Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií
FIIT-5212-102904

Matej Dudák

Komunikácia medzi mikroslužbami

Vedúci práce: Ing. Molnár Eugen, PhD.

Máj 2023

Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií
FIIT-5212-102904

Matej Dudák

Komunikácia medzi mikroslužbami

Študijný program: Informatika

Študijný odbor: Informatika

Školiace pracovisko: Ústav informatiky, informačných systémov a softvérového inžinierstva

Vedúci práce: Ing. Molnár Eugen, PhD.

Máj 2023

Slovenská technická univerzita v Bratislave
Ústav informatiky, informačných systémov a softvérového inžinierstva

Fakulta informatiky a informačných technológií
Akademický rok: 2022/2023
Evidenčné číslo: FIIT-16768-102904

ZADANIE BAKALÁRSKEJ PRÁCE

Študent:	**Matej Dudák**
ID študenta:	102904
Študijný program:	informatika
Študijný odbor:	informatika
Vedúci práce:	Ing. Eugen Molnár, PhD.
Vedúci pracoviska:	doc. Ing. Valentino Vranić, PhD.

Názov práce: **Komunikácia medzi mikroslužbami**

Jazyk, v ktorom sa práca vypracuje: slovenský jazyk

Špecifikácia zadania:

Aplikácia založená na mikroslužbách predstavuje distribuovaný systém zahrňujúci spoluprácu viacerých mikroslužieb, zvyčajne dokonca na viacerých serveroch alebo hostiteľoch. Každá inštancia služby je zvyčajne proces. Služby preto musia navzájom interagovať a komunikácia medzi mikroslužbami zohráva kľúčovú úlohu. Mikroslužby sú často súčasťou cloudu a nasadzované prostredníctvom procesov DevOps v kontajneroch. Udalosťami riadená architektúra (event-driven architecture) umožňuje efektívnu implementáciu logiky riadenia mikroslužieb (ich choreografiu). Analyzujte spôsoby komunikácie medzi mikroslužbami v rámci vybranej platformy na orchestráciu kontajnerov. Určte, ktorý prístup komunikácie medzi mikroslužbami, nasadenými v jednotlivých podoch (Kubernetes) / kontajneroch (Docker) je vhodný pri implementácii udalosťami riadenej architektúry (event-driven architecture). Implementujte navrhnuté scenáre, analyzujte dosiahnuté výsledky a špecifikujte plusy a mínusy každého z nich. Ako systém pre orchestráciu kontajnerov je preferovaný Kubernetes, alternatívne je možné použiť Docker Swarm.

Rozsah práce: 40

Termín odovzdania bakalárskej práce:	22. 05. 2023
Dátum schválenia zadania bakalárskej práce:	23. 11. 2021
Zadanie bakalárskej práce schválil:	doc. Ing. Valentino Vranić, PhD. – garant študijného programu

ČESTNÉ PREHLÁSENIE

Čestne vyhlasujem, že som túto prácu vypracoval samostatne, na základe konzultácií a s použitím uvedenej literatúry.

V Bratislave, 22. máj 2023

.....

Matej Dudák

POĎAKOVANIE

Chcem sa poďakovať Ing. Eugenovi Molnárovi, PhD. za jeho usmerňovanie a cenné rady pri písaní tejto práce. Takisto sa chcem poďakovať mojim kamarátom z Univerzitného pastoračného centra v Bratislave, ktorí ma pri písaní práce podporovali.

Anotácia

Slovenská technická univerzita v Bratislave

Fakulta informatiky a informačných technológií

Študijný program: Informatika

Autor: Matej Dudák

Bakalárska práca: Komunikácia medzi mikroslužbami

Vedúci bakalárskej práce: Ing. Molnár Eugen, PhD.

Máj 2023

Komunikácia medzi mikroslužbami v udalosťami riadenej architektúre je asynchrónna a prebieha prostredníctvom sprostredkovateľa udalostí (event broker). Dôležitou oblasťou je konzumovanie správ, uložených v sprostredkovateľovi udalostí (event broker) v takom poradí, v akom boli do neho zapísané. Táto bakalárska práca sa zameriava na návrh riešenia uvedeného problému. Cieľom práce je popísať prístupy pri riešení uvedeného problému, navrhnúť a implementovať riešenie, ktoré umožní aplikácii konzumovať správy zo sprostredkovateľa udalostí v takom poradí, v akom boli do neho publikované. V práci sú analyzované vybrané scenáre a navrhnuté riešenie. Výsledkom práce je návrh, implementácia a testovanie riešenia.

Annotation

Slovak University of Technology Bratislava

Faculty of Informatics and Information Technologies

Degree Course: Informatika

Author: Matej Dudák

Bachelor Thesis: Communication among microservices

Supervisor: Ing. Molnár Eugen, PhD.

May 2023

Communication among microservices in event driven architecture is asynchronous and runs by event broker. Important part is consuming messages, which are stored in event broker, in such order, as they were written to it. This bachelor thesis aims to design solutions to the mentioned problem. The aim is to describe approaches in solving the problem, design and implement solution, which enables consuming messages from event broker in such order, as they were published to it. The result of the work is design, implementation, and testing of the solution.

OBSAH

ZOZNAM OBRÁZKOV

ZOZNAM TABULIEK

ÚVOD

Systémy využívajúce mikroslužby sú čoraz populárnejšie. Oproti monolitickým systémom majú mnohé výhody. Niektoré prínosy architektúry mikroslužieb podľa Richardsona (1) :

- Umožňujú kontinuálne doručovanie a nasadzovanie komplexných aplikácií
- Jednotlivé mikroslužby sú jednoducho udržateľné, samostatne nasaditeľné a nezávisle škálovateľné
- Architektúra mikroslužieb umožňuje tímom pracovať na vývoji autonómne
- Umožňuje jednoduché experimentovanie a adaptovanie nových technológií
- Lepšie izolujú chyby a podporujú vysokú dostupnosť

Tieto výhody sú nepochybne prínosom pre vývoj systémov, avšak, architektúra mikroslužieb predstavuje určitú komplexitu, ktorá sa najviac odráža v komunikácii medzi mikroslužbami. V monolitickom systéme je celá hlavná biznis logika vykonávaná v jednej aplikácii, zvyčajne na jednom stroji. Komunikácia prebieha iba medzi databázou, klientskymi aplikáciami, prípadne nejakými externými službami, z ktorých sú získavané dodatočné údaje. Táto aplikácia zahŕňa všetky potrebné funkcionality, ktoré môžu byť navzájom poprepájané. Nevýhodou monolitických systémov je neefektívna škálovateľnosť, slabá spoľahlivosť, alebo nízka flexibilita. Ak by sme ju rozdelili a každá funkcionalita by bola implementovaná ako samostatná mikroslužba, bol by počet interakcií medzi jednotlivými biznis logikami výrazne väčší a komplexnejší.

Komunikácia medzi mikroslužbami môže byť synchrónna, alebo asynchrónna. Synchrónna komunikácia je najčastejšie realizovaná pomocou komunikačnej architektúry REST, ktorá využíva HTTP protokol. Každá služba má definované rozhranie API, ktoré poskytuje možnosti pre získavanie, vytváranie, aktualizovanie a mazanie údajov. Nevýhodou synchrónnej komunikácie je vysoká sieťová závislosť a nevyhnutná dostupnosť všetkých navzájom prepojených mikroslužieb. To znamená, že každá mikroslužba musí mať informácie o všetkých mikroslužbách, s ktorými potrebuje komunikovať. Tiež je potrebné, aby v danom momente, kedy jedna mikroslužba posiela alebo sa dopytuje na údaje od inej mikroslužby, boli obe mikroslužby – odosielateľ a prijímateľ, dostupné.

Asynchrónna komunikácia tento nedostatok odstraňuje. V asynchrónnej komunikácii sa neposielajú údaje medzi dvomi mikroslužbami priamo, ale mikroslužby posielajú údaje na tému (kanál) a po prihlásení sa na odber z nej ich prijímajú. Tým pádom jednotlivé

mikroslužby nemusia navzájom o sebe vedieť, ale stačí, ak poznajú tému a štruktúru správy, ktorá sa po danej téme posiela. Asynchrónna komunikácia tiež rieši problém s dostupnosťou. Jedna mikroslužba údaje odošle, a druhá si ich prečíta, keď bude k dispozícii. Navrhnutie systému s asynchrónnou komunikáciou však môže byť veľmi náročné. Preto existujú architektúry, ktoré na to môžeme využiť.

Udalosťami riadená architektúra je jedným z riešení na navrhnutie asynchrónnej komunikácie. Tá pozostáva z troch komponentov: producent (producer), konzument (consumer) a sprostredkovateľ udalostí (event broker). Najčastejšie sa udalosťami riadená architektúra implementuje pomocou sprostredkovateľa udalostí Apache Kafka. V Apache Kafka producent odosiela udalosti na témy a sprostredkovateľ udalosti ukladá do partícií. Tie ďalej preposiela konzumentovi, ktorý je prihlásený na odber udalostí z jednej alebo viacerých tém. Keďže prebieha komunikácia asynchrónne, čo znamená, že odosielateľ nemusí čakať na potvrdenie o prijatí správy, môže sa stať, že správy dorazia v inom poradí, v akom boli odoslané. Tento problém je aktuálnou témou a jeho riešenie závisí od konkrétneho prípadu použitia. Z uvedeného vyplýva, že medzi mikroslužbami existuje úložisko záznamov, do ktorého jednotlivé mikroslužby záznamy ukladajú a odkiaľ ich aj čítajú. Zákonite vyvstáva otázka: "*Ako dosiahnuť zaručenie toho, že záznamy budú jednotlivými mikroslužbami čítané presne v tom poradí, v akom boli do úložiska zapísané*?"

V tejto práci je práve rozobrané riešenie uvedeného problému pomocou najpoužívanejšieho sprostredkovateľa udalostí Apache Kafka. V analytickej časti je všeobecne opísaná udalosťami riadená architektúra sprostredkovateľ udalostí a podrobne opísaný Apache Kafka, jeho komponenty a ich konfigurácia súvisiaca s našim problémom. Ďalej je popísaná špecifikácia problému a požiadavky na jeho riešenie. V návrhu riešenia je popísaná taká konfigurácia Apache Kafka, ktorá umožňuje konzumovanie správ v rovnakom poradí, v akom boli produkované. Implementácia tohto návrhu je popísaná v šiestej kapitole. V siedmej kapitole je popísané overenie riešenia – dôkaz správneho fungovania testami.

ANALÝZA RIEŠENÉHO PROBLÉMU

1. Udalosťami riadená architektúra

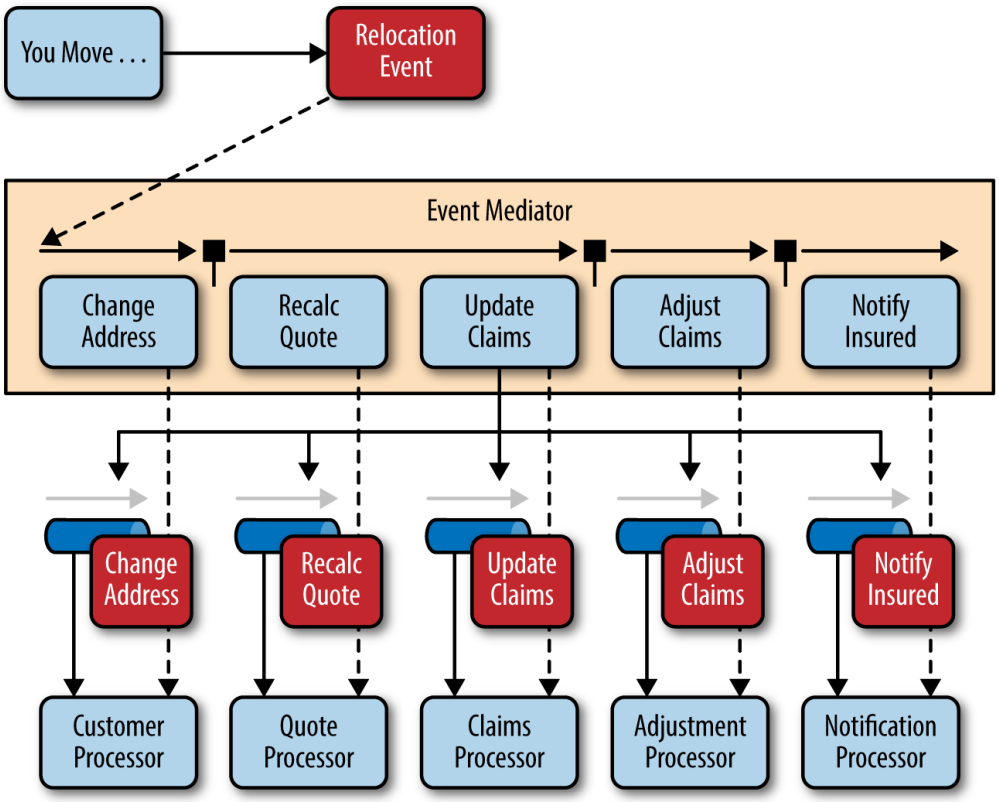
Udalosťami riadená architektúra je vzor architektúry softvéru, ktorý podporuje produkovanie a konzumovanie významných reakcií na významné zmeny stavu systému (známe ako udalosti) a reakciu na ne. Táto architektúra sa tiež označuje ako asynchrónna komunikácia, keďže nahrádza klasickú synchrónnu request/response komunikáciu. Uplatňuje sa to prostredníctvom návrhu a implementácie aplikácií a systémov, ktoré prenášajú udalosti medzi voľne prepojenými softvérovými komponentmi a službami. V mnohých prípadoch sa udalosťami riadená architektúra používa spolu s architektúru orientovanou na služby, ktorú dopĺňa, pretože tieto služby sa môžu aktivovať spúšťačmi na prichádzajúce udalosti (2).

Na implementáciu udalosťami riadenej architektúry sa najčastejšie používajú dve topológie: mediátor a sprostredkovateľ (3).

1.1 Mediátor

Topológia mediátor sa používa pri udalostiach, ktoré vyžadujú viac krokov pre ich spracovanie, a tak je potrebná istá orchestrácia (3). Táto topológia sa skladá zo štyroch komponentov: rad správ, mediátor, kanál správ a služba spracovania udalostí. Priebeh udalosti v topológii mediátora začína odoslaním počiatočnej udalosti klientom na rad správ, cez ktorý sa udalosť dostane k mediátorovi. Následne mediátor rozhodne, do ktorého kanálu správ ju má poslať. Z kanálu správ bude udalosť prečítaná a spracovaná vhodnou službou, ktorá udalosť spracuje. Príklad takéhoto priebehu je zobrazený na Obr. 1.1, kde je popísaná potrebná komunikácia v prípade zmeny bydliska. Pre každú počiatočnú udalosť, mediátor vytvorí udalosť spracovania (zmena adresy, prepočítanie cenovej ponuky, ...), odošle udalosť spracovania do kanála udalostí a čaká, kým danú udalosť spracuje proces udalostí. To pokračuje, až kým nie sú všetky počiatočné udalosti spracované. Na obrázku sú udalosti, ktoré je možné spracovať súčasne, zobrazené jednou šípkou nad nimi.

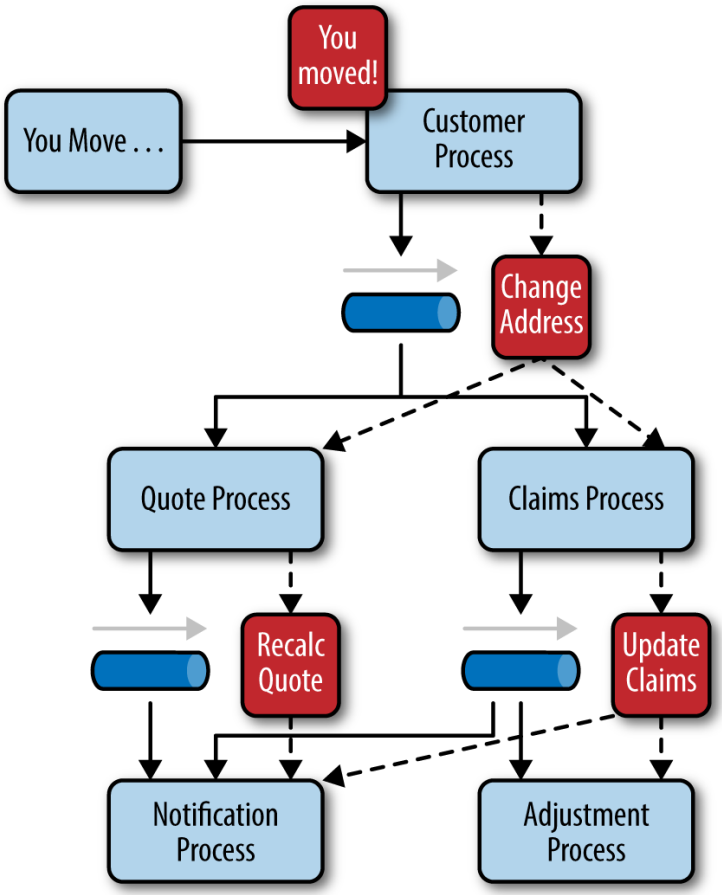
Mediátor je zodpovedný za orchestráciu krokov počiatočnej udalosti, ale nevykonáva žiadnu biznis logiku. Kanály správ sa používajú na asynchrónnu komunikáciu medzi mediátorom s službami.



Obr. 1.1 Diagram topológie Mediátor (3 s. 15)

1.2 Sprostredkovateľ udalostí

Ďalšou topológiou použitou na implementáciu udalosťami riadenej architektúry je sprostredkovateľ udalostí. Topológia sprostredkovateľa udalostí na rozdiel od topológie mediátora nemá centrálneho spracovateľa udalostí. Prúd udalostí sa reťazovo rozdeľuje medzi službami prostredníctvom sprostredkovateľa. Táto topológia je veľmi užitočná, keď je našou úlohou vytvorenie relatívne jednoduchého toku spracovania udalostí bez potreby centrálnej orchestrácie (3).



Obr. 1.2 Diagram topológie Sprostredkovateľ udalostí (3 s. 17)

V rámci topológie sprostredkovateľa udalostí existujú dva hlavné typy komponentov: samotný sprostredkovateľ udalostí a služby spracovania udalostí. Sprostredkovateľ udalostí obsahuje všetky kanály správ, ktoré sa používajú v rámci toku udalostí. Kanály môžu byť buď rady (Message Queues), témy (Topics) alebo ich kombinácia. V danej realizácii je každá služba zodpovedná za spracovanie udalostí a prípadné publikovanie novej udalosti označujúcej akciu, ktorá bola práve vykonaná. Reakciou na novú udalosť je jej spracovanie ďalšou službou. Na Obr. 1.2 je opäť prípad zmeny bydliska, realizovaný s použitím sprostredkovateľa. V tejto realizácii môžeme hovoriť o komunikácii medzi službami ako o predávaní štafety. Po prijatí udalosti prvým procesom sa táto udalosť spracuje a výsledok sa pošle ďalšiemu procesu. Tak to ide, až kým sa celý proces nedostane do cieľa. Ako je na obrázku zobrazené, udalosti sa v rámci spracovania môžu rozdeľovať na viaceré a posielať do niekoľkých kanálov. Takisto je možné udalosti spájať do jednej (4). Výhodu tejto architektúry predstavuje veľmi vysoká škálovateľnosť komponentov softvérového produktu.

Na rozdiel od realizácie pomocou mediátora môže daná topológia v každom mieste zväčšiť svoju priepustnosť pomocou vytvorenia nových inštancií. Ak sprostredkovateľ nebude zvládať množstvo vstupných udalostí, tak môžeme vytvoriť ďalšie inštancie sprostredkovateľov a spojiť ich do klastra. Klaster obsahuje niekoľko inštancií rovnakých aplikácií, ktoré si navzájom rozdeľujú vstupné udalosti, najčastejšie podľa metódy Round-Robin (5). Tým pádom môže niekoľko sprostredkovateľov zvládnuť oveľa väčšiu záťaž, ktorú by iba jedna inštancia nezvládla, pretože sa sieťová a výpočtová záťaž medzi nich rovnomerne rozdelí. Okrem toho si môžu aj samotné služby rozdeľovať záťaž medzi svojimi inštanciami.

Ďalšia veľká výhoda, ktorá spravila túto realizáciu veľmi populárnou na trhu softvérových produktov, je veľká odolnosť voči chybám a poruchám. Vďaka 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. Ak jedna inštancia prestane fungovať, napríklad kvôli výpadku siete, alebo nedostatku výpočtových zdrojov, aplikácia bude stále dostupná pomocou iných inštancií, ktoré bežia na iných funkčných serveroch. Tiež sa dá pomocou sprostredkovateľa udalostí predísť strate údajov v prípade chybového stavu aplikácie. Ak by došlo k chybe pri spracovávaní údajov a služba by sa reštartovala, údaje sa nestratia, pretože sprostredkovateľ ich ukladá a odošle ich až po vyžiadaní službou. Služby v tejto architektúre sú úplne nezávislé a majú v porovnaní s topológiou mediátora menšiu viazanosť (5). Vďaka tomu sa môžu vyvíjať, meniť, alebo sa paralelne vylepšovať s ostatnými službami. Okrem toho sa jednotlivé služby dajú pomocou svojej nezávislosti bez potreby spustenia ďalších služieb ľahko a dôkladne testovať (4).

Sprostredkovateľ udalostí umožňuje dve základné formy komunikácie:

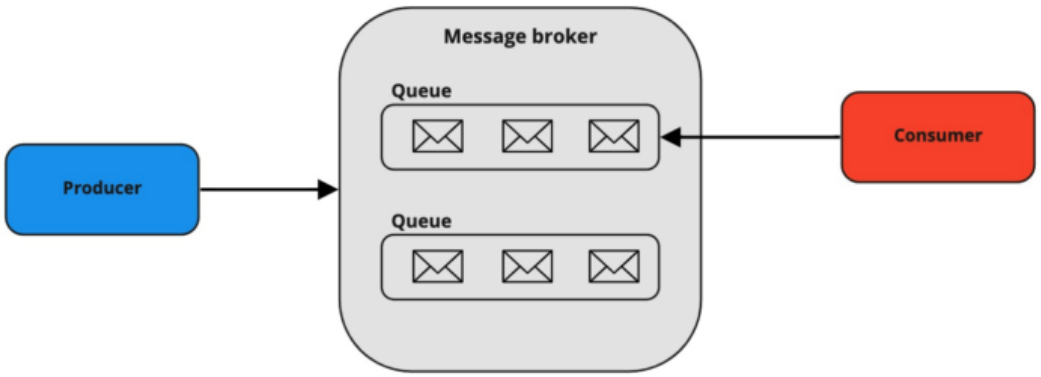
- Point-to-point - využíva rady (queues)
- Publish/Subscribe - využíva témy (topics)

1.2.1 Point-to-point

Komunikácia point-to-point, tiež nazývaná push-pull, funguje na forme jedného odosielateľa a jedného čitateľa (Obr. 1.3). Tento štýl správ zvyčajne používa frontu na ukladanie udalostí odosielaných producentom, až kým ich nedostane konzument. Producent odošle udalosť sprostredkovateľovi, ktorý ju uloží do fronty. Konzument si udalosť od sprostredkovateľa vyžiada a jej prijatie potvrdí sprostredkovateľovi. Následne je správa z fronty zmazaná.

Táto forma komunikácie vyžaduje, aby každá udalosť bola odoslaná a prijatá práve raz. Ako príklad môžeme uviesť spracovanie výplat alebo finančných transakcií.

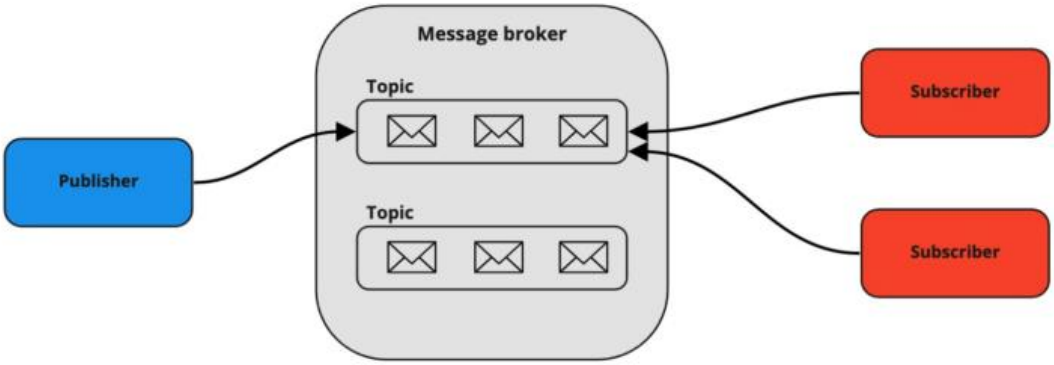
Výhodou point-to-point komunikácie je, že implicitne podporuje vyrovnanie záťaže systému (load balancing). Tiež umožňuje, aby sa systém, v prípade výpadku zotavil.



Obr. 1.3 Typ komunikácie point-to-point (6)

1.2.2 Publish/Subscribe

V komunikačnom systéme typu publish/subscribe odosielateľ publikuje udalosti na tému. Jeden alebo viac odosielateľov môže publikovať na tú istú tému a mnoho príjemcov môže prijať udalosť od jedného alebo od viacerých odosielateľov (Obr. 1.4). Príjemcovia sa prihlásia na odber tém a každému z nich budú doručené všetky udalosti uverejnené na danej téme. V prípade, že príjemca nie je v danom momente dostupný, publikované správy nikdy nedostane. Tento model poskytuje jednoduché dodanie udalostí na základe záujmu, podľa toho, ktoré témy si príjemca vyberie.



Obr. 1.4 Typ komunikácie Publish/Subscribe (6)

1.3 Udalosť

Ako už z názvu vyplýva, základnou jednotkou údajov v udalosťami riadenej architektúre je udalosť. Slovo udalosť sa taktiež používa na označenie objektu v programovacom jazyku, ktorým vyjadrujeme takýto výskyt (7). Udalosť, ako jednota údajov, je v niektorých prípadoch nahrádzaná pojmami správa (message), alebo záznam (record).

Udalosť môže napríklad reprezentovať uskutočnenie bezhotovostnej platby v nákupnom stredisku. Pri platbe kartou sa automaticky vytvorí udalosť, ktorá obsahuje atribúty, akými sú napríklad čas uskutočnenia platby, suma na zaplatenie, číslo účtu zákazníka, typ udalosti a množstvo ďalších.

1.3.1 Logická štruktúra udalosti

Každý typ udalosti obsahuje špecifické informácie - atribúty udalostí. Tie pozostávajú z názvu atribútu a jeho dátového typu. Hodnoty týchto atribútov pomáhajú odpovedať na otázky: Čo sa stalo?, Kedy sa to stalo?, Kde sa to stalo? (7).

Štruktúra udalosti pozostáva z dvoch častí: hlavička a telo. Hlavička obsahuje tzv. metaúdaje - dôležité atribúty, napríklad časovú pečiatku, vyjadrujúcu časový okamih, v ktorom bola táto udalosť vytvorená, alebo údajový typ tela udalosti, ktorý je unikátny pre množinu udalostí s rovnakou štruktúrou a rovnakým sémantickým významom (7). Údaje v hlavičke udalosti sú používané aj sprostredkovateľom, resp. mediátorom. Telo udalosti obsahuje údaje podstatné pre koncového príjemcu. Štruktúra udalosti môže byť vyjadrená pomocou ľubovoľného značkovacieho jazyka. Medzi najznámejšie patrí JavaScript Object Notation (JSON) a Extensible Markup Language (XML).

2. Apache Kafka

Apache Kafka, uvádzaná aj iba ako Kafka, je open-source softvér vytvorený spoločnosťou Linkedin. Okrem Linkedin ju používajú veľké spoločnosti, ako napríklad Twitter, Netflix, Strava, Cisco, a mnoho ďalších. Kafka umožňuje distribuované publikovanie a konzumovanie veľkého objemu dát medzi rôznymi systémami v reálnom čase.

O Kafke môžeme hovoriť ako o ekosystéme technológií, pomocou ktorého môžeme vytvoriť celý udalosťami riadený systém. Kafka poskytuje perzistenciu udalostí, transformáciu a sémantiku spracovania (8). Je napísaná v programovacom jazyku Java, takže je možné ju jednoducho nasadiť, či už na čisté železo, v cloude, alebo Kubernetes klastri. Navyše, Kafka má knižnice napísane takmer pre každý programovací jazyk, čo znamená, že prakticky každý vývojár môže pomocou Kafky začať využívať streamovanie udalostí.

Ako bolo spomenuté v časti *1.3 Udalosť,* na pomenovanie jednotky údajov v udalosťami riadenej architektúre, sa používajú viaceré názvy v závislosti od kontextu. V kontexte Kafky sa používa slovo záznam (record). Toto pomenovanie je asi najvýstižnejšie, pretože sa v skutočnosti neposielajú udalosti, ale iba záznamy o nich (9).

Kafka nám ponúka riešenie na viaceré spomenuté výzvy, ktoré prináša udalosťami riadená architektúra. Na rozdiel od mnohých sprostredkovateľov udalostí, Kafka podporuje oba komunikačné modely, point-to-point a publish/subscribe. Záznamy sa ukladajú do neustále rastúceho sekvenčného radu - logu, v ktorom má každý záznam unikátne poradové číslo - ofset. Jej komponenty je možné jednoducho a efektívne škálovať. Na rozdiel od klasického point-to-point modelu, prečítaním sa záznam nezmaže. Rozdiel je aj oproti modelu publish-subscribe, pretože príjemca môže záznamy prečítať kedykoľvek neskôr, teda nemusí ho spracovať okamžite, ako bol odoslaný. Okrem toho Kafka umožňuje čítať záznamy od ktoréhokoľvek ofsetu (napr. od nejakého časového bodu, ...), takže novo pridaný konzument môže spracovať aj historické záznamy.

Kafka je distribuovaný systém pozostávajúci zo 4 základných komponentov, ktoré v tomto systéme tvoria uzly (ndoes):

- Sprostredkovateľ (broker node)
- ZooKeeper
- Producent
- Konzument

Okrem uzlov v Kafke vystupujú ďalšie komponenty, dôležité pre komunikáciu a uchovávanie záznamov - téma (topic) a partícia (partition).

2.1 Téma a partícia

Téma je adresovateľná abstrakcia, ktorá umožňuje prístup k dátovému toku podľa jej jedinečného názvu. Každá téma môže obsahovať ľubovoľné množstvo záznamov, ktoré sú usporiadané do jednotlivých partícií. V rámci témy sa nastavuje napríklad počet partícií, počet replík a ich maximálna veľkosť. Pre nás podstatným nastavením je *min.in.sync.replicas*. Toto nastavenie hovorí o počte požadovaných replikácií záznamu na rôznych sprostredkovateľoch, aby mohol líder partície prijímať ďalšie záznamy.

Partícia je úplne zoradená množina záznamov. Publikované záznamy sú priradené na jej koniec. Spôsob, akým sa na ne záznamy ukladajú je popísaný v kapitolách *2.5 Producent* a *2.6 Konzument*.

2.2 Záznam (Record)

Záznamy sa posielajú na tému a rovnako sa z nej čítajú. Po odoslaní je záznam uložený na partíciu, na ktorej má každý záznam jedinečný identifikátor, tzv. ofset.

V kapitole *1.3.1 Logická štruktúra udalosti* sme opísali štruktúru udalosti. Záznam v Kafke ju rozširuje o ďalšie špecifické atribúty: (8 s. 24)

- kľúč (key) – kľúč v zázname funguje ako klasifikátor, ktorý spája súvisiace záznamy. Podľa neho sa môže vyberať, na ktorú partíciu sa záznam uloží. Kľúč môže byť akákoľvek hodnota, ale nemusí byť definovaný vôbec.
- hlavičky (headers) – páry kľúč-hodnota, ktoré pridávajú dodatočné metaúdaje do záznamu, ako napr. údajový typ tela záznamu.
- číslo partície (partition number) – určuje partíciu, na ktorej je záznam uložený. Nemusí byť špecifikované pri odosielaní záznamu.
- Ofset (offset) – 64-bitové celé číslo, ktorým sa identifikuje záznam v rámci partície

2.3 Sprostredkovateľ (Broker)

Sprostredkovateľ je zodpovedný za prijímanie záznamov od producentov ako aj za ich perzistentné uchovanie na partíciách a následné odoslanie konzumentom. Anglické slovo broker označuje *osobu alebo firmu, ktorá zabezpečuje transakcie medzi kupujúcim a*

predávajúcim...[1]. Ak kupujúceho nahradíme konzumentom a predávajúceho producentom, tak táto definícia vystihuje úlohu sprostredkovateľa v Kafke. Sprostredkovateľ teda zabezpečuje interakciu medzi dvomi komunikujúcimi stranami, čo rieši problém s väzbou medzi nimi. V udalosťami riadenej architektúre tieto strany o sebe nevedia, a tiež nemusia byť spoločne aktívne v rovnakom okamihu. To znamená, že sprostredkovateľ musí byť stavový (stateful). Inými slovami, sprostredkovateľ musí uchovať správy vyslané producentom, aby ich následne mohol odoslať konzumentovi, keď to bude možné. Prijaté správy musia byť trvalé a odolné voči strate - v angličtine sa táto vlastnosť nazýva *durability*.

Ako už bolo spomenuté, sprostredkovateľ uchováva správy v partíciách. Každá partícia je spravovaná práve jedným sprostredkovateľom - lídrom partície. Partície môžu byť replikované žiadnym, alebo viacerými sprostredkovateľmi - nasledovateľmi. Líder a nasledovatelia sa spolu nazývajú replikami. Každý sprostredkovateľ môže byť súčasne lídrom a nasledovateľom. Tieto role sa môžu meniť, napríklad keď dôjde k odpojeniu jedného zo sprostredkovateľov. Repliky zabezpečujú trvalosť a odolnosť záznamov voči strate. Čím viac replík v klastri, tým menšia pravdepodobnosť, že sa záznamy stratia kvôli osamotenej replike.

Kafka sprostredkovateľ je v praxi Java proces, ktorý je súčasťou klastra. Tento proces sa dá jednoducho škálovať. Zvýšením ich počtu spolu s partíciami môžeme dosiahnuť zlepšenú priepustnosť, dostupnosť a *durability*. Keďže všetci sprostredkovatelia v klastri musia navzájom spolupracovať, aby mohli fungovať ako lídri a repliky, je potrebné ich v rámci klastra spravovať. Jeden sprostredkovateľ je zvolený ako kontrolór, a ten je zodpovedný za správu stavu všetkých replík a partícií, aj za ich preraďovanie medzi sprostredkovateľov. Minimálny počet sprostredkovateľov v klastri je jeden.

2.3.1 Konfigurácia sprostredkovateľa

Sprostredkovateľ môže byť konfigurovaný staticky cez konfiguračný súbor, alebo dynamicky. Dynamická konfigurácia môže byť aplikovaná per-broker, teda iba pre daného sprostredkovateľa, alebo cluster-wide, čo znamená, že daná konfigurácia bude účinná pre všetkých sprostredkovateľov v klastri. Pri konfigurácii sprostredkovateľa platí, že dynamická per-broker konfigurácia má prednosť pred cluster-wide konfiguráciou, a tá má

1 https://en.wikipedia.org/wiki/Broker

prednosť pred statickou. Statická konfigurácia je iba na čítanie (read-only), takže pri zmene konfiguračného súboru je potrebné sprostredkovateľa reštartovať (8 s. 142-143).

V statickej konfigurácii nie sú povinné žiadne nastavenia, okrem *zookeeper.connect*. Toto nastavenie definuje zoznam adries ZooKeeperov v tvare *host1:port1,host2:port2....* Napriek tomu, že ostatné nastavenia sprostredkovateľa nie sú povinné, niektoré z nich je vhodné nataviť, aby nedošlo k chybám pri jeho spustení.

a) *listeners*

   Zoznam adries a ich názvy, na ktorých sprostredkovateľ počúva, a na ktoré sa pripíja producent a konzument. Ak má názov inú hodnotu ako bezpečnostný protokol, je potrebné ho pridať do *listener.security.protocol.map*. Predvolená hodnota je *PLAINTEXT://:9092*.

b) *log.dir*

   Adresár, do ktorého sa ukladajú všetky údaje, vrátane záznamov. Tento adresár musí byť pre každého sprostredkovateľa unikátny. Je možné špecifikovať aj zoznam adresárov v nastavení *log.dirs*. Predvolená hodnota je */tmp/kafka-logs*.

c) *broker.id*

   Identifikátor Kafka servera. Ak nie je nastavená žiadna hodnota, alebo je hodnota -1, nataví sa číslo, ktoré začína od *reserved.broker.max.id + 1*. Identifikátor je vhodné nastaviť, aby sme mali prehľad pri úprave konfigurácii sprostredkovateľa. Predvolená hodnota je -1.

2.4 ZooKeeper

ZooKeeper slúži ako centralizovaná služba pre riadenie Kafky. Jeho úlohou je zvolenie kontrolóra spomedzi sprostredkovateľov. Po spustení sa naňho pripoja všetci sprostredkovatelia v klastri. ZooKeeper zabezpečuje, aby vždy jeden sprostredkovateľ mal status kontrolóra. V prípade, že by kontrolór zlyhal, ZooKeeper okamžite zvolí nového kontrolóra. Okrem toho ZooKeeper funguje aj ako konzistentné úložisko, kde udržuje metaúdaje klastra, informácie o stavoch sprostredkovateľov, kvóty, a ďalšie potrebné informácie.

Zookeeper je rovnako dostupný a škálovateľný, ako sprostredkovateľ, takže v klastri môže existovať viacej ZooKeeperov. V klastri sa všetci ZooKeeperi dohodnú na jednom, ktorý bude mať funkciu lídra a iba on môže spracovávať požiadavky na zápis. Ostatné inštancie

ZooKeepera fungujú ako nasledovatelia a iba poskytujú operácie čítania pre prichádzajúce požiadavky. Aby bolo možné vytvoriť dohodu, v rámci klastra musí existovať nepárny počet ZooKeeperov (8 s. 49).

ZooKeeper nie je priamo súčasťou Kafky, ale samostatný open-source projekt. V najnovšej verzii Apache Kafka 3.4.0 je možné použiť KRaft namiesto ZooKeepera, ktorý je súčasťou Kafky (10). Avšak, použitá verzia Kafky v tejto práci, Apache Kafka 3.3.0, nedokáže pracovať bez ZooKeepera.

2.5 Producent

Producent je klientska aplikácia, ktorá produkuje údaje a posiela ich na zadefinovanú tému. Všeobecne platí, že iba producenti môžu vytvárať záznamy v Kafka klastri. Producent môže posielať záznamy na jednu alebo viaceré témy, pričom na jednu tému môžu posielať záznamy viacerí producenti. Producent komunikuje s klastrom pomocou sady TCP spojení, kde jednotlivé spojenia sú nadviazané s každým sprostredkovateľom v klastri. Producent má definovaný zoznam adries sprostredkovateľov. Tento zoznam nemusí obsahovať adresy všetkých sprostredkovateľov, pretože každý sprostredkovateľ pozná topológiu celého klastra a tú pošle producentovi v metaúdajoch. V prípade výpadku jedného zo sprostredkovateľov nie je potrebné riešiť zmenu sprostredkovateľa na strane producenta. Túto zmenu rieši samotný klaster na základe nasledovateľov (8 s. 20).

2.5.1 Odosielanie záznamov

Producent neodosiela záznamy po jednom, ale ich zabalí do tzv. dávky (batch), takže odosiela viaceré záznamy naraz. Tým sa zefektívni prenášanie záznamov po sieti. Producent odosiela záznamy zavolaním metódy *send()*. V závislosti od jeho nastavenia sa následne čaká určitú dobu, aby sa záznamy naakumulovali do dávky. Ďalej prebieha kontrola podmienok, za ktorých je možné dávku odoslať a v prípade, že všetky podmienky nie sú splnené, volanie metódy je uspaté. Po jeho zobudení sa dávka odošle sprostredkovateľovi. Producent následne čaká na odpoveď. V prípade, že potvrdenie neprišlo do stanoveného času, producent opätovne odošle dávku. To môže zopakovať definovaný počet krát. Ak sú všetky pokusy neúspešné, dávka sa zahodí.

2.5.2 Nastavenia producenta

To, ako sú záznamy odosielané a ukladané na partície je definované nastaveniami producenta. Kafka umožňuje upraviť fungovanie producenta pre konkrétny prípad použitia

jeho nastaveniami. Kľúčovými nastaveniami producenta v kontexte správneho ukladania záznamov na partíciách sú:

a) *acks*[2]
b) retries a *retry.backoff.ms*
c) enable.idempotence
d) max.in.flight.request.per.connection
e) partitioner.class
f) *batch.size* a *linger.ms*

2.5.3 Potvrdenie prijatia záznamu na partícii

Producent môže vyžadovať od sprostredkovateľa potvrdenie o prijatí záznamu. To je definované nastavením *acks*. Toto nastavenie hovorí o tom, koľko replík musí poslať potvrdenie producentovi o prijatí záznamu, aby mohol daný záznam považovať za úspešne odoslaný. Potvrdenia o prijatí záznamu poskytuje aj ďalšie informácie, ako je číslo partície, na ktorú bol záznam uložený, a jeho ofset. Potvrdenia teda zaručujú, že sprostredkovateľ záznam prijal. Vlastnosť *acks* môže mať nasledovné 3 hodnoty (8 s. 214-217) (9):

- 0
- 1
- -1 alebo *all*

V prípade nastavenia *acks*=0, producent nevyžaduje od sprostredkovateľa žiadne potvrdenie. Toto nastavenie nezaručuje, že budú záznamy spoľahlivo uložené na partícii. V prípade zlyhania producenta alebo sprostredkovateľa sa práve posielané záznamy stratia. Výhodou je, že sa zvýši priepustnosť a zníži sa latencia, pretože sa vynechajú čakania na potvrdenie záznamov. Táto možnosť sa môže použiť iba v prípade, že strata niektorých záznamov by mala zanedbateľný dopad na správne fungovanie celého systému. V praxi sa toto nastavenie môže použiť napríklad pri zberaní údajov v reálnom čase zo zariadení internetu vecí, alebo agregácii logov.

Keď je *acks*=1, producent vyžaduje potvrdenie len od sprostredkovateľa, ktorý je lídrom danej partície. Sprostredkovateľ asynchrónne prepošle záznamy všetkým nasledovateľom

[2] Skratka z angl. slova Acknowledgements, slov. potvrdenia

a odpovie producentovi. Táto odpoveď sa môže odoslať skôr, ako prijme potvrdenie od nasledovateľov. V ideálnom prípade tohto nastavenia sa v prípade výpadku sprostredkovateľa pošle záznam novému sprostredkovateľovi, ktorý bol zvolený za lídra partície. Sprostredkovateľ nezapisuje údaje na disk zavolaním metódy *flush()* po prijatí každého jedného záznamu. Preto sa môže stať, že sprostredkovateľ odošle potvrdenie o prijatí záznamu a zlyhá ešte predtým, ako by bol záznam uložený na disk, alebo zreplikovaný. Producent by pokračoval v jeho činnosti, pretože dostal potvrdenie o prijatí záznamu. Rozdielom oproti *acks*=0 je, že v prípade *acks*=1 nedôjde k strate záznamov, ak by zlyhal producent, ale len v prípade zlyhania sprostredkovateľa. Keďže je v praxi väčšinou sprostredkovateľ nasadený na kvalitnejšom hardvéri a s väčšou redundanciou ako producent, miera trvalosti a odolnosti záznamov voči strate je vyššia, no ani táto možnosť nezaručuje spoľahlivé uloženie záznamu na partícii. *Acks*=1 sa odporúča použiť v prípade, že môžeme stratiť niektoré záznamy, ale potrebujeme mať informáciu o ofsete záznamu a o partícii, na ktorej bol záznam uložený.

V prípade nastavenia *acks*=-1 (synonymom je *acks*=all), producent vyžaduje potvrdenia od všetkých tzv. in-sync replík (ISR - repliky, ktoré sú synchronizované s lídrom partície, teda majú uložené všetky záznamy, ktoré má aj líder). Toto nastavenie zaručuje že záznam bude zapísaný za predpokladu, že aspoň jedna zo zoznamu ISR zostane funkčná. Zoznam ISR sa môže líšiť od celkového počtu replík a dynamicky meniť. Keďže latencia publikovania záznamov závisí od najpomalšej ISR, zo zoznamu ISR sa dočasne odstavia tie repliky, v ktorých práve nastalo vyhladovanie. Ak chceme mať istotu, že všetky publikované záznamy budú uložené, je potrebné vybrať toto nastavenie. Hlavnou nevýhodou je, že nastavenie *acks*=-1 zvyšuje latenciu kvôli blokujúcim volaniam medzi producentom a lídrom partície, a medzi lídrom a nasledovateľmi.

2.5.4 Idempotencia producenta

Wikipédia definuje idempotenciu ako vlastnosť operácií v matematike alebo počítačových vedách, ktoré možno aplikovať viackrát bez toho, aby sa výsledok zmenil po počiatočnej operácii (11).

V prípade publikovania záznamov v Kafke to znamená, že záznam môžeme poslať koľkokoľvek krát, vždy sa uloží do partície práve raz. To je zabezpečené takým spôsobom, že Kafka pridá monotónne rastúce sekvenčné číslo každému odoslanému záznamu, ktoré v kombinácii s jedinečným identifikátorom producenta (PID), nám umožní ukladať záznamy

na partíciách bez duplikátov a v takom poradí, v akom boli odoslané. Sprostredkovateľ si ukladá do mapy posledné sekvenčné číslo pre dané PID na každej partícii. Ak producent pošle záznam, ktorého sekvenčné číslo je väčšie ako +1, sprostredkovateľ odpovie s chybou OUT_OF_ORDER_SEQUENCE_NUMBER, na čo producent odošle záznam opätovne. V prípade záznamu s menším alebo rovnakým sekvenčným číslom je tento záznam zahodený (12) (8 s. 163-165).

Ak má toto natavenie hodnotu *true* (hovoríme, že idempotencia je zapnutá), tak Kafka zaručuje:

- každý záznam bude uložený na príslušnej partícii práve raz
- záznamy sú na partíciách uložené v poradí, v ktorom boli odoslané
- skôr, ako sprostredkovateľ odošle producentovi potvrdenie, záznamy sú uložené na všetkých ISR.

Predvolená hodnota tohto nastavenia je *false*. Na to, aby bolo možné nastaviť *enable.idempotence* na *true*, musia byť splnené určité nastavenia:

- *acks* musí byť nastavené na -1 (all)
- *max.in.flight.retries.per.connection* musí byť menšie ako 5
- *retries* musí byť väčšie ako 0

2.5.5 Asynchrónne odosielanie záznamov

Producent môže posielať viacej požiadaviek naraz bez toho, aby na ne dostal potvrdenie o prijatí od sprostredkovateľov, aj keď je *ack=1* alebo *-1*. Ich počet sa dá definovať natavením *max.in.flight.requests.per.connection*. Toto nastavenie sa viaže na jedno spojenie (požiadavky odoslané jednému sprostredkovateľovi - lídrovi partície). Pri dosiahnutí tejto hranice je producent blokovaný a čaká na potvrdenie odoslaných požiadaviek. To umožňuje zvýšenie priepustnosti producenta, najmä pri vysokej odozve siete (8 s. 162). Toto nastavenie je veľmi dôležité v kontexte poradia záznamov. Ak by bola hodnota tohto nastavenia väčšia ako 1 a *enable.*idempotence by bolo nastavené na *false*, mohlo by dôjsť k zmene poradia uložených záznamov, v prípade, že by došlo k chybe pri niektorej požiadavke.

Zoberme si nasledujúcu situáciu, v ktorej predpokladáme, že nastavenie *enable.idempotence*=false. Odošleme 3 záznamy. Pre prvý príde potvrdenie, pri druhom

dôjde k dočasnej chybe a tretí je potvrdený, rovnako ako prvý. Keďže neprišlo potvrdenie pre druhý záznam, tak je poslaný znovu. V tejto situácii prídu záznamy v poradí 1,3,2. Keďže je predvolená hodnota tohto nastavenia 5, je potrebné brať ho do úvahy, najmä v prípade potreby správneho poradia záznamov.

Ak je ale *enable.idempotence* nastavené na *true*, poradie záznamov bude zabezpečené aj keď bude *max.in.flight.requests.per.connection>1*.

2.5.6 Opakované odoslanie záznamu

Producent podporuje nastavenie *retries* a *retry.backoff.ms*, ktoré špecifikujú počet pokusov na opakované odoslanie záznamu a dĺžku čakania medzi nimi v prípade, že odoslanie záznamov zlyhá. Tieto nastavenia umožňujú producentovi vykonávať automatické opätovné odosielanie záznamov v prípade výpadkov, alebo dočasných chýb. Predtým, ako producent opäť odošle záznam, zníži počet pokusov (*retries*) o jedna a čaká *retry.backoff.ms* milisekúnd. Predvolené nastavenie *retries* má hodnotu 2147483647 (Integer.MAX). Predvolená hodnota *retry.backoff.ms* je 100ms.

2.5.7 Priraďovanie partícií záznamom

Partícia, do ktorej sa odoslaný záznam zapíše sa určuje na strane producenta. V hlavičke záznamu je možné definovať partíciu pre daný záznam, alebo je možné použiť stratégiu nastavením *partitioner.class*. Toto nastavenia určuje triedu, v ktorej je stratégia priraďovania partícií implementovaná. Predvolene je hodnota tohto nastavenia *null*, a priraďovanie partícií funguje nasledovne :

1. Ak producent explicitne definuje partíciu v odoslanom zázname, použije sa tá partícia
2. Ak partícia nie je nastavená v odoslanom zázname, partícia sa vyberie podľa hash hodnoty kľúča
3. Ak nie je špecifikovaná ani partícia ani kľúč, priradí sa tzv. sticky partícia – partícia vybratá spôsobom Round-Robin. Tá sa zmení po tom, ako prijaté správy na partícii prekročia veľkosť nastavenú v *batch.size*.

Ak je nastavenia *partition.ignore.keys=false*, druhý krok sa vynechá. To ale neplatí, ak použijeme vlastnú stratégiu priraďovania partícií, ktorú môžeme vytvoriť implementovaním *org.apache.kafka.clients.producer.Partitioner*. Kafka poskytuje aj implementáciu *RoundRobinPartitioner*, ktorá priraďuje záznamom partície postupne dookola, takže sú záznamy rovnomerne rozložené medzi všetky partície. (9)

2.5.8 Odosielanie záznamov v dávkach

Ako už bolo spomenuté, záznamy sa neodosielajú po jednom, ale v dávkach, pre ich efektívnejší prenos po sieti. Celé nastavenie dávky spravuje producent. Nastavenie *batch.size* limituje veľkosť dávky, ktorú môže producent poslať. *Linger.ms* je čas v milisekundách, ktorý producent čaká po zavolaní metódy *send(),* kým sa záznamy naakumulujú do dávky. To má za cieľ posielať čo najväčšie množstvo údajov naraz. Predvolené nastavenie *batch.size* je 16384 bytov a *linger.ms* je 0 milisekúnd.

2.6 Konzument

Konzument je klientska aplikácia, ktorá funguje ako zberač dát. Konzument sa prihlási na odber na jednu alebo viac tém, z ktorých konzumuje záznamy. Každý konzument konzumuje záznamy vo vlastnom tempe, takže v prípade dlhšie trvajúceho spracovania, alebo dočasného výpadku, konzument môže spracovať všetky záznamy. Konzument nedostáva záznamy po jednom, ale, podobne ako producent, mu ich sprostredkovateľ posiela v dávkach. Ich veľkosť sa dá kontrolovať nastaveniami konzumenta. Konzument môže fungovať samostatne, alebo v skupine. Samostatný konzument sa neprihlasuje na odber záznamov z témy, ale manuálne si priradí určité partície z témy. Tento spôsob použitia konzumenta sa využíva na monitorovanie tém, alebo na implementáciu komunikačného vzoru *sync-over-async*. V tejto práci sa budeme zaoberať iba konzumentom, ktorý je súčasťou skupiny (8 s. 40-41).

2.6.1 Skupina konzumentov

Viacerí konzumenti pracujú spolu v skupine, v ktorej si rozdeľujú záťaž tak, že každá partícia je priradené práve jednému konzumentovi (jeden konzument môže konzumovať správy z viacerých partícií) (13) (8 s. 34). Partície sa prerozdeľujú medzi konzumentov podľa stratégií definovaných v nastaveniach konzumenta. Ak je konzumentov v skupine viac, ako je počet partícií, partície sa prerozdelia podľa stratégie a zvyšným konzumentom nebude priradená žiadna partícia, ale budú slúžiť ako záloha, v prípade výpadku niektorého z aktívnych konzumentov. Priraďovanie a prerozdeľovanie partícií má na starosti líder skupiny.

Skupina je definovaná nastavením *group.id* pre dynamickú skupinu, alebo *group.instance.id* pre statickú skupinu. Skupina je manažovaná koordinátorom skupiny – sprostredkovateľom, ktorý uchováva informácie o skupine. Koordinátor skupiny je implicitne vybraný spomedzi lídrov partícií na téme _consumer_offsets, podľa hash hodnoty kľúča skupiny. Každý

konzument v skupine posiela koordinátorovi skupiny tzv. heartbeat v pravidelnom intervale, čím informuje o tom, že je aktívny. Ak konzument nepošle heartbeat do určitého času, bude zo skupiny vyhodení a koordinátor skupiny prerozdelí partície aktívnym konzumentom.

2.6.2 Pripájanie konzumentov do skupiny

Pripájanie konzumentov do skupiny sa dá rozdeliť na dve fázy: vytvorenie členstva v skupine a synchronizácia stavu. Pri vytváraní členstva sa identifikujú všetci aktívny konzumenti v skupine a určí sa líder skupiny. Po pripojení sa do klastra dostane konzument topológiu sprostredkovateľov, s informáciou o koordinátorovi skupiny. Konzument odošle požiadavku JoinGroupRequest na pridanie sa do skupiny, koordinátor skupiny mu ju potvrdí odpoveďou JoinGroupResponse s identifikátorom, ktorý bude konzument používať až do jeho odpojenia zo skupiny.

Lídrom skupiny sa stane konzument, ktorý sa ako prvý pripojí na koordinátora skupiny. Ten dostane odpoveď ako posledný, pričom v odpovedi dostane aj zoznam všetkých konzumentov v skupine. Po ukončení tejto fázy sa koordinátor skupiny dostane do stavu AwaitSync a čaká na priradenie partícií konzumentom. V synchronizačnej fáze líder skupiny priraďuje konzumentom partície. Tie sa priradia podľa implementovanej stratégie, ktorá je definovaná v nastavení *partition.assignment.strategy*. Konzumenti po priradení partícií pošlú koordinátorovi SyncGroupRequest s priradenými partíciami. Koordinátor odošle odpoveď SyncGroupResponse a prejde do stavu Stable. (8 s. 235-241)

2.6.3 Riadenie zmien v skupine

Koordinátor skupiny udržuje ID generácie - monotónne zvyšujúci sa kladné celé číslo – ktoré korešponduje so stavom skupiny v aktuálnom čase. Keď konzument pošle JoinGroupRequest pri pripájaní sa do skupiny, alebo LeaveGroupRequest pri odchádzaní z nej, koordinátor sa dostane do stavu Joining a čaká na všetkých členov, aby sa opätovne pripojili. Keď je koordinátor v stave Joining, odpovie na heartbeat požiadavky chybovou hláškou REBALANCE_IN_PROGRESS, čo informuje konzumentov o tom, že majú poslať JoinGroupRequest. Keď sú všetci konzumenti v skupine pripojený, koordinátor inkrementuje ID generácie a prejde do stavu AwaitSync. Nastane synchronizačná fáza, v ktorej sa prerozdelia partície konzumentom v skupine. Po nej prejde koordinátor skupiny opäť do stavu Stable. Konzumenti posielajú ID generácie v heartbeat požiadavkách. Môže sa stať, že konzumentovi sa nedostane informácia o Joining stave koordinátora, napríklad

kvôli vysokému heartbeat intervalu. Konzument sa stále správa ako keby bol členom skupiny, ale koordinátor ho už z nej vymazal. Hovoríme o tzv. zombie konzumentovi. Keď zombie konzument pošle heartbeat, koordinátor naň odpovie chybovou hláškou ILLEGAL_GENERATION, čo informuje konzumenta o tom, aby sa znovu pripojil do skupiny. (8 s. 235-241)

2.6.4 Priraďovanie partícií konzumentom v skupine

Ako už bolo spomenuté, priraďovanie partícií v rámci skupiny prebieha na konzumentovi, ktorý je jej lídrom. Hlavnou výhodou toho je, že vývojári môžu jednoducho zmeniť stratégiu priraďovania partícií bez toho, aby museli upravovať nastavenia každého sprostredkovateľa (8 s. 267-268). Stratégia priraďovania partícií je implementovaná v triede, ktorá implementuje org.apache.kafka.clients.consumer.ConsumerPartition. To, ktorá implementácia sa použije je definované ich zoznamom v nastavení *partition.assignment.strategy*. Všetci konzumenti v skupine musia mať aspoň jednu hodnotu v tomto zozname rovnakú, inak líder skupiny odpovie chybovou hláškou INCONSISTENT_GROUP_ERROR. Je možné použiť vlastnú stratégiu, alebo použiť jednu zo štyroch implementácií, ktoré poskytuje Kafka (8 s. 261-268):

- RangeAssignor - táto stratégia funguje tak, že pre jednu tému sa rozložia partície v číselnom poradí a konzumenti v lexikografickom poradí. Počet partícií sa vydelí počtom konzumentov, čím sa určí rozsah, koľko partícií by mal mať každý konzument priradených. Keďže počet partícií nemusí byť deliteľný počtom konzumentov, prvých niekoľko konzumentov môže mať priradenú o jednu partíciu viac. Toto priraďovanie sa vykoná pre každú tému, na ktorú je skupina konzumentov prihlásená. V prípade že je vo viacerých témach počet partícií menší, ako počet konzumentov, partície budú rozdelené nerovnomerne, priradené iba prvým konzumentom.
- RoundRobinAssignor - priraďuje partície tzv. Round-Robin spôsobom, ktorým rozdelí partície rovnomerne medzi všetkých konzumentov v skupine. Funguje to tak, že partície zo všetkých odoberaných tém sa dajú do jedného zoznamu. Z neho sa postupne priraďuje po jednej partícii každému konzumentovi, tak, že prvá partícia sa priradí prvému konzumentovi, druhá druhému, až kým sa všetky partície zo zoznamu neminú. Keď sa prejdú všetci konzumenti, algoritmus sa vráti opäť k prvému.

- StickyAssignor - udržuje partície rovnomerne rozložené medzi všetkými konzumentmi v skupine. Táto stratégia sa snaží zachovávať rovnaké pridelenie partícií konzumentom aj po zmenách v skupine. To sa dosiahne tak, že pri zmene v klastri sa budú zohľadňovať predošlé pridelenia partícií konzumentom v skupine. Tým sa minimalizuje rozdiel po sebe nasledujúcich priradení.
- CooperativeStickyAssignor - variácia predchádzajúcej stratégie. Využíva kooperatívny protokol, aby sa prerozdelenie partícií vykonalo efektívnejšie.

Keďže všetci konzumenti v skupine musia mať definovanú aspoň jednu spoločnú stratégiu priraďovania partícií, je potrebné vykonávať jej zmenu v dvoch krokoch. Najprv treba pridať všetkým konzumentom stratégiu, ktorú chceme použiť, do zoznamu aktuálne používaných stratégií. Následne môžeme bezpečne predošlú stratégiu zo zoznamu vymazať. Pri zmene predvolenej stratégie, tá musí byť tiež explicitne v zozname. Predvolená stratégia je RangeAssignor spolu s CooperativeStickyAssignor (9).

2.6.1 Statická skupina

V statickej skupine sú partície priradené konzumentom stabilne na dlhšiu dobu. Konzumenti v tejto skupine z nej môžu odísť a znovu sa pripojiť bez toho, aby sa partície prerozdelili, za predpokladu, že konzument nie je odpojený dlhšie ako stanovený čas. To znižuje dopad prerozdeľovania partícií a prerušení pri tzv. *nepružných* konzumentoch. Nepružný konzument je taký, pri ktorom predpokladáme, že sa odpojí iba na krátku dobu, kvôli plánovanému nasadzovaniu, alebo dočasnému výpadku (8 s. 242-247).

Pripájanie konzumenta funguje podobne, ako pri dynamickom konzumentovi, ktoré bolo spomenuté vyššie. Koordinátor skupiny pošle konzumentovi identifikátor v odpovedi na JoingGroupRequest. Avšak, pri odpojení sa zo skupiny, konzument neposiela LeaveGroupRequest, ale koordinátor skupiny čaká, kým vyprší čas relácie (\textit{session.timeout.ms}). Koordinátor bude stále odpovedať na heartbeat od konzumenta a ak ho konzument nepošle do vypršania času relácie, koordinátor zo skupiny konzumenta vyhodí a nastane prerozdelenie partícií.

Konzumenti v statickej skupine majú veľké využitie najmä v spojení s externým nástrojom, ktorý kontroluje zdravie aplikácie, ako je napríklad *Kubernetes*. Kubernetes spúšťa aplikácie v kontajneroch, ktoré kontrolujú jej zdravie. V prípade chyby sa kontajner reštartuje a aplikácia sa nanovo spustí. Keďže v statickej skupine sa partície prerozdeľujú až

po vypršaní *session.timeout.ms*, je potrebné nastaviť tento čas dlhší ako trvá zotavenie Kubernetes kontajnera (8 s. 177-178).

2.6.2 Komitovanie ofsetu

V Kafke konzumovanie záznamu neodstráni záznam z partície. Preto je potrebné udržiavať stav jednotlivých skupín a ofsetov na partíciách, z ktorých záznamy už v danej skupine boli prečítané. Tento stav udržiava koordinátor skupiny, ale tiež je uložený na téme _consumer_offsets, pre prípad, že by koordinátor skupiny zlyhal, a stal sa ním iný sprostredkovateľ.

V rámci skupiny z jeden partície konzumuje záznamy v danom čase maximálne jeden konzument. Po pripojení do skupiny konzumenti získajú partície a im prislúchajúce ofsety od lídra skupiny. Každý konzument v skupine vykonáva tzv. komit ofsetu – odošle ofset každému lídrovi partícií, ktoré mu boli priradené. Komit ofsetu sa vykonáva v pravidelnom intervale, ktorý je definovaný nastavením *auto.commit.interval.ms* (predvolená hodnota tohto nastavenia je 5 sekúnd). Konzumentovi je možné nastaviť *auto.offset.reset*, čo určuje, či, v prípade že ešte žiadny ofset nebol komitnutý, bude po pripojení do skupiny bude čítať všetky záznamy od začiatku partície, alebo bude čítať iba nové záznamy od bodu, kedy sa pripojil (8 s. 37-38).

3. Kubernetes

Kubernetes, tiež sa uvádza aj ako K8s, je open-source platforma pre orchestráciu kontajnerov, ktorá umožňuje spravovať, škálovať a nasadzovať aplikácie v kontajnerovom prostredí. Vytvorený bol spoločnosťou Google, ktorá ho v roku 2014 sprístupnila ako open-source platformu. Kubernetes odstraňuje mnoho ručných procesov, ktoré sa týkajú nasadenia a manažovania kontajnerových aplikácií. Poskytuje ideálne prostredie pre nasadenie a správu Kafky, pretože kombinácia týchto technológií prináša výhody ako škálovateľnosť, odolnosť voči chybám a efektívne využívanie zdrojov (14). Každý komponent Kafky je možné zabaliť do kontajnera, ktorý môže byť nasadený do tzv. podov v Kubernetes klastri. Kontajner je softvérový balík, ktorý obsahuje všetko potrebné na spustenie aplikácie: kód samotnej aplikácie, systémové knižnice a nastavenia.

Orchestrácia v systéme Kubernetes označuje riadenie kontajnerov. Ide o pokročilú automatizáciu, vďaka ktorej je možné v prípade pádu kontajnera kontajner reštartovať, v prípade pádu fyzického servera spustiť ďalšie kontajnery na inom serveri a v prípade veľkého zaťaženia siete presunúť komunikáciu na menej vyťaženú infraštruktúru. Vďaka orchestrácii kontajnerov teda dokážeme automatizovať riešenia pre najčastejšie kritické scenáre.

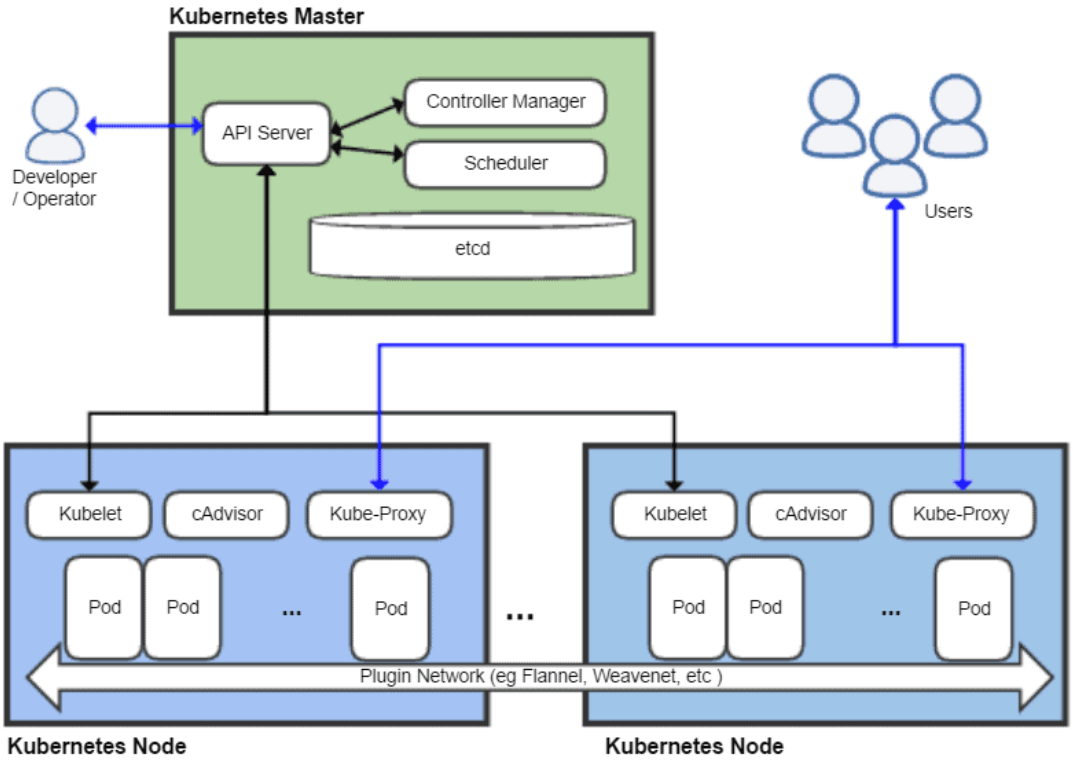
3.1 Architektúra Kubernetes

Architektúra Kubernetes je založená na distribuovanom modeli, ktorý umožňuje rozsiahle škálovanie a odolnosť voči chybám. Kubernetes sa skladá z dvoch hlavných častí, ako je zobrazené na Obr. 3.1: Správcu (Master node) a uzly (Worker nodes) (15).

Správca je centrálny prvok v Kubernetes klastri, ktorý riadi a spravuje všetky operácie. Správca obsahuje API server (kube-apiserver), pomocou ktorého komunikuje s uzlami v klastri. Tiež zahŕňa nástroj na manažovanie (kube-controller-manager), nástroj na plánovanie (kube-scheduler) a databázu (etcd). Pri komunikácii s uzlami im správca dáva vedieť, aké objekty majú vytvoriť, zmeniť alebo odstrániť. Najjednoduchší a zároveň najdôležitejší objekt je tzv. *pod*, pretože obsahuje kontajnery, ktoré má Kubernetes orchestrovať. Všetky ostatné objekty iba slúžia podu. Príkladom objektov sú volume, service, deployment, ingress, secret, alebo configMap.

Uzly sú pracovníci (workers), ktorí vykonajú všetko presne tak, ako im zadá správca (master). Na uzloch teda beží kontajnerizovaná aplikácia nasadená v podoch podľa pokynov

od správcu. Každý uzol musí mať nainštalovaný Kubernetes runtime a komponenty, vrátane kubelet. Kubelet je nástroj, ktorý riadi životný cyklus kontajnerov na danom uzle. Priamo komunikuje s API serverom, prijíma príkazy na spustenie, zastavenie alebo odstránenie kontajnerov a monitoruje ich stav. Kontajnery sú spustené v spomínaných podoch. Pod kontajnery zoskupuje a poskytuje im zdieľané zdroje, ako sú sieťové pripojenie a úložný priestor. Kontajnery v rámci jedného podu môžu medzi sebou komunikovať pomocou lokálneho rozhrania (15).



Obr. 3.1 Architektúra systému Kubernetes (15)

3.2 Kubernetes objekty

Objekty Kubernetes sú konštantné entity v systéme Kubernetes. Používajú sa na reprezentáciu stavu klastra. Konkrétne popisujú:

- Ktoré aplikácie v kontajneroch sú spustené a na ktorých uzloch?
- Dostupné zdroje pre aplikácie.
- Politika, podľa ktorej sa aplikácie správajú - politik reštartu, odolnosť voči chybám a upgrade.

Kubernetes objekty majú definovaný zámer, ktorý chceme dosiahnuť. Po vytvorení objektu systém Kubernetes zabezpečí, že objekt bude existovať v požadovanom stave. Vytvorením

objektu v skutočnosti hovoríme systému Kubernetes, že toto je spôsob, akým chceme, aby klaster vyzeral, teda je to požadovaný stav klastra. Na prácu s objektmi Kubernetes, či už na ich vytváranie, úpravu alebo odstraňovanie, používame Kubernetes API. Môžeme na to použiť kubect1 CLI, ktorá vytvára potrebné API volania. Tiež je možné API volania vytvoriť aj priamo v programoch pomocou jednej z klientskych knižníc.

Všetky objekty Kubernetes obsahujú 2 vnorené prvky, ktoré riadia konfiguráciu objektu. Tieto prvky sú *status* (stav objektu) a *spec* (špecifikácia objektu). Špecifikáciu objektu pre jeho vytvorenie musíme poskytnúť. Popisuje požiadavky stavu, do ktorého sa má objekt po spustení dostať. Stav na druhej strane popisuje skutočný stav objektu, ktorý poskytuje a aktualizuje systém Kubernetes (16).

Základným objektom v Kubernetes je *pod*. Je najmenším a najjednoduchším objektom, ktorý môžeme vytvoriť a nasadiť. Pody predstavujú bežiace procesy v klastri. Každý pod obsahuje jeden, alebo viacero kontajnerov s aplikáciou, jedinečnú IP adresu, zdroje, a spôsob, ktorým sa pod bude v danej situácii správať. Najčastejšie sa používa pod s jedeným kontajnerom. Kontajnerizovaná aplikácia v pode je izolovaná. Ak by sme potrebovali, aby jednotlivé aplikácie vzájomne zdieľali úložisko, museli by sme ich nasadiť do jedného podu (17).

OPIS RIEŠENIA

4. Opis riešeného problému

Táto práca sa zaoberá komunikáciou medzi mikroslužbami, ktoré sú nasadené v kontajneroch a umiestnené do jednotlivých podov v Kubernetes. Ako scenár komunikácie bola vybraná asynchrónna komunikácia v prostredí udalosťami riadenej architektúry.

Práca konkrétne rieši aktuálny problém dodržania poradia spracovávania správ. V analýze sme popísali sprostredkovateľa udalostí Kafku. Architektúra umožňuje existenciu viacerých partícií v rámci jednej témy a v takomto prípade nemáme zaručené dodržanie postupnosti spracovávania prijatých záznamov.

4.1 Špecifikácia požiadaviek

- Navrhnúť scenár komunikácie medzi mikroslužbami prostredníctvom udalosťami riadenej architektúry
- Navrhnúť konfiguráciu udalosťami riadeného systému tak, aby záznamy boli konzumentom prijímané v tom istom poradí, v akom boli odoslané producentom

4.2 Funkcionálne požiadavky

- Vytvoriť producenta a dvoch konzumentov udalostí
- Simulovať odosielanie a prijímanie záznamov a analyzovať výsledky
- Prijímané záznamy budú konzumenty zapisovať do logov.

4.3 Nefunkcionálne požiadavky

- Spoľahlivosť fungovania systému - dodržanie poradia spracovania záznamov
- Nastavenie systému udalosťami riadenej architektúry

5. Návrh riešenia

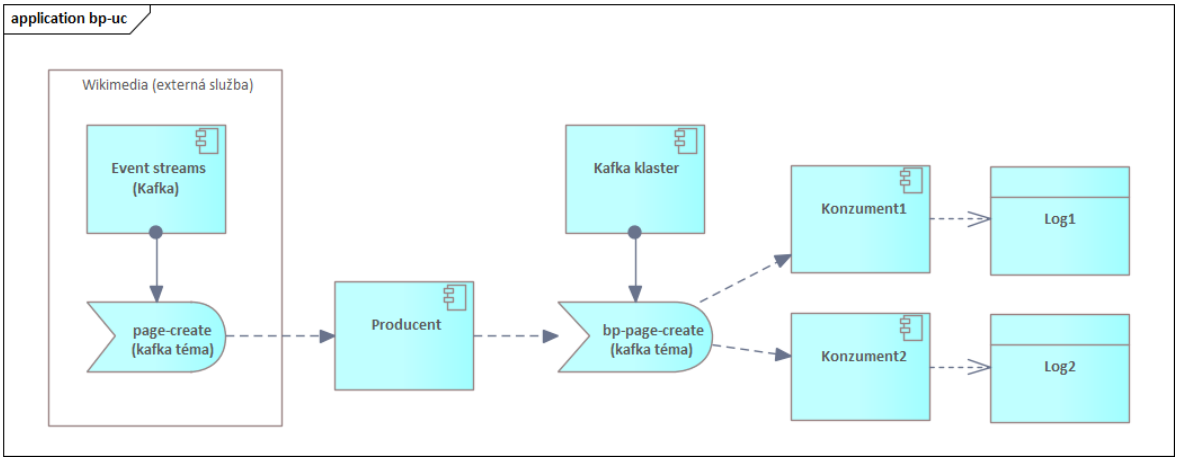
Konzumovanie záznamov v poradí, v akom boli odoslané producentom má viacero prípadov použitia. Typickým príkladom sú finančné transakcie. Zachovanie poradia je však dôležité pri akejkoľvek aktualizácii nejakého stavu. V niektorých prípadoch môže byť na jednu tému pripojených viac typov konzumentov. Tiež často býva veľmi dôležitá, okrem zachovania poradia, aj spoľahlivosť doručenia všetkých záznamov. Na tomto type prípadu použitia budeme demonštrovať dôležitosť zachovania rovnakého poradia záznamov u producenta a konzumentov. Obsahom ďalších častí tejto kapitoly bakalárskej práce je návrh riešenia zadaného problému. Pri návrhu vychádzame zo špecifikácie požiadaviek a analytickej časti tejto práce.

5.1 Popis navrhovaného riešenia

V tejto časti navrhneme scenár komunikácie medzi mikroslužbami s použitím udalosťami riadenej architektúry, pričom využijeme Kafku. V scenári bude vystupovať jeden producent a dvaja konzumenti. Systém bude poskytovať udalosti získavané z verejnej streamovacej služby. Základnou funkcionalitou systému bude čítanie streamovaných údajov z externej služby a ich publikovanie na tému pre našich konzumentov.

Na Obr. 5.1 je zobrazená aplikačná architektúra nášho systému, ktorý pozostáva zo štyroch hlavných častí:

1. Externá služba
2. Producent
3. Kafka
4. Konzumenti



Obr. 5.1 Diagram aplikačnej architektúry

Služba Wikimedia Event streams je externou súčasťou nášho systéme. Poskytuje verejne dostupné toky záznamov, na ktoré je možné sa prihlásiť a prijímať z nich údaje. V našom systéme ju budeme používať ako generátor záznamov, s ktorými budeme pracovať.

V našom návrhu producent prijíma záznamy z toku s názvom *page-create*[3] od externej služby. Tieto záznamy modifikuje, a posiela na tému *bp-page-create*. Tá sa nachádza už v nami vytvorenom systéme, ktorý obsahuje Kafka klaster. Prípad použitia tiež obsahuje dvoch konzumentov, ktorí konzumujú záznamy z témy *bp-page-create* a ukladajú ich na disk.

5.2 Návrh systému

Navrhovaný systém musí spĺňať požiadavky definované v kapitole *4 Opis riešeného problému*. Systém sa skladá z troch častí:

1. Producent
2. Konzument
3. Kafka klaster

Jednotlivé časti navrhneme a urobíme k nim konfiguráciu tak, aby spĺňali definované požiadavky.

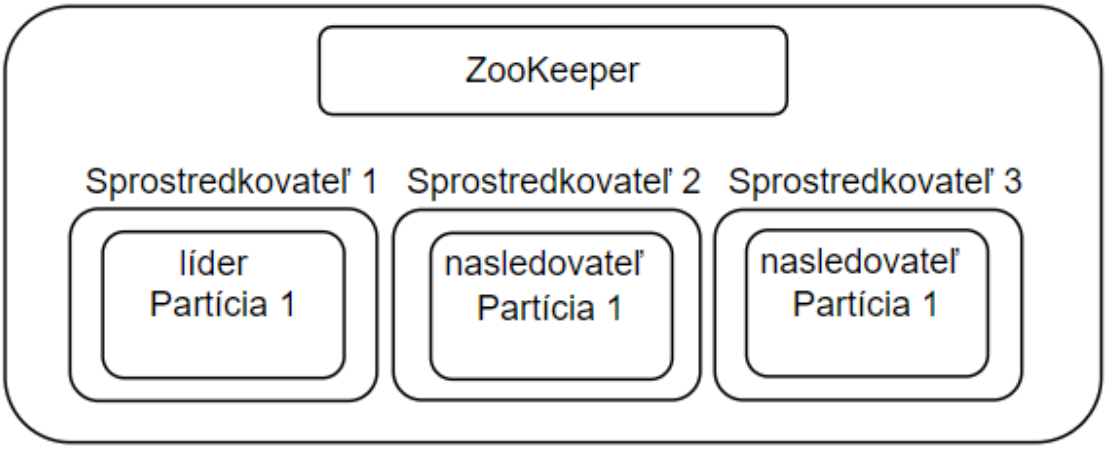
Podstatným bodom medzi požiadavkami je navrhnutie systému tak, aby boli záznamy prijímané konzumentom v tom istom poradí, v akom boli odoslané producentom. Kafka

[3] https://stream.wikimedia.org/v2/stream/page-create

sprostredkovateľ použitý v návrhu, umožňuje použiť viacej partícií pre zvýšenie rýchlosti posielania záznamov. Avšak, zachovanie poradia záznamov je zaručené iba pri použití jednej partície. Odôvodníme si to zvlášť pri návrhu konfigurácie producenta a konzumenta. Najskôr si však navrhneme klaster, na ktorý sa obaja klienti pripájajú.

5.2.1 Návrh klastra

Kafka klaster je v zásade skupina sprostredkovateľov a ZooKeeper, ktorý ich manažuje. Aby bolo posielanie záznamov spoľahlivé, odporúča sa mať aspoň troch sprostredkovateľov v jednom klastri, ktorí si vzájomne replikujú partície. Každá partícia má jedného lídra – sprostredkovateľa, ktorý do nej zapisuje a číta znej, ostatní sprostredkovatelia – nasledovatelia, robia jej zálohu. Navrhovaný Kafka klaster v našom systéme je zobrazený na Obr. 5.2 Návrh Kafka klastra.



Obr. 5.2 Návrh Kafka klastra

Inštancia sprostredkovateľa v Kafke je nazvaná Kafka server. Pre jednoduchosť budeme inštanciu sprostredkovateľa nazývať iba server. Keďže sa naň klienti pripájajú, navrhneme jeho konfiguráciu ako prvú.

V analytickej časti *2.3.1 Konfigurácia sprostredkovateľa* sme popísali 3 spôsoby konfigurácie servera. Pri návrhu použijeme hlavne statickú konfiguráciu, pretože v našom scenári nám postačujú predvolené dynamické nastavenia.

Nastavenie *zookeeper.connect* je jediným nastavením, ktoré je povinné a musí obsahovať adresu ZooKeepera v klastri. Nastavenia *listeners*, *log.dir* a *broker.id* musia byť unikátne. Tieto nastavenia sú dôležité pre správne spustenie serverov a ich fungovanie. Pre

zabezpečenie spoľahlivosti a bezpečný návrh klientov je potrebné vykonať ešte ďalšie podstatné nastavenia:

a) *auto.create.topics.enable*

   Keď producent začne posielať záznamy na nejakú tému, alebo konzument sa pripojí na čítanie záznamov a téma neexistuje, automaticky sa vytvorí, ak je toto nastavenie *true* (automatické vytváranie tém je zapnuté. Ak má hodnotu *false*, server odpovie s chybovou hláškou *UNKNOWN_TOPIC_OR_PARTITION*) a klient po ich spustení budú vypisovať upozornenie.

   Odporúča sa vypnúť automatické vytváranie tém, pre vyvarovanie sa chybám pri konfigurácii klientov.

b) *default.replication.factor*

   Predvolený počet replík, ktoré sa vytvoria, ak tento počet nie je špecifikovaný pri vytváraní témy.

c) *min.insync.replicas*

   Minimálny počet replík, ktoré musia poslať potvrdenia producentovi, ak má producent nastavené *acks=all* (alebo *-1*). Ak neexistuje dostatočný počet replík, producent dostane odpoveď s chybovou hláškou *NOT_ENOUGH_REPLICAS*.

Pre zabezpečenie spoľahlivosti je typickým scenárom nastavenie *replícation.factor* na 3, *min.in.sync.replicas* na 2 a *acks* natavené pre producenta na *all*. To zabezpečí, že ak väčšina replík nie je funkčná, producent dostane chybovú hlášku a záznamy môže poslať opätovne.

Statická konfigurácia serverov je zobrazená v tabuľke Tabuľka 5.1.

Kľúč	Hodnota		
	Server 1	Server 2	Server 3
zookeeper.connect	Adresa ZooKeepera		
listeners	PLAINTEXT://:9092	PLAINTEXT://:9093	PLAINTEXT://:9094
log.dir	/logs/kafka-broker-1	/logs/kafka-broker-2	/logs/kafka-broker-3
broker.id	1	2	3
auto.create.topics.enable	false		
default.replication.factor	3		
min.insync.replicas	2		

Tabuľka 5.1 Nastavenie sprostredkovateľov

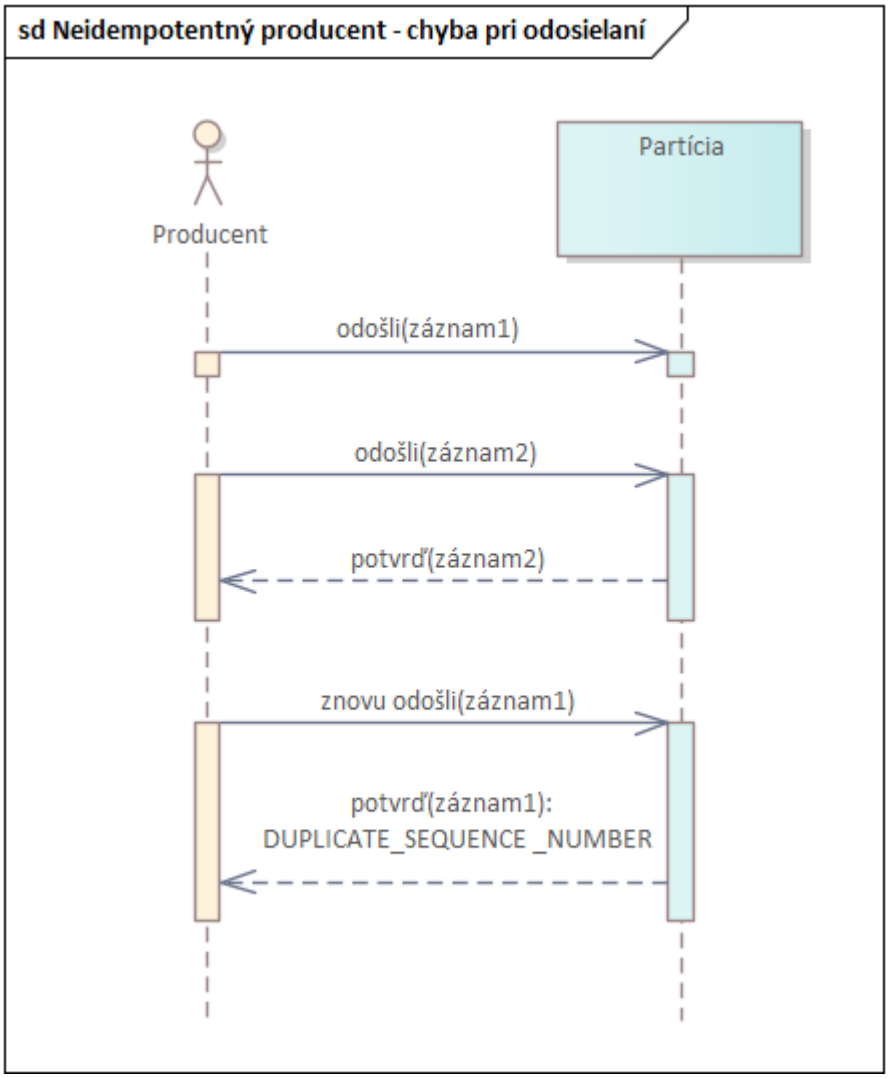
Pri spúšťaní témy sa nastavuje adresa servera, názov témy a počet partícií. Počet partícií je vhodnejšie nastavovať pre každú tému zvlášť, podľa daného prípadu použitia. V našom prípade nám záleží na zachovaní poradia odoslaných a prijatých záznamov, preto budeme používať jednu partíciu. Ďalšie nastavenia témy, ako replikačný faktor a minimálny počet synchronizovaných replík, sú predvolene definované nastaveniami servera.

5.2.2 Návrh producenta

Producent je jediným komponentom, ktorý publikuje údaje v našom systéme. Na Obr. 5.1 je zobrazené, že producent získava údaje z externej služby. To však nie je podstatné pri jeho návrhu, ale budeme sa tomu venovať v implementačnej časti. Aby sme splnili druhú zo špecifický požiadaviek, podstatné je, aby sa záznamy odoslané producentom spoľahlivo uložili na určitú partíciu.

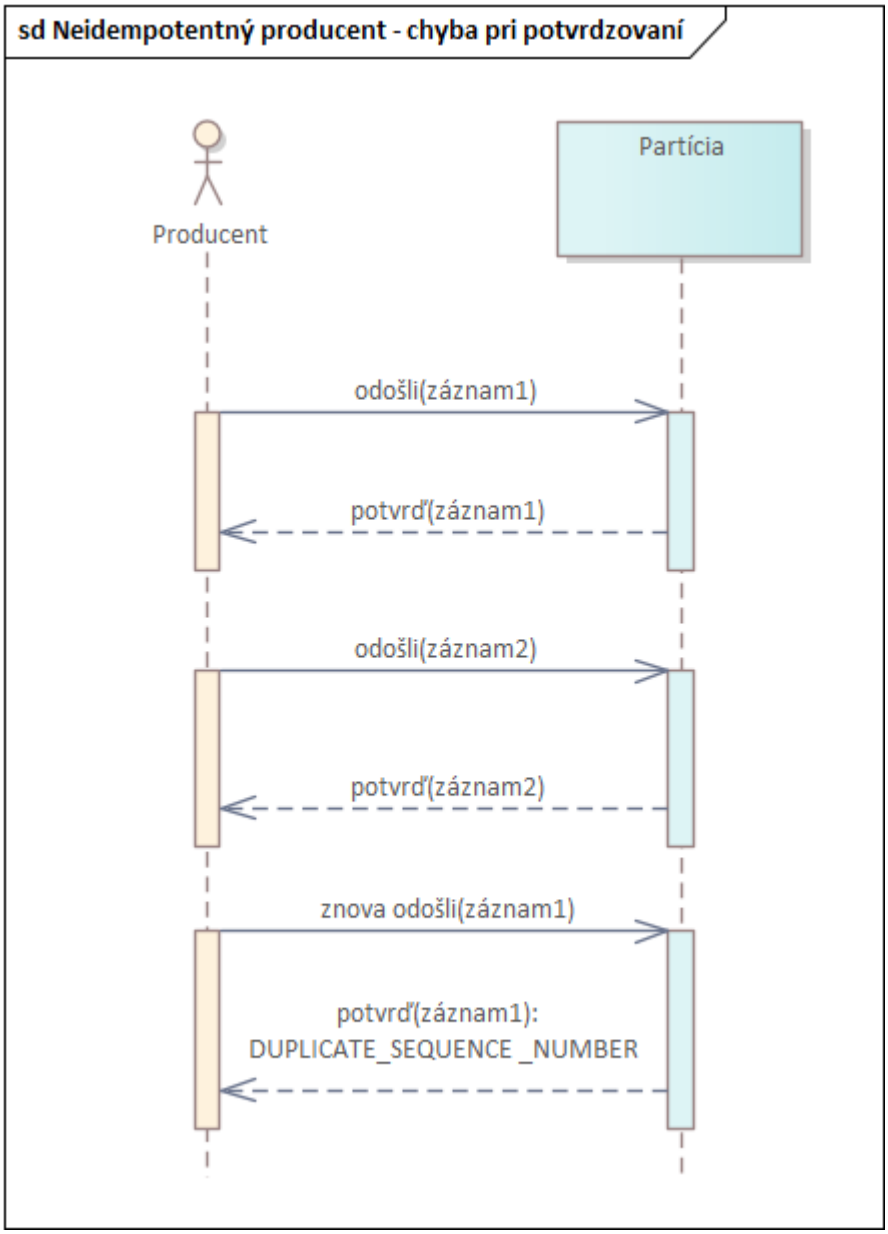
Pre spoľahlivé, neduplicitné uloženie záznamov odoslaných producentom a zachovanie ich poradia na partícii, musí byť producent idempotentný. To pri Kafka producentovi dosiahneme nastavením *enable.idempotence* na hodnotu *true*. Ak by producent idempotentný nebol, mohlo by dôjsť k duplicitám, tým pádom aj k zmene poradia záznamov uložených na partícii, ako je zobrazené v scenároch na Obr. 5.3 a Obr. 5.4Obr. 5.4. V prvom scenári dôjde k chybe po odoslaní prvého záznamu, ešte pred jeho zapísaním na partíciu. Producent odošle druhý záznam, ktorého prijatie je potvrdené. Keďže potvrdenie pre prvý záznam nedostal, po vypršaní *request.timeout* ho odoslal opäť. V druhom scenári chyba nastane pri potvrdzovaní prvého záznamu, až po jeho zapísaní na partíciu. Ten je neskôr opäť poslaný, takže sa na partíciu zapíše záznam1 duplicitne. Tento problém tiež rieši zapnutie idempotencie na producentovi.

Nastavením *enable.idempotence* na *true* tieto problémy vyriešime. Zapnutím idempotencie producent pridá do záznamu sekvenčné číslo, ktoré inkrementálne rastie, a server si udržuje mapu posledného sekvenčného čísla na partícii. V prvom scenári na Obr. 5.3 server prijme záznam so sekvenčným číslom, ktoré nie je od predošlého väčšie o 1, ale o viac, takže po prijatí záznamu2 odpovie chybovou hláškou OUT_OF_ORDER_SEQUENCE_NUMBER. Tým upozorní producenta, aby odoslal záznam2 znova.



Obr. 5.3 Neidempotentný producent – chyba pri odosielané

V druhom scenári, ktorý je na Obr. 5.4 server prijme záznam1 druhý krát, čo znamená, že prijme menšie sekvenčné číslo, ako aktuálne. Server záznam zahodí a pošle potvrdenie s hláškou DUPLICATE_SEQUENCE _NUMBER, čo značí, že záznam bol zahodený.



Obr. 5.4 Neidempotentný producent – chyba pri potvrdzovaní

V analytickej časti sme spomenuli, že sekvenčné číslo si server ukladá do mapy pre každého producenta a každú partíciu zvlášť. V tomto prípade sú zasielané záznamy na viacero partícií. Ich poradie síce bude zachované čiastočne na každej partícii, ale nebude zachované na všetkých partíciách dokopy.

Ako bolo spomenuté v analytickej časti, nastavenie *enable.idempotence* na *true* vyžaduje, aby mali ďalšie 3 nastavenia určitú hodnotu:

- *acks=all* (alebo *-1*)
- *max.in.flight.requests.per.connection* <= 5
- *retries > 0*

V prípade prvého nastavenia, *acks*, musíme použiť hodnotu *all*. Druhé nastavenie sa odporúča nastaviť aspoň na hodnotu 2, pretože pri nastavení na túto hodnotu je priepustnosť záznamov a latencia medzi producentom a Kafka serverom podstatne lepšia, ako pri použití hodnoty 1. Pri vyšších hodnotách (v prípade zapnutej idempotencie môžeme použiť ešte hodnoty 3, 4 a 5) zlepšenie priepustnosti a latencie nie je veľmi výrazné. Tretie nastavenie, *retires*, sa odporúča nastaviť na maximálnu hodnotu celého čísla, MAX_INT, ktorá je rovná 2147483647 (18). Je to najväčšia možná hodnota, ktorú toto nastavenie môže mať. Tým zaručíme doručenie záznamov na partíciu aj v prípade dlhšieho výpadku sprostredkovateľa.

Na akú partíciu sa záznam uloží je určené stratégiou priraďovania partícií. Podľa implicitnej stratégie producenta v Kafke sa záznamy uložia na tú partíciu, ktorá je špecifikovaná v hlavičke. Pre rôzne prípady použitia môžeme implementovať vlastnú stratégiu priraďovania partícií. My však používame iba jednu partíciu, takže nám stačí použiť predvolenú stratégiu, ktorú producent použije implicitne.

Celkové nastavenia producenta bude nasledovné:

Kľúč	Hodnota
bootstrap.servers	localhost:9092,localhost:9093,localhost:9094
value.serializer	org.apache.kafka.common.serialization.Json
acks	all
max.in.flight.requests.per.connection	2
retries	MAX_INT

Tabuľka 5.2 Nastavenia producenta

5.2.3 Návrh konzumentov

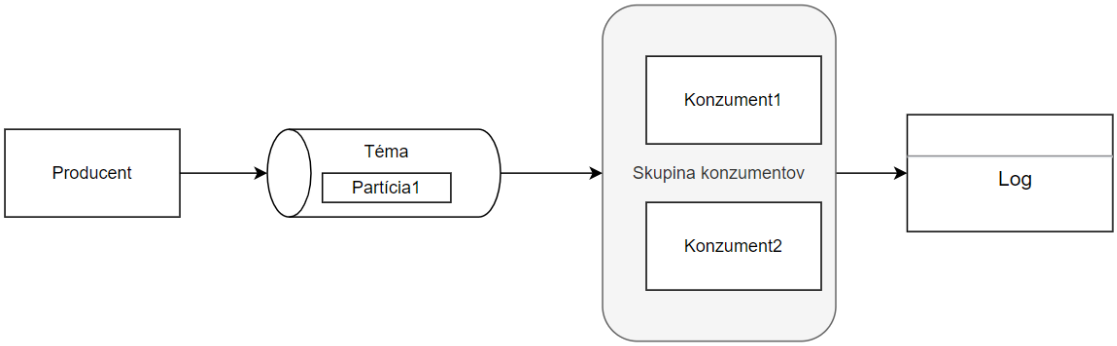
V systéme máme dvoch konzumentov. Kafka umožňuje rozložiť záťaž medzi konzumentov, ktorý sú v rovnakej skupine. Avšak, jedna partícia môže byť priradená iba jednému konzumentovi v skupine. Keďže v našom prípade používame iba jednu partíciu, dvaja konzumenti sa dajú použiť dvomi rôznymi spôsobmi:

1. V rovnakej skupine
2. V rôznych skupinách

Prvý prípad je zobrazený na Obr. 5.5. Skupina je spravovaná jej koordinátorom, ktorým je jeden zo sprostredkovateľov. Keď sa do skupiny pripojí konzument, informuje o tom koordinátora, ktorý vyberie lídra skupiny a informuje ho o všetkých konzumentoch v nej. Následne líder skupiny prerozdelí partície.

Keďže sa obaja konzumenti nachádzajú v jednej skupine, a čítajú údaje z témy, ktorá má jednu partíciu, aktívny je iba jeden z nich. Neaktívny konzument nečinne čaká, kým sa partícia uvoľní. To znamená, že všetky záznamy budú zapísané do jedného log súbor, ako je na obrázku znázornené. Uvoľnenie partície môže nastať v dvoch prípadoch. Uvažujme situáciu, že Konzument1 číta záznamy z témy a Konzument2 nečinne čaká. Keď sa Konzument1 zo skupiny odpojí, pošle LeaveGroupRequest koordinátorovi, čím vykoná tzv. graciózne odpojenie. Ten informuje všetkých konzumentov, aby sa znovu pripojili, pritom vyberie nového lídra. Teraz už máme v skupine iba Konzuemnta2, ktorému bola partícia priradená.

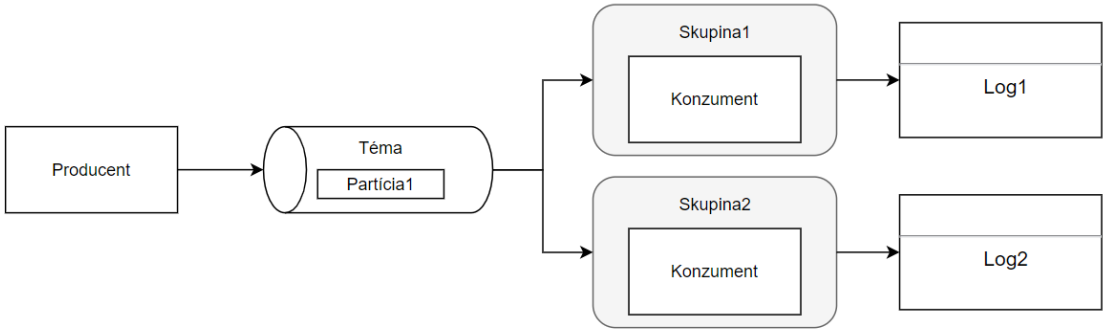
Druhý prípad môže nastať, keď sa aktívny konzument neodpojí graciózne, ale pri spracovaní údajov došlo k chybe, alebo zaseknutiu. Koordinátor skupiny zahájí prerozdelenie po vypršaní heartbeat intervalu aktívneho konzumenta a vyhodí ho zo skupiny. Následne o tejto zmene informuje ostatných konzumentov a proces pokračuje, ako v predošlej situácii.



Obr. 5.5 Konzumenti v jednej skupine

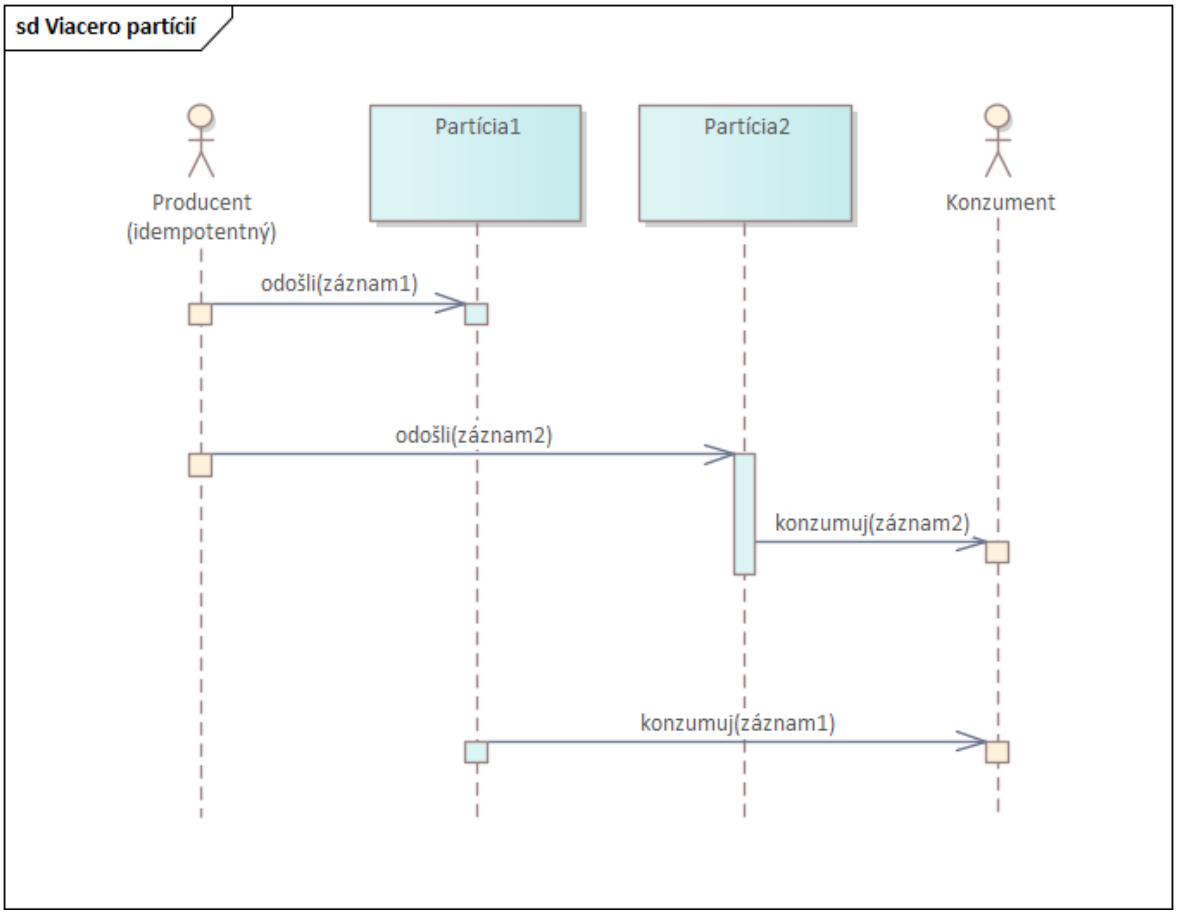
Na obrázku Obr. 5.6 je znázornený druhý prípad. Keďže je každý konzument v samostatnej skupine, obaja môžu čítať záznamy z tej istej partície. Tento prípade je použitý aj v nami

navrhovanom riešení, pretože, na rozdiel od prvého prípadu, v ktorom konzumenti v jednej skupine ukladali záznamy do jedného log súboru, každý z konzumentov ukladá prijaté záznamy do samostatného log súboru.



Obr. 5.6 Dve skupiny konzumentov

Pri producentovi sme na jednej partícii zabezpečili úplné zachovanie poradia záznamov, a čiastočné s použitím viacerých partícií, pomocou sekvenčného čísla. Na strane konzumenta to však nie je možné. Konzumenti si v rámci skupiny delia prácu tak, že viacerí z nich môžu prijímať záznamy z viacerých partícií asynchrónne v jednom čase. To by znamenalo, že by medzi nimi musela existovať určitá synchronizácia. Navyše, ak by boli záznamy produkované na viaceré partície, ich poradie by bolo iba čiastočné. Na Obr. 5.7 je zobrazené, že v prípade použitia viacerých partícií, poradie prijatia záznamov nie je zhodné s poradím, v ktorom boli odoslané, napriek tomu, že na partície boli odoslané v správnom poradí.



Obr. 5.7 Viacero partícií

Na základe uvedeného by konfigurácia konzumenta vyzerala nasledovne:

Kľúč	Hodnota	
	Konzument1	Konzument2
bootstrap.servers	localhost:9092,localhost:9093,localhost:9094	
value.serializer	org.apache.kafka.common.serialization.Json	
group_id	bp_group1	bp_group2

Tabuľka 5.3 Nastavenie konzumentov

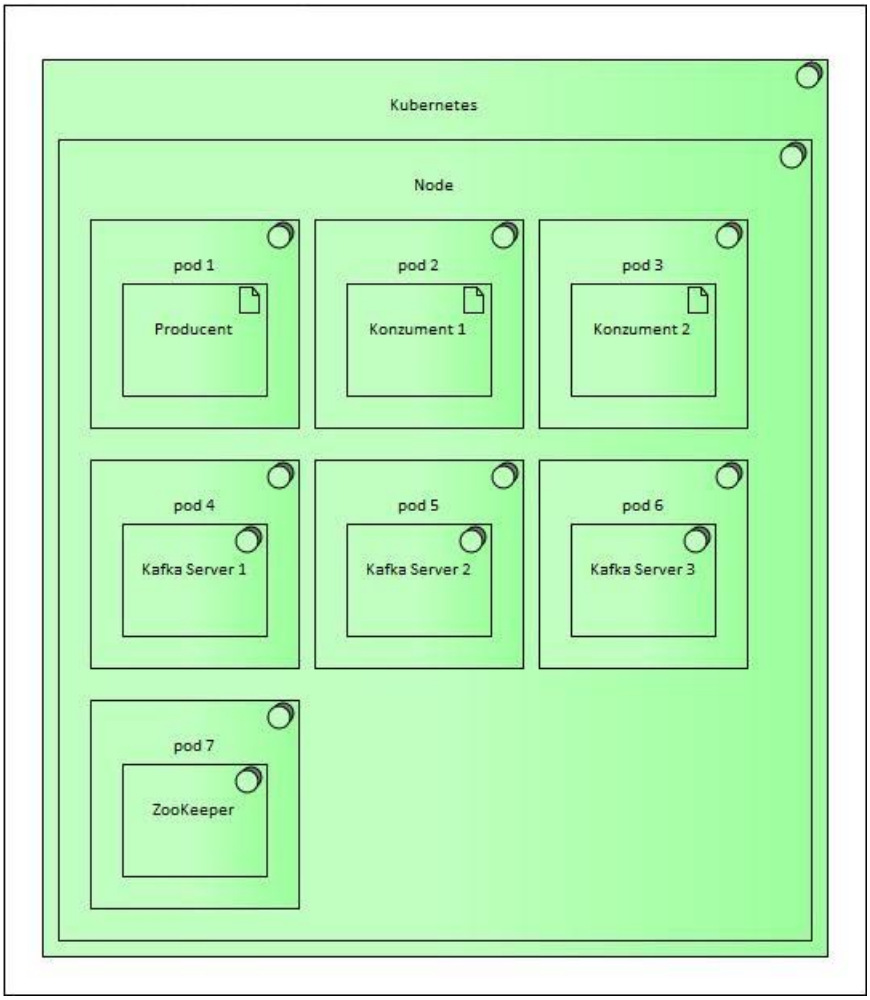
5.3 Kubernetes

Náš systém bude nasadený v prostredí Kubernetes. Kubernetes je opísaný v kapitole *3 Kubernetes*, kde sme spomenuli základný typ objektu – pod. Komponenty systému bude uložené v kontajneroch a nasadené v podoch. V jednom pode môžeme nasadiť jeden, alebo

viac kontajnerov. Pri použití troch Kafka serverov, nasadiť ich do jedného podu by sa mohlo zdať vhodné. Avšak, veľkou nevýhodou by bola chýbajúca izolácia. Ak by jeden zo serverov prestal fungovať, pod by bol nasadený znova a všetky servery by boli dočasne nedostupnú. Preto každý komponent nášho systému bude nasadená v samostatnom pode. Vytvoríme teda 7 podov, ako je zobrazené na Obr. 5.8:

- Aplikácia producent
- 2x Aplikácia konzumenta
- 3x Kafka server
- ZooKeeper

Keďže aplikácie v jednotlivých podoch sú od seba izolované, musíme zabezpečiť, aby bolo možné medzi nimi komunikovať. Musíme sprístupniť porty, na ktorých budú aplikácie komunikovať. Porty vieme sprístupniť v špecifikácii objektu. Pre ZooKeepera a servery sprístupníme porty, ktoré sme špecifikovali v *5.2.1 Návrh klastra*.



Obr. 5.8 Návrh Kubernetes klastra

6. Implementácia riešenia

Táto kapitola obsahuje popis implementácie mikroslužieb, ako aj výber programovacieho jazyka, frameworku a knižníc, potrebnej na overenie nášho riešenia.

6.1 Výber programovacieho jazyka a frameworku

Pre Kafka klientov existujú implementované knižnice v mnohých jazykoch a frameworkoch. Najpopulárnejšími sú Pyton, GO, .NET, alebo Node.js (19). Keďže je naše riešenie postavené na mikroslužbách, nie je závislé od použitého programovacieho jazyka. Kafka je napísaná v programovacích jazykoch Scala a Java, pre ktorých natívne podporuje aj API klientov. Pre Kafka klientov existuje knižnica vo frameworku Spring Boot (20). Riešenie je implementované v programovacom jazyku Java s použitím frameworku Spring Boot.

Pri implementácii mikroslužieb bol použitý build-tool *maven*, Modely projektov pre jednotlivých klientov dedia dependencie od artefaktu *spring-boot-starter-parent*. Jeho použitie zabezpečí verzie pre ostatné dependencie, ktoré Spring framework podporuje.

Hlavnou knižnicou je *spring-kafka*. Tá poskytuje API pre všetkých Kafka klientov – producenta, konzumenta a administrátora. Ďalšou dôležitou knižnicou je *jackson-databind*, ktorú používame na serializáciu a deserializáciu objektov. Údaje objektov sa serializujú a ako byty sa posielajú po téme na partíciu. Po prijatí na strane konzumenta sa deserializujú na objekt, ktorý môžeme jednoducho použiť v kóde. Pre prístupové metódy k jednotlivým hodnotám objektov použijeme knižnicu *lombok*, s ktorou vieme jednoducho vytvoriť getre, setre a konštruktory. V návrhu producent získava údaje z externej služby Wikimedia streams. Na to, aby bolo možné tieto údaje získať, použijeme knižnicu *okhttp-eventsource* od *com.launchdarkly*.

6.2 Implementácia klientov

S použitím frameworku Spring Boot a uvedených knižníc je implementácia veľmi jednoduchá. Na strane producenta je potrebné pripojiť sa na stream udalostí poskytovaný od Wikimeda streams. Wikimedia síce používa Kafku, ale jej témy nie sú verejne dostupné. Záznamy sú zverejnené iba pomocou streamov. Na ich čítanie použijeme triedu EventSource z knižnice *okhttp-eventsource*, ktorá sa pripojí na daný stream a vráti iterátor, do ktorého sa

pridávajú prijaté správy. Tie namapujeme na triedu vytvorenú podľa definovanej schémy[4]. Následne vytvoríme objekt, ktorý posielame na nami definovanú tému. Jeho hodnoty sú skopírované z prijatého objektu. Knižnica Kafka poskytuje generickú triedu KafkaTemplate. Pri jej vytváraní špecifikujeme typy kľúča a hodnoty, aké sa posielajú. Po sieti sa však objekty posielať nedajú, ale iba bity. Preto je potrebné objekt serializovať. Na serializáciu sa používa trieda *JsonSerializer*. V Spring Boot producent predvolene posiela v hlavičke triedu objektu, z ktorého bol serializovaný. Ak by sme mali v module konzumenta triedu s rovnakým názvom a premennými, nachádzala by sa v inom balíku, a teda trieda nie je rovnaká. Po prijatí záznamu by aplikácia vyhodila výnimku *SerializationException* s hláškou *Class not found* (Trieda nebola nájdená). Riešením môže byť buď vytvorenie modulu, ktorý bude obsahovať triedu modelu, ktorý budeme posielať. Tento modul pridáme obom klientom ako dependenciu.

Ako bolo spomenuté, framework Spring Boot poskytuje API aj pre Kafka administrátora, pomocou ktorého môžeme vytvoriť tému. V návrhu sme v konfigurácii Kafka servera zakázali automatické vytváranie témy. Buď by sme tému museli vytvoriť cez príkazový riadok, alebo môžeme použiť API administrátora. Je vhodné vytvoriť tému na strane producenta, pretože ak by neexistovala, producent by po odoslaní záznamu dostal odpoveď s chybovou hláškou. Preto je vhodné vytvoriť tému, na ktorú bude producent posielať záznamy, cez API administrátora po spustení mikroslužby producenta.

V službe konzumenta je implementovaná iba metóda *consume()*, ktorá má anotáciu *@KafkaListener*. V nej je definovaný názov témy, z ktorej konzument záznamy konzumuje. Táto metóda je zavolaný vždy, keď príde nový záznam. V nej pracujeme už s deserializovaným objektom. Na deserializáciu používame triedu *JsonDeserializer*. Aby mohli byť údaje deserializované, musíme nastaviť dôveryhodné balíky, v ktorých sa musí nachádzať objekt, inak aplikácia vyhodí výnimku *SerializationException*. To môžeme urobiť nastavením *spring.json.trusted.packages=**, čím povolíme triedu z akéhokoľvek balíka.

[4]https://schema.wikimedia.org/repositories/primary/jsonschema/mediawiki/revision/create/1.1.0.yaml

6.3 Kubernetes

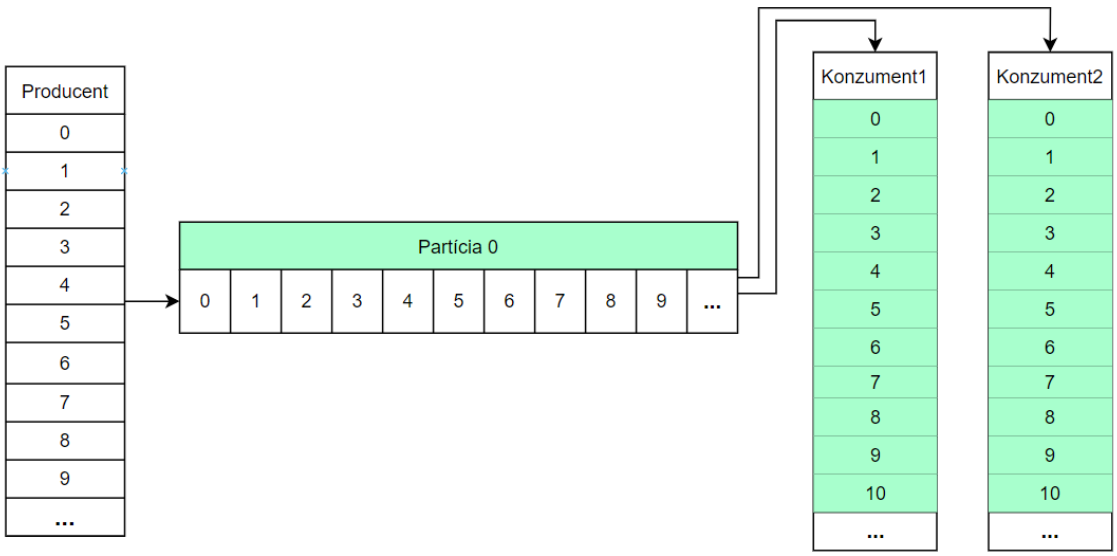
Tieto mikroslužby sú uložené v docker kontajneroch. Na ich orchestráciu je použitá platforma Kubernetes, ako sme popísali v návrhu. Pre každý pod je vytvorený .yaml súbor, v ktorom je uvedená špecifikácia daného podu. Ich popis je uvedený v prílohe A.

7. Overenie riešenia

V tejto časti sa budeme venovať overeniu implementovaného riešenia. Riešenie spĺňa požiadavky, ktoré sú spísané v kapitole *4 Opis riešeného problému.*

V rámci implementácie sme vytvorili jedného producenta, ktorý posiela záznamy na tému, a z nej ich prijímajú dve mikroslužby. Tieto záznamy by mali byť prijaté v takom poradí, v akom boli odoslané. Pre overenie tejto požiadavky bude producent pridávať do záznamu identifikačné číslo, ktoré inkrementálne rastie. Nastavením idempotencie producenta sme zabezpečili, že záznamy budú uložené na partíciách v takom poradí, v akom boli odoslané. Preto nám pri konzumovaní stačí kontrolovať, či záznamy prichádzajú s rastúcim sekvenčným číslom, teda v každom zázname o 1 väčším. Pre porovnanie si prijaté záznamy budeme ukladať do súboru.

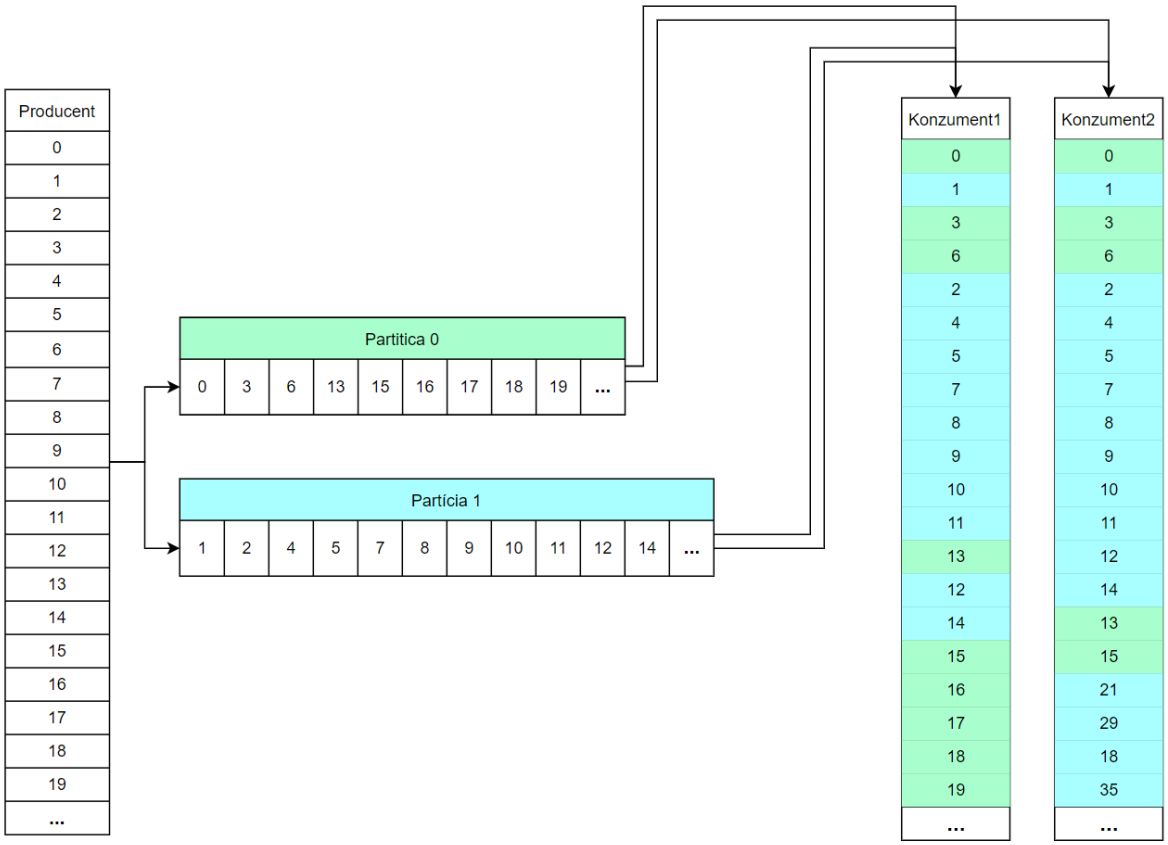
Riešenie sme overili dvomi scenármi – pozitívnym a negatívnym. V návrhu sme uviedli, že zachovanie poradia dokážeme zabezpečiť ukladaním záznamov na práve jednu partíciu. V pozitívnom scenári sme teda použili jednu partíciu na téme bp-page-create. Na ňu producent posielal záznamy rozšírené o sekvenčné číslo a konzumujúce mikroslužby ich z nej prijímali. Každá konzumujúca mikroslužba bola v samostatnej skupine. Na Obr. 7.1 sú zobrazené identifikačné čísla prvých desiatich záznamov v poradí, v akom boli produkované, zapísané na partícii a konzumované. Všetky záznamy sú v prílohe B. Na obrázku je vidieť, že záznamy boli prijaté konzumujúcimi mikroslužbami v takom poradí, v akom boli odoslané producentom.



Obr. 7.1 Graf sekvenčných čísel prijatých záznamov s použitím jednej partície

V negatívnom scenári sme demonštrovali, že pri použití viacej ako jednej partície nie je poradie záznamov zachované. V scenári sme použili dve partície. Stratégiu priraďovania produkovaných záznamov na partície sme nechali predvolenú. Záznamy, ktoré posielame nemajú veľa bytov a pretože sa pri predvolenej stratégii partícia mení po tom, ako sa na jednu zapíšu záznamy s minimálnou veľkosťou, ktorá je definovaná nastavením *batch.size*, nastavili sme ho na 10. To spôsobí, že záznamy budú rovnomernejšie rozdistribuované na obe partície.

Pri testovaní negatívnym scenárom sme zistili, že k narušeniu poradia konzumovaných záznamov dôjde iba vtedy, ak je na oboch partíciách záznam, ktorý ešte nebol skonzumovaný. To môže nastať v prípade, že sa konzumujúca mikroslužba prihlási na odber záznamov na danej téme neskôr, ako na ňu začne producent záznamy posielať, alebo ak dôjde k nejakému oneskoreniu, ktoré môže byť spôsobené dlhšie trvajúcim spracovávaním údajov, dočasnou chybou, alebo sieťovým výpadkom. Ak by konzument dokázal záznamy prijímať rovnakou alebo väčšou rýchlosťou, ako sú produkované, ich poradie by bolo zachované. Pre demonštrovanie a lepšie zobrazenie narušenia zachovania poradia prijatých záznamov v prípade použitia viacerých partícií sme pridali čakanie po prijatí záznamu. Prvej konzumujúcej mikroslužbe sme pridali čakanie 100ms a druhej mikroslužbe 300ms. Na Obr. 7.2 je zobrazené, na ktorú partíciu bol zapísaný záznam s daným identifikačným číslom. Z obrázka je vidieť, v akom poradí boli záznamy z partícií konzumované. Môžeme si tiež všimnúť, že pri pomalšom konzumovaní (Konzument2 s čakaním 300ms po skonzumovaní záznamu) sú medzi identifikačnými číslami konzumovaných záznamov väčšie rozdiely, pretože sa najprv konzumovali všetky záznamy z jednej partície, a následne z druhej. Keďže spracovanie záznamu Konzumentovi2 trvalo dlhšie, na partíciu prišlo viacej záznamov, ktoré musel najprv skonzumovať.



Obr. 7.2 Graf sekvenčných čísel prijatých záznamov s použitím dvoch partícií

ZHODNOTENIE

Táto bakalárska práca sa zameriava na dôležitý a aktuálny problém v oblasti softvérového inžinierstva a distribuovaných systémov. Práca úspešne splnila svoj cieľ preskúmať, navrhnúť a implementovať riešenie, ktoré zabezpečuje zachovanie správneho poradia záznamov udalostí v systéme udalosťami riadenej architektúry s využitím Apache Kafka.

Dôležitou súčasťou riešenia problematiky práce bolo pochopenie základných princípov a fungovania udalosťami riadenej architektúry ako takej, architektúru a činnosť smerovača udalostí Kafky a platformu na správu kontajnerových pracovných zaťažení a služieb Kubernetes. Z tohto dôvodu sme podrobne analyzovali základné topológie udalosťami riadenej architektúry a vybrali sme z nich vhodnú pre náš problém. Ďalej sme podrobne analyzovali princípy Kafky, jej komponenty a ich konfiguráciu. Popísali sme dôležité nastavenia konfigurácie pre náš problém. V závere analytickej časti sme popísali Kubernetes a výhody jeho použitia s Kafkou.

Podľa vypracovanej analýzy a zadaného problému sme spísali požiadavky na jeho riešenie. V návrhu riešenia sme najprv popísali spôsob, akým budeme riešenie realizovať. Navrhli sme systém, ktorý pozostával z Kafky, producenta a dvoch konzumentov. Návrh tiež obsahuje diagramy ako celého systému, tak aj jednotlivých komponentov. Pri každom komponente vystupujúcom v systéme je navrhnutá konfigurácia a sú zdôvodnené zvolené hodnoty v nastaveniach konfigurácie a ich dôležitosť pre riešenie problému. Nakoniec sme navrhli spôsob nasadenia komponentov systému na platformu Kubernetes. Navrhnuté riešenie sme implementovali v jazyku Java s použitím SpringBoot frameworku. Implementačná časť tiež obsahuje spôsob uloženia komponentov do podov v systéme Kubernetes.

Riešenie sme overili dvomi scenármi – pozitívnym a negatívnym. V pozitívnom scenári bolo poradie konzumovaných záznamov zhodné s poradím, v akom boli produkované. V negatívnom scenári sme popísali, za akých podmienok poradie záznamov nebude zachované . Výsledky potvrdili naše predpoklady a sú prehľadne zobrazené na priložených diagramoch.

Práca poskytuje dôležité poznatky a prínosy pre implementáciu riešenia, pri ktorom je nevyhnutné zachovanie poradia záznamov v udalostiach riadenej architektúry s využitím Apache Kafka. Navrhnuté riešenie môže prispieť k zvýšeniu dôveryhodnosti, konzistencie

a spoľahlivosti takýchto systémov. Je však dôležité zdôrazniť, že navrhované riešenie má určité obmedzenia. Riešenie sa zameriava primárne na zachovanie poradia záznamov v rámci jednej partície, a v prípade použitia viacerých partície poradie záznamov nie je zachované. Možné smerovanie pre budúce rozšírenie práce by mohlo zahŕňať skúmanie spôsobov, ako zachovať poradie medzi s použitím viacerých partícií, implementovaním vlastného klienta, alebo optimalizácie systému.

LITERATÚRA

1. **Richardson, Chris.** *Microservice Patterns.* New York, NY : Manning Publications, 2019. s. 14-17. 9781617294549.

2. **Rouse, Margaret.** What is Event-Driven Architecture (EDA)? - Definition from Techopedia. [Online] 29. 12 2022. http://www.techopedia.com/definition/3718/event-driven-architecture-eda.

3. **Richards, Mark.** *Software Architecture Patterns:*. Sebastopol, CA : O'Reilly Media, 2015. 9781491924242.

4. **Dean, Alexander.** *Event Streams in Action.* New York, NY : Manning Publications, 2019. 9781617292347.

5. **Kleppmann, Martin.** *Designing Data-Intensive Applications.* New York, NY : O'Reilly Media, 2017. 9781449373320.

6. **Jena, Manisha.** Message Brokers: Key Models, Use Cases & Tools Simplified 101. [Online] 19. 4 2022. [Dátum: 4. 11 2022.] https://hevodata.com/learn/message-brokers/.

7. **Etzion, Opher and Niblett, Peter.** *Event Processing in Action.* New York, NY : Manning Publications, 2011. 9781935182214.

8. **Koutanov, Emil.** *Effective Kafka.* s.l. : Leanpub, 2021. 9798628558515.

9. **Apache Kafka.** Apache Kafka. [Online] 2023. [Dátum: 01. 15 2023.] https://kafka.apache.org/intro#intro_concepts_and_terms.

10. **McCabe, Colin.** KIP-500: Replace ZooKeeper with a Self-Managed Metadata Quorum. [Online] 09. 07 2020. [Dátum: 03. 14 2023.] https://cwiki.apache.org/confluence/display/KAFKA/KIP-500%3A+Replace+ZooKeeper+with+a+Self-Managed+Metadata+Quorum.

11. **Wikipedia. *Idempotence.* [Online] 10. 4 2023. https://en.wikipedia.org/wiki/Idempotence.**

12. Waldron, Wade. Idempotency and ordering in event-driven systems. [Online] 12. 05 2022. [Dátum: 28. 03 2023.] https://www.cockroachlabs.com/blog/idempotency-and-ordering-in-event-driven-systems/.

13. Big Data In Real World. Can multiple Kafka consumers read the same message from a partition? [Online] 26. 05 2021. [Dátum: 18. 02 2023.] https://www.bigdatainrealworld.com/can-multiple-kafka-consumers-read-the-same-message-from-a-partition/.

14. Overview | Kubernetes. [Online] 3. 1 2023 . [Dátum: 20. 4 2023.] https://kubernetes.io/docs/concepts/overview/.

15. Websupport, s.r.o. Kubernetes – 1. História, architektúra a inštalácia. [Online] 14. 7 2022. [Dátum: 8. 5 2023.] https://www.websupport.sk/podpora/kb/kubernetes-1-historia-architektura-a-instalacia/.

16. V.P., Akash. *7-Day Guide for Easy Understanding of Kubernetes for Developers and IT Professionals.* s.l. : Akash V.P., 2022. 1393677908.

17. Sarvepalli, Krishna Chaitanya. [Online] 24. 11 2018. [Dátum: 10. 05 2023.] https://chkrishna.medium.com/kubernetes-objects-e0a8b93b5cdc.

18. Mehta, Apurva. KIP-185: Make exactly once in order delivery per partition the default producer setting. [Online] 26. 8 2017. [Dátum: 12. 5 2023.] https://cwiki.apache.org/confluence/display/KAFKA/KIP-185%3A+Make+exactly+once+in+order+delivery+per+partition+the+default+Producer+setting.

19. Rao, Jun a Mastrodicasa, Mario. Clients - Apache Kafka. [Online] 15. 3 2023. [Dátum: 5. 16 2023.] https://cwiki.apache.org/confluence/display/KAFKA/Clients.

20. Russell, Gary, a iní. Spring for Apache Kafka. [Online] Apache software fundation, 15. 5 2023. [Dátum: 16. 5 2023.] https://docs.spring.io/spring-kafka/reference/html/.