéra partícií

Potreba v manažérovi partícií vznikla až pri testovaní systému, ktorý paralelne spracováva udalosti pomocou niekoľkých inštancií. Dôkladný popis problematiky

pridelenia používateľských identifikátorov nejakému množstvu partícií bude vykonaný v časti: „Výsledok testovania paralelného spracovania udalostí bez MP“.

Pre realizáciu manažéra partícií bol zvolený programovací jazyk Java a aplikačný rámec (z angl. Framework) Spring Boot pre jednoduchosť vytvorenia webových aplikácií, rozsiahlosť poskytovanej funkcionality a osobné skúsenosti s týmto jazykom a aplikačným rámcem. Aplikácia bola spravená pre komunikáciu pomocou rozhrania REST s klientskymi aplikáciami a pre pridelovanie im čísla vhodnej partície. Po prvom spustení aplikácia vytvorí dve databázové tabuľky: *Users* pre používateľov a *Partitions* pre dostupné partície. Hned po naštartovaní sa aplikácia pripojí k vstupnej téme *local\_security\_incident* sprostredkovateľa správ, prečíta si a uloží do príslušnej databázovej tabuľky všetky dostupné partície. Pri prvom odosielaní partícií do databázy označí partíciu číslo 0 ako nasledujúcemu pre priradenie. Ďalej aplikácia očakáva na adrese: [http://localhost:8080/partition/set/{používateľský\\_identifikátor}](http://localhost:8080/partition/set/{používateľský_identifikátor}) POST požiadavku, z ktorej si vyberie používateľský identifikátor ako parameter cesty.

```
    @Autowired
    public PartitionController(UserService userService, PartitionService partitionService) {
        this.userService = userService;
        this.partitionService = partitionService;
    }

    @PostMapping("/{username}")
    public ResponseEntity<Integer> setPartitionForUser(@PathVariable String username) throws ValidationException {
        log.info("Handling partition for user: {}", username);
        User user = userService.findByName(username);
        Partition partition;
        if (isNull(user)) {
            log.info("Saving new user: {}", username);
            partitionService.actualizingExistingPartitions();
            partition = partitionService.getPartitionByNextInLineToUse();
            partitionService.moveOnToNextPartition();
            user = userService.saveUser(new User(username, partition));
        } else {
            partition = user.getPartition();
        }
        log.info("User: {} get partition number: {}", user.getName(), partition.getNumber());
        return new ResponseEntity<>(partition.getNumber(), HttpStatus.OK);
    }
}
```

Obrázok 14 – Ukážka kódu kontroléra manažéra partícií.

Na obrázku č. 14 je zobrazená časť kódu, v ktorej vidíme implementačnú logiku manažéra partícií. Po prijatí požiadavky od používateľa si manažér partícií skontroluje, či je taký používateľ už zaregistrovaný v databáze. Ak áno, tak mu vráti už uložené číslo partície, aby sa všetky dátá jedného používateľa spracovávali jednou a tou istou službou.

Ak sa tento používateľský identifikátor nenachádza v databáze, tak ho pridá s číslom partície, ktoré je aktuálne a hned' nato zväčší číslo partície o 1 a zmení jej status na nasledujúcu partíciu v poradí pre ďalšieho používateľa. Ak partícia zväčšená o 1 v zozname dostupných partícií neexistuje, tak začneme znova od prvej a tak dookola. Pri každej požiadavke od nového používateľa zároveň aj kontrolujeme zoznam aktuálnych partícií pre prípad, že sa pridajú nové alebo sa vymažú niektoré, ktoré sa používali. Takýmto spôsobom má každý používateľ priradené číslo partície a bud' ho vráti z databázy alebo dostane spolu s registráciou svojho identifikátora v databáze nové číslo.

### 3.1.3 Implementácia sprostredkovateľa správ

Ako implementácia sprostredkovateľa správ bola vybraná Apache Kafka z niekoľkých dôvodov. V porovnaní s inými sprostredkovateľmi, ako napríklad RabbitMQ a ActiveMQ, Kafka nevymazáva svoje správy z radu správ po ich prečítaní, čo umožňuje v budúcnosti pridať iné spôsoby spracovania bezpečnostných incidentov a uchovávať odoslané udalosti tak dlho, ako je to potrebné. Ako realizáciu Apache Kafka používame Docker obraz od spoločnosti Confluent. Okrem toho spoločnosť Confluent poskytuje veľké množstvo funkcionality a dobre spracovanej dokumentácie pre Apache Kafka. Z danej funkcionality ešte používame Kafka JDBC Connect Sink, ktorého funkcia je popísaná v časti „Implementácia monitorovania pomocou databázy“. Spustenie a nastavenie sprostredkovateľa správ je popísané v prílohe A.

Po spustení sprostredkovateľa správ musíme vytvoriť vstupnú a výstupnú tému s určitým množstvom partícií. Počet partícií by mal byť približne rovnaký ako počet konzumentov v skupine nato, aby spracovanie dát bolo čo najrýchlejšie, ale aj nato, aby žiadna inštancia konzumenta nepracovala nadarmo v prípade, že inštancií konzumentov bude viac ako partícií. Apache Kafka automaticky spája voľnú partíciu s voľným konzumentom v rámci konzumnej skupiny a taktiež sa automaticky stará o to, aby každý konzument v skupine dostával iba nové udalosti, ktoré ešte neboli prečítané žiadnymi inými konzumentmi v rámci skupiny, ak to nie je nastavené inak. Pre väčší výkon sme zavedli manažér partícií, ktorý rozdeľuje používateľské identifikátori pred produkovaním dát do najvhodnejšej partície a každý konzument pri pripojení zadáva číslo partície, aby striktne čítal udalosti z jednej partície aj po prebalansovaní témy, pretože potrebujeme, aby

sa všetky dátá ľubovoľného používateľa spracovali iba jednou a tou istou inštanciou služby.

Po nastavení uvedenom v prílohe A je Apache Kafka pripravená na prácu s bezpečnostnými udalosťami. V skutočnosti sa po spustení klientskej aplikácie v pozadí uskutoční ešte jedno nastavenie sprostredkovateľa správ, a tým je pridanie nami vytvorenej Avro schémy pre serializáciu a deserializáciu udalostí do Kafka Schema Registry, ktorá je zodpovedná za správne formátovanie všetkých udalostí vstupnej témy pri ich prijatí a pri odoslaní. Ale táto činnosť sa vykoná automaticky pre vstupnú tému po odoslaní prvej udalosti z klientskej aplikácie a pre výstupnú tému pri odoslaní prvej udalosti zo serverovej služby.

### 3.1.4 Implementácia serverovej služby

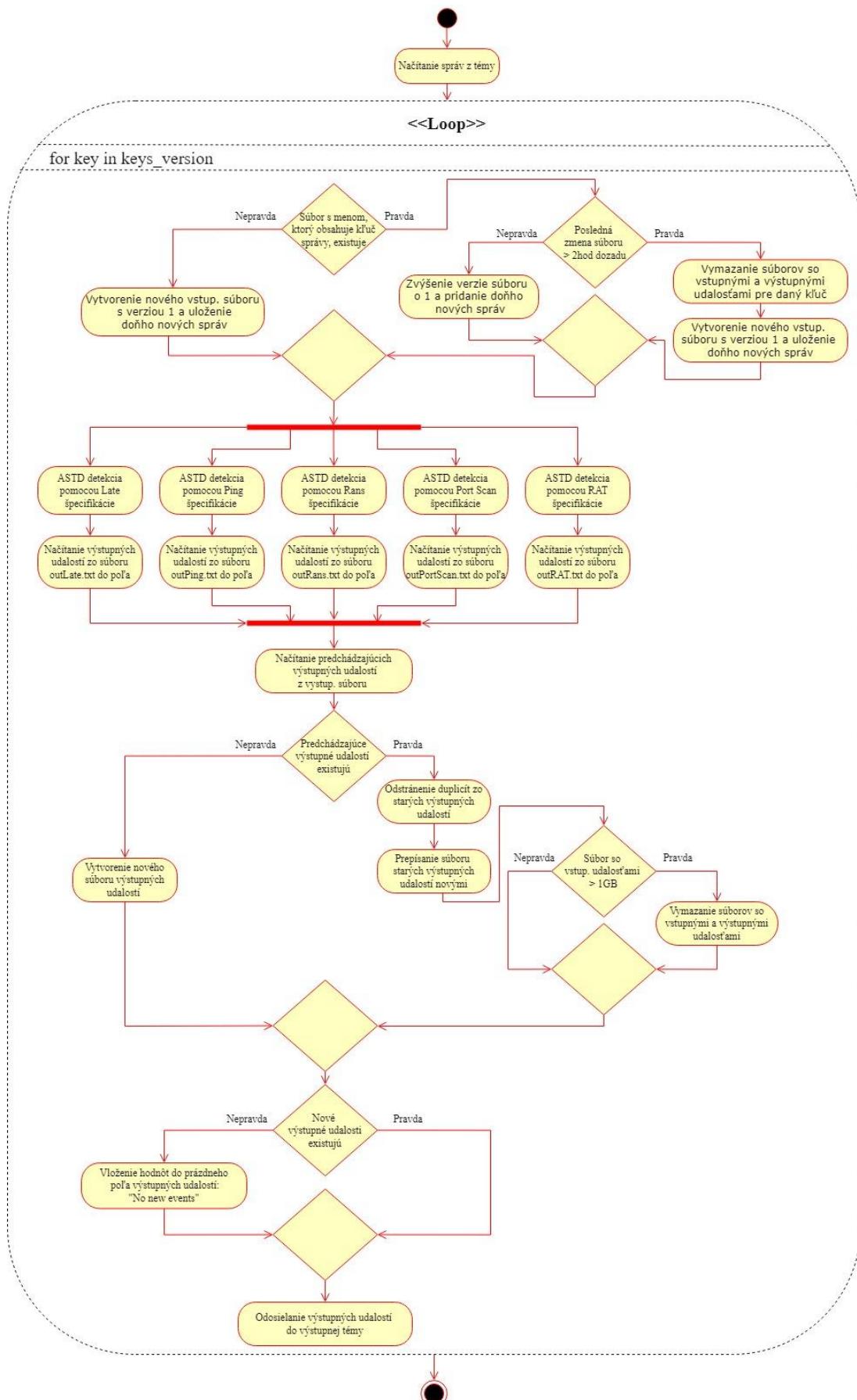
Serverová služba predstavuje aplikáciu v programovacom jazyku Python, ktorá má podľa vyššie uvedenej architektúry vykonávať úlohu služby spracovania udalostí. Z vysokoúrovňového hľadiska si má služba prečítať vstupné udalosti zo sprostredkovateľa správ, spracovať ich a výsledok spracovania odoslať späť do inej témy sprostredkovateľa správ. Ako jadro spracovania udalostí používame aplikáciu ASTD, ktorá bola popísaná v teoretickej časti práce a zapríčinuje v našej službe niektoré obmedzenia. Prvé z daných obmedzení je formát vstupných dát. Aplikácia ASTD spracováva iba udalosti, ktoré sú vytvorené v špeciálnom tvare, ktorý bol zobrazený na obrázku č. 13. Do daného formátu taktiež vieme premeniť Sysmon žurnálové správy z formátu XML, ktorých ukážka bola zobrazená vyššie na obrázku č. 12. Ostatné formáty vstupných udalostí ASTD nevie spracovať. Ak taká udalosť prejde klientskou aplikáciou a odošle sa do sprostredkovateľa správ, tak ASTD nebude vedieť danú udalosť spracovať a do výstupnej témy bude odoslaná správa: „*Bad format of events*“. Dané obmedzenie má byť vyriešené klientskou aplikáciou, ktorá filtriuje vstupné dátá. Ale následné významné obmedzenie používania ASTD aplikácie je realizované na serverovej časti.

Tým, že ASTD je rozšírením stavových automatov a tabuliek, nie je pre ASTD detegovaná samotná udalosť ako hrozba alebo bezpečnostný incident. Pre detegovanie hrozieb a zaradenie do určitého typu útoku musí ASTD spracovať určité množstvo udalostí, v ktorých hľadá určitú postupnosť udalostí (ktorá je definovaná pomocou jednotlivej špecifikácie), a preto daná aplikácia na vstupe očakáva hned' všetky dostupné

udalosti. To znamená, že ak zoberieme súbor so vstupnými ASTD udalosťami, ktorých spracovanie nám určite vyhodnotí nejaké hrozby a spracujeme daný súbor pomocou našej služby dvakrát oddelene tak, že najprv spracujeme prvú polovicu súboru a potom druhú polovicu súboru, tak na výstupe nemusíme dostať žiadnu hrozbu alebo incident, pretože sa nemusí nachádzať v samotných polovičiach daného súboru celá postupnosť udalostí zo špecifikácie. Čiže pri prečítaní novej udalosti z Kafka pre spracovanie incidentov na vstup do ASTD musíme poslat' všetky dostupné udalosti pre určitého používateľa, inak nastáva možnosť, že bezpečnostné incidenty nebudú objavené. Pre riešenie daného obmedzenia by sme mohli pri prijatí novej správy vygumovať ofset pre danú partíciu a čítať nie iba novú správu, ale aj všetky predchádzajúce. Avšak dané riešenie by nám spôsobilo veľké oneskorenia kvôli odosielaniu obrovského množstva dát cez internetovú sieť a každá nasledujúca správa by spôsobovala lineárny rast množstva posielaných správ. Preto sme sa rozhodli pri navrhovaný našej služby použiť ukladanie určitého množstva správ v serverovej službe. Po naštartovaní aplikácie sa spustí nekonečný cyklus, ktorý sa snaží prečítať nové správy z jemu pridelenej partície sprostredkovateľa správ. Po prečítaní nových správ naša serverová aplikácia najprv skontroluje, či už má vytvorený textový súbor s názvom, ktorý obsahuje kľúč správy a číslo verzie detekcie. Ak taký súbor už existuje, tak nakoniec k nemu pridá nové správy a zvýši jeho verziu o 1. Ak taký súbor neexistuje, tak ho vytvorí a uloží doňho dátu. Ďalej spustí paralelne ASTD detegovanie incidentov zo vstupných udalostí v danom súbore pomocou 5 dostupných špecifikácií, ktoré boli spomenuté v teoretickej časti pri popise ASTD. Následne sa paralelne prečítajú výsledky spracovania a uložia sa do slovníka, v ktorom sa ako kľúč použije názov použitej špecifikácie a ako hodnota – pole výsledkov detegovania.

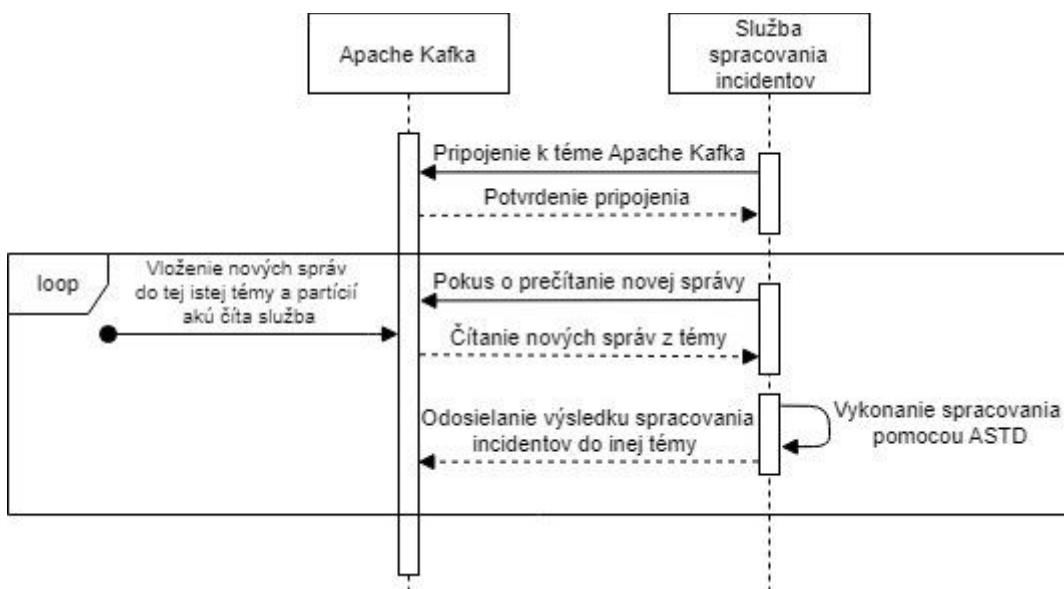
Samořejme, ak na vstup budeme posielat' stále všetky dostupné udalosti, tak hned', ako sa objaví prvý incident, tak sa bude objavovať po každom ďalšom detegovaní vo výsledkoch. Kvôli tomu potrebujeme ukladať na serveri nielen zoznam vstupných udalostí, ale aj zoznam výstupných udalostí, aby sme vedeli rozoznať a odoslať do výslednej témy iba nové výsledné incidenty. Čiže po prečítaní výstupných incidentov z ASTD aplikácie musíme skontrolovať, či sa na serveri už nachádza súbor s výstupnými incidentami predchádzajúceho detegovania pre daného používateľa. Ak sa nachádza, odstraňujeme staré výsledky zo slovníka a potom pridávame do súboru nové výsledky. Ak taký súbor neexistuje, naša aplikácia ho vytvorí a uloží doňho všetky výsledky. Nakoniec potrebujeme

pred odoslaním výsledných incidentov do výslednej témy skontrolovať, či sa na výstupe ASTD niečo nachádzalo, aby používateľ videl výsledok detegovania, a nie, aby nekonečne niečo očakával v databáze alebo vo výslednej téme a tam sa nič nepridá, pretože ASTD pre daný vstup nič našlo. Ak výsledné incidenty pre nejakú špecifikáciu nie sú, naša aplikácia tam vloží ako incident správu: „*No new events*“. Po tejto činnosti budú výsledné incidenty odosланé do výslednej témy sprostredkovateľa správ spolu s verziou detegovania. Následne sa aplikácia bude snažiť nové správy znova prečítať a po ich prečítaní zopakuje vyššie uvedený cyklus. Cely proces, ktorý prebieha vo vnútri služby, je zobrazený na obrázku č. 15.



Obrázok 15 – Diagram aktivít pre serverovú službu.

Okrem diagramu aktivít sú zobrazené aj na sekvenčnom diagrame na obrázku č. 16 komunikácia sprostredkovateľa správ a služby a spracovanie udalostí touto službou.



Obrázok 16 – Sekvenčný diagram serverovej aplikácie.

Pri sériovej činnosti jednej inštancie služby spracovania incidentov nevzniká žiadny problém. Inštancia aplikácie sa pripojí ku Kafka téme a bude dostávať udalosti zo všetkých partií témy. Avšak sa pri prechode na paralelné používanie niekoľkých inštancií stretávame s ďalším obmedzením. Pri ukladaní vstupných dát do súboru do vnútra nejakej inštancie sa stávame od nej závislými, pretože, ak nové správy začne spracovať nová inštancia serverovej služby, do ASTD na spracovanie budú odoslané iba posledné udalosti bez predchádzajúcich, ktoré sú uložené vo vnútri inej inštancie. Riešenie daného obmedzenia predstavuje zabezpečenie spracovania ľubovoľného množstva dát pre určitého používateľa vždy tou istou inštanciou služby. Apache Kafka nám pomáha vo vykonaní danej úlohy tým, že zabezpečuje spôsob rozdeľovania dát do partií podľa hašovacieho kódu klíča správy a následne umožňuje čítanie udalostí zo striktne vybranej partície.

Tento spôsob však nezaručuje, že ak máme 5 používateľov s rôznym klíčom, tak ich správy budú rovnomerne rozdelené do 5 partií a následne paralelne spracované pomocou 5 služieb. Táto situácia nie je jednoznačná, pretože o zariadení do určitej partície rozhoduje hašovací kód klíča udalostí, ktorý môže byť rovnaký pre viaceru klíčov.

S ohľadom na tento problém sme vytvorili manažér partícií pre jednoznačné zaručenie rovnomerného rozdelenia kľúčov správ do jednotlivých partícií. Práve tento spôsob používame pri implementácii našej služby, čiže vytvárame niekoľko partícií pre vstupnú a výstupnú tému a potom každú partíciu prepájame s nejakou inštanciou serverovej služby. Takýmto štýlom vieme, že počet inštancií služieb nesmie byť väčší ako počet partícií používanej témy, pretože súčasťou môžeme čítať niekoľko partícií jednou službou, ale jednu partíciu spracovávať niekoľkými službami kvôli našej biznis logike nemôžeme.

### 3.1.5 Implementácia monitorovania pomocou databázy

Pri navrhovaní ľubovoľného systému existuje vždy nejaký vstup a nejaký výstup. V našom systéme je vstup reprezentovaný ASTD udalosťami alebo Sysmon XML žurnálovými správami a výstup je reprezentovaný ako výstupná udalosť serializovaná pomocou Avro, ktorá obsahuje kľúč používateľa (taký istý ako aj vstupná udalosť) a 4 textové premenné a jednu číselnú premennú. Do textových premenných sa ukladá samotná správa z výstupu ASTD alebo jej náhrada („No new events“, „Bad format for events“), názov špecifikácie, podľa ktorej boli vstupné udalosti vyhodnotené, názov inštancie služby, ktorá spracovávala a odosielala bezpečnostné incidenty do Kafka, a identifikátor používateľa pre možnosť ukladania do databázy a pridelovania výsledkov bezpečnostných incidentov jednotlivým používateľom. Číselná hodnota reprezentuje verziu odoslaných udalostí, aby sa dalo rozlíšiť, pre ktorú iteráciu odosielania dát boli dané výsledné udalosti spracované. Tieto výsledné udalosti sa posielajú do výslednej témy Apache Kafka. Aby používateľ zistil výsledky spracovania svojich vstupných dát alebo aby mal nejaký administrátor spracovania hrozí a útokov prehľad o vyskytnutých udalostach, sme sa rozhodli ukladať výstupné udalosti do databázy. Taktiež sme pre prípad monitorovania alebo ručného spracovanie vstupných udalostí nesprávneho formátu implementovali ukladanie aj vstupných udalostí do databázy. Ako už bolo vyššie spomenuté, pre posielanie dát z témy do databázy sa používa program Apache Kafka Connect, ktorý podporuje prepojenie daného sprostredkovateľa správ s väčšinou dostupných databáz alebo s inými sprostredkovateľmi správ či nejakými aplikáciami. Čiže v našej práci používame presne Apache Kafka JDBC Sink Connect, ktorý vytvára dve prepojenia – jedno zo vstupnej témy do databázy a druhé z výstupnej témy do databázy. Samotné prepojenie sa vytvára

pomocou konfiguračného súboru. Okrem potrebného nastavenia pre posielanie dát (typ deserializácie správ, prihlasovacie údaje do databázy a nastavenie primárneho kľúča pre databázovú tabuľku) umožňuje daný konfiguračný súbor aj modifikáciu dát, ktoré sa odosielajú do databázy (pridať nejaké pole z udalosti ako primárny kľúč, pridať statickú premennú s nejakou hodnotou a transformovať systémové dáta z sprostredkovateľa správ do databázy).

	<b>id</b>	<b>data</b>	<b>version</b>	<b>specification</b>	<b>user_id</b>	<b>instance_name</b>	<b>kafka_partition</b>	<b>kafka_offset</b>	<b>kafka_topic</b>
1	2750834	"c\Process creation","2022-04-17 13:24:51.1944311","C:\Windows\system32\search\SearchHost.exe","C:\Windows\system32\SearchProtocolHost.exe",0x1000,0x1000,4345-4e75-0e585865f94	0	7516423942	0	0	0	0	local_security_incident
2	2750835	"c\Process creation","2022-04-17 13:24:51.1716677","C:\Windows\system32\SearchProtocolHost.exe","C:\Windows\system32\SearchProtocolHost.exe",0x1000,0x1000,4345-4e75-0e585865f94	0	7516423942	0	1	0	0	local_security_incident
3	2750836	"c\Network connection","2022-04-17 13:24:53.3714949","System","4","107","192.168.0.117","192.168.23.107",0x1000,0x1000,4345-4e75-0e585865f94	0	7516423942	0	2	0	0	local_security_incident
4	2750837	"c\Process termination","2022-04-17 13:24:53.1099647","C:\Windows\system32\SearchProtocolHost\SearchProtocolHost.exe",0x1000,0x1000,4345-4e75-0e585865f94	0	7516423942	0	3	0	0	local_security_incident
5	2750838	"c\Network connection","2022-04-17 13:24:53.2161445","C:\Windows\system32\SearchProtocolHost\SearchProtocolHost.exe",0x1000,0x1000,4345-4e75-0e585865f94	0	7516423942	0	4	0	0	local_security_incident
6	2750839	"c\Process creation","2022-04-17 13:24:53.3549047","C:\Windows\system32\SearchProtocolHost\SearchProtocolHost.exe",0x1000,0x1000,4345-4e75-0e585865f94	0	7516423942	0	5	0	0	local_security_incident
7	2750840	"c\Process terminated","2022-04-17 13:24:53.3548079","C:\Program Files\dotnet\MicrosoftEdge\Application\MicrosoftEdge.exe",0x1000,0x1000,4345-4e75-0e585865f94	0	7516423942	0	6	0	0	local_security_incident
8	2750841	"c\Process creation","2022-04-17 13:24:53.3460079","C:\Program Files\dotnet\MicrosoftEdge\Application\MicrosoftEdge.exe",0x1000,0x1000,4345-4e75-0e585865f94	0	7516423942	0	7	0	0	local_security_incident
9	2750842	"c\Process creation","2022-04-17 13:24:56.9702147","C:\Windows\system32\SearchHost.exe","C:\Windows\system32\SearchHost\SearchHost.exe",0x1000,0x1000,4345-4e75-0e585865f94	0	7516423942	0	8	0	0	local_security_incident
10	2750843	"c\Process creation","2022-04-17 13:24:56.9861427","C:\Windows\system32\SearchProtocolHost\SearchProtocolHost.exe",0x1000,0x1000,4345-4e75-0e585865f94	0	7516423942	0	9	0	0	local_security_incident
11	2750844	"c\Network connection","2022-04-17 13:24:56.9861427","C:\Windows\system32\SearchProtocolHost\SearchProtocolHost.exe",0x1000,0x1000,4345-4e75-0e585865f94	0	7516423942	0	10	0	0	local_security_incident
12	2750845	"c\Network connection","2022-04-17 13:24:56.9861427","C:\Windows\system32\SearchProtocolHost\SearchProtocolHost.exe",0x1000,0x1000,4345-4e75-0e585865f94	0	7516423942	0	11	0	0	local_security_incident
13	2750846	"c\Process terminated","2022-04-17 13:24:56.9861427","C:\Windows\system32\SearchProtocolHost\SearchProtocolHost.exe","",1,0x1000,0x1000,4345-4e75-0e585865f94	0	7516423942	0	12	0	0	local_security_incident
14	2750847	"c\Network connection","2022-04-17 13:26:51.7115257","System","4","107","192.168.0.117","192.168.0.117",0x1000,0x1000,4345-4e75-0e585865f94	0	7516423942	0	13	0	0	local_security_incident
15	2750848	"c\Network connection","2022-04-17 13:28:57.7915177","System","4","107","192.168.0.117","192.168.0.117",0x1000,0x1000,4345-4e75-0e585865f94	0	7516423942	0	14	0	0	local_security_incident
16	2750849	"c\Network connection","2022-04-17 13:28:57.8747167","C:\Windows\system32\SearchProtocolHost\SearchProtocolHost.exe",0x1000,0x1000,4345-4e75-0e585865f94	0	7516423942	0	15	0	0	local_security_incident
17	2750850	"c\Network connection","2022-04-17 13:28:55.8747167","C:\Program Files\dotnet\Google\Chrome\Application\Google Chrome.exe",0x1000,0x1000,4345-4e75-0e585865f94	0	7516423942	0	16	0	0	local_security_incident

Obrázok 17 – Zobrazenie dát z tabuľky *local\_security\_incident* pomocou pgAdmin4.

	<b>id</b>	<b>data</b>	<b>version</b>	<b>specification</b>	<b>user_id</b>	<b>instance_name</b>	<b>kafka_partition</b>	<b>kafka_offset</b>	<b>kafka_topic</b>
1	57203	Alert Portscan attack - 45 scanned ports	1	late	e1813737752a	service-3	2	0	scanning_result
2	57204	Exploit phase started	1	late	e1813737752a	service-3	2	1	scanning_result
3	57205	192.168.1.129 exploits Standard API on the victim machine 10.10.8.136	1	late	e1813737752a	service-3	2	2	scanning_result
4	57206	Alert Metasploit Privilege Escalation	1	late	e1813737752a	service-3	2	3	scanning_result
5	57207	Alert attempted user privilege gain - cmdline\ net user /add Sy Hacker from host 192.168.1.129 on port 4444	1	late	e1813737752a	service-3	2	4	scanning_result
6	57208	Info User permission changed	1	late	e1813737752a	service-3	2	5	scanning_result
7	57209	Alert attempted user privilege gain - cmdline\ net user /add Sy Hacker from host 192.168.1.129 on port 4444	1	late	e1813737752a	service-3	2	6	scanning_result
8	57210	Info User permission changed	1	late	e1813737752a	service-3	2	7	scanning_result
9	57211	Alert attempted user privilege gain - cmdline\ net user /add Sy Hacker from host 192.168.1.129 on port 4444	1	late	e1813737752a	service-3	2	8	scanning_result
10	57212	Info User permission changed	1	late	e1813737752a	service-3	2	9	scanning_result
11	57213	Alert attempted user privilege gain - cmdline\ net user /add Sy Hacker from host 192.168.1.129 on port 4444	1	late	e1813737752a	service-3	2	10	scanning_result
12	57214	Info User permission changed	1	late	e1813737752a	service-3	2	11	scanning_result
13	57215	Alert attempted user privilege gain - cmdline\ net user /add Sy Hacker from host 192.168.1.129 on port 4444	1	late	e1813737752a	service-3	2	12	scanning_result
14	57216	Info User permission changed	1	late	e1813737752a	service-3	2	13	scanning_result
15	57217	Alert attempted user privilege gain - cmdline=C:\Windows\system32\net user /add Sy Hacker from host 192.168.1...	1	late	e1813737752a	service-3	2	14	scanning_result
16	57218	Info User permission changed	1	late	e1813737752a	service-3	2	15	scanning_result
17	57219	Alert attempted user privilege gain - cmdline=C:\Windows\system32\net1 user /add Sy Hacker from host 192.168.1...	1	late	e1813737752a	service-3	2	16	scanning_result

Obrázok 18 – Zobrazenie dát z tabuľky *scanning\_result* pomocou pgAdmin4.

Práve transformovanie dát z Apache Kafka, ako sú identifikátor partície, identifikátor offsetu a názov témy, používame pre obidve naše prepojenia Kafka a databázy. Na obrázkoch č. 17 a č. 18 sú zobrazené dátá z tabuľiek *local\_security\_incident* a *scanning\_result*, ktoré obsahujú všetky potrebné údaje pre identifikáciu incidentov určitého používateľa a monitorovanie spracovania udalostí. Pod to spadajú odpovede na otázky, ktorému používateľovi dátá patria, na ktorej inštancii bolo spracovanie vykonané, podľa ktorej špecifikácie boli vyhodnotené určité výsledky spracovania, v ktorej partícií sa nachádzajú dátá s určitým klúčom, v akom poradí sa dostali dátá do sprostredkovateľa správ a v ktorej verzii spracovania boli dátá vyhodnotené. Podľa toho aj vieme určiť všetky dátá, čo boli odosланé do ASTD pre vyhodnotenie danej verzie.

## 3.2 Testovanie systému

Po implementovaní systému spracovania bezpečnostných incidentov je ho potrebné otestovať. Ako druh testovania sme použili záťažové testovanie, pri ktorom sa systém spúšťa na serveri a pri ktorom sa testuje funkcionálita pri podmienkach, ktoré prekračujú hranice bežnej prevádzky. Cieľom daného testovania je overiť, či systém splňa požiadavky, ktoré boli spomenuté v časti: „Implementácia serverovej služby“, či pracuje tak, ako sa od neho očakáva a zistiť približný čas od odoslania veľkého množstva dát (pri prvom testovaní je to 1 514 770 udalostí, pri druhom a treťom – 2 272 155 udalostí) rozdelených do piatich iterácií do odoslania výsledkov spracovania k výslednej téme. Testovanie prebieha troma spôsobmi – sekvenčným spracovaním pri použití jednej služby, paralelným spracovaním niekoľkými službami bez manažéra partícií a paralelným spracovaním niekoľkými službami pomocou manažéra partícií. V každej časti výsledkov testovania najprv budú zobrazené obrázky konzolových výpisov počas testovania a ďalej na základe týchto výpisov budú vyhodnotené výsledky pre daný typ testovania. Daná časť končí celkovým zhodnením testovania.

### 3.2.1 Popis testovania

Testovanie bude prebiehať nasledovne: na serveri, ktorý ma výpočtovú kapacitu 15 gigabajtov RAM pamäte, 4-jadrový procesor Intel Xeon E3-1220 a SSD úložisko s kapacitou 100 gigabajtov, bude spustený sprostredkovateľ správ a jedna inštancia služby spracovania incidentov. Pre sekvenčné testovanie do vstupnej témy sprostredkovateľa

správ sa budú paralelne odosielat' testovacie udalosti z dvoch rôznych počítačov rozmiestnených v dvoch rôznych mestách pre simulovanie paralelného pripojenia a odosielania dát viacerých používateľov do Apache Kafka. Testovanie paralelného spracovania bude prebiehať v dvoch fázach – bez manažéra partícií a s manažérom partícií. Pre paralelne testovanie vstupné dáta budú odosielat' traja používatelia z rôznych miest pre zväčšenie záťaže na systém. Pri testovaní bez použitia manažéra partícií na vyššie uvedenom serveri budú spustené ešte dve inštancie služby spracovania bezpečnostných incidentov pre rýchlejšie spracovanie dát a pre testovanie rozdelenia používateľských dát do rôznych inštancií služieb. Pri testovaní s manažérom partícií budú dodatočné dve inštancie služieb pridané na iné zariadenie, ktoré má nasledovnú výpočtovú kapacitu: 16 gigabajtov RAM pamäte, 6-jadrový procesor Intel Core i7-9850H a SSD úložisko s 60 gigabajtovou kapacitou. Čiže pri poslednom testovaní služba spracovania bude distribuovaná do troch inštancií: jedna inštancia na serveri a dve inštancie na pridanom zariadení.

Ako bolo vyššie uvedené, sekvenčné testovanie sme pripravili tak, že všetky správy bude spracovávať jedna inštancia služby. Pre simulovanie pribúdajúcich udalostí sme modifikovali klientsku aplikáciu spôsobom, že načíta naše testovacie dáta (spolu je to 151 477 udalostí) a ďalej ich v piatich iteráciách bude odosielat' do vstupnej témy, ktorá je rozdelená do dvoch partícií, pretože dáta budú odosielat' dvaja používatelia paralelne. Po každom odoslaní cyklus zastaví beh programu na minútu, aby služba mala možnosť si dát načítať a začať ich spracovať. Ak by sme nezastavovali cyklus odosielania, tak by nás systém načítal všetky dáta v jednom behu a spracoval všetky naraz. Avšak my potrebujeme skontrolovať, či systém správne a ako rýchlo zvláda spracovanie priebežne pribúdajúcich skupín dát pre tých istých používateľov, ako to bude prebiehať v prevádzke, ale pri menšom množstve udalostí a menšej intenzite odosielania. Pred paralelným spustením testovacej klientskej aplikácie sme na serveri nastavili sprostredkovateľa správ podľa návodu, ktorý je uvedený v prílohe A. Ďalej sme spustili inštanciu služby detegovania bezpečnostných udalostí pomocou prvej polovice návodu z prílohy B, iba namiesto štyroch partícií sme vytvorili dve a namiesto štyroch služieb sme spustili iba jednu. Po týchto krokoch sme s kolegom, ktorý sa nachádzal v inom meste, paralelne spustili aplikáciu testovacieho klienta. Výsledok sekvenčného testovania je uvedený v časti: „Výsledok testovania sekvenčného spracovania udalostí“.

Prvé testovanie paralelného spracovania udalostí sa vykonávalo bez použitia manažéra partícií. Pred týmto testovaním sme vymazali všetky predchádzajúce dátá a nanovo sme nastavili sprostredkovateľa správ podľa prílohy A už len s tromi partíciami pre každú tému, pretože pri danom testovaní do sprostredkovateľa správ sme s dvoma kolegami paralelne vkladali dátá z 3 rôznych počítačov. Po nastavení sprostredkovateľa správ sme spustili službu spracovania udalostí podľa prvej časti prílohy B, ale s troma inštanciami služby. Pre správne čítanie dát z Apache Kafka si každá inštancia čítala svoju partíciu pomocou ručného priradenia identifikátora partície inštanciám. Hned' po spustení serverových služieb sme s kolegami v jednom čase spustili ten istý testovací klient, ktorého funkcia už bola vyššie popísaná. Výsledok daného testovania sa nachádza v časti: „Výsledok testovania paralelného spracovania udalostí bez MP“.

Druhé testovanie paralelného spracovania udalostí sa vykonávalo pomocou použitia manažéra partícií. Pred týmto testovaním sme vymazali všetky predchádzajúce dátá a nastavili sme sprostredkovateľa správ tak isto ako aj pri prvom testovaní paralelného spracovania. Služba spracovania dát pri tomto testovaní sa inštalovala nasledovne: najprv sme na serveri spustili jednu inštanciu služby spracovania s prednastaveným číslom partície, ktorú bude čítať, a ďalej sme spustili dve inštancie služby spracovania na inom zariadení, ktoré mali prednastavené zvyšné dve čísla partície, z ktorých si čítali dátu. Okrem toho sme spustili na serveri aj manažér partícií podľa prílohy B. Tak isto ako aj pri prvom testovaní sme s kolegami približne v jednom čase spustili každý svoju klientsku aplikáciu, ktorá však bola modifikovaná pridaním volania do manažéra partície a následne použitím prideleného čísla pri odosielaní udalostí, aby sa každá udalosť pre určitého používateľa odoslala do tej istej partície a aby každý používateľ dostál najvhodnejšiu partíciu v danom okamihu. Výsledok testovania je popísaný v nasledujúcej časti.

### 3.2.2 Výsledok testovania sekvenčného spracovania udalostí

Po vykonaní testovania sekvenčného spracovania udalostí sme dostali nasledujúce žurnálové správy.

```
User record b'27:ed:43:7f:5a:f1' successfully produced to local_security_incident [1] at offset 757382
User record b'27:ed:43:7f:5a:f1' successfully produced to local_security_incident [1] at offset 757383
User record b'27:ed:43:7f:5a:f1' successfully produced to local_security_incident [1] at offset 757384
local      2022-05-08 15:58:34,834 INFO    generate_data_to_kafka.py:<module> Iteration: 4.Data was sent: 151477
local      2022-05-08 15:59:34,894 INFO    generate_data_to_kafka.py:<module> All data was sent.
kubo@kubo-X510UNR:~/Desktop/Test_Clients$ kubo@kubo-X510UNR:~/Desktop/Test_Clients$
```

Obrázok 19 – Konzola s výpisom testovacej aplikácie kolegu.

Na obrázku č. 19 je zobrazený výstup z konzoly nášho kolegu, ktorému testovacia aplikácia odoslala dátu rýchlejšie ako nám. Z tohto obrázku je vidno, že aplikácia odoslala do partície vstupnej témy číslo 1 *local\_security\_incident* celkovo 757 385 incidentov. Ešte jedna dôležitá informácia je pre nás presný čas, kedy klient odoskal všetky dátu. To nastalo práve o 15 hodine, 58 minúte a 34 sekunde. Na obrázku č. 20 vidíme výpis z konzoly serveru, kde beží naša služba spracovania udalostí a pod tým v dolnom rohu vidíme čas, kedy sa nám odoslali všetky udalosti. To nastalo presne o 16 hodine, 3 minúte a 19 sekunde. Taktiež z výpisu spodnej konzoly vidíme, že nám aplikácia odoslala také isté množstvo udalostí do tej istej témy ako aj kolegovi, avšak už do partície číslo 0. Z výpisu serverovej konzoly, ktorá bola zobrazená na tom istom obrázku hned' nad našou, vidíme, že služba spracovania udalostí po prečítaní správ prijala dva používateľské klúče, ktoré sa zhodujú s našim a kolegovým a spracovala všetky údaje o 16 hodine, 7 minúte a 28 sekunde (serverový čas je nastavený na UTC +0 a náš lokálny je UTC +2, preto je v serverových žurnálových správach čas posunutý o dve hodiny dozadu).

```
dp2022@fel-dp-monitoring-2022:~/vasyl/Server$ docker-compose up
Starting server_Instance-1 ... done
Attaching to server_Instance-1
server_Instance-1  instance-1  2022-05-08 13:53:34,980 INFO  kafka_utils.py:consume_from_topic Try to consume data from topic: local_security_incident
server_Instance-1  instance-1  2022-05-08 13:55:02,562 INFO  kafka_utils.py:consume_from_topic Data was read for users: dict_keys(['21:30:90:7a:7d:44', '27:ed:43:7f:5a:f1'])
server_Instance-1  instance-1  2022-05-08 13:55:02,571 INFO  main.py:<module> Start of processing incidents
server_Instance-1  instance-1  2022-05-08 13:56:01,350 INFO  kafka_utils.py:produce_to_topic Producing records of events is started
server_Instance-1  instance-1  2022-05-08 13:56:02,488 INFO  kafka_utils.py:produce_to_topic Producing records of events has been finished
server_Instance-1  instance-1  2022-05-08 13:57:00,964 INFO  kafka_utils.py:produce_to_topic Producing records of events is started
server_Instance-1  instance-1  2022-05-08 13:57:02,003 INFO  kafka_utils.py:produce_to_topic Producing records of events has been finished
server_Instance-1  instance-1  2022-05-08 13:57:02,970 INFO  main.py:<module> Processing incidents finished in 120.4 second(s)
server_Instance-1  instance-1  2022-05-08 13:57:02,970 INFO  main.py:<module> ...
server_Instance-1  instance-1  2022-05-08 13:57:02,970 INFO  kafka_utils.py:consume_from_topic Try to consume data from topic: local_security_incident
server_Instance-1  instance-1  2022-05-08 13:57:30,457 INFO  kafka_utils.py:consume_from_topic Data was read for users: dict_keys(['21:30:90:7a:7d:44', '27:ed:43:7f:5a:f1'])
server_Instance-1  instance-1  2022-05-08 13:57:30,468 INFO  main.py:<module> Start of processing incidents
server_Instance-1  instance-1  2022-05-08 13:59:28,168 INFO  main.py:<module> Deleting data for user: 21:30:90:7a:7d:44 if it's more than 100MB
server_Instance-1  instance-1  2022-05-08 13:59:28,238 INFO  kafka_utils.py:produce_to_topic Producing records of events is started
server_Instance-1  instance-1  2022-05-08 13:59:29,262 INFO  kafka_utils.py:produce_to_topic Producing records of events has been finished
server_Instance-1  instance-1  2022-05-08 14:03:20,128 INFO  main.py:<module> Deleting data for user: 27:ed:43:7f:5a:f1 if it's more than 100MB
server_Instance-1  instance-1  2022-05-08 14:03:20,207 INFO  kafka_utils.py:produce_to_topic Producing records of events is started
server_Instance-1  instance-1  2022-05-08 14:03:21,225 INFO  kafka_utils.py:produce_to_topic Producing records of events has been finished
server_Instance-1  instance-1  2022-05-08 14:03:22,218 INFO  main.py:<module> Processing incidents finished in 351.75 second(s)
server_Instance-1  instance-1  2022-05-08 14:03:22,218 INFO  main.py:<module> ...
server_Instance-1  instance-1  2022-05-08 14:03:45,796 INFO  kafka_utils.py:consume_from_topic Try to consume data from topic: local_security_incident
server_Instance-1  instance-1  2022-05-08 14:03:45,816 INFO  kafka_utils.py:consume_from_topic Data was read for users: dict_keys(['21:30:90:7a:7d:44', '27:ed:43:7f:5a:f1'])
server_Instance-1  instance-1  2022-05-08 14:06:29,949 INFO  main.py:<module> Start of processing incidents
server_Instance-1  instance-1  2022-05-08 14:06:30,979 INFO  kafka_utils.py:produce_to_topic Producing records of events has been finished
server_Instance-1  instance-1  2022-05-08 14:07:26,048 INFO  kafka_utils.py:produce_to_topic Producing records of events is started
server_Instance-1  instance-1  2022-05-08 14:07:27,068 INFO  kafka_utils.py:produce_to_topic Producing records of events has been finished
server_Instance-1  instance-1  2022-05-08 14:07:28,052 INFO  main.py:<module> Processing incidents finished in 222.24 second(s)
server_Instance-1  instance-1  2022-05-08 14:07:28,052 INFO  main.py:<module> ...
server_Instance-1  instance-1  2022-05-08 14:07:28,052 INFO  kafka_utils.py:consume_from_topic Try to consume data from topic: local_security_incident
User record b'21:30:90:7a:7d:44' successfully produced to local_security_incident [0] at offset 757378
User record b'21:30:90:7a:7d:44' successfully produced to local_security_incident [0] at offset 757379
User record b'21:30:90:7a:7d:44' successfully produced to local_security_incident [0] at offset 757380
User record b'21:30:90:7a:7d:44' successfully produced to local_security_incident [0] at offset 757381
User record b'21:30:90:7a:7d:44' successfully produced to local_security_incident [0] at offset 757382
User record b'21:30:90:7a:7d:44' successfully produced to local_security_incident [0] at offset 757383
User record b'21:30:90:7a:7d:44' successfully produced to local_security_incident [0] at offset 757384
local      2022-05-08 16:03:19,860 INFO  generate_data_to_kafka.py:<module> Iteration: 4. Data was sent: 151477
local      2022-05-08 16:04:19,874 INFO  generate_data_to_kafka.py:<module> All data was sent.
★ (venv)  vklevanykvb:~/Documents/DP/Test_Client$ [ ]
```

Obrázok 20 – Moja konzola s výpisom testovacej aplikácie.

Čiže služba spracovania udalostí spracovala sekvenčne 757 385 udalostí pre jedného používateľa a 757 385 udalostí pre iného používateľa, čo spolu predstavuje milión a pol

udalostí za 13 minút vrátane očakávania dát medzi jednotlivými iteráciami odosielania, čítania a zápisu dát do sprostredkovateľa správ. Zároveň najdlhšie spracovanie trvalo necelých 6 minút, avšak v tom čase je zahrnuté spracovanie dát pre dvoch používateľov, takže približne 3 minúty trvalo najdlhšie spracovanie dát pre jedného používateľa. Taktiež posledné spracovanie dát trvalo necelé 4 minuty pre dátu 2 používateľských identifikátorov, preto prichádzame k záveru, že spracovanie takého množstva dát prináša pre nás systém oneskorenie približne 2 minúty pre každého používateľa. Ďalej sa pozrieme na výsledky spracovania udalostí, aby sme zhodnotili správnosť detegovania incidentov.

The screenshot shows a pgAdmin4 interface with a query editor containing the following SQL:

```

SELECT * FROM public.local_security_incident
ORDER BY kafka_offset DESC

```

The results grid has columns: Data Output, Explain, Messages, Notifications, session\_id, user\_id, kafka\_partition, kafka\_offset, kafka\_topic. The data shows 11 rows of log entries from Kafka, each with a timestamp and a detailed log message.

	session_id	user_id	kafka_partition	kafka_offset	kafka_topic
1	39737fc0-6d1b-468c-9660-db3bd30ed4cc	21:30:90:7a:7d:44	0	757384	local_security_incident
2	cfe14133-e102-4265-b703-d9910047af93	27:ed:43:7f:5a:f1	1	757384	local_security_incident
3	cfe14133-e102-4265-b703-d9910047af93	27:ed:43:7f:5a:f1	1	757384	local_security_incident
4	39737fc0-6d1b-468c-9660-db3bd30ed4cc	21:30:90:7a:7d:44	0	757382	local_security_incident
5	39737fc0-6d1b-468c-9660-db3bd30ed4cc	21:30:90:7a:7d:44	0	757382	local_security_incident
6	cfe14133-e102-4265-b703-d9910047af93	27:ed:43:7f:5a:f1	1	757382	local_security_incident
7	cfe14133-e102-4265-b703-d9910047af93	27:ed:43:7f:5a:f1	1	757381	local_security_incident
8	39737fc0-6d1b-468c-9660-db3bd30ed4cc	21:30:90:7a:7d:44	0	757381	local_security_incident
9	39737fc0-6d1b-468c-9660-db3bd30ed4cc	21:30:90:7a:7d:44	0	757380	local_security_incident
10	cfe14133-e102-4265-b703-d9910047af93	27:ed:43:7f:5a:f1	1	757380	local_security_incident

Obrázok 21 – Dáta zo vstupnej témy zobrazené pomocou pgAdmin4.

Na obrázku č. 21 vidíme vstupné udalosti a pomocné dátá, ktoré sú vyfiltrované od posledného Apache Kafka ofsetu, čo nám umožňuje sa uistíť, že dátu boli skutočne odoslané v rovnakom množstve pre každý používateľský identifikátor a že boli rozmiestnené do dvoch rôznych partícii.

The screenshot shows a pgAdmin4 interface with a query editor containing the following SQL:

```

SELECT * FROM scanning_result
ORDER BY kafka_offset DESC

```

The results grid has columns: Data Output, Explain, Messages, Notifications, version, specification, user\_id, instance\_name, kafka\_partition, kafka\_offset, kafka\_topic. The data shows 17 rows of log entries from Kafka, each with a timestamp and a detailed log message.

	version	specification	user_id	instance_name	kafka_partition	kafka_offset	kafka_topic
1	RAT	21:30:90:7a:7d:44	instance-1	0	6784	scanning_result	
2	RAT	27:ed:43:7f:5a:f1	instance-1	1	6784	scanning_result	
3	RAT	27:ed:43:7f:5a:f1	instance-1	1	6783	scanning_result	
4	RAT	21:30:90:7a:7d:44	instance-1	0	6783	scanning_result	
5	PortScan	21:30:90:7a:7d:44	instance-1	0	6782	scanning_result	
6	PortScan	27:ed:43:7f:5a:f1	instance-1	1	6782	scanning_result	
7	Rans	27:ed:43:7f:5a:f1	instance-1	1	6781	scanning_result	
8	Rans	21:30:90:7a:7d:44	instance-1	0	6781	scanning_result	
9	Rans	21:30:90:7a:7d:44	instance-1	0	6780	scanning_result	
10	Rans	27:ed:43:7f:5a:f1	instance-1	1	6780	scanning_result	
11	Rans	27:ed:43:7f:5a:f1	instance-1	1	6779	scanning_result	
12	Rans	21:30:90:7a:7d:44	instance-1	0	6779	scanning_result	
13	Rans	21:30:90:7a:7d:44	instance-1	0	6778	scanning_result	
14	Rans	27:ed:43:7f:5a:f1	instance-1	1	6778	scanning_result	
15	Rans	21:30:90:7a:7d:44	instance-1	0	6777	scanning_result	
16	Rans	27:ed:43:7f:5a:f1	instance-1	1	6777	scanning_result	
17	Rans	27:ed:43:7f:5a:f1	instance-1	1	6776	scanning_result	

Obrázok 22 – Dáta z výstupnej témy zobrazené pomocou pgAdmin4.

Pri testovaní sme využili 151 477 vstupných udalostí, ktorých lokálne vyhodnotenie nám vrátilo 1 357 výstupných incidentov pri výstupe z ASTD aplikácie. Pri testovaní sme posielali do sprostredkovateľa správ tie isté dátá päťkrát, čiže na výstupe máme dostať 6 785 zraniteľností. Na obrázku č. 22 sú zobrazené výstupné dátá po vyhodnotení bezpečnostných incidentov a tým, že offset pre Apache Kafka začína od 0, tak vidíme, že v databáze je presne 6 785 bezpečnostných udalostí pre každého používateľa, čiže aplikácia spracovala bezpečnostné incidenty správne. Okrem toho je v databázovej tabuľke vidno, že verzia posledných udalostí je 1 kvôli tomu, že po spracovaní dát v druhom cykle služba spracovania spustila vymazávanie starých súborov dát pri podmienke, že vstupný súbor musí byť väčší ako 100 megabajtov. To bola určite pravda, a tým pádom si v poslednej tretej iterácii služba vytvorila nové súbory dát a priradila im verziu 1. Takýmto spôsobom môžeme kontrolovať v databáze, kedy došlo k vymazávaniu starých dát a kedy ASTD dostalo na spracovanie iba nové dátá.

### 3.2.3 Výsledok testovania paralelného spracovania udalostí bez MP

Po vykonaní testovania paralelného spracovania bez použitia manažéra partícii sme dostali nasledujúce žurnálové správy.

```

dp2022@fel-dp-monitoring-2022:~/vasyl/Server$ docker-compose up
Creating service-3 ... done
Creating service-1 ... done
Creating service-2 ... done
Attaching to service-1, service-2, service-3
service-2 | service-2 2022-05-09 13:56:13,883 INFO kafka_utils.py:consume_from_topic Try to consume data from topic: local_security_incident
service-3 | service-3 2022-05-09 13:56:13,885 INFO kafka_utils.py:consume_from_topic Try to consume data from topic: local_security_incident
service-1 | service-1 2022-05-09 13:56:13,886 INFO kafka_utils.py:consume_from_topic Try to consume data from topic: local_security_incident
service-3 | service-3 2022-05-09 14:00:13,984 INFO kafka_utils.py:consume_from_topic Data was read for users: dict_keys(['e1:81:37:75:2a'])
service-3 | service-3 2022-05-09 14:00:34,004 INFO kafka.py:<module> Start to process incidents
service-2 | service-2 2022-05-09 14:01:04,304 INFO kafka.py:<module> Deleting old data for users: dict_keys(['b5:de:6e:03:50:8d', '1d:18:de:61:e4:3c'])
service-2 | service-2 2022-05-09 14:01:04,310 INFO main.py:<module> Start to process incidents
service-3 | service-3 2022-05-09 14:01:27,931 INFO kafka_utils.py:produce_to_topic Producing records of events for user: e1:81:37:75:2a is started
service-3 | service-3 2022-05-09 14:01:28,996 INFO kafka_utils.py:produce_to_topic Producing records of events for user: e1:81:37:75:2a has been finished
service-3 | service-3 2022-05-09 14:01:29,954 INFO main.py:<module> Processing incidents finished in 55.95 second(s)
service-3 | service-3 2022-05-09 14:01:29,954 INFO kafka_utils.py:consume_from_topic Try to consume data from topic: local_security_incident
service-3 | service-3 2022-05-09 14:01:29,954 INFO kafka_utils.py:consume From topic was read for users: dict_keys(['e1:81:37:75:2a'])
service-3 | service-3 2022-05-09 14:01:49,695 INFO main.py:<module> Start to process incidents
service-3 | service-3 2022-05-09 14:01:49,702 INFO main.py:<module> Start to process incidents
service-2 | service-2 2022-05-09 14:02:00,487 INFO kafka_utils.py:produce_to_topic Producing records of events for user: b5:de:6e:03:50:8d is started
service-2 | service-2 2022-05-09 14:02:01,553 INFO kafka_utils.py:produce_to_topic Producing records of events for user: b5:de:6e:03:50:8d has been finished
service-2 | service-2 2022-05-09 14:03:02,031 INFO kafka_utils.py:produce_to_topic Producing records of events for user: id:18:de:61:e4:3c is started
service-2 | service-2 2022-05-09 14:03:03,103 INFO kafka_utils.py:produce_to_topic Producing records of events for user: id:18:de:61:e4:3c has been finished
service-2 | service-2 2022-05-09 14:03:04,047 INFO main.py:<module> Processing incidents finished in 119.74 seconds(s)
service-2 | service-2 2022-05-09 14:03:04,048 INFO main.py:<module> ...
service-2 | service-2 2022-05-09 14:03:21,493 INFO kafka_utils.py:consume_from_topic Try to consume data from topic: local_security_incident
service-2 | service-2 2022-05-09 14:03:21,493 INFO kafka_utils.py:consume From topic Data was read for users: dict_keys(['id:18:de:61:e4:3c', 'b5:de:6e:03:50:8d'])
service-3 | service-3 2022-05-09 14:03:43,816 INFO main.py:<module> Start to process incidents
service-3 | service-3 2022-05-09 14:03:43,863 INFO kafka_utils.py:produce_to_topic Deleting data for user: e1:81:37:37:75:2a if it's more than 100MB
service-3 | service-3 2022-05-09 14:03:43,921 INFO kafka_utils.py:produce_to_topic Producing records of events for user: e1:81:37:37:75:2a is started
service-3 | service-3 2022-05-09 14:03:45,881 INFO main.py:<module> Processing records of events for user: e1:81:37:37:75:2a has been finished
service-3 | service-3 2022-05-09 14:03:45,881 INFO main.py:<module> ...
service-3 | service-3 2022-05-09 14:03:45,881 INFO kafka_utils.py:consume_from_topic Try to consume data from topic: local_security_incident
service-3 | service-3 2022-05-09 14:03:45,881 INFO kafka_utils.py:consume From topic Data was read for users: dict_keys(['e1:81:37:75:2a'])
service-3 | service-3 2022-05-09 14:03:55,103 INFO main.py:<module> Start to process incidents
service-3 | service-3 2022-05-09 14:04:54,050 INFO kafka_utils.py:produce_to_topic Producing records of events for user: e1:81:37:75:2a is started

```

Obrázok 23 – Výpis zo serverovej konzoly spracovania udalostí niekoľkými službami.

Na obrázku č. 23 vidíme výpis zo serverovej konzoly, ktorý reprezentuje beh serverových služieb. Ako môžeme sledovať z výpisu, služba číslo jedna, ktorá číta dátu z jednej partície nedostala dátu pre žiadneho používateľa vtedy, keď služba číslo 3 číta dátu pre práve jedného používateľa a služba číslo 2 dostala dátu hned' dvoch používateľov pre

spracovanie. Samozrejme, výkon tejto vyťaženej služby je oveľa menší ako je výkon služby, ktorá spracováva dátá práve jedného používateľa.

```

dp2022@fe1-dp-monitoring-2022: ~/vasyl/Server
[service-3] | service-3 2022-05-09 14:04:56,069 INFO main.py:<module> Processing incidents finished in 60.97 second(s)
[service-3] | service-3 2022-05-09 14:04:55,069 INFO main.py:<module> ...
[service-3] | service-3 2022-05-09 14:04:55,070 INFO kafka_utils.py:consume From topic Try to consume data from topic: local_security_incident
[service-3] | service-3 2022-05-09 14:05:05,100 INFO kafka_utils.py:consume From topic Data was read for users: dict_keys(['e1:81:37:37:75:2a'])
[service-3] | service-3 2022-05-09 14:05:05,107 INFO main.py:<module> Start of processing incidents
[service-2] | service-2 2022-05-09 14:06:09,665 INFO main.py:<module> Deleting data for user: id:18:de:61:e4:3c if it's more than 100MB
[service-2] | service-2 2022-05-09 14:06:09,728 INFO kafka_utils.py:produce To topic Producing records of events for user: id:18:de:61:e4:3c is started
[service-2] | service-2 2022-05-09 14:06:10,853 INFO kafka_utils.py:produce To topic Producing records of events for user: id:18:de:61:e4:3c has been finished
[service-3] | service-3 2022-05-09 14:06:59,096 INFO main.py:<module> Deleting data for user: e1:81:37:37:75:2a if it's more than 100MB
[service-3] | service-3 2022-05-09 14:06:59,138 INFO kafka_utils.py:produce To topic Producing records of events for user: e1:81:37:37:75:2a is started
[service-3] | service-3 2022-05-09 14:07:00,207 INFO kafka_utils.py:produce To topic Producing records of events for user: e1:81:37:37:75:2a has been finished
[service-3] | service-3 2022-05-09 14:07:01,151 INFO main.py:<module> Processing incidents finished in 116.04 second(s)
[service-3] | service-3 2022-05-09 14:07:01,152 INFO main.py:<module> ...
[service-3] | service-3 2022-05-09 14:07:01,152 INFO kafka_utils.py:consume From topic Try to consume data from topic: local_security_incident
[service-3] | service-3 2022-05-09 14:07:01,153 INFO main.py:<module> ...
[service-3] | service-3 2022-05-09 14:07:10,275 INFO main.py:<module> Start of processing incidents
[service-2] | service-2 2022-05-09 14:08:02,053 INFO main.py:<module> Deleting data for user: b5:de:6e:03:50:8d if it's more than 100MB
[service-2] | service-2 2022-05-09 14:08:02,096 INFO kafka_utils.py:produce To topic Producing records of events for user: b5:de:6e:03:50:8d is started
[service-2] | service-2 2022-05-09 14:08:03,165 INFO kafka_utils.py:produce To topic Producing records of events for user: b5:de:6e:03:50:8d has been finished
[service-2] | service-2 2022-05-09 14:08:04,112 INFO main.py:<module> Processing incidents finished in 282.7 second(s)
[service-2] | service-2 2022-05-09 14:08:04,113 INFO main.py:<module> ...
[service-2] | service-2 2022-05-09 14:08:04,113 INFO kafka_utils.py:consume From topic Try to consume data from topic: local_security_incident
[service-2] | service-2 2022-05-09 14:08:04,991 INFO main.py:<module> ...
[service-3] | service-3 2022-05-09 14:08:06,038 INFO kafka_utils.py:produce To topic Producing records of events for user: e1:81:37:37:75:2a is started
[service-3] | service-3 2022-05-09 14:08:07,010 INFO main.py:<module> Processing incidents finished in 56.71 second(s)
[service-3] | service-3 2022-05-09 14:08:07,010 INFO main.py:<module> ...
[service-3] | service-3 2022-05-09 14:08:07,010 INFO kafka_utils.py:consume From topic Try to consume data from topic: local_security_incident
[service-3] | service-3 2022-05-09 14:08:07,010 INFO main.py:<module> ...
[service-2] | service-2 2022-05-09 14:08:10,314 INFO main.py:<module> Start of processing incidents
[service-2] | service-2 2022-05-09 14:08:11,180 INFO kafka_utils.py:produce To topic Producing records of events for user: id:18:de:61:e4:3c is started
[service-2] | service-2 2022-05-09 14:08:11,722 INFO kafka_utils.py:produce To topic Producing records of events for user: id:18:de:61:e4:3c has been finished
[service-2] | service-2 2022-05-09 14:10:16,724 INFO kafka_utils.py:produce To topic Producing records of events for user: b5:de:6e:03:50:8d is started
[service-2] | service-2 2022-05-09 14:12:54,136 INFO kafka_utils.py:produce To topic Producing records of events for user: b5:de:6e:03:50:8d has been finished
[service-2] | service-2 2022-05-09 14:12:55,276 INFO kafka_utils.py:produce To topic Producing records of events for user: b5:de:6e:03:50:8d is started
[service-2] | service-2 2022-05-09 14:12:56,160 INFO main.py:<module> Processing incidents finished in 264.97 second(s)
[service-2] | service-2 2022-05-09 14:12:56,161 INFO main.py:<module> ...
[service-2] | service-2 2022-05-09 14:12:56,161 INFO kafka_utils.py:consume From topic Try to consume data from topic: local_security_incident

User record b'55:de:6e:03:50:8d' successfully produced to local_security_incident [1] at offset 1514766
User record b'55:de:6e:03:50:8d' successfully produced to local_security_incident [1] at offset 1514767
User record b'55:de:6e:03:50:8d' successfully produced to local_security_incident [1] at offset 1514768
User record b'55:de:6e:03:50:8d' successfully produced to local_security_incident [1] at offset 1514769
local 2022-05-09 16:08:28,200 INFO generate_data_to_kafka.py:<module> Iteration: 4. Data was sent: 151477
local 2022-05-09 16:09:28,236 INFO generate_data_to_kafka.py:<module> All data was sent.
★ (venv) vklevalnyk@vb:~/Documents/DP/Test_Client$ 

```

Obrázok 24 – Pokračovanie výpisu zo serverovej konzoly.

Na obrázku č. 24 je zobrazené pokračovanie výpisu konzoly, kde vidíme veľký rozdiel medzi časmi spracovania správ pre službu č. 2 a č. 3, pričom služba č. 1 kvôli tomu, že nedostala dátá žiadneho používateľa, je stále spustená, čiže zaberá serverové výpočtové kapacity s nulovým prínosom pre nás systém. Službe č. 2 trvalo spracovanie dát priemerne necelých 4 minút pre dvoch používateľov (120 sekúnd trvala prvá iterácia, 282 sekúnd druhá iterácia a 265 sekúnd tretia), čo je trošku lepší výsledok ako pri sekvenčnom spracovaní, kdeže službe č. 3 detekcia incidentov trvala v priemere necelú minútu a pol pre jedného používateľa (56 sekúnd v prvej iterácii, 116 sekúnd v druhej iterácii, 61 sekundu v tretej iterácii, 116 sekúnd v štvrtej iterácii a 56 sekúnd v piatej iterácii). Okrem týchto časových rozdielov vidíme, že služba č. 3 čítala dátá kvôli rýchlejšiemu spracovávaniu častejšie, vďaka čomu bol objem prenášaných dát menší, spracovanie rýchlejšie a používateľ rýchlejšie dostával výsledky každého spracovania vo výslednej téme a následne v databáze.

Tento scenár so zlým pridelením používateľského identifikátora do jednotlivých partícií, kvôli čomu jeden alebo niekoľko služieb nespracovávajú žiadne dátá a zvyšné sú používateľmi preťažené, sa veľmi často vyskytuje, pretože funkcia, ktorá vypočítava hašovací kód na základe kľúča správ pre pridelenie jednotlivého čísla partície

používateľským identifikátorom očakáva zložitejší alebo kompozitný kľúč. Pre porovnanie sme vyskúšali vytvoriť aj 10 partícií a zopakovať testovanie, ale tento scenár sa zopakoval – osem partícií ostalo prázdných, jedna partícia dostala jeden používateľský identifikátor a jedna partícia zase čítala udalosti dvoch používateľov. Kvôli tomuto problému prichádzame o výkonnosť systému spracovania udalostí – niektoré služby a partície dostavajú veľké množstvo dát a niektoré pracujú naprázdno.

Práve pre riešenie daného problému a pre kontrolu rozdelenia záťaže po partíciách sme sa rozhodli pridať webovú aplikáciu, ktorá nám umožní maximálny výkon pri rozdelení témy na partície a pri paralelnom spracovaní. Musíme však podotknúť, že rozdelenie témy na partície používame aj pre výstupnú tému, ale z hľadiska biznis požiadaviek sa môžu výstupné udalosti ukladať pre viacerých používateľov do jednej partície, pretože množstvo výstupných udalostí je oveľa menšie ako vstupných udalostí (6 785 výstupných zo 757 384 vstupných udalostí). Okrem toho sa výstupné dáta nespracovávajú sekvenčne pre každý kľúč v rámci služby, ako to je pri vstupných udalostiah, ale sa hned po prečítaní odosielajú do našej relačnej databázy, čiže neprichádza v tomto prípade k výraznému spomaleniu. Avšak aj tak by sme mali používať pre maximálny výkon rovnaké množstvo partícií pre vstupnú a výstupnú tému, čo by nám dovolilo používať, ak by sme potrebovali, ten istý identifikátor partícií pre obidve témy pomocou manažéra partícií.

### 3.2.4 Výsledok testovania paralelného spracovania udalostí pomocou MP

Pre vykonanie testovania paralelného spracovania pomocou manažéra partícií sme rozdelili tri inštancie služby tak, ako to bolo popísané v časti „Popis testovania“. Tým, že náš systém aktívne používa čítanie a zápis na úložisko, a zároveň sprostredkovateľ správ ho používa taktiež, tak väčšina inštancií serverovej aplikácie by mala byť umiestnená mimo sprostredkovateľa správ. Najlepšie by k tomu ešte bolo, keby boli aj samotné inštancie rozmiestnené na niekoľkých serveroch pre väčšiu spoľahlivosť systému a väčší výkon (pri stabilnom sietovom pripojení).

Na obrázku č. 25 vidíme výpis z konzoly zariadenia, ktorý bol popísaný v predošlých častiach, na ktorom sme spustili dve služby detegovania incidentov. Hned po štarte paralelného odosielania správ do vstupnej témy manažér partícií rozdelil každého používateľa do inej partície.

```

Creating server_instance-2_1 ... done
Creating server_instance-3_1 ... done
Attaching to server_instance-2_1, server_instance-3_1
(instance-2_1) | instance-2 2022-05-12 06:08:37,173 INFO kafka_utils.py:consume_from_topic Try to consume data from topic: local_security_incident
(instance-3_1) | instance-3 2022-05-12 06:08:37,177 INFO kafka_utils.py:consume_from_topic Try to consume data from topic: local_security_incident
(instance-2_1) | 1 kafka_utils.py:consume_from_topic Data was read for users: dict_keys(['99:93:3d:7d:06:2a'])
(instance-2_1) | instance-2 2022-05-12 06:19:41,281 INFO math.py:<module> Start of processing Incidents
(instance-3_1) | 2 kafka_utils.py:consume_from_topic Data was read for users: dict_keys(['5f:72:fa:f5:f8:fa'])
(instance-3_1) | instance-3 2022-05-12 06:19:41,975 INFO main.py:<module> Start of processing Incidents
(instance-3_1) | instance-3 2022-05-12 06:19:42,007 INFO kafka_utils.py:produce_to_topic Producing records of events for user: 5f:72:fa:f5:f8:fa is started
(instance-3_1) | instance-3 2022-05-12 06:20:14,580 INFO kafka_utils.py:produce_to_topic Producing records of events for user: 99:93:3d:7d:06:2a is started
(instance-2_1) | instance-2 2022-05-12 06:20:14,580 INFO kafka_utils.py:produce_to_topic Producing records of events for user: 5f:72:fa:f5:f8:fa has been finished
(instance-2_1) | instance-2 2022-05-12 06:20:15,660 INFO kafka_utils.py:produce_to_topic Producing records of events for user: 99:93:3d:7d:06:2a has been finished
(instance-2_1) | instance-2 2022-05-12 06:20:15,660 INFO main.py:<module> Processing Incidents finished in 35.12 second(s)
(instance-3_1) | instance-3 2022-05-12 06:20:17,124 INFO main.py:<module>
(instance-3_1) | instance-3 2022-05-12 06:20:17,124 INFO kafka_utils.py:consume_from_topic Try to consume data from topic: local_security_incident
(instance-3_1) | instance-3 2022-05-12 06:20:17,146 INFO main.py:<module> Processing Incidents finished in 35.84 second(s)
(instance-2_1) | instance-2 2022-05-12 06:20:17,146 INFO main.py:<module>
(instance-2_1) | instance-2 2022-05-12 06:20:17,147 INFO kafka_utils.py:consume_from_topic Try to consume data from topic: local_security_incident
(instance-3_1) | 2 kafka_utils.py:consume_from_topic Data was read for users: dict_keys(['5f:72:fa:f5:f8:fa'])
(instance-3_1) | instance-3 2022-05-12 06:20:56,903 INFO main.py:<module> Start of processing Incidents
(instance-3_1) | instance-3 2022-05-12 06:20:57,266 INFO kafka_utils.py:consume_from_topic Data was read for users: dict_keys(['99:93:3d:7d:06:2a'])
(instance-2_1) | 1 kafka_utils.py:<module> Start of processing Incidents
(instance-2_1) | instance-2 2022-05-12 06:21:10,510 INFO main.py:<module> Deleting data for user: 5f:72:fa:f5:f8:fa if it's more than 100MB
(instance-3_1) | instance-3 2022-05-12 06:22:02,411 INFO main.py:<module> Deleting data for user: 99:93:3d:7d:06:2a if it's more than 100MB
(instance-3_1) | instance-3 2022-05-12 06:22:02,447 INFO kafka_utils.py:produce_to_topic Producing records of events for user: 5f:72:fa:f5:f8:fa is started
(instance-3_1) | instance-3 2022-05-12 06:22:03,507 INFO kafka_utils.py:produce_to_topic Producing records of events for user: 5f:72:fa:f5:f8:fa has been finished
(instance-3_1) | instance-3 2022-05-12 06:22:04,594 INFO main.py:<module> Processing Incidents finished in 67.33 second(s)
(instance-3_1) | instance-3 2022-05-12 06:22:04,594 INFO main.py:<module>
(instance-3_1) | instance-3 2022-05-12 06:22:04,594 INFO kafka_utils.py:consume_from_topic Try to consume data from topic: local_security_incident
(instance-2_1) | instance-2 2022-05-12 06:22:17,092 INFO main.py:<module> Deleting data for user: 99:93:3d:7d:06:2a if it's more than 100MB
(instance-2_1) | instance-2 2022-05-12 06:22:17,117 INFO kafka_utils.py:produce_to_topic Producing records of events for user: 99:93:3d:7d:06:2a is started
(instance-2_1) | instance-2 2022-05-12 06:22:18,869 INFO kafka_utils.py:produce_to_topic Producing records of events for user: 99:93:3d:7d:06:2a has been finished
(instance-2_1) | instance-2 2022-05-12 06:22:21,035 INFO main.py:<module> Processing Incidents finished in 70.5 second(s)
(instance-2_1) | instance-2 2022-05-12 06:22:21,035 INFO main.py:<module>
(instance-2_1) | instance-2 2022-05-12 06:22:21,035 INFO kafka_utils.py:consume_from_topic Try to consume data from topic: local_security_incident

```

Obrázok 25 – Výpis z konzoly spracovania udalostí niekolkými službami.

Naše dve služby čítajú dátá z partícií č. 1 a č. 2 a každá začala čítať svoje dátá o 8 hodine a 19 minúte podľa UTC +2 alebo o 6 hodine a 19 minúte podľa UTC +0. Zároveň vidíme v logovacích správach aj číslo partície, z ktorej číta služba, ktoré bolo pridané iba pre monitorovacie účely. Hned' vidíme časový rozdiel pri spracovaní udalostí s pomocou manažéra partícií a rovnomerného pridelenia používateľov partíciám a predchádzajúcim testovacím scenárom bez použitia manažéra partícií. Na obrázku č. 26 vidíme, že služba č. 3 ukončila spracovanie poslednej skupiny dát o 8 hodine, 25 minúte a 23 sekunde. A na obrázku č. 27 vidíme kolegovu konzolu, na ktorej je zafixovaný čas odosielania poslednej iterácie dát, a to je o 8 hodine, 24 minúte a 36 sekunde. Ako vidíme z konzoly služby, samotné spracovanie trvalo iba 39 sekúnd, ale služba začala čítať dátá kvôli predchádzajúcemu náročnému spracovaniu neskôr.

```

vklevlanv@vb: ~/Documents/DP/Server
[instance-2_1] 1
[instance-2_1] instance-2 2022-05-12 06:22:43,287 INFO kafka_utils.py:consume_from_topic Data was read for users: dict_keys(['99:93:3d:7d:06:2a'])
[instance-2_1] instance-2 2022-05-12 06:22:43,288 INFO main.py:<module> Start of processing Incidents
[instance-3_1] instance-3 2022-05-12 06:23:12,012 INFO kafka_utils.py:produce_to_topic Producing records of events for user: 5f:72:fa:f5:f8:fa is started
[instance-3_1] instance-3 2022-05-12 06:23:13,008 INFO kafka_utils.py:produce_to_topic Producing records of events for user: 5f:72:fa:f5:f8:fa has been finished
[instance-3_1] instance-3 2022-05-12 06:23:14,142 INFO main.py:<module> Processing Incidents finished in 34.41 second(s)
[instance-3_1] instance-3 2022-05-12 06:23:14,143 INFO main.py:<module>
[instance-3_1] instance-3 2022-05-12 06:23:14,143 INFO kafka_utils.py:consume_from_topic Try to consume data from topic: local_security_incident
[instance-2_1] instance-2 2022-05-12 06:23:14,896 INFO kafka_utils.py:produce_to_topic Producing records of events for user: 99:93:3d:7d:06:2a is started
[instance-2_1] instance-2 2022-05-12 06:23:15,982 INFO kafka_utils.py:produce_to_topic Producing records of events for user: 99:93:3d:7d:06:2a has been finished
[instance-2_1] instance-2 2022-05-12 06:23:17,094 INFO main.py:<module> Processing Incidents finished in 33.81 second(s)
[instance-2_1] instance-2 2022-05-12 06:23:17,094 INFO main.py:<module>
[instance-2_1] instance-2 2022-05-12 06:23:17,094 INFO kafka_utils.py:consume_from_topic Try to consume data from topic: local_security_incident
[instance-3_1] 2
[instance-3_1] instance-3 2022-05-12 06:23:26,136 INFO kafka_utils.py:consume_from_topic Data was read for users: dict_keys(['5f:72:fa:f5:f8:fa'])
[instance-3_1] instance-3 2022-05-12 06:23:26,139 INFO main.py:<module> Start of processing Incidents
[instance-2_1] 1
[instance-2_1] instance-2 2022-05-12 06:24:09,254 INFO kafka_utils.py:consume_from_topic Data was read for users: dict_keys(['99:93:3d:7d:06:2a'])
[instance-2_1] instance-2 2022-05-12 06:24:09,255 INFO main.py:<module> Start of processing Incidents
[instance-3_1] instance-3 2022-05-12 06:24:32,104 INFO main.py:<module> Deleting data for user: 5f:72:fa:f5:f8:fa if it's more than 100MB
[instance-3_1] instance-3 2022-05-12 06:24:32,131 INFO kafka_utils.py:produce_to_topic Producing records of events for user: 5f:72:fa:f5:f8:fa is started
[instance-3_1] instance-3 2022-05-12 06:24:33,189 INFO kafka_utils.py:produce_to_topic Producing records of events for user: 5f:72:fa:f5:f8:fa has been finished
[instance-3_1] instance-3 2022-05-12 06:24:34,556 INFO main.py:<module> Processing Incidents finished in 68.41 second(s)
[instance-3_1] instance-3 2022-05-12 06:24:34,556 INFO main.py:<module>
[instance-3_1] instance-3 2022-05-12 06:24:34,556 INFO kafka_utils.py:consume_from_topic Try to consume data from topic: local_security_incident
[instance-3_1] 2
[instance-3_1] instance-3 2022-05-12 06:24:44,478 INFO kafka_utils.py:consume_from_topic Data was read for users: dict_keys(['5f:72:fa:f5:f8:fa'])
[instance-3_1] instance-3 2022-05-12 06:24:44,501 INFO main.py:<module> Start of processing Incidents
[instance-2_1] instance-2 2022-05-12 06:25:17,346 INFO main.py:<module> Deleting data for user: 99:93:3d:7d:06:2a if it's more than 100MB
[instance-2_1] instance-2 2022-05-12 06:25:17,371 INFO kafka_utils.py:produce_to_topic Producing records of events for user: 99:93:3d:7d:06:2a is started
[instance-2_1] instance-2 2022-05-12 06:25:19,121 INFO kafka_utils.py:produce_to_topic Producing records of events for user: 99:93:3d:7d:06:2a has been finished
[instance-2_1] instance-2 2022-05-12 06:25:20,955 INFO kafka_utils.py:produce_to_topic Producing records of events for user: 5f:72:fa:f5:f8:fa is started
[instance-2_1] instance-2 2022-05-12 06:25:20,964 INFO main.py:<module> Processing Incidents finished in 71.71 second(s)
[instance-2_1] instance-2 2022-05-12 06:25:20,965 INFO main.py:<module>
[instance-3_1] instance-3 2022-05-12 06:25:21,536 INFO kafka_utils.py:produce_to_topic Producing records of events for user: 5f:72:fa:f5:f8:fa has been finished
[instance-3_1] instance-3 2022-05-12 06:25:23,304 INFO main.py:<module> Processing Incidents finished in 38.8 second(s)
[instance-3_1] instance-3 2022-05-12 06:25:23,304 INFO main.py:<module>
[instance-2_1] instance-2 2022-05-12 06:25:45,001 INFO kafka_utils.py:consume_from_topic Data was read for users: dict_keys(['99:93:3d:7d:06:2a'])
[instance-2_1] instance-2 2022-05-12 06:25:45,023 INFO main.py:<module> Start of processing Incidents
[instance-2_1] instance-2 2022-05-12 06:26:14,215 INFO kafka_utils.py:produce_to_topic Producing records of events for user: 99:93:3d:7d:06:2a is started
[instance-2_1] instance-2 2022-05-12 06:26:15,316 INFO kafka_utils.py:produce_to_topic Producing records of events for user: 99:93:3d:7d:06:2a has been finished
[instance-2_1] instance-2 2022-05-12 06:26:16,786 INFO main.py:<module> Processing Incidents finished in 31.76 second(s)
[instance-2_1] instance-2 2022-05-12 06:26:16,786 INFO main.py:<module>
[instance-2_1] instance-2 2022-05-12 06:26:16,786 INFO kafka_utils.py:consume_from_topic Try to consume data from topic: local_security_incident

```

Obrázok 26 – Pokračovanie výpisu z konzoly.

```

User record b'5f:72:fa:f5:f8:fa' successfully produced to local_security_incident [2] at offset 757383
User record b'5f:72:fa:f5:f8:fa' successfully produced to local_security_incident [2] at offset 757384
local      2022-05-12 08:24:36,435 INFO      generate_data_to_kafka.py:<module> Iteration: 4.Data was sent: 151477
local      2022-05-12 08:25:36,493 INFO      generate_data_to_kafka.py:<module> All data was sent.
kubo@kubo-X510UNR:~/Desktop/Test_Client$ 

```

Obrázok 27 – Výpis z konzoly prvého kolegu.

Taktiež aj služba č. 2, ktorá spracovala udalosti z nášho počítača, zvládla posledné spracovanie do minúty, čo vyplýva z porovnania logovacích správ na obrázku č. 28 a žurnálových správ služieb.

```

User record b'99:93:3d:7d:06:2a' successfully produced to local_security_incident [1] at offset 757384
local      2022-05-12 08:25:36,839 INFO      generate_data_to_kafka.py:<module> Iteration: 4.Data was sent: 151477
local      2022-05-12 08:26:36,883 INFO      generate_data_to_kafka.py:<module> All data was sent.

```

Obrázok 28 – Výpis z našej konzoly.

Paralelne so spracovaním vyššie uvedených udalostí ešte jedna služba, ktorá bola spustená na serveri, detegovala incidenty pre ďalšieho kolegu. Výpis žurnálových správ je vidno na obrázku č. 29. Vidíme, že napriek tomu, že posielame tie isté údaje v tom istom množstve, sa časy detegovania incidentov líšia kvôli tomu, že sú na serveri spustené aj iné aplikácie (sprostredkovateľ správ a manažér partícii) a na inom zariadení, na ktorom sme spúšťali vyššie uvedené dve služby, žiadna iná aplikácia nevyužívala výpočtové kapacity zariadenia. Samotné spracovanie posledných 151 tisíc udalostí síce trvalo 54 sekúnd, ale kvôli predchádzajúcemu spracovaniu používateľ čakal po poslednom klientskom odoslaní

správ (žurnálové správy kolejovej konzoly sú zobrazené na obrázku č.32 ) na výsledok detegovania 2 minúty.

```
dp2022@fel-dp-monitoring-2022:~/vasyl/Server$ docker-compose up
Creating instance-1 ... done
Attaching to instance-1
instance-1 | Instance-1 2022-05-12 06:10:29,779 INFO kafka_utils.py:consume_from_topic Try to consume data from topic: local_security_incident
instance-1 | Instance-1 2022-05-12 06:19:24,854 INFO kafka_utils.py:consume_from_topic Data was read for users: dict_keys(['cd:7a:c4:78:ff:9a'])
main.py:<module> Start of processing Incidents
instance-1 | Instance-1 2022-05-12 06:20:19,022 INFO kafka_utils.py:produce_to_topic Producing records of events for user: cd:7a:c4:78:ff:9a is started
instance-1 | Instance-1 2022-05-12 06:20:20,075 INFO main.py:<module> Processing Incidents finished in 56.18 second(s)
instance-1 | Instance-1 2022-05-12 06:20:21,034 INFO main.py:<module> -----
instance-1 | Instance-1 2022-05-12 06:20:21,035 INFO kafka_utils.py:consume_from_topic Try to consume data from topic: local_security_incident
instance-1 | Instance-1 2022-05-12 06:20:32,371 INFO kafka_utils.py:consume_from_topic Data was read for users: dict_keys(['cd:7a:c4:78:ff:9a'])
main.py:<module> Start of processing Incidents
instance-1 | Instance-1 2022-05-12 06:20:32,375 INFO main.py:<module> Deleting data for user: cd:7a:c4:78:ff:9a if it's more than 100MB
kafka_utils.py:produce_to_topic Producing records of events for user: cd:7a:c4:78:ff:9a is started
instance-1 | Instance-1 2022-05-12 06:22:19,151 INFO main.py:<module> Processing Incidents finished in 108.89 second(s)
main.py:<module> -----
instance-1 | Instance-1 2022-05-12 06:22:19,251 INFO kafka_utils.py:produce_to_topic Producing records of events for user: cd:7a:c4:78:ff:9a is started
instance-1 | Instance-1 2022-05-12 06:22:20,302 INFO main.py:<module> Processing Incidents finished in 108.89 second(s)
instance-1 | Instance-1 2022-05-12 06:22:21,265 INFO main.py:<module> -----
instance-1 | Instance-1 2022-05-12 06:22:21,265 INFO kafka_utils.py:consume_from_topic Try to consume data from topic: local_security_incident
instance-1 | Instance-1 2022-05-12 06:22:21,266 INFO kafka_utils.py:consume_from_topic Data was read for users: dict_keys(['cd:7a:c4:78:ff:9a'])
main.py:<module> Start of processing Incidents
instance-1 | Instance-1 2022-05-12 06:22:30,455 INFO kafka_utils.py:produce_to_topic Producing records of events for user: cd:7a:c4:78:ff:9a is started
main.py:<module> -----
instance-1 | Instance-1 2022-05-12 06:22:30,460 INFO main.py:<module> Processing Incidents finished in 55.81 second(s)
instance-1 | Instance-1 2022-05-12 06:23:24,257 INFO kafka_utils.py:produce_to_topic Producing records of events for user: cd:7a:c4:78:ff:9a is started
instance-1 | Instance-1 2022-05-12 06:23:25,309 INFO main.py:<module> Processing Incidents finished in 55.81 second(s)
instance-1 | Instance-1 2022-05-12 06:23:26,268 INFO main.py:<module> -----
instance-1 | Instance-1 2022-05-12 06:23:26,268 INFO kafka_utils.py:consume_from_topic Try to consume data from topic: local_security_incident
instance-1 | Instance-1 2022-05-12 06:23:35,239 INFO kafka_utils.py:consume_from_topic Data was read for users: dict_keys(['cd:7a:c4:78:ff:9a'])
main.py:<module> Start of processing Incidents
instance-1 | Instance-1 2022-05-12 06:23:35,246 INFO main.py:<module> Deleting data for user: cd:7a:c4:78:ff:9a if it's more than 100MB
kafka_utils.py:produce_to_topic Producing records of events for user: cd:7a:c4:78:ff:9a is started
instance-1 | Instance-1 2022-05-12 06:25:21,951 INFO main.py:<module> Processing Incidents finished in 108.77 second(s)
instance-1 | Instance-1 2022-05-12 06:25:22,005 INFO kafka_utils.py:produce_to_topic Producing records of events for user: cd:7a:c4:78:ff:9a is started
instance-1 | Instance-1 2022-05-12 06:25:23,065 INFO main.py:<module> Processing Incidents finished in 108.77 second(s)
instance-1 | Instance-1 2022-05-12 06:25:24,017 INFO main.py:<module> -----
instance-1 | Instance-1 2022-05-12 06:25:24,017 INFO kafka_utils.py:consume_from_topic Try to consume data from topic: local_security_incident
instance-1 | Instance-1 2022-05-12 06:25:24,018 INFO kafka_utils.py:consume_from_topic Data was read for users: dict_keys(['cd:7a:c4:78:ff:9a'])
main.py:<module> Start of processing Incidents
instance-1 | Instance-1 2022-05-12 06:25:32,917 INFO main.py:<module> -----
instance-1 | Instance-1 2022-05-12 06:25:32,921 INFO kafka_utils.py:produce_to_topic Producing records of events for user: cd:7a:c4:78:ff:9a is started
instance-1 | Instance-1 2022-05-12 06:26:24,588 INFO kafka_utils.py:produce_to_topic Producing records of events for user: cd:7a:c4:78:ff:9a has been finished
instance-1 | Instance-1 2022-05-12 06:26:25,840 INFO main.py:<module> Processing Incidents finished in 53.68 second(s)
instance-1 | Instance-1 2022-05-12 06:26:26,602 INFO main.py:<module> -----
instance-1 | Instance-1 2022-05-12 06:26:26,602 INFO kafka_utils.py:consume_from_topic Try to consume data from topic: local_security_incident
instance-1 | Instance-1 2022-05-12 06:26:26,602 INFO main.py:<module> -----
```

Obrázok 29 – Výpis zo serverovej konzoly.

```
User record b'cd:7a:c4:78:ff:9a' successfully produced to local_security_incident [0] at offset 757381
User record b'cd:7a:c4:78:ff:9a' successfully produced to local_security_incident [0] at offset 757382
User record b'cd:7a:c4:78:ff:9a' successfully produced to local_security_incident [0] at offset 757383
User record b'cd:7a:c4:78:ff:9a' successfully produced to local_security_incident [0] at offset 757384
local 2022-05-12 08:23:58,302 INFO generate_data_to_kafka.py:<module> Iteration: 4.Data was sent: 151477
local 2022-05-12 08:24:58,360 INFO generate_data_to_kafka.py:<module> All data was sent.
tomas@tomas-ThinkPad-E550:~/Downloads/Test_Client$
```

Obrázok 30 – Výsledný výpis z konzoly druhého kolegu.

Približný priemerný čas spracovania jednej iterácie pre službu č. 2 a č. 3 je 48 sekúnd a pre službu č. 1 predstavuje 1 minútu a 17 sekúnd. Priemerne sedem minút trvalo každej službe pre spracovanie 757 tisíc vstupných udalostí vrátane odosielania dát a do sprostredkovateľa správ a čakania na čítanie ďalších dát od klienta. Celkovo naša služba spracovala za tento čas 2 milióny a 272 tisíce udalostí a výpočtová kapacita za jednotku času je obmedzená iba počtom služieb a počtom sprostredkovateľov správ v klastri. Čiže pri paralelnom pridaní ďalších inštancií služby do našej serverovej aplikácie, ďalších partícií a od určitého momentu ďalšej inštancie sprostredkovateľa správ by náš systém za tých istých sedem minút spracoval oveľa väčšie množstvo vstupných udalostí. Samozrejme, po pridaní každej jednej inštancie zariadenie prichádza o výpočtovú kapacitu a pri používaní distribuovaných inštancií môže byť výkon systému obmedzený aj prenosovou rýchlosťou dát z a do témy sprostredkovateľa správ.

### 3.2.5 Zhrnutie výsledkov testovania

Podľa vyššie uvedených výsledkov testovania sme vytvorili tabuľku č. 1, v ktorej sme uviedli vypočítané výsledky všetkých typov testovania podľa rôznych kritérií.

Typ vyhodnotenia výsledkov/ Typ testovania	Sekvenčné spracovanie	Paralelné spracovanie bez MP	Paralelné spracovanie s MP
Počet vyhodnotených vstupných udalostí za testovanie	1 514 770	2 272 155	2 272 155
Priemerný čas spracovania udalostí za 1 iteráciu 1 službou	139 sek	133 sek	62 sek
Celkový čas spracovania všetkých udalostí v rámci testovania	750 sek	780 sek	446 sek

*Tabuľka 1 – Porovnanie výsledkov testovania rôznych typov spracovania udalostí.*

Ako vidíme v tabuľke, paralelné spracovanie bez použitia manažéra partícií celkovo vložilo výsledky o 30 sekúnd neskôr ako sekvenčné spracovanie, pretože na tom istom serveri bežali hned tri inštancie, ktoré používali výpočtové kapacity serveru namiesto jednej inštancie ako pri sekvenčnom vykonaní. Ale vďaka tomu, že mal systém pri paralelnom spracovaní dve inštancie služby, ktoré pracovali paralelne, vyhodnotil tento typ spracovania incidentov dva krát viac incidentov (tretia inštancia nespracovala žiadne dátu). Avšak výkon prvých dvoch typov spracovania je oveľa menší ako výkon pri spracovaní incidentov pomocou manažéra partícií.

Náš systém zvláda sekvenčné spracovanie dát jednou službou s časovým oneskorením približne 2 minút na spracovanie každého používateľa pri odosielaní 150 tisíc dát každou iteráciou. Pre menšie skupiny dát a pri minimálnom množstve používateľov, ktorí paralelne posielajú svoje udalosti na vstup, by náš systém zvládal spracovanie v čase priblíženom k reálnemu. Avšak pre reálne alebo produkčné používanie systému sekvenčné spracovanie udalostí nezvládne spracovanie paralelne odosielaného veľkého množstva dát bez oneskorenia, ktoré je kritické pre reagovanie na detegovanie bezpečnostných útokov.

Taktiež používanie paralelného spôsobu detegovania incidentov bez manažéra partícií zvyšuje množstvo vyhodnotených udalostí za ten istý čas, ako aj sekvenčné spracovanie,

skoro dvojnásobne. Ale jeho správnosť pridelenia používateľských udalostí k jednotlivým partíciám je veľmi závislé od hašovacieho kódu, čo spôsobuje nerovnomerné rozdelenie záťaže počas paralelného vkladania udalostí viacerými používateľmi. Práve pre riešenie danej problémy bol zavedený manažér partícií, ktorý rovnomerne rozdeľuje množstvo používateľov do každej partície, a tým pádom nám zaručuje maximálny výkon pri paralelnom spracovaní incidentov a vylučuje prípad, keď množstvo používateľov je rovné alebo väčšie ako množstvo partícií, a pri tom by nejaká služba spracovávala niekoľkých používateľov a nejaká služba by bola spustená naprázdno.

# Záver

V danej práci sme rozobrali princípy navrhovania softvérových produktov pomocou rôznych architektúr a zhodnotili sme výhody a nevýhody každej architektúry s ohľadom na naše požiadavky pre systém detegovania a spracovávania bezpečnostných incidentov. Taktiež sme navrhli a implementovali klientsku aplikáciu pre vkladanie udalostí do nášho systému na detekciu, manažér partícií pre rovnomenné rozdelenie používateľských dát do dostupných partícií, serverovú službu, ktorá paralelne spracováva bezpečnostné incidenty pomocou viacerých inštancií a rozbehali a nastavili sme pre vlastné potreby už vytvorený sprostredkovateľ správ s tým, že sme pridali ukladanie dát do relačnej databázy PostgreSQL.

V poslednej časti sme otestovali svoje riešenie pomocou záťažového testovania pri paralelnom odosielaní veľkého množstva vstupných udalostí. Pri testovaní sme sa uistili, že systém pracuje tak, ako bol navrhnutý – zvláda paralelné spracovanie dát a výkon je postačujúci pre dlhodobé odosielanie Sysmon žurnálových správ alebo ASTD udalostí aj vo veľmi veľkom množstve v rovnakom čase. Pri testovaní sme potvrdili, že prieplustnosť nášho systému priamo závisí od množstva služieb, ktoré ich spracovávajú, od správneho rozdelenia používateľských identifikátorov do jednotlivých partícií, čo sme ošetrili pomocou manažéra partícií, a od výpočtových kapacít zariadenia, na ktorom inštancia služby bežia.

Systém pre detegovanie bezpečnostných incidentov je spravený a pripravený na prevádzku. Je navrhnutý tak, že pridanie novej služby žiadnym spôsobom neovplyvní beh už vytvorenej služby. Stačí pridať novú službu a pripojiť ju k vstupnej téme s iným *group-id* – identifikátorom skupiny. Taktiež sa celý systém okrem klientskej aplikácie nachádza vo virtuálnom prostredí Docker kontajnerov, čo umožňuje spustenie na hoci ktorom zariadení bez toho, aby to zariadenie potrebovalo inštalovanie nejakých závislostí okrem samotných Docker a Docker Compose aplikácií. Spustenie a vypnutie týchto kontajnerov je automatizované pomocou Docker Compose súboru a klientska aplikácia sa spúšťa štartovacím skriptom.

Samozrejme, ako každý softvérový produkt, má náš systém ešte priestor na zlepšenie. Pridanie jednotkových testov, automatizovanie spúšťania ďalších inštancií služby spolu s pridaním ďalších partícií pri zväčšení záťaže na systém a pridanie aplikácií pre zbieranie

metrík výstupných hrozieb, ako sú Grafana, Kibana a Prometheus, by zlepšilo hodnotu nášho systému v prevádzke a zjednodušilo by reagovanie na rôzne zmeny v systéme.

# Zoznam použitej literatúry

- [1] RICHARDS, Mark. *Software Architecture Patterns*. USA: O'Reilly Media, Inc., 2015. Print. 55 s. ISBN: 978-1-491-92424-2.
- [2] FORD, Neal, PARSONS, Rebecca, KUA, Patrick. *Building Evolutionary Architectures*. USA: O'Reilly Media, Inc., 2017 Print. 176s. ISBN: 978-1-491-98636-3.
- [3] KLEPPMANN, Martin. *Designing Data-Intensive Applications*. USA: O'Reilly Media, Inc., 2017. Print. 640s. ISBN: 978-1-449-37332-0.
- [4] STOPFORD, Ben. *Designing Event-Driven Systems*. USA: O'Reilly Media, Inc., 2018. Print. 199s. ISBN: 978-1-492-03824-5.
- [5] DEAN, Alexander, CRETTEAZ, Valentin. *Event Streams in Action*. USA: Manning Publications., 2019. Print. 344s. ISBN: 978-1-617-29234-7.
- [6] KLEPPMANN, Martin. *Designing Data-Intensive Applications*. USA: O'Reilly Media, Inc., 2017. Print. 640s. ISBN: 978-1-449-37332-0.
- [7] SCOTT, Dylan, GAMOV, Viktor, KLEIN, Dave. *Kafka in Action*. USA: Manning Publications., 2022. Print. 272s. ISBN: 978-1-617-29523-2.
- [8] MIELL, Ian, SAYERS, Hobson. *Docker in Practice, 2nd Edition*. USA: Manning Publications., 2019. Print. 384s. ISBN: 978-1-617-29480-8.
- [9] TIDJON, Lionel, FRAPPIER, Marc, LEUSCHEL, Michael, MAMMAR, Amel. *Extended Algebraic State-Transition Diagrams*. Austrália: IEEE, 2018. [online]. Medzinárodná konferencia. ISBN: 978-1-5386-9341-4. Dostupné z : <https://ieeexplore.ieee.org/document/8595068>

# **Prílohy**

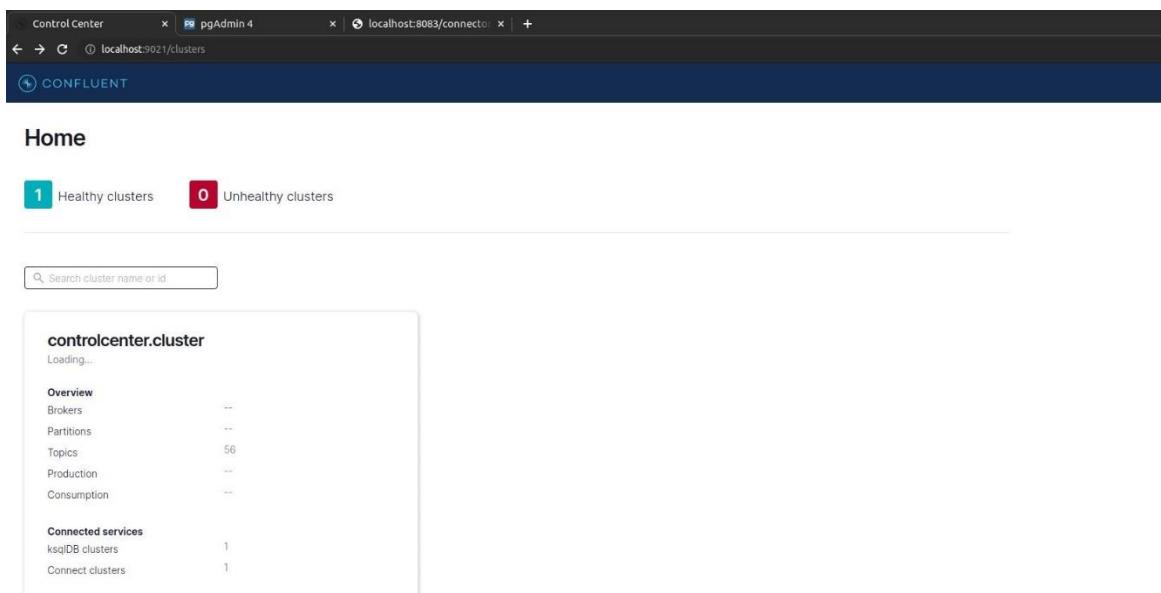
<b>A Nastavenie sprostredkovateľa správ</b>	II
<b>B Spustenie systému spracovania incidentov</b>	VI
<b>C Používateľská príručka</b>	VII

# A Nastavenie sprostredkovateľa správ

Sprostredkovateľa správ Apache Kafka ako aj databázu pre monitorovanie stavu v priečinku */DP/Kafka/* spúšťame pomocou Docker Compose súboru príkazom:

```
$ docker-compose up -d
```

Týmto príkazom sa naštartujú všetky potrebné komponenty, ako napríklad Zookeeper, Schema-Registry alebo Kafka Connect, a spolu s tým aj Kafka Control Center, ktorý je realizáciou GUI pre Apache Kafka od spoločnosti Confluent.



Obrázok A.1 – Dostupný Apache Kafka klaster v Control Center.

Po spustení */docker-compose.yml* súboru na adrese localhost:9021/clusters sa zobrazí dostupný Apache Kafka klaster, ako je zobrazené na obrázku A.1. Po kliknutí na klaster sa otvorí prehľad jeho možností a nastavenia. Po prechode na riadok *Topics* v ľavom menu prechádzame na stránku nastavenia tém pre náš klaster, ako je zobrazené na obrázku A.2. Ďalej potrebujeme vytvoriť dve nové témy – *local\_security\_incident* pre vstupné udalosti a *scanning\_result* pre výstupné udalosti.

Topic name	Partitions
_connect-configs	1
_connect-offsets	25
_connect-status	5
confluent_rmoff_0ksql_processing_log	1

Obrázok A.2 – Prehľad vytvorených témy v Kafka Control Center.

Vytvoriť novu tému sa dá dvoma spôsobmi. Pomocou použitia GUI môžeme kliknúť na tlačidlo *+ Add a topic* a v novom otvorenom okne zadáme názov témy a počet partícií a stlačíme tlačidlo *Create with defaults*. Druhý spôsob je pomocou konzoly zadať príkaz, ktorý sa dostane dovnútra Docker kontajneru Apache Kafka sprostredkovateľa a vytvorí tému pomocou priameho konzolového príkazu:

```
$ docker-compose exec broker kafka-topics --bootstrap-server broker:9092 --topic local_security_incident --create --replication-factor 1 --partitions 4
```

Týmto príkazom vytvárame pre sprostredkovateľa *broker1* tému *local\_security\_incident* so štyrmi partíciami a jednou replikáciou. Analogickým príkazom vytvoríme aj výstupnú tému:

```
$ docker-compose exec broker kafka-topics --bootstrap-server broker:9092 --topic scanning_result --create --replication-factor 1 --partitions 4
```

Po zadaní týchto príkazov do konzoly alebo ručného vytvorenia tému pomocou Control Center sa objavia tieto témy na stránke *Topics*, ako to je ukázané na obrázku A.3.

The screenshot shows the 'Topics' section of the Confluent Control Center. On the left, there's a sidebar with links like 'Cluster overview', 'Brokers', 'Topics' (which is selected and highlighted in blue), 'Connect', 'ksqlDB', 'Consumers', 'Replicators', and 'Cluster settings'. The main area has a search bar ('Search topics') and a toggle switch ('Hide internal topics'). A button '+ Add a topic' is at the top right. Below is a table with columns 'Topic name' and 'Partitions'. The table lists several topics with their partition counts:

Topic name	Partitions
_connect-configs	1
_connect-offsets	25
_connect-status	5
confluent_rmoff_01ksel_processing_log	1
local_security_incident	4
scanning_result	4

Obrázok A.3 – Zobrazenie novovytvorených tém pre sprostredkovateľa správ.

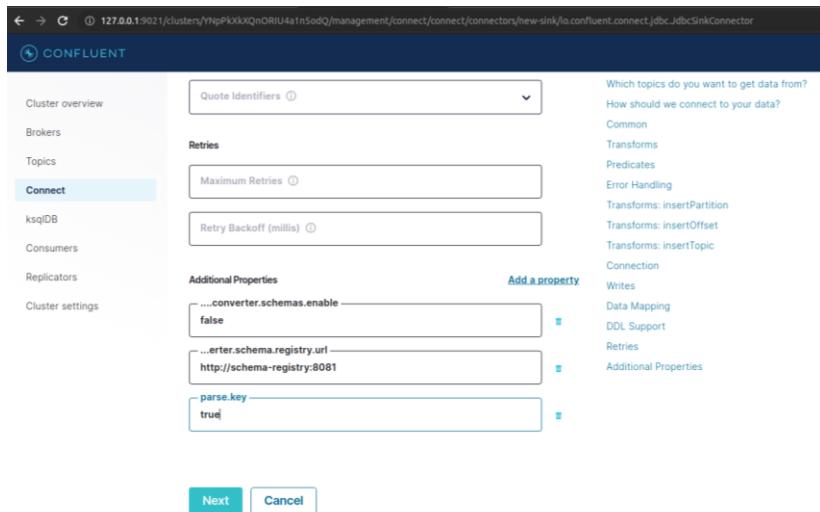
Ďalej potrebujeme pre monitorovanie výsledkov nastaviť vstupné a výstupné prepojovače Kafka JDBC Connect Sink. V ľavom menu zvolíme položku *Connect*. V otvorenom okne klikneme na riadok *connect* a vďaka tomu sa otvorí nové okno pridania prepojovača, ako je zobrazené na obrázku A.4.

The screenshot shows the 'Connectors' section of the Confluent Control Center. The left sidebar includes 'Cluster overview', 'Brokers', 'Topics', and 'Connect' (selected). The main area has a 'Browse' heading, a 'Filter by category' dropdown, and an 'Upload connector config file' button. It displays several connector components arranged in a grid:

- JdbcSinkConnector**: Sink, with a 'Connect' button.
- JdbcSourceConnector**: Source, with a 'Connect' button.
- FileStreamSinkConnector**: Sink, with a 'Connect' button.
- FileStreamSourceConnector**: Source, with a 'Connect' button.
- MirrorHeartbeatConnector**: Source, with a 'Connect' button.
- MirrorSourceConnector**: Source, with a 'Connect' button.
- MirrorCheckpointConnector**: Source, with a 'Connect' button.

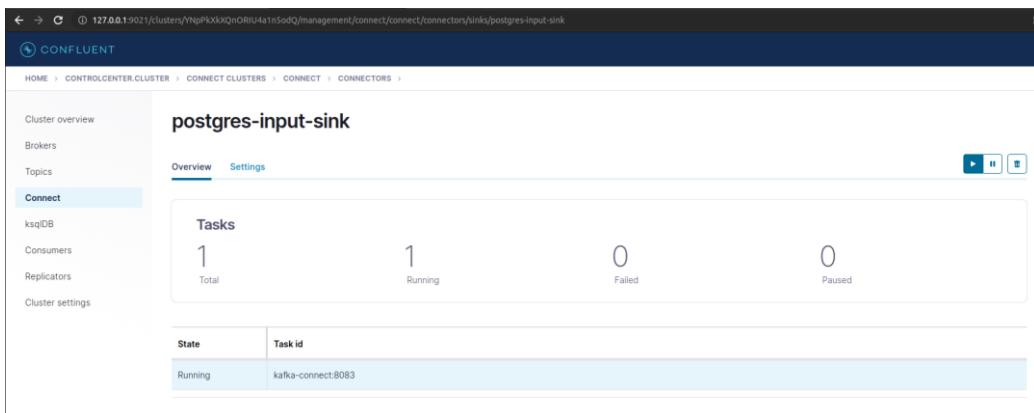
Obrázok A.4 – Menu prepojovačov.

V otvorenom menu klikneme na tlačidlo *Upload connector config file* pre načítanie konfiguračných súborov, ktoré sa nachádzajú v priečinku */DP/Kafka/* a to sú: */postgres-input-sink.json* a */postgres-output-sink.json*. Tieto súbory sa načítavajú postupne, najprv jeden, potom druhý v ľubovoľnom poradí. Po vybraní jedného z nich sa nám zobrazí jeho konfigurácia, ktorá je zobrazená na obrázku A.5.



Obrázok A.5 – Nastavenie prepojovača.

V tomto konfiguračnom okne môžeme pridať nejakú konfiguráciu alebo zmeniť existujúcu (napr. pri zmene prihlásovacích údajov k databáze – v danom okne ich treba prepísať na nové pre správne prepojenie k databáze). Ďalej treba potvrdiť konfiguráciu stlačením tlačidla *Next* a ešte raz ju potvrdiť v novom okne stlačením *Launch*. Po pridaní prepojovača sa nám zobrazí okno, ktoré je uvedené na obrázku A.6, čo znamená, že nás prepojovač beží.



Obrázok A.6 – Okno s jedným spusteným prepojovačom.

Po zopakovaní procesu pridania druhého konfiguračného súboru je sprostredkovateľ správ pripravený na použitie. Pre zobrazenie stavu prepojovača môžeme prejsť na URL a skontrolovať jeho stav:

```
$ http://localhost:8083/connectors/{názov_prepojovača}/status
```

## B Spustenie systému spracovania incidentov

Náš systém je kontajnerovaný pomocou Docker a Docker Compose, čo veľmi uľahčuje spustenie aplikácií a inštaláciu závislosti. Pred spustením nášho systému je potrebné sa uistíť, že máme nainštalované aplikácie Docker a Docker Compose. Ak nie sú nainštalované, tak ich potrebujeme nainštalovať. Potom otvoríme priečinok */DP/Kafka/* a zopakujeme postup spustenia a nastavovania sprostredkovateľa správ z prílohy A. Ďalej otvoríme priečinok */DP/Server/* a vytvoríme obraz našej serverovej služby pomocou príkazu:

```
$ docker build . -f Dockerfile -t backend_diploma_thesis --rm
```

Hned' po vytvorení obrazu spustíme 4 služby spracovania udalostí (pri potrebe zmeniť množstvo inštancií služieb treba otvoriť súbor */docker-compose.yml*, pridať alebo odstrániť potrebné množstvo inštancií služieb a prispôsobiť množstvo partícií pre vstupnú a výstupnú tému) pomocou príkazu na spustenie kontajnerov:

```
$ docker-compose up -d
```

Týmito príkazmi máme spustený sprostredkovateľ správ a štyri inštancie služieb detegovania a spracovania správ. Ďalej spúšťame manažér partícií v priečinku */DP/partition-manager/*. Tak ako aj pri službe spracovania potrebujeme najprv vytvoriť obraz aplikácie pomocou príkazu:

```
$ docker build . -f Dockerfile --build-arg JAR_FILE=target/partition-manager-1.0.0.jar -t partition_manager --rm
```

A ukončujeme inštaláciu systému pomocou spustenia posledného obrazu už známym príkazom:

```
$ docker-compose up -d
```

## C Používateľská príručka

Po spustení všetkých komponentov služby spracovania incidentov pomocou prílohy B potrebujeme spustiť klientsku aplikáciu na svojom zariadení. Ak máme vlastne údaje, tak použijeme klientsku aplikáciu alebo ak chceme použiť testovacie dátá, použijeme testovaciu klientsku aplikáciu. Klientska aplikácia sa nachádza v priečinku */DP/Client/* a spúšťa sa pomocou príkazu, ktorý nainštaluje všetky potrebné závislosti a spustí klientsku aplikáciu:

```
$ ./startup.sh
```

Počas behu programu môžeme vkladať dátá pre spracovanie: ASTD udalostí do priečinku */astd\_events/* a Sysmon žurnálové správy vo formáte XML do priečinku */sysmon\_logs/*. Po vyhodnotení udalostí nájdete výsledky detegovania incidentov v databáze *Kafka* v schéme *public* a v tabuľke *scanning\_result* podľa vlastného *user\_id*, ktorý sa počas odosielania dát zobrazí v konzole. V prípade potreby môžeme skontrolovať aj vstupné udalosti, ktoré sa nachádzajú v tej istej databáze a schéme, ale v tabuľke *local\_security\_incident*. Aplikácia sa vypína pomocou príkazu:

```
$ ./shutdown.sh
```

Testovacia klientska aplikácia sa nachádza v priečinku */DP/Test\_Client/* a spúšťa sa tak isto ako aj obyčajná, len nepotrebuje dátá a vypínací skript, pretože sa vypne približne po šiestich až siedmych minútach svojej činnosti. Výstupné dátá budú tak isto zobrazené v databáze ako aj pre obyčajnú klientsku aplikáciu.