Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Matija Kljun

Integracija pretočnih dogodkov in mikrostoritev z uporabo Apache Kafka

DIPLOMSKO DELO

UNIVERZITETNI ŠTUDIJSKI PROGRAM
PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Mentor: prof. dr. Matjaž Branko Jurič

Ljubljana, 2017

Copyright. Rezultati diplomske naloge so intelektualna lastnina avtorja in Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavo in koriščenje rezultatov diplomske naloge je potrebno pisno privoljenje avtorja, Fakultete za računalništvo in informatiko ter mentorja.

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Opišite pomen mikrostoritev pri razvoju sodobnih aplikacij in opišite modele interakcije. Podrobno analizirajte dogodkovno interakcijo med storitvami in napravite primerjavo različnih modelov. Proučite sporočilne sisteme in platforme za pretakanje dogodkov ter jih primerjajte. Podrobno analizirajte platformo Apache Kafka. Zasnujte in realizirajte integracijo med Apache Kafko in ogrodjem za razvoj mikrostoritev v Javi. Razvijte praktični primer, na katerem boste prikazali in ovrednotili delovanje.

Posebno bi se rad zahvalil mentorju prof. dr. Matjažu Branku Juriču in ostalim članom Laboratorija za integracijo informacijskih sistemov za strokovno pomoč in vodenje. Prav tako bi se rad zahvalil vsem bližnjim, družini in prijateljem za podporo in spodbudo.

Kazalo

Povzetek

Abstract

1 Uvod **1**

1.1 Motivacija 1

1.2 Cilji 2

1.3 Struktura 2

2 Mikrostoritve in interakcija mikrostoritev **3**

2.1 Mikrostoritve 3

2.2 Interakcija mikrostoritev 8

2.3 Dogodkovna interakcija 15

2.4 Primerjava interakcij mikrostoritev 18

2.5 Ogrodje KumuluzEE 20

3 Sporočilni in dogodkovni sistemi **21**

3.1 Sporočilni sistemi in sporočilne vrste 21

3.2 Platforma za pretakanje dogodkov 23

3.3 Primerjava 28

4 Platforma Apache Kafka **31**

4.1 Pregled arhitekture 31

4.2 Teme in particije 34

4.3 Kafka posredniki in gruče 35

4.4 Kafka proizvajalci 36
4.5 Kafka potrošniki 37
4.6 Procesiranje tokov s Kafko 38

5 Integracija Apache Kafke in mikrostoritev 41
5.1 Razširitev KumuluzEE Event Streaming 41
5.2 Implementacija vzorcev dogodkovnih virov in CQRS s pomočjo razširitve KumuluzEE Event Streaming 47

6 Sklepne ugotovitve 53

Literatura 57

Seznam uporabljenih kratic

kratica	angleško	slovensko
API	Application Programming Interface	aplikacijski programski vmesnik
AWS	Amazon Web Services	Amazonove spletne storitve
TCP	Transmission Control Protocol	protokol za nadzor prenosa
IPC	Inter-Process Communication	medprocesna komunikacija
REST	Representational State Transfer	predstavitveni prenos stanja
HTTP	Hypertext Transfer Protocol	protokol za prenos hiperteksta
URI	Uniform Resource Identifier	enolični identifikator vira
ALPN	Application-Layer Protocol Negotiation	pogajanje o protokolu aplikacijskega sloja
JSON	JavaScript Object Notation	notacija objektov JavaScript
XML	Extensible Markup Language	razširljiv označevalni jezik
HATEOAS	Hypertext As The Engine Of Application State	hipertekst kot vodilo aplikacijskega stanja
2PC	Two-Phase Commit	dvosmerna potrditev
MOM	Message-Oriented Middleware	sporočilno usmerjena vmesna oprema
CDI	Contexts and Dependency Injection	vstavljanje konteksta in odvisnosti

AMQP	Advanced Message Queuing Protocol	napreden protokol za sporočilne vrste
STOMP	Simple Text Oriented Messaging Protocol	enostaven tekstovno usmerjen sporočilni protokol
URL	Uniform Resource Locator	enolični krajevnik vira
ETL	Extract, Transform, Load	izvleček, preoblikovanje, nalaganje
IoT	Internet of Things	internet stvari
CQRS	Command Query Responsibility Separation	ločitev odgovornosti ukazov in poizvedb

Povzetek

Naslov: Integracija pretočnih dogodkov in mikrostoritev z uporabo Apache Kafka

Avtor: Matija Kljun

Skozi zadnje desetletje so velika in uspešna internetna podjetja, kot so Netflix, Amazon in LinkedIn, arhitekturo mikrostoritev uporabile kot temelj svojega delovanja. Procesiranje tokov podatkov, izmenjava in agregiranje le teh s t. i. platformo za pretakanje dogodkov prav tako postaja vse bolj pomembno, saj prinaša večjo konkurenčnost, agilnost in odzivnost. V diplomskem delu so analizirane interakcije med mikrostoritvami in opisane platforme za pretakanje ter običajne sporočilne vrste. Natančneje je predstavljena platforma Apache Kafka in njeno delovanje. V sklopu naloge je bila razvita razširitev KumuluzEE Event Streaming, ki omogoča lažjo integracijo pretočnih dogodkov in mikrostoritev na platformi Java. Z razširitvijo se lahko Apache Kafka enostavno uporabi v sistemu mikrostoritev, saj nudi anotacije za pošiljanje in prejemanje sporočil ter procesiranje toka. Kot prikaz uporabe razširitve je bila razvita aplikacija za spletno knjigarno v arhitekturi dogodkovno vodenih mikrostoritev po vzorcih dogodkovnih virov in CQRS, ki temelji na uporabi Apache Kafke.

Ključne besede: mikrostoritve, Apache Kafka, pretakanje dogodkov, sporočilni sistemi, dogodkovni viri.

Abstract

Title: Integration of event streaming and microservices with Apache Kafka

Author: Matija Kljun

Over the last decade, the microservice architecture has become a standard for big and successful internet companies, like Netflix, Amazon and LinkedIn. The importance of stream processing, aggregation and exchange of data is growing, as it allows companies to compete better and move faster. In this diploma, we have analyzed the interactions between microservices, described the streaming platform and ordinary message queues. We have described the Apache Kafka platform and how it works. We have developed the KumuluzEE Event Streaming extension for the Java platform that allow easy integration of event streaming and microservices. With the use of the extension, you can easily implement the Kafka platform in our microservice system. Our extension provides easy to use annotations for producing, consuming messages and stream processing. As a proof of concept, we have developed a sample bookstore application. It has been designed in the event driven microservice architecture with the use of event sourcing and CQRS patterns.

Keywords: microservices, Apache Kafka, event streaming, messaging systems, event sourcing.

Poglavje 1

Uvod

1.1 Motivacija

S porastom sodobnih spletnih aplikacij, mobilnih in IoT naprav se količina podatkov veča in vse bolj pomembno postaja, kako s temi podatki ravnamo. Podjetje Netflix se je na tem področju izkazalo, saj uspešno analizira in obdeluje povprečno več kot 500 milijard dogodkov na dan. Na pretočnih dogodkih in mikrostoritvah zasnovan sistem se je izkazal za uspešnega in vodi do višje odzivnosti in večje konkurenčnosti podjetja.

Pri arhitekturi mikrostoritev je pomembna interakcija med storitvami, ki mora biti zasnovana tako, da ohranja šibko sklopljenost in neodvisnost posameznih storitev. V različnih scenarijih so primerni različni načini interakcije bodisi sinhroni, kot je REST ali Trift, bodisi asinhroni, kot so sporočilni sistemi. Novost na tem področju so platforme za pretakanje dogodkov, kot je Apache Kafka, katerih uporaba strmo narašča [17]. Ključne prednosti platform za pretakanje dogodkov so visoka pretočnost in nizke zakasnitve ter zmožnosti procesiranja toka v realnem času.

1.2 Cilji

V diplomskem delu bomo opisali arhitekturo mikrostoritev in predstavili načine interakcij med storitvami. Predstavili bomo sporočilne vrste in platforme za pretakanje dogodkov ter ju primerjali. Podrobneje se bomo osredotočili na platformo Apache Kafka ter opisali njeno zasnovo in delovanje. Cilj diplomske naloge je razviti razširitev za odprtokodno ogrodje KumuluzEE, ki bo omogočala enostavno uporabo pretočnih dogodkov v arhitekturi mikrostoritev v Javi. Razširitev smo zasnovali kot priročne anotacije, ki bodo razvijalcem omogočale enostavno in hitro implementacijo pošiljanja in prejemanja dogodkov ter procesiranje toka s platformo Kafka. Kot zgled uporabe razširitve smo razvili spletno aplikacijo v dogodkovno vodeni arhitekturi mikrostoritev po vzorcu dogodkovnih virov in CQRS.

1.3 Struktura

Diplomsko delo je razdeljeno na šest poglavij. V poglavju 2 in 3 je podana teoretična podlaga snovi. Poglavje 2 opisuje arhitekturo mikrostoritev in načine interakcije med storitvami. Predstavljen je tudi arhitekturni pristop, ki temelji na pretakanju dogodkov in vzorca dogodkovnih virov (*event sourcing*) ter CQRS. V poglavju 3 sledi podroben opis sporočilnih vrst in platform za pretakanje ter navedba razlik med njimi.

V poglavju 4 je natančneje opisana platforma za pretakanje Apache Kafka, njena arhitekturna zasnova in delovanje.

V poglavju 5 je predstavljena odprtokodna razširitev KumuluzEE Event Streaming, ki je bila razvita kot del diplomske naloge. Razloženo je delovanje in zasnova razširitve ter prikazana njena uporaba na primeru aplikacije za spletno knjigarno, ki je zasnovana na principu dogodkovno vodene arhitekture mikrostoritev po vzorcu dogodkovnih virov in CQRS.

Poglavje 2

Mikrostoritve in interakcija mikrostoritev

Arhitektura mikrostoritev je nov pristop h grajenju in oblikovanju aplikacij. Pojavila se je kot posledica novih tehnologij, arhitekturnih vzorcev, kot je *domain-driven design*, novih načinov virtualizacije, avtomatizacije infrastrukture ipd. Arhitektura mikrostoritev predstavlja dekompozicijo monolitnega sistema v samostojne manjše gradnike oziroma storitve. Posamezne storitve so običajno nameščene na različnih računalniških procesih, kar zahteva medsebojno komunikacijo preko omrežja. Interakcija med mikrostoritvami mora biti zasnovana tako, da ne prinaša odvisnosti med njimi in tako ohranja šibko sklopljenost storitev.

V nadaljevanju tega poglavja bomo podrobneje opisali koncept mikrostoritev, njegove prednosti in nove izzive, ki jih prinaša, ter predstavili različne načine interakcij med storitvami.

2.1 Mikrostoritve

Koncept mikrostoritev je izgradnja porazdeljene aplikacije iz več manjših storitev, kjer vsaka teče v svojem procesu in komunicira preko definiranih vmesnikov [11, 16, 23]. Vsaka storitev je zgrajena okoli posameznega poslovnega

področja, njeno nameščanje je neodvisno in avtomatizirano. Mikrostoritve se primerja s starejšo monolitno arhitekturo, ki temelji na grajenju celotne aplikacije kot samostojne enote. Takšna arhitekturna zasnova omogoča enostaven razvoj, testiranje in postavitev enotne aplikacije, vendar pri večjih monolitnih aplikacijah pride do številnih težav, kot je problem skaliranja sistema, kompleksnost programske kode in njeno vzdrževanje, težje posodabljanje, popravljanje in testiranje novih funkcionalnosti. Uvajanje novih tehnologij in programskih jezikov je tvegano in dolgotrajno. Razvijalci potrebujejo več časa za obvladovanje celotnega sistema in organizacija dela v manjših skupinah je težje dosegljiva.

Razlog za uvajanje mikrostoritev so problemi z monolitno arhitekturo, saj ti rešujejo številne probleme monolitnih sistemov. Mikrostoritve so majhne in obvladljive komponente, ki upravljajo določeno poslovno funkcijo. Posamezno mikrostoritev lahko ločeno razvijajo manjše skupine razvijalcev. Storitve so med seboj neodvisne in tako lahko delujejo z različnimi tehnologijami in programskimi jeziki. Za komunikacijo med storitvami uporabljamo različne pristope, ki morajo ohranjati neodvisnost posameznih komponent. Vsaka storitev izpostavlja vmesnik API, prek katerega komunicirajo ostale storitve.

2.1.1 Ključne prednosti arhitekture mikrostoritev

Mikrostoritve omogočajo neodvisno in učinkovito skaliranje posameznih storitev. Pri monolitni aplikaciji je, četudi je preveč obremenjen le majhen del aplikacije, potrebno skalirati celoten sistem. Z arhitekturo mikrostoritev lahko skaliramo le določene najbolj obremenjene storitve in izvajamo ostale manj zahtevne storitve na manj zmogljivi in cenejši strojni opremi. S storitvami, ki jih ponujajo Amazonov AWS in podobni ponudniki, lahko izvajamo skaliranje storitev na zahtevo glede na trenutno obremenitev. Tako lahko veliko bolj učinkovito upravljamo stroške gostovanja našega sistema [24].

Pri mikrostoritvah se aplikacijo razčleni na ločene storitve glede na njihovo poslovno področje oziroma funkcijo. Vsaka storitev je organizirana okoli neke poslovne enote. Odgovornost za posamezno storitev prevzame

ločena ekipa razvijalcev, ki skrbi za celoten sklop tehnologij od podatkovne baze do postavitve storitve in testiranja. Razvijalske ekipe so manjše in je zato komunikacija lažja. Razvijalcem ni treba obvladati celotne programske kode aplikacije ampak le posamezne storitve, kar prinaša večjo produktivnost. Pri mikrostoritvah je prišlo do spremembe v načinu načrtovanja in organizacije dela. V monolitnih sistemih je običajno šlo za ureditev delovnih skupin na silose glede na tehnološko področje dela (razvoj programske opreme, upravljanje s podatkovnimi bazami, sistemska administracija itd.). Takšna ločitev privede do slabše komunikacije znotraj organizacije. Conwayev zakon pravi, da zasnova sistema, ki ga izdeluje neka organizacija, prezrcali strukturo komunikacije v organizaciji. Pri mikrostoritvah se tako stremi k manjšim ekipam, ki obvladujejo vsa tehnološka področja, povezana s storitvijo, ki jo razvijajo. Nov način, nasproten silosni ureditvi organizacije dela, se imenuje *DevOps* [21, 6].

Z razčlenitvijo na ločene neodvisne storitve omogočimo tudi hitrejši razvoj, testiranje in ponovno postavitev. Dokler so spremembe omejene na kontekst ene storitve in ne kršijo uveljavljenih pogodb z ostalim sistemom, lahko posamezno storitev posodabljamo neodvisno od preostalega sistema. S tem omogočimo pogostejše in hitrejše uvajanje sprememb in posodobitev v sistemu ob manjšem tveganju za napake, saj so te izolirane na posamezno storitev. To tudi pomeni, da lahko nove funkcionalnosti pridejo hitreje do uporabnikov, kar je za podjetja zelo pomembno.

Mikrostoritve omogočajo razvijanje posamezne storitve z različnimi tehnologijami in orodji. Programski jezik lahko prilagodimo zahtevam, ki jih postavlja določen problem. Posamezne storitve lahko hranijo podatke na različne načine. Izbiramo lahko podatkovno bazo, ki je najbolj primerna tipu podatkov, ki jih storitev obdeluje. Prav tako lahko lažje in hitreje uvajamo nove tehnologije, saj ni potrebno prepisati celotne aplikacije, ampak le posamezno storitev. Običajen problem uvajanja novih tehnologij je z njimi povezano tveganje. Pri mikrostoritvah to tveganje minimiziramo, saj lahko novo tehnologijo postopoma preizkušamo na posamezni storitvi in

tako omejimo morebiten negativen vpliv. Pri monolitnih aplikacijah imamo običajno eno relacijsko podatkovno bazo, katere prednost je uporaba transakcij ACID, ki zagotavljajo atomarnost, konsistentnost, izolacijo in trajnost podatkov. Zaradi porazdeljenosti postane pri arhitekturi mikrostoritev dostop do podatkov bolj zapleten. Podatki vsake storitve so drugim dostopni le preko vmesnika API. Ločitev podatkov vsake storitve je potrebna zato, da so mikrostoritve med seboj šibko sklopljene in neodvisne ena od druge. Posamezne storitve imajo lahko tudi različne tipe podatkovnih baz, ki najbolj ustrezajo vrsti podatkov, katere storitev upravlja. Pristopu upravljanja podatkov pri mikrostoritvah, kjer se uporablja kombinacija različnih SQL in NoSQL podatkovnih baz, pravimo *Polyglot Persistence* [10].

Posebnost mikrostoritev je tudi odpornost na napake in izpade. Storitve morajo biti zasnovane tako, da se napaka ali izpad ene storitve izolira in ne vpliva verižno na preostali del sistema. Če neka storitev odpove pri monolitni arhitekturi, preneha delovati celoten sistem, z mikrostoritvami pa lahko zgradimo sistem, ki vzdrži popoln izpad posamezne storitve in ohrani neko poslabšano delovanje celotnega sistema. Vgradnja odpornosti na napake in izpade ni enostavna, potrebna je pozornost na nove vire napak v distribuiranem sistemu in omrežju. Potrebno je odkrivanje napak in nepravilnega delovanja, zato je pomembno spremljanje (*monitoring*) in zbiranje metrik sistema v realnem času. Za preprečevanje napak pri sinhronih klicih na oddaljene storitve se uporablja arhitekturni vzorec odklopnikov (*circuit breaker*). Odklopniki delujejo tako, da preko njih izvedemo klic na oddaljeno storitev. Če število neuspešnih klicev preseže neko mejo, odklopnik prekine tok in za določen čas za vse nadaljne klice vrne napako. Po določenem času odklopnik ponovno odpre tok za nekaj testnih klicev, in če so ti uspešni, popolnoma odpre povezavo s storitvijo. Z odklopniki preprečimo čakanje odjemalcev na neodzivno storitev in možnost večje odzivnosti na napake [19].

2.1.2 Izzivi arhitekture mikrostoritev

Kakor vsak arhitekturni koncept tudi mikrostoritve prinašajo dodatna tveganja in izzive. Zaradi porazdeljenosti sistema nastajajo nove ovire in zapleti, med drugim s hitrostjo. Težava nastane pri interakciji med oddaljenimi storitvami. Oddaljeni klici potrebujejo več časa kot funkcijski klici znotraj procesov pri monolitnih aplikacijah. Problem nastane, ko naša storitev kliče več oddaljenih storitev in se tako časi odgovorov kopičijo. Možni rešitvi sta zmanjšanje razdrobljenosti klicev, tako da jih izvedemo čim manj, ali uporaba asinhronih klicev, pri katerih ne ustavimo procesa pri čakanju na odgovor, vendar s slednjim vpeljemo princip asinhronega programiranja, ki je bolj kompleksno in težje za učenje in upravljanje. Pri porazdeljenih sistemih je problematična tudi zanesljivost, saj so lahko oddaljeni klici zaradi nezanesljivega omrežja neuspešni.

Zaradi decentraliziranega upravljanja podatkov pri mikrostoritvah ni mogoče zagotoviti popolne konsistentnosti podatkov. Mikrostoritve dosegajo eventualno konsistentnost (*eventual consistentcy*), kar pa lahko privede do težav, kot je izvedba poslovne logike na nekonsistentnih podatkih. Potrebna je dodatna pozornost glede konsistentnosti podatkov ter način, kako preverjati, ali so podatki v sistemu sinhronizirani. Pojavi se nov izziv izvajanja transakcij preko več storitev in preko različnih tipov podatkovnih baz v primeru *Polyglot Persistence*. Transakcije morajo ohranjati konsistentnost podatkov. Pri monolitnih aplikacijah se takšen problem običajno rešuje s porazdeljenimi transakcijami *two-phase commit* 2PC, vendar se teh zaradi težav s porazdeljenim sistemom in slabe podpore podatkovnih baz NoSQL pri arhitekturi mikrostoritev izogibamo.

Pri mikrostoritvah je problematična tudi operacijska kompleksnost, ki jo prinaša upravljanje s števlnimi storitvami v sistemu. Čeprav so posamezne storitve majhne in lažje razumljive, se kompleksnost prestavi na povezave med storitvami. Potrebnega je veliko znanja in izkušenj s postavljanjem, testiranjem in spremljanjem storitev v sistemu, da lahko dosežemo vse prednosti mikrostoritev. Poleg tehničnega znanja je pomembna tudi organizacija

dela in komunikacija, ki jo prinaša kultura *DevOps* [12].

2.2 Interakcija mikrostoritev

Pri monolitni aplikaciji se komponente med seboj kličejo na ravni programskega jezika preko klicev metod ali funkcij. Pri mikrostoritvah to ni mogoče, saj tečejo posamezne storitve na več računalnikih in je vsaka storitev pogosto ločen proces. Za to je potreben drugačen pristop, ki se imenuje medprocesna komunikacija (*inter-process communication*), s kratico IPC. Poznamo več specifičnih tehnologij IPC, po zasnovi se razlikujejo po dveh dimenzijah [20]. Delimo jih glede na način interakcije ena-na-ena ali ena-na-mnogo:

- ena-na-ena - Vsako poslano zahtevo obravnava le ena storitev.
- ena-na-mnogo - Vsako poslano zahtevo predela več storitev.

Druga dimenzija ločitve so asinhrone in sinhrone interakcije:

- Sinhrone - Ko odjemalec pošlje zahtevo, pričakuje odgovor v določenem časovnem obdobju, med čakanjem običajno ustavi izvajanje procesa.
- Asinhrone - Odjemalec med čakanjem na odgovor ne zaustavi izvajanja, ko ta prispe, se avtomatsko proži obdelava odgovora.

Glede na omenjeni dimenziji ločimo več načinov interakcije, ki so podane v tabeli 2.1.

	ena-na-ena	**ena-na-mnogo**
Sinhrono	Zahteva / Odgovor	/
Asinhrono	Obvestilo	Objava / Naročilo
	Zahteva / Asinhron odgovor	Objava / Asinhron odgovor

Tabela 2.1: Načini medprocesne komunikacije.

Med ena-na-ena interakcije, tako asinhrone kot sinhrone, štejemo naslednje načine komunikacij:

- Zahteva / Odgovor - Odjemalec pošlje zahtevo storitvi in čaka na odgovor. Odgovor pričakuje v nekem časovnem obdobju in med čakanjem običajno ustavi delovanje.
- Obvestilo (enosmerna zahteva) - Odjemalec pošlje zahtevo, vendar odgovora ne pričakuje.
- Zahteva / Asinhron odgovor - Odjemalec pošlje zahtevo storitvi, ki nanjo odgovori asinhrono. Odjemalec med čakanjem ne blokira delovanja. Odgovor na zahtevo lahko potrebuje tudi več časa, da prispe.

Ena-na-mnogo interakciji, obe asinhroni, pa sta:

- Objava / Naročilo - Odjemalec pošlje obvestilo, ki ga prejme in obdela nobena ali več naročenih storitev.
- Objava / Asinhron odgovor - Odjemalec pošlje sporočilo in določen čas čaka na odgovore naročenih storitev.

V naslednjem poglavju bomo podrobneje opisali specifične tehnološke implementacije sinhronih in asinhronih medprocesnih interakcij mikrostoritev.

2.2.1 Sinhrona interakcija

Pri uporabi sinhrone komunikacije, ki temelji na zahtevi in odgovoru, odjemalec pošlje zahtevo storitvi, ki jo ta obdela in vrne odgovor. Odjemalec, ki pošlje zahtevo, v nekaterih primerih med čakanjem blokira izvajanje. Temu se lahko izognemo z uporabo asinhrone programske kode, tj. s knjižnico, kot je *ReactiveX*. Obstaja več protokolov za sinhrono komuniciranje, med najbolj znanimi je protokol REST, katerega bomo podrobneje opisali v naslednjem podpoglavju, drugi so še Trift in gRPC.

REST

REST je način medprocesne komunikacije, ki v večini primerov temelji na protokolu HTTP. Ključni koncept REST so viri, ki tipično predstavljajo neko

poslovno enoto, na primer kupca ali produkt. Z pomočjo metod HTTP lahko manipuliramo z viri, na katere se sklicujemo z naslovom URL. Vire običajno vračamo v obliki objekta JSON ali dokumenta XML.

Model, ki ga je razvil Leonard Richardson za razvoj vmesnikov po principu REST, je sestavljen iz naslednjih nivojev [9]:

- Nivo 0 - pri tem nivoju je vmesnik zasnovan na osnovi protokola HTTP kot transportnega sistema za oddaljene klice. Izpostavljen je samo en naslov URL, na katerega se pošiljajo zahteve HTTP POST z specificirano akcijo in morebitnimi parametri.

- Nivo 1 - Vmesnik nivoja 1 podpira koncept virov. Za izvedbo akcije nad virom se pošlje zahteva HTTP POST na določen naslov z specificirano akcijo za izvedbo in parametri.

- Nivo 2 - Na nivoju 2 vmesnik uporablja metode HTTP za izvedbo akcij: GET za prevzem virov, POST za kreacijo, PUT za posodabljanje itd. Parametre akcije v zahtevi določimo s parametri za poizvedbo (*query parameters*) ter dodatnimi parametri v telesu zahteve. Z uporabo metod HTTP izkoriščamo spletno infrastrukturo, kot je predpomnjenje zahtevkov GET.

- Nivo 3 - Vmesnik, zasnovan na nivoju 3, temelji na principu HATEOAS (*Hypertext As The Engine Of Application State*). Deluje tako, da predstavitev vira, vrnjena z metodo GET, vsebuje povezave do dovoljenih operacij nad virom. Prednost takšnega pristopa je, da ni potrebno poznati vseh možnih operacij in njihovih naslovov.

Vse bolj se za dokumentacijo vmesnikov REST uporabljajo t. i. jeziki za definicijo vmesnikov (*interface definition language*), na primer OpenAPI ali RAML. Nekatere rešitve, kot je OpenAPI, omogočajo definicijo formata zahtev in odgovorov, druge, kot je RAML, pa uporabljajo druge vrste specifikacije, na primer shemo JSON.

Ključne prednosti vmesnikov REST, ki uporablja protokol HTTP, so:

- Enostavnost uporabe in razširjenost protokola HTTP.
- Enostavno testiranje, na voljo je vrsta različnih orodij (npr. Postman) ali iz ukazne vrstice z ukazom `curl`.
- Pri komunikaciji ne potrebujemo vmesnega posrednika, kar poenostavi arhitekturo sistema.
- Vmesniki REST so neodvisni od vrste platforme ali programskega jezika, kar omogoča komunikacijo med storitvami v različnih tehnologijah in implementacijo novih tehnologij brez dodatnih težav.

Slabosti uporabe vmesnikov REST in protokola HTTP pa so:

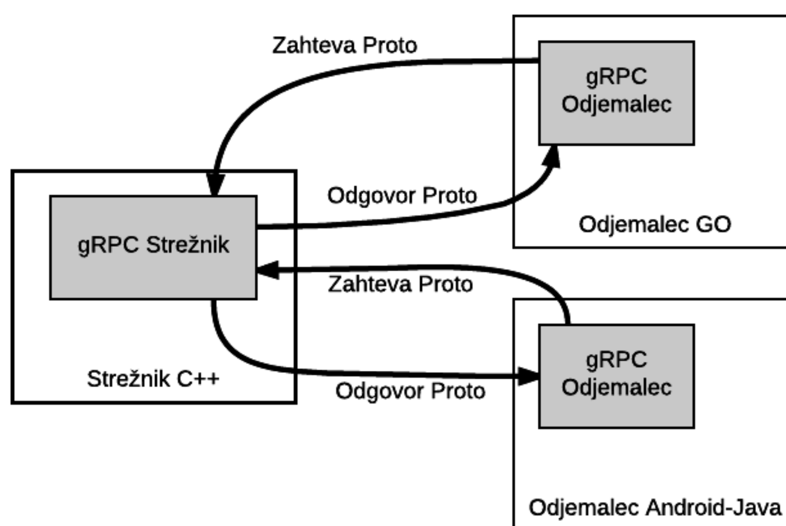
- HTTP podpira le komunikacijo zahteva / odgovor. Tudi pri uporabi za pošiljanje obvestil mora storitev vrniti odgovor.
- Ker odjemalec in strežnik komunicirata neposredno, brez posrednika, morata med trajanjem izmenjave delovati oba.
- Odjemalec mora poznati lokacijo vseh storitev (naslov URL), s katerimi želi komunicirati, kar pa pri porazdeljenih sistemih, kot so mikrostoritve, ni enostavno. Potrebna je uporaba mehanizma odkrivanja storitev (*service discovery*) za iskanje naslovov storitev.

Nova verzija protokola HTTP/2 prinaša številne izboljšave v primerjavi s starejšo verzijo HTTP 1.1, hkrati pa ohranja kompatibilnost s prejšnjimi izdajami. HTTP/2 ohranja visokonivojsko sintakso HTTP 1.1, kot so metode, statusne kode, URI in polja v glavi. Tako od obstoječih spletnih aplikacij ne zahteva prilagoditev za HTTP/2. Prednosti in izboljšave protokola so naslednje [4]:

- HTTP/2 format je binaren namesto tekstovni.
- HTTP/2 je popolno multipleksiran, kar pomeni, da omogoča pošiljanje več zahtevkov za podatke vzporedno preko ene povezave TCP.

- Uporablja stiskanje glave zahtevkov s HPACK, s čimer zmanjša režijske stroške.
- Omogoča strežnikom, da potiskajo podatke v predpomnilnike odjemalcev, namesto da bi čakali na nove zahteve za vsak vir.
- Uporablja razširitev ALPN, ki omogoča hitrejše šifrirane povezave.

gRPC



Slika 2.1: Prikaz komunikacije gRPC med strežnikom in odjemalcema, implementiranima v različnih programskih jezikih.

Odprtokodno ogrodje gRPC omogoča izvajanje univerzalnih oddaljenih klicev podprograma RPC. Prednost uporabe gRPC je visoka zmogljivost in varnost oddaljenih klicev. Z gRPC odjemalec izvaja oddaljene klice funkcij na strežnik, kot da bi bile v istem procesu. Kakor v številnih sistemih za RPC tudi gRPC temelji na definirani storitvi, ki opredeljuje funkcije, ki so na voljo za oddaljen klic, njihove parametre in odgovore. Na strežniku implementiramo storitev in izvajamo strežnik gRPC, ki obravnava oddaljene klice odjemalcev. Na strani odjemalca je implementiran gRPC odjemalec

(*stub*), ki izpostavlja definirane funkcije strežnika. Strežniki in odjemalci gRPC lahko med seboj komunicirajo v različnih okoljih, izvajajo se lahko v različnih operacijskih sistemih in so lahko napisani v različnih programskih jezikih, ki jih podpira gRPC, kar je prikazano na sliki 2.1. Kot format sporočil gRPC uporablja odprtokodni mehanizem za serializacijo strukturiranih podatkov Protocol Buffers. V primerjavi z drugimi formati žičnih protokolov (*wire-format protocol*) je Protocol Buffers manjši, hitrejši in bolj učinkovit. Za implementacijo visoko zmogljivih in skalabilnih vmesnikov API gRPC uporablja protokol HTTP/2, ki prinaša številne prednosti, kot je multipleksiranje zahtevkov preko povezave TCP.

2.2.2 Asinhrona interakcija

Pri asinhroni interakciji med mikrostoritvami odjemalec pošlje zahtevo, na katero je naročenih nič ali več storitev, ki zahtevo prejmejo in jo obdelajo. Če odjemalec zahteva odgovor, ga storitev vrne, vendar odjemalec med čakanjem nanj ne ustavi izvajanja. Med načine asinhrone interakcije štejemo:

- Enosmerna komunikacija - odjemalec pošlje zahtevo in ne pričakuje odgovora.
- Izpraševanje (*polling*) - odjemalec konstantno v intervalih pošilja zahtevke za nove ali posodobljene podatke na storitev, ta mu jih v odgovoru vrne. Pri dolgem izpraševanju *long polling* odjemalec pošlje zahtevo, storitev pa odgovori šele ko pridobi nove ali posodobljene podatke. Ko odjemalec prejme odgovor, pošlje novo zahtevo.
- Povratni klic (*callback*) - odjemalec pošlje zahtevo z naslovom za povratni klic, storitev odgovori z izvedbo povratnega klica na ta naslov.

Sporočilni sistemi

Pri sporočilnih sistemih odjemalec pošlje sporočilo na kanal, na katerega je naročenih nič ali več storitev, ki sporočilo prejmejo in ga obdelajo. Če

odjemalec zahteva odgovor, storitev pošlje nazaj sporočilo. Odjemalec je napisan tako, da pri čakanju na odgovor ne ustavi procesa ter ne pričakuje takojšnjega odziva. Sporočila so običajno sestavljena iz glave sporočila, ki vsebuje metapodatke, kot je podatek o pošiljatelju, in iz telesa sporočila z vsebino. Sporočila se pošiljajo preko kanala, na katerega lahko proizvajajo sporočila več odjemalcev, prav tako lahko iz njega bere več storitev. Poznamo dve vrsti kanalov:

- Točkovni kanal, na katerega pošljemo sporočilo eni storitvi, ki bere iz njega. To je primer ena-na-ena komunikacije.
- Objavi/naroči kanal dostavi vsako sporočilo vsem naročenim storitvam. Takšen kanal se uporablja za ena-na-mnogo način komunikacije.

Obstaja več različnih implementacij sporočilnih sistemov, ki se med seboj razlikujejo po protokolu, ki ga uporabljajo, formatih sporočil, programskih jezikih, ki jih podpirajo itd. Nekateri sporočilni sistemi uporabljajo standardne protokole, kot so AMQP ali STOMP, drugi pa imajo lastne protokole. Prednosti, ki jih prinašajo sporočilni sistemi, so:

- Sporočilni sistemi ločijo storitve med seboj in vpeljejo šibko sklopljenost med njimi. Z uporabo sporočil se storitve izognejo odvisnostim od drugih delov sistema.
- Pošiljatelju sporočila ni potrebno poznati lokacije storitev, ki prejemajo sporočila, tako ni potrebe po mehanizmu za odkrivanje storitev.
- Sporočila se v kanalu hranijo dokler jih storitve ne preberejo, tako ni potrebe, da je storitev v delujočem stanju, ko odjemalec pošlje sporočilo.
- Sporočilni sistemi omogočajo fleksibilno komunikacijo, saj podpirajo vse vrste interakcij, naštetih v tabeli 2.1.

Kot slabost sporočilnih sistemov lahko navedemo dodatno kompleksnost, ki jo prinese dodaten sistem, ki ga je potrebno namestiti, vzdrževati in konfigurirati. Poleg tega je za implementacijo načina interakcije zahteva/odgovor

potrebno dodatno delo. Vsako sporočilo z zahtevo mora definirati kanal za odgovor ter vsebovati korelacijski identifikator. Storitev pošlje v odgovor sporočilo, ki vsebuje korelacijski identifikator, na definiran kanal za odgovor. Odjemalec uporabi korelacijski identifikator, da poveže zahteve z odgovori.

Več o sporočilnih sistemih bomo opisali v poglavju 3.

Webhooks

Koncept Webhooka je povratni klic HTTP. Ko se določen dogodek zgodi, se izvede metoda POST. Vse skupaj deluje kot nekakšno obvestilo o dogodku prek HTTP metode POST. Z Webhooki lahko dobimo informacijo o dogodku takoj ko se zgodi, ne da bi po njih nenehno izpraševali (*polling*). Trije glavni načini uporabe Webhookov so:

- Potiskanje podatkov - Raje kot bi za podatki stalno izpraševali, lahko registriramo Webhook, ki vrne podatke takoj, ko so ti na voljo.
- Cevovod podatkov - Ko Webhook prejme podatke, jih na nek način obdela in jih uporabi ali pošlje dalje.
- Vtičnik - omogoča procesiranje podatkov in izvedbo dodatnih funkcionalnosti, kar omogoča uporabnikom, da aplikacijo razširijo.

Problem Webhookov je, da izvajajo neposredne klice med storitvami, kar povzroči odvisnost med storitvami in težave v primeru izpada storitve, ki jo želimo klicati.

2.3 Dogodkovna interakcija

Arhitektura aplikacij, kjer interkacija poteka preko objave in porabe dogodkov, se imenuje dogodkovno vodena arhitektura (*event-driven architecture*) [5, 20, 8]. Pri takšni arhitekturi storitve objavljajo in porabljajo dogodke. Storitev objavi dogodek, kadar se stanje entitete spremeni, neka druga storitev pa se lahko na te dogodke naroči in ob prejetem dogodku posodobi svoje

entitete ter mogoče objavi še kakšen drug dogodek. Na ta način lahko vgradimo poslovno transakcijo, ki posodobi več entitet kot zaporedje korakov, od katerih vsak posodobi entiteto in objavi dogodek, ki sproži naslednji korak v transakciji. Prav tako lahko s takšno zasnovo ohranjamo konsistentnost podvojenih podatkov, tako da storitev spremlja dogodke, ki so objavljeni ob spremembi glavne entitete.

Dogodkovno vodena arhitektura je rešitev izzivov porazdeljenega upravljanja s podatki pri mikrostoritvah. Prinaša šibko sklopljenost storitev, saj so storitve, ki objavljajo dogodke, popolnoma neodvisne od storitev, ki so nanje naročene. Posledično je takšen sistem bolj skalabilen in prilagodljiv. Prednost je tudi zmožnost implementacije transakcij, ki potekajo prek več storitev, in s tem dosežena eventualna konsistentnost podatkov.

Izziv arhitekture so predvsem zapleten in neobičajen način programiranja z dogodki. Potrebni so mehanizmi, ki v primeru neuspelih transakcij povrnejo prvotno stanje, ter upravljanje z nekonsistentnimi podatki, kar prinaša še dodatna tveganja. Predpogoj za delovanje dogodkovno vodene arhitekture je zmožnost atomarne izvedbe posodobitve entitete (npr. v podatkovni bazi) in objave dogodka. V tradicionalni monolitni aplikaciji lahko podatkovna baza in sistem za objavo dogodka sodelujeta pri porazdeljeni transakciji, vendar ta ni primerna za porazdeljene sisteme, kot so mikrostoritve. Rešitev za ta problem je pristop dogodkovnih virov (*event sourcing*), ki ga bomo podrobneje predstavili v naslednjem podpoglavju.

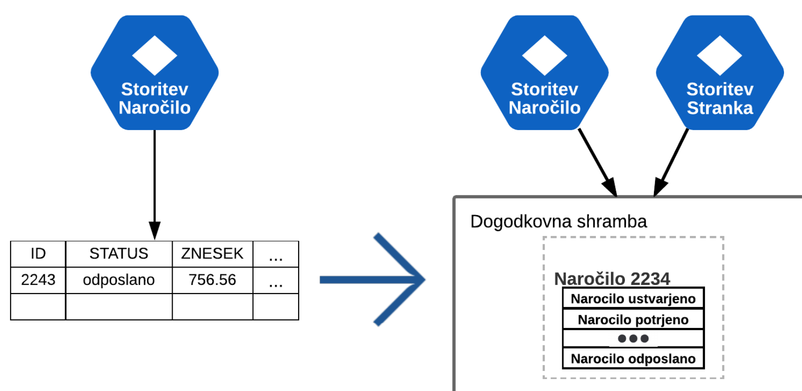
Dogodkovni viri in CQRS

Arhitekturni pristop dogodkovnih virov(*event sourcing*) je povsem drugačen pristop k upravljanju s podatki, ki temelji na dogodkih. Omogoča enostavnejši razvoj dogodkovno vodenih aplikacij. Ključna ideja pristopa dogodkovnih virov je, da namesto, da aplikacija hrani trenutno stanje entitete, hrani zaporedje sprememb entitete v obliki dogodkov; glej sliko 2.2. Primer dogodkov so na primer pri storitvi za bančni račun „nov račun“, „priliv na račun“ in „odliv z računa“. Ko želi storitev pridobiti trenutno stanje enti-

tete, predela vse dogodke o spremembah entitete. Ko se izvede sprememba nad entiteto, se na zaporedje dogodkov pripne nov dogodek s spremembo. Zaporedja dogodkov se hranijo v dogodkovni shrambi (*event store*), ki ima funkcionalnosti podatkovne baze, saj hrani in pridobiva dogodke za entiteto preko njenega primarnega ključa, hkrati pa opravlja delo posrednika sporočil, saj posreduje nove dogodke naročenim storitvam. Dogodkovna shramba je hrbtenica aplikacije, grajene v dogodkovno vodeni arhitekturi mikrostoritev.

Pristop dogodkovnih virov ima številne prednosti. Prinaša rešitev ključnega problema implementacije dogodkovno vodene arhitekture, saj, ko objavimo in shranimo dogodek o posodobitvi entitete, se ta samodejno posreduje vsem naročenim storitvam. Posledično s tem rešimo težave konsistentnosti podatkov v arhitekturi mikrostoritev. Dodatna prednost dogodkovnih virov je, da so vsi objavljeni dogodki sistema natančen revizijski dnevnik vseh sprememb v sistemu. V tradicionalnih sistemih je za to potrebna dodatna programska oprema za beleženje sprememb v podatkovni bazi. Zaradi shrambe dogodkov namesto domenskih objektov se tudi izognemo problemom objektno-relacijskega preslikovanja. Kot vsak arhitekturni vzorec imajo tudi dogodkovni viri svoje slabosti, predvsem gre tu za nov pristop do zasnove aplikacij, hranjenja podatkov in načina programiranja, ki večini razvijalcem ni domač in ni enostaven za obvladovanje. Zaradi omejitev lahko dogodkovna shramba direktno pridobiva dogodke le po primarnem ključu entitete, kar ne zadostuje večini aplikacij, ki zahtevajo kompleksnejše poizvedbe. Zaradi tega se pristop dogodkovnih virov pogosto uporablja v povezavi z vzorcem, imenovanim ločitev odgovornosti ukazov in poizvedb, s kratico CQRS.

Z vzorcem CQRS ločimo procesiranje poizvedb od procesiranja ukazov, ki posodabljajo entitete s poslovno logiko, implementirano po principu dogodkovnih virov. Del za poizvedbe predeluje dogodke, ki jih objavi ukazni del, in posodablja agregate dogodkov in poglede. Pogledi so posebno prilagojeni tipu poizvedb in so večinoma implementirani z uporabo podatkovnih baz NoSQL. Poleg rešitve izvajanja poizvedb pri dogodkovnih virih ima CQRS še druge prednosti. Z ločitvijo ukazov od poizvedb lahko ločeno implementiramo



Slika 2.2: Prikaz običajne arhitekture s podatkovno bazo in pristopa dogodkovnih virov z dogodkovno shrambo.

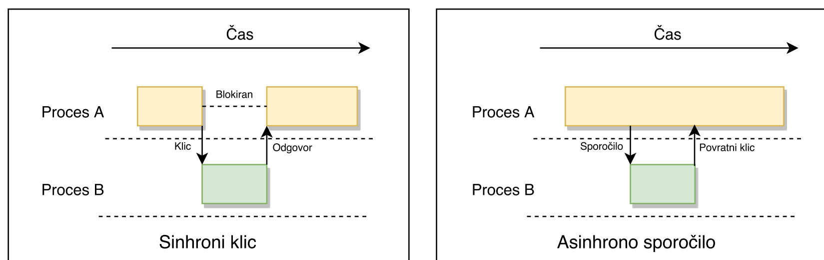
ožje specializirana modela, ki optimizirata delovanje poizvedb in ukazov. S tem lahko tudi ločeno skaliramo poizvedbeni del od ukaznega, kar je koristno v primeru aplikacij, ki so bolj bralno intenzivne. Vzorec CQRS dodatno izboljša zmogljivost in skalabilnost, saj lahko del za poizvedbe uporablja denormalizirane poglede, ki optimizirajo specifične poizvedbe [5, 20].

2.4 Primerjava interakcij mikrostoritev

2.4.1 Sinhrona in asinhrona interakcija

Pri arhitekturi mikrostoritev je običajno za interakcijo med storitvami bolje implementirati asinhrono komunikacijo. Pri sinhroni interakciji odjemalec blokira izvajanje med čakanjem na odgovor, kar vodi v počasen in neodziven sistem. Prav tako s sinhrono interakcijo vpeljemo odvisnosti med storitvami, kar je v nasprotju z načelom arhitekture mikrostoritev o šibki sklopljenosti komponent. Prednost asinhrone interakcije je izvajanje brez blokiranja izvajanja pri odjemalcu in čakanja na odgovor storitve, kar posledično vodi do šibke sklopljenosti med komponentami. Asinhrona komunikacija omogoča tudi pošiljanje več prejemnikom hkrati, za kar bi pri sinhronem komuni-

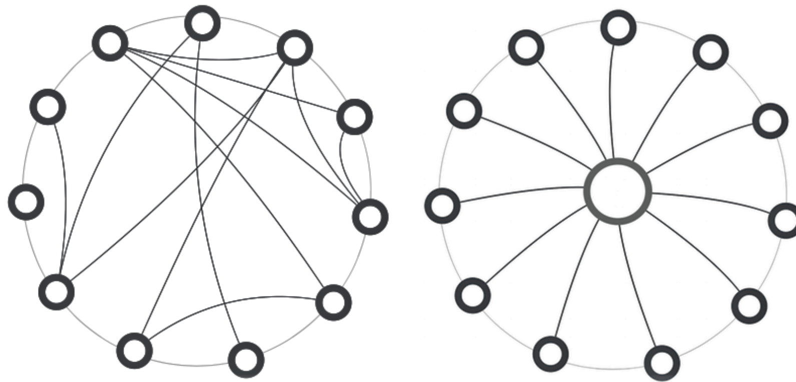
ciranju potrebovali pošiljanje posameznega sporočila vsakemu prejemniku posebej. Slika 2.3 prikazuje razliko med sinhrono in asinhrono interakcijo.



Slika 2.3: Prikaz sinhrone in asinhrone komunikacije.

Medtem ko je sinhrono interakcijo lažje implementirani, je pri asinhroni interakciji težji razvoj in razhroščevanje, saj gre za drugačen in neobičajen princip programiranja, t. i. reaktivno programiranje (*reactive programming*).

Pri arhitekturi mikrostoritev pride do problema pri logiki, ki poteka prek več ločenih storitev, kar rešujemo z interakcijo mikrostoritev. Pri tem lahko v našem sistemu uporabimo dve metodi implementacije interakcije, in sicer metodo orkestracije in koreografije. Pri metodi orkestracije imamo centralne storitve, ki komunicirajo z ostalimi storitvami, ki so ključne za izvajanje njene logike. Problem te metode je sklopljenost storitev, saj imamo eno centralno storitev, ki je pri izvajanju svoje logike odvisna od drugih. Boljša rešitev je metoda koreografije, kjer so vse storitve med seboj neodvisne in se odzivajo na dogodke v sistemu. Tu ni centralne storitve in vsaka storitev izvaja svoj del logike. Asinhrona interakcija je idealna za metodo koreografije. Storitve objavljajo dogodke in so nanje naročene. Na sliki 2.4 je prikazan graf odvisnosti pri metodi orkestracije in koreografije. Krogi predstavljajo storitve, črte med njimi pa odvisnosti. Pri metodi koreografije je prikazana interakcija prek toka dogodkov.



Slika 2.4: Graf odvisnosti pri metodi orkestracije in koreografije.

2.5 Ogrodje KumuluzEE

Ogrodje KumuluzEE je eno prvih ogrodij za razvoj mikrostoritev v tehnologiji Java EE. Je odprtokodna rešitev, ki omogoča enostavno selitev obstoječih Java EE aplikacij v arhitekturo mikrostoritev. Prednosti ogrodja so predvsem majhno pakiranje, nizki režijski stroški, hitri zagon ter prilagojenost na delovanje v vsebniški tehnologiji. Ogrodje ponuja tudi številne razširitve za implementacijo različnih rešitev v oblačni arhitekturi, kot so razširitve za konfiguracijo, beleženje dnevnikov, zbiranje metrik, odkrivanje storitev, odklopniki (*circuit-breakers*) in varnost [2].

Poglavje 3

Sporočilni in dogodkovni sistemi

Sporočilni sistemi so najpogostejši način asinhrone interakcije pri arhitekturi mikrostoritev, saj zagotavljajo šibko sklopljenost storitev v porazdeljenem sistemu. Omogočajo tudi pisanje storitev v različnih programskih jezikih ter nameščanje na različne platforme. Poleg navadnih sporočilnih sistemov se pojavljajo nove rešitve t. i. dogodkovnih sistemov oziroma platform za pretakanje dogodkov, kot je Apache Kafka, ki poleg funkcionalnosti sporočilnega sistema prinašajo še dodatne funkcionalnosti, kot so procesiranje tokov sporočil v realnem času in hramba sporočil. V nadaljnjih podpoglavjih bomo podrobneje opisali obe rešitvi ter ju med seboj primerjali.

3.1 Sporočilni sistemi in sporočilne vrste

Sporočilne vrste zagotavljajo asinhron komunikacijski protokol, pri čemer ni potrebno, da sta pošiljatelj in prejemnik med komunikacijo ves čas prisotna in povezana s sporočilno vrsto. Sporočila se pošiljajo v vrsto, kjer se hranijo, dokler jih prejemnik ne prebere; glej sliko 3.1.

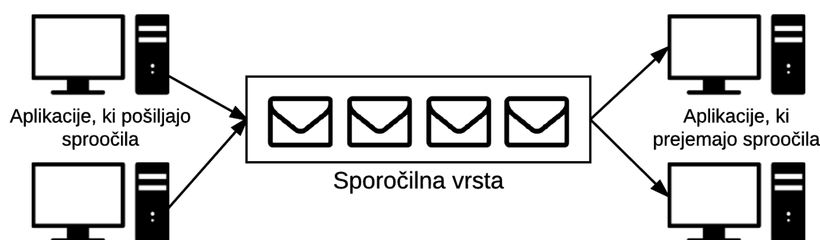
Za sporočanje in sporočilno vrsto skrbi ločen sistem, t. i. *Message-oriented Middleware* (MOM). Obstajajo številne implementacije sporočilnih

vrst v obliki licenčnih in odprtokodnih produktov ter rešitev, ki temeljijo na posebni strojni opremi. Pri odprtokodnih implementacijah sporočilne vrste so najbolj znani trije standardi:

1. AMQP (*Advanced Message Queuing Protocol*) - funkcijsko bogat protokol za sporočilne vrste,

2. STOMP (*Simple Text Oriented Messaging Protocol*) - enostaven tekstoven sporočilni protokol,

3. MQTT (*Message Queue Telemetry Transport*) - lahek protokol za sporočilne vrste, posebej namenjen vgrajenim sistemom.



Slika 3.1: Prikaz sporočilne vrste.

AMQP

AMQP je odprtokodni standardizirani binarni žični protokol. Izraz žični protokol (*wire protocol*) pomeni, da opisuje format podatkov in način, kako se ti pošiljajo preko omrežja. Protokol opredeljuje usmerjanje in orientiranje sporočil, vstavljanje v sporočilno vrsto, varnost in zanesljivost prenosa. Pri AMQP protokolu sporočilo naprej prispe v t. i. izmenjavo (*exchange*), kar si lahko predstavljamo kot nekakšen poštni nabiralnik; od tu naprej se sporočila razvrstijo v sporočilne vrste glede na določena pravila. Nato AMQP posrednik posreduje sporočila iz vrst naročnikom, ali pa naročniki sami zahtevajo sporočila. Med najbolj znanimi implementacijami protokola AMQP sodijo RabbitMQ, OpenAMQ, StormMQ, Apache Qpid in Apache ActiveMQ.

STOMP

STOMP je enostaven tekstovno orientiran sporočilni protokol (*Simple (or Streaming) Text Oriented Message Protocol*). Protokol omogoča odjemalcem STOMP komunikacijo z vsemi posredniki sporočil, ki podpirajo protokol, neodvisno od programskega jezika in implementacije. STOMP deluje preko protokola TCP in ima podobno zasnovo kot protokol HTTP. Preko protokola se pošiljajo okvirji, ki imajo ukaz, specifična polja v glavi ter opcijsko telo. Čeprav je protokol tekstovno usmerjen, omogoča tudi prenos binarnih podatkov. Znani sistemi MOM, ki podpirajo protokol STOMP, so HornetQ, RabbitMQ in Apache ActiveMQ.

MQTT

Protokol MQTT je bil zasnovan kot lahek protokol, ki temelji na vzorcu „objava/naročilo“. Njegova prednost je kompaktnost in majhni režijski stroški, zato je posebno primeren za komunikacijo z oddaljenimi lokacijami, kjer je omrežje omejeno, in napravami IoT z malo procesorske moči. Sporočilni vzorec „objava/naročilo“ zahteva posrednika sporočil, ki je odgovoren za posredovanje sporočil naročenim odjemalcem glede na temo sporočila. Sistemi MOM, ki podpirajo protokol MQTT, so IBM Websphere MQ Telemetry, Mosquitto, HiveMQ in drugi.

3.2 Platforma za pretakanje dogodkov

V zadnjih letih se je zgodil premik v arhitekturi in zasnovi sistemov, ki ne temeljijo samo na hrambi podatkov v pasivnih shrambah, temveč dajo več poudarka na pretok podatkov in realnočasovne tokove podatkov. S tem se vzpostavi pogled na poslovanje z vidika toka dogodkov. Tako ima na primer prodaja tok naročil, prodaje, prilagoditve cen in tako dalje. V takšnem pogledu podjetje sprejema vhodne tokove podatkov, ki jih predela v izhodne tokove. S takšno interpretacijo poslovanja nastane potreba po platformi, ki omogoča shranjevanje, prenašanje in procesiranje dogodkov v sistemu. S

tem smo opisali platformo za pretakanje dogodkov, ki ima dva glavna načina uporabe [13]:

1. **Procesiranje tokov**: omogoča aplikacije, ki se nenehno in v realnem času odzivajo na tokove podatkov, jih spreminjajo in procesirajo. To je naravna evolucija sporočilnih sistemov, ki se osredotočajo na dostavo posameznih sporočil. Procesiranje tokov gradi nad tem in ponuja še dodatne funkcionalnosti.

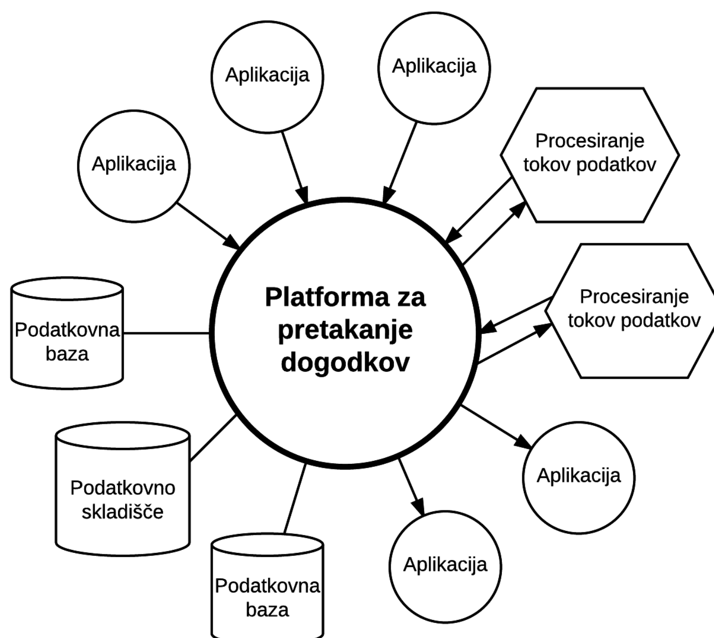
2. **Integracija podatkov**: platforma za pretakanje zajema tokove dogodkov in sprememb podatkov ter jih dovaža v druge podatkovne sisteme, kot so relacije baze, „ključ-vrednost“ shrambe ali druge podatkovne shrambe. To je pretočna različica obstoječih podatkovnih tehnologij, kot so sistemi ETL.

Platforma za pretakanje deluje kot centralna enota za tokove podatkov, kar prikazuje slika 3.2. Aplikacijam, ki prejemajo podatkovne toke, se ni potrebno ukvarjati z izvorom toka, prav tako proizvajalci podatkov niso obremenjeni z poznavanjem sistemov, ki bodo podatke brali in procesirali. S platformo za pretakanje dogodkov tako v sistem vpeljemo šibko sklopljenost med posameznimi proizvajalci ter potrošniki podatkovnih tokov.

Ključne zmogljivosti in spodobnosti pretočne platforme so [13, 1]:

- **Objava / naročanje na tokove dogodkov**: Platforma omogoča objavo in naročanje na tokove dogodkov, podobno kot delujejo sporočilne vrste ali sporočilni sistemi. Pri tem mora zagotavljati nizko zakasnitev pri pošiljanju, tako da je uporabna za realnočasovne aplikacije. Zdržati mora velike obremenitve in obdelati velike količine podatkov, ki so značilni za dnevniške zapise ali podatke IoT. Poleg tega mora zagotavljati tudi integracijo z drugimi sistemi in aplikacijami, kot so pasivne shrambe podatkov (podatkovne baze) ipd.

- **Procesiranje tokov dogodkov**: Omogoča procesiranje tokov podatkov v realnem času in izvajati manipulacijo nad podatkovnimi tokovi.



Slika 3.2: Prikaz platforme za pretakanje dogodkov.

- **Shramba tokov dogodkov**: Omogoča shranjevanje tokov sporočil na način, ki je odporen na napake in izpade. Pri tem mora zagotoviti podvajanje in zaporedje podatkov, tako da je primerna za uporabo v kritičnih primerih, kot je prenašanje sprememb iz podatkovnih baz, brez izgub podatkov in v pravilnem vrstnem redu. Omogoča naj tudi hrambo podatkovnih tokov za daljši čas, kar dovoljuje večkratno procesiranje starih podatkov ter integracijo z sistemi za paketno obdelavo podatkov (npr. podatkovnimi shrambami), ki podatke predelujejo periodično in ne v realnem času.

3.2.1 Tehnologija za prenašanje dogodkov

Platforma za pretakanje dogodkov temelji na tehnologiji za prenašanje sporočil, ki mora imeti nekatere ključne značilnosti, da lahko celotna platforma za pretakanje dogodkov deluje. Značilnosti, ki so bistvene za tehnologijo za sporočanje [7], so naslednje:

- Neodvisnost med proizvajalci sporočil in potrošniki; ni potrebno poznati potrošnikov, ki bodo brali in procesirali ta sporočila.

- Obstojnost (hramba) sporočil; sporočila se hranijo v temah, tako lahko zagotovimo izolacijo med proizvajalcem in potrošnikom, saj sporočil ni potrebno prebrati takoj ko so poslana.

- Velika frekvenca sporočil na sekundo; sistem za sporočanje mora za potrebe modernih pretočnih sistemov prenesti ogromno število sporočil.

- Poimenovane teme; tako lahko potrošniki izberejo podatkovne toke, ki jih potrebujejo.

- Zaporedje toka dogodkov, ki ga lahko poljubno večkrat preberemo, hkrati pa ohranja strog vrstni red. Potrošniki lahko tako določijo točko, od katere želijo brati tok podatkov, ali pa celoten tok ponovno preberejo. Proizvajalci lahko zaporedno proizvajajo sporočila z zagotovljenim vrstim redom, kar omogoča logično odvisnost med posameznimi sporočili.

- Odpornost na napake in izpade; kot vsak kritičen sistem mora zagotavljati konstantno delovanje, brez napak in izpadov.

- Lokacijska porazdeljenost (*Geo-distributed replication*), kar omogoča arhitekturo, ki deluje na več podatkovnih centrih na različnih lokacijah.

Sistem za sporočanje z zgornjimi značilnostmi najdemo v tehnologijah, kot so Apache Kafka, ki je podrobneje opisana v poglavju 4, in MapR Streams, ki uporablja Kafka API.

3.2.2 Procesiranje tokov

Procesiranje tokov je neprekinjena, istočasna in zaporedna obdelava in analiza podatkov v realnem času. Pri procesiranju tokov si podatke predstavljamo kot neskončen tok, ki ga obdelujemo podatek za podatkom, ali preko

nekega časovnega okna, drugače kot s paketno obdelavo podatkov, kot je Hadoopov MapReduce, kjer se obdelujejo podatki v večjih paketih in pride pri tem do večjih zakasnitev.

Za dosego učinkovitosti pri procesiranju mora sistem za procesiranje tokov slediti osmim zahtevam, določenih v članku [22]:

- Procesiranje v toku – za zagotovitev nizkih zakasnitev mora sistem procesirati podatke v toku, brez zamudnih operacij shranjevanja v podatkovne baze ali na disk.

- Poizvedbe z visokonivojskim jezikom – sistem za procesiranje tokov naj podpira poizvedbe nad podatki z visokonivojskim jezikom (npr. StreamsSQL).

- Upravljanje z težavami tokov (zamude, manjkajoči podatki in napačen vrstni red) – sistem naj ima mehanizme za upravljanje z težavami podatkov v realnem času.

- Proizvajanje predvidenega rezultata – rezultat operacij procesiranja tokov mora biti predvidljiv in ponovljiv.

- Integracija shranjevanja tokov podatkov – sistem mora zagotoviti učinkovito hrambo, dostop in upravljanje s stanjem podatkov, ki ga lahko primerja in kombinira s tokom podatkov v realnem času. Tako lahko med procesiranjem primerja nove podatke s starimi shranjenimi stanji.

- Zagotovitev varnosti in dostopnosti podatkov – sistem za procesiranje tokov mora zagotoviti odpornost na izpade in ponujati mehanizme za stalno razpoložljivost.

- Avtomatsko skaliranje in porazdelitev – sistem mora biti zmožen porazdeljenega procesiranja preko več procesov in s tem zagotoviti skalabilnost.

- Procesiranje in odgovarjanje z minimalno zakasnitvijo – sistem mora imeti nizke režijske stroške in proizvajati rezultate v realnem času oziroma z nizko zakasnitvijo tudi pri velikih obremenitvah.

Za procesiranje tokov podatkov je na voljo vrsta tehnologij, od odprtokodnih ogrodij, kot so Apache Spark Streaming, Apache Storm, Apache Flink, in Apache Apex, do lastniških produktov, kot sta Google DataFlow in AWS Lambda. Z verzijo 0.10 nudi tudi Apache Kafka implementacijo za procesiranje tokov imenovano Kafka Streams. Te tehnologije omogočajo aplikacijam realnočasovno (oziroma z zelo majhno zakasnitvijo) obdelavo in transformacijo podatkovnih tokov.

3.3 Primerjava

Sporočilni in dogodkovni sistemi imajo nekaj skupnih lastnosti, vendar se razlikujejo v zasnovi in upravljanju s sporočili oziroma dogodki ter v namenu uporabe. Sporočilne sisteme uporabljamo, ko želimo, da storitev sporoči drugi storitvi naj nekaj naredi ali vrne, pri dogodkovnih sistemih pa storitve objavljajo dogodke, ko se nekaj zgodi, druge storitve pa so lahko na te dogodke naročene. Ko upoštevamo te scenarije uporabe v tipični aplikaciji, hitro ugotovimo, da gre pri primeru objave dogodka, ko se nekaj zgodi, za večjo količino podatkov, ki jo morajo dogodkovni sistemi prenesti. Zasnova dnevnika dogodkov, na katerega se samo pripenjajo novi dogodki, je za takšen sistem bolj smiselna in učinkovita, kot so sporočilne vrste pri sporočilnih sistemih.

Platforme za pretakanje dogodkov nudijo večjo pretočnost in manjše zakasnitve kot sporočilne vrste in lahko tako obdelujejo zaporedje sporočil kot konstanten tok podatkov. Platforme za pretakanje so grajene tako, da vzdržijo večje količine podatkov in veliko število potrošnikov sporočil, pri tem pa ohranjajo nizke režijske stroške. Glavni razlog za večji pretok je, da se v platformi za pretakanje ne potrjuje pošiljanja oz. sprejemanja vsakega posameznega sporočila, ampak poteka pošiljanje/sprejemanje v paketih sporočil.

Običajno sporočilne vrste hranijo sporočila dokler jih vsi naročniki ne preberejo, pri platformah za pretakanje pa se sporočila avtomatsko hranijo za daljši čas brez posledic za zmogljivost, tudi pri gigabajt ali več sporočilih na sekundo na posameznem strežniku. S hranjenjem sporočil pridobimo vrsto prednosti, kot so možnost branja starejših sporočil, lažja obdelava in procesiranje toka sporočil. Platforma za pretakanje nudi več kot le sistem za prenos sporočil, v sklopu platforme so tudi orodja za procesiranje, analizo in predelavo tokov podatkov. Na tak način lahko podatke obdelamo in uporabimo v realnem času, kar je v sodobnem svetu mobilnih in IoT naprav bistvenega pomena. Platforme za pretakanje, kot je Apache Kafka, so v zasnovi porazdeljeni sistemi, ki omogočajo lažje skaliranje in nudijo večjo odpornost na izpade kot sporočilni sistemi.

Poglavje 4

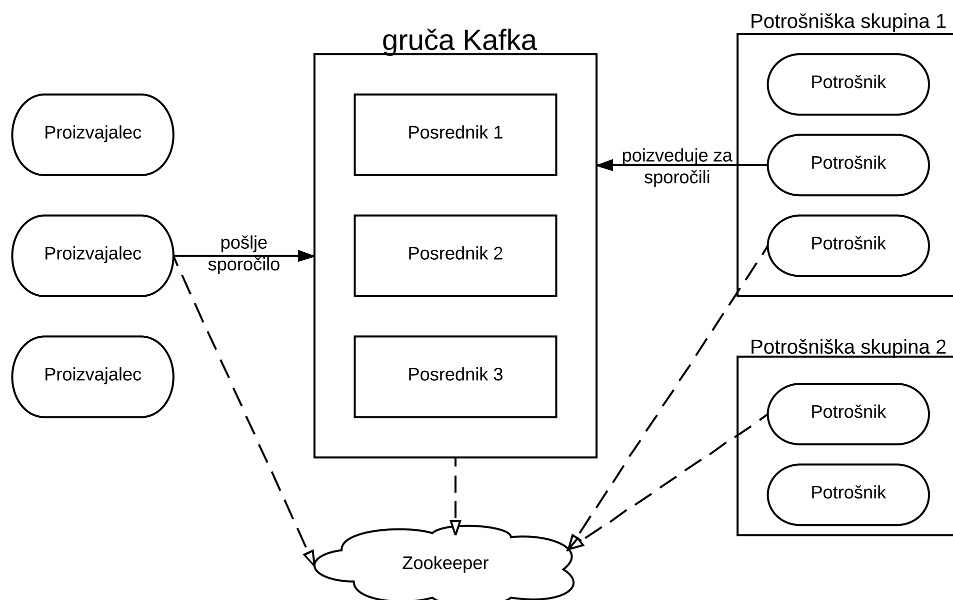
Platforma Apache Kafka

Apache Kafka je odprtokodna porazdeljena pretočna platforma (*streaming platform*). Razvita je bila pri podjetju LinkedIn, prva odprtokodna izdaja je iz začetka leta 2011. Kasneje je razvoj prevzela odprtokodna fundacija *Apache Software Fundation*. Platforma Apache Kafka stremi k visoki pretočnosti in hitrosti, nizki latenci ter odpornosti na izpade. Kafko lahko uporabimo za sporočanje kot nadomestilo bolj tradicionalnim sporočilnim sistemom, drugi primeri uporabe so procesiranje tokov podatkov, sledenje aktivnostim na spletnih straneh, zbiranje in spremljanje meritev in agregacija dnevnikov (*log aggregation*) in njihovo shranjevanje v centralno shrambo, kot je na primer Hadoopov porazdeljen datotečni sistem (HDFS). Prav tako lahko Kafko uporabljamo za dovoz tokov podatkov v druga realnočasovna ogrodja za obdelavo in agregiranje podatkovnih tokov, kot so Spark Streaming, Flume in Flink.

4.1 Pregled arhitekture

Kafka je porazdeljen sistem, torej deluje kot gruča (*cluster*) posrednikov na enemu ali več strežnikih, glej sliko 4.1.

Kafka posrednik hrani tokove sporočil v kategorijah imenovanih teme (*topics*). Za vsako temo Kafka upravlja s porazdeljenim dnevnikom (*log*) parti-



Slika 4.1: Primer arhitekture gruče Kafka

cij. Kafka odjemalci so uporabniki sistema, na voljo so štiri API-ji: osnovna dva za proizvajanje sporočil *Producer API* in branje *Consumer API* ter napredna *Connect API* za integracijo v podatkovne baze ali shrambe in *Streams API* za procesiranje tokov podatkov. Napredna odjemalca uporabljata proizvajalce in potrošnike kot gradnike in ponujata višjenivojske funkcionalnosti. Proizvajalci in potrošniki v Kafki delujejo na principu potisni/povleci (*push/pull*). To pomeni, da proizvajalci potiskajo sporočila posrednikom, potrošniki pa izprašujejo posrednika za nova sporočila. Kafka posrednik ne ohranja stanja (*stateless*), ne spremlja katere ali koliko sporočil so potrošniki prebrali, kar je naloga posameznega potrošnika. Takšna zasnova manjša kompleksnost in režijske stroške posrednika. Kafka je porazdeljena platforma, kar zahteva dodatno koordinacijo. Pri zasnovi platforme se razvijalci niso odločili za centralizirano glavno vozlišče, saj bi tako sistem dodatno zakomplicirali ter ustvarili enotno točko odpovedi. Za rešitev koordinacije uporablja Kafka storitev Zookeeper, ki je centralizirana storitev za vzdrževanje konfiguracij,

porazdeljeno sinhronizacijo, imenske storitve (*naming services*) in upravljanje skupin (*group membership*).

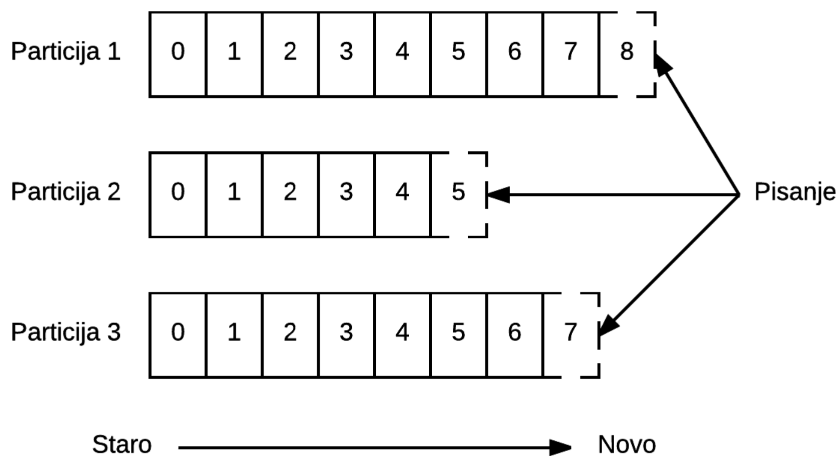
Posebnost Kafke je avtomatsko hranjenje vseh sporočil, pri čemer se za shranjevanje in predpomnjenje sporočil zanaša na datotečni sistem. Z uporabo linearnega branja in pisanja z diska, ki je precej optimizirano s strani operacijskega sistema, dosega Kafka učinkovito shranjevanje in branje podatkov na disk. Pri tem si pomaga z predpomnjenjem operacijskega sistema, t. i. *pagecache*. Takšna zasnova je boljša od upravljanja predpomnjenja znotraj procesa *in-memory* ali kakšne druge strukture, saj ima manj režijskih stroškov. Prav tako se, v nasprotju z predpomnjenjem znotraj procesa, ki bi zahtevalo obnovitev v pomnilniku, v primeru ponovnega zagona *pagecache* ohrani. Dodatno je Kafka za doseganje hitrejšega shranjevanja in dostopa do podatkov zasnovan tako, da minimizira število vhodno-izhodnih operacij in kopiranj podatkov. To v operacijskem sistemu Linux doseže z uporabo sistemskega klica *sendfile*, ki optimizira prenos podatkov iz *pagecache* v vtičnico [1].

Komunikacija med odjemalci in strežniki je v Kafki implementirana z enostavnim in visokozmogljivim protokolom preko TCP. Protokol ni odvisen od programskega jezika, tako da obstajajo odjemalci v različnih programskih jezikih. Fundacija Apache vodi razvoj odjemalca v programskem jeziku Java, na voljo pa so tudi številni odprtokodni odjemalci v več drugih programskih jezikih, kot so C, C++, Python, .Net in drugi [18].

4.1.1 Sporočila v Kafki

Sporočilo v Kafki je zgolj tabela bajtov, podatki v njem nimajo posebnega formata ali pomena za Kafko. Sporočilo ima lahko dodatno polje imenovano ključ, ki se uporablja za določanje particije za sporočilo. Najenostavnejši način določanja particije je, da se najprej izračuna *hash* vrednost ključa, nato pa se po modulu števila particij v temi izračuna številka particije. Na tak način zagotovimo, da so vsa sporočila z enakim ključem zapisana na isto particijo. Zaradi učinkovitosti se sporočila v Kafki prenašajo v paketih

(*batch*). Paket je zbirka sporočil, proizvedenih za določeno temo in particijo, ki lahko vsebuje eno ali več sporočil. S pošiljanjem paketov se izognemo prekomernih režijskih stroškov (*overhead*), ki bi nastali pri pošiljanju posameznih sporočil preko omrežja. Pri tem gre za kompromis med zakasnitvijo in prepustnostjo, večji kot so paketi, več sporočil lahko predelamo na časovno enoto, vendar prenos posameznega sporočila traja več časa. Velikost paketov sporočil in čas čakanja na dodatna sporočila lahko nastavimo v konfiguraciji posameznega proizvajalca. Navadno so paketi sporočil tudi stisnjeni (*compressed*), kar v zameno za nekaj procesorske moči doprinese k učinkovitosti prenosa in hrambe podatkov.



Slika 4.2: Prikaz arhitekture Kafka teme

4.2 Teme in particije

Sporočila v Kafki se hranijo v temah, to so kategorije ali imenovani vir (*feed name*), podobno kot datoteke v datotečnem sistemu. Teme v Kafki lahko imajo nič, enega ali več potrošnikov, ki je naročen na temo in iz nje bere sporočila. Teme so dodatno razčlenjene na eno ali več particij. Za vsako temo se vzdržuje porazdeljen dnevnik particij, glej sliko 4.2. Particija je kot strukturiran dnevnik, urejeno in nespremenljivo zaporedje sporočil, kateremu

zaporedno pripenjamo nova sporočila. Vsako sporočilo v particiji ima dodeljeno zaporedno identifikacijsko število imenovano odmik (*offset*), ki enolično definira sporočilo v particiji. Glede na to, da ima lahko tema več particij, ni zagotovila, da bodo sporočila brana iz nje v pravilnem časovnem vrstnem redu, to zagotovilo velja le za posamezno particijo. Z uporabo particij Kafka omogoča razširljivost (*scalability*) in redundanco. Vsaka particija je lahko podvojena ali hranjena na različnih strežnikih, kar nudi večjo odpornost na izpade ter boljšo zmogljivost, saj so teme tako horizontalno skalirane in presegajo zmogljivost posameznega strežnika. Vsaka particija ima en strežnik, ki igra vlogo „vodje“ in nič ali več strežnikov, ki igrajo vlogo „sledilca“. Vodja izvaja vse zahteve za branje in pisanje na particijo, sledilci pa skrbijo za pasivno podvajanje particij vodje.

4.3 Kafka posredniki in gruče

Posamezen Kafka strežnik je imenovan posrednik (*broker*). Njegov namen je sprejemanje sporočil od proizvajalcev, določanje odmika ter shranjevanje sporočil na disk. Prav tako se odziva na zahteve potrošnikov sporočil in vrača sporočila, shranjena na disku. Posamezen posrednik lahko, glede na strojno opremo, upravlja s tisočimi particijami ter miljoni sporočil na sekundo [15].

Posredniki delujejo v sklopu gruče (*cluster*). V vsaki gruči je avtomatično izvoljen upravljavec (*controller*) gruče, ki je odgovoren za administrativne operacije, kot so spremljanje izpadov posrednikov in določanje particij posameznim posrednikom. Particija je v lasti enega posrednika, ki je imenovan vodja. Za redundanco je lahko particija določena večim posrednikom, te imenujemo sledilci, ki podvajajo particijo vodje. V primeru izpada vodje se enega izmed sledilcev avtomatično določi za novega vodjo. Za namen uravnoteženja obremenitve v gruči je posamezen posrednik v vlogi vodje nekaterih svojih particij in sledilec drugih.

Kafka ohranja vsa proizvedena sporočila glede na nastavljeno vrednost obdobja hranjenja (*retention period*). Posredniki imajo nastavljeno vrednost

hranjenja za teme, ki je lahko nastavljena kot časovna omejitev (na primer 7 dni) ali pa mejna velikost teme v bajtih (na primer 1 GB). V primeru, da je meja presežena, se sporočila izbrišejo. Obdobje hranjenja lahko nastavimo tudi ločeno za posamezno temo in tako določimo hrambo sporočil glede na njihovo uporabnost. Dodatno lahko nastavimo temo tako, da ohranja samo določena sporočila z specifično vrednostjo ključa.

V primeru uporabe Kafke v večjem sistemu nastane potreba po uporabi večjega števila gruč. Glavne prednosti so segregacija tipov podatkov, izolacija zaradi varnostnih zahtev ter večje število podatkovnih centrov. Mehanizmi replikacije v Kafka gruči delujejo zgolj v posamezni gruči in ne med več gručami. V ta namen nudi platforma orodje *MirrorMaker*, ki je sestavljeno iz proizvajalca in potrošnika, ki sta med seboj povezana z vrsto. Orodje bere sporočila iz ene gruče in jih proizvaja za drugo gručo.

4.4 Kafka proizvajalci

Glavna naloga proizvajalca je, da proizvaja sporočila v teme. Proizvajalec pošlje sporočilo neposredno tistemu posredniku, ki je vodja particije, na katero želi shraniti sporočilo. V pomoč proizvajalcem vsa Kafka vozlišča odgovarjajo na poizvedbe z metapodatki, ki vsebujejo informacije o tem, kateri posredniki so aktivni ter naslove vodij particij posameznih tem. Ko posrednik prejme sporočilo, vrne odgovor. V primeru uspešno prejetega sporočila vrne podatke o temi, particiji in odmiku sporočila znotraj particije, če pa je bilo prejemanje ali shranjevanje sporočila neuspešno, posrednik vrne napako. Ko proizvajalec prejme napako, lahko, odvisno od njegovih nastavitev, pošiljanje večkrat ponovi preden obupa in vrne napako o neuspešnem pošiljanju sporočila. Kafka proizvajalec ima tri glavne načine pošiljanja sporočil:

- *Pošlji in pozabi* (*Send and forget*): sporočilo pošljemo, odgovor posrednika o uspehu pošiljanja pa nas ne zanima. Ker je Kafka visokodosegljiva in proizvajalci avtomatsko ponovijo neuspešna pošiljanja,

je velika verjetnost, da bo sporočilo uspešno prispelo, vendar se kljub temu nekatera sporočila pri tem načinu izgubijo.

- *Sinhrono pošiljanje*: ko pošljemo sporočilo, čakamo, dokler ne prejmemo odgovora posrednika, ali je bilo pošiljanje uspešno ali ne.

- *Asinhrono pošiljanje*: s pomočjo funkcije povratnega klica (*function*) prejmemo odgovor posrednika, ne da bi nanj čakali in ustavili delovanje procesa.

Proizvajalec sam skrbi za to, na katere particije pošilja sporočila. Privzeto sporočila enakomerno in naključno porazdeli po vseh particijah, lahko pa tudi določimo funkcijo za delitev sporočil (*partitioning function*) na določene particije, kar se običajno naredi z vrednostjo ključa sporočila. Za doseganje visoke učinkovitosti skušajo potrošniki shranjevati podatke v spominu ter hkrati pošiljati več sporočil v paketu. Pošiljanje v paketu je nastavljivo in ga lahko po želji izključimo. Določimo lahko maksimalno število sporočil in maksimalen čas čakanja na sporočila, preden se paket odpošlje .

4.5 Kafka potrošniki

Kafka potrošnik je lahko naročen na več tem in iz njih bere sporočila. Branje poteka tako, da potrošnik najprej pošlje zahtevo za pridobitev sporočil na posrednika, ki je vodja particije, iz katere želi brati. V zahtevi poda tudi odmik v particiji in tako prejme paket sporočil, ki se nahajajo od tistega mesta dalje. Kafka potrošniki običajno delujejo v sklopu potrošniške skupine (*consumer group*). V primeru, ko je več potrošnikov naročenih na isto temo in spadajo v isto potrošniško skupino, bo vsak potrošnik prejemal sporočila iz različne podmnožice particij teme. Na tak način lahko več potrošnikov hkrati bere iz teme in si porazdeli sporočila med seboj. Potrošnik lahko pri tem uporablja več niti za izvajanje, ki so lahko tudi porazdeljene po več procesih. V primeru, da so vsi potrošniki v različnih potrošniških skupinah, pa bo vsak prejel vsa sporočila iz teme. Z možnostjo hkratnega branja sporočil

z več potrošniki v potrošniški skupini omogočamo večjo propustnost, skalabilnost in odpornost na izpade. V primeru, ko se nov potrošnik pridruži skupini, prevzame nekaj particij od drugih članov skupine. Ko potrošnik iz skupine preneha delovati, se njegove particije porazdelijo drugim članom. Načeloma ni smiselno imeti več potrošnikov v skupini kot je particij v temi, saj so takrat potrošniki brez particij nedejavni. Kafka zagotavlja vrstni red sporočil le znotraj posamezne particije in ne med več particijami v temi, kar v kombinaciji z možnostjo izbiranja particije pri proizvajanju sporočil zadošča večini aplikacij. Potrošniku se odmik naslednjega sporočila znotraj particije pri branju določi avtomatsko, tako da ni možnosti, da bi sporočila potrjeval v napačnem vrstnem redu. Odmik se lahko eksplicitno nastavi tudi na določeno sporočilo, na prvo sporočilo, ki ga vsebuje posrednik, ali pa na konec zadnjega sporočila. Tudi pri tako nastavljenemu odmiku se naslednja sporočila berejo v pravilnem vrstnem redu, dokler odmika ne eksplicitno ponovno nastavimo. V primeru, da želimo ohraniti pravilni vrstni red znotraj celotne teme, mora ta vsebovati samo eno particijo, vendar s tem zgubimo možnost hkratnega branja sporočil od več potrošnikov [15, 7].

4.6 Procesiranje tokov s Kafko

Od verzije 0.10 naprej ponuja Apache Kafka tudi Kafka Streams API. To je enostavna javanska knjižnica za grajenje skalabilnih, porazdeljenih in na izpade odpornih aplikacij za procesiranje in analiziranje tokov podatkov, shranjenih v Kafki. Knjižnica deluje v odjemalcu in razen Kafke nima drugih sistemskih odvisnosti, tako da jo lahko vključimo v katerokoli javansko aplikacijo, ki je lahko pakirana, postavljena in kontrolirana kot navadna javanska aplikacija.

Procesiranje tokov s Kafka Streams zagotavlja, da je vsak podatek procesiran samo enkrat tudi v primeru napake v odjemalcu ali na posredniku med procesiranjem. Knjižnica omogoča tudi, da se med procesiranjem podatkov stanje ohranja. Tako lahko izvajamo kompleksnejše operacije nad podatki,

kot so združitev, grupiranje in agregiranje. Za hranjenje stanja ponuja Kafka Streams lokalno shrambo stanja (*state store*), ki jo lahko uporabljamo v aplikaciji za procesiranje tokov za hranjenje podatkov. Shramba je lahko implementirana kot obstojna „ključ-vrednost“ shramba ali podatkovna struktura v spominu. Posebnost shrambe stanja je njena porazdeljenost. Vse aplikacije, ki vključujejo Kafka Streams, hranijo podmnožico stanja aplikacije kot del ali particijo shrambe stanja. Shramba stanja je odporna na izpade, saj shrani vse lokalne spremembe v visokodostopno in obstojno temo v Kafki. Tako lahko v primeru izpada storitve in izgube njenega lokalnega dela shrambe stanja Kafka Streams ponovno ustvari del shrambe stanja tako, da bere iz teme v Kafki in napolni shrambo stanja. Prednost uporabe Kafka Streams je tudi uravnoteženje obremenitev, saj ob primeru izpada ali nove storitve porazdeli particije shrambe stanja med vse storitve.

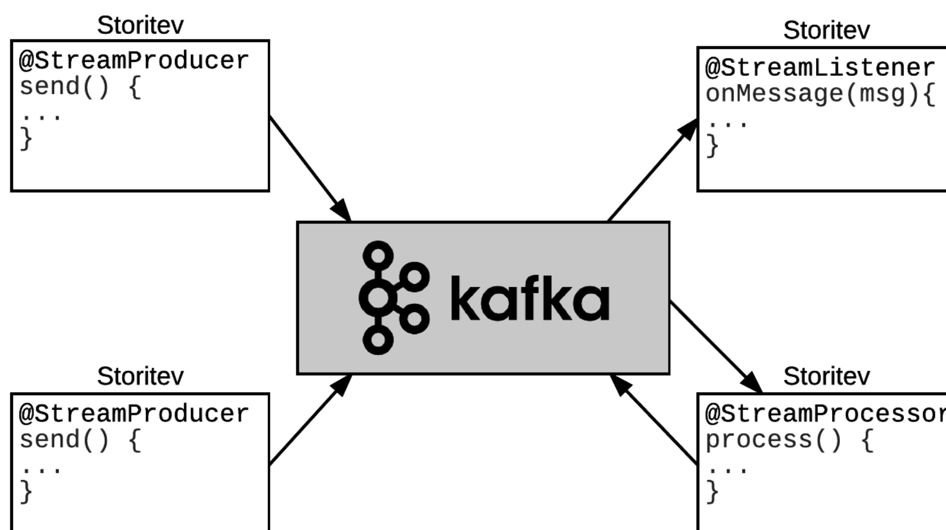
Poglavje 5

Integracija Apache Kafke in mikrostoritev

V sklopu diplomske naloge smo se odločili, da bomo razvili razširitev ogrodja KumuluzEE za implementacijo pretakanja dogodkov s platformo Apache Kafka. Cilj razširitve je enostavna implementacija Kafke v ogrodje, ki razvijalcem olajša delo. Kot prikaz uporabe razširitve smo pripravili primer aplikacije za spletno knjigarno, ki je razvita v arhitekturi mikrostoritev po vzorcih dogodkovnih virov in CQRS.

5.1 Razširitev KumuluzEE Event Streaming

Razširitev KumuluzEE Event Streaming omogoča enostavno uporabo platforme za pretakanje dogodkov v aplikacijo, grajeno v odprtokodnem ogrodju KumuluzEE. Cilj je bil razviti razširitev, ki bo omogočala enostavno pošiljanje in prejemanje sporočil iz platforme Kafka ter nudila enostavno uporabo Kafka Streams API, ki omogoča procesiranje toka sporočil. Razširitev smo zasnovali tako, da je v prihodnosti možno dodajanje dodatnih implementacij drugih platform za pretakanje; je odprtokodna in prosto dostopna na repozitoriju Github [3]. Na sliki 5.1 je prikazano njeno delovanje.



Slika 5.1: Prikaz delovanja razširitve KumuluzEE Event Streaming za pisanje, branje in procesiranja sporočil s platforme Apache Kafka.

Razširitev nudi anotacije programskega jezika Java za proizvajanje, prejemanje in procesiranje sporočil iz platforme Apache Kafka. Za implementacijo anotacij smo uporabili prenosno razširitev CDI, ki omogoča priključitev procesu skeniranja CDI, ki se zgodi ob zagonu. S tem lahko spremenimo ali dodamo metapodatke, ki jih je ustvaril vsebnik CDI. Za kreiranje potrošnika Kafka in procesorja tokov smo ustvarili razširitev CDI, v kateri smo ob zagonu aplikacije naročeni na dogodke CDI (*ProcessAnnotatedType*, *AfterTypeDiscovery*, *ProcessBean* in *AfterDeploymentValidation*). V razširitvah CDI pridobimo parametre iz anotacij, preberemo konfiguracije iz konfiguracijskih datotek, ustvarimo potrošnik Kafka oziroma procesor toka in ga vstavimo v kodo.

V sklopu ogrodja KumuluzEE lahko s pomočjo razširitve KumuluzEE Config enostavno nalagamo konfiguracijo za proizvajalce, potrošnike in procesorje sporočil iz spremenljivk okolja, konfiguracijskih datotek ali konfiguracijskih strežnikov, kot so Consul ali etcd. Kot primer konfiguracijske datoteke s konfiguracijo za proizvajalca in potrošnika glej izsek programske kode 5.1.

Dodatno lahko v konfiguraciji nastavimo tudi vrednost parametra *timeout* metode `poll()` potrošnika Kafka, ki specificira čas čakanja na nova sporočila v milisekundah.

```
1  kumuluzee:
2    streaming:
3      kafka:
4        producer:
5          bootstrap-servers: localhost:9092
6          batch-size: 16384
7          buffer-memory: 33554432
8          key-serializer: org.apache.kafka.common.serialization.
               StringSerializer
9          value-serializer: org.apache.kafka.common.serialization.
               StringSerializer
10       consumer:
11         bootstrap-servers: localhost:9092
12         group-id: group1
13         enable-auto-commit: true
14         auto-commit-interval-ms: 1000
15         key-deserializer: org.apache.kafka.common.serialization.
               StringDeserializer
16         value-deserializer: org.apache.kafka.common.
               serialization.StringDeserializer
```

Izsek programske kode 5.1: Primer konfiguracije Kafka proizvajalca in potrošnika iz konfiguracijske datoteke yaml.

5.1.1 Anotacija za proizvajanje sporočil

Za proizvajanje sporočil je na voljo anotacija `StreamProducer`, ki jo moramo navesti v povezavi z anotacijo `Inject`. S pomočjo anotacije `StreamProducer` v kodo vstavimo referenco proizvajalca Kafka, kot je to prikazano v izseku programske kode 5.2. Anotacija ima parameter `config`, s katerim podamo oznako konfiguracije proizvajalca, privzeta vrednost je „producer“. S pomočjo anotacije se proizvajalec Kafka s podano konfiguracijo generira v

ozadju in vstavi v kodo, kjer ga lahko normalno uporabljamo za proizvajanje sporočil, kar je tudi prikazano v izseku kode 5.2.

```
1  @Inject
2  @StreamProducer(config = "customProducer")
3  private Producer producer;
4
5  public void produceMessage(ProducerRecord<String, String> msg) {
6   producer.send(msg, (metadata, e) -> {
7    if (e != null)
8     e.printStackTrace();
9    else
10    log.info("Odmik poslanega sporocila: " + metadata.offset());
11  });
12 }
```

Izsek programske kode 5.2: Primer uporabe anotacije *StreamProducer* in pošiljanja sporočila.

5.1.2 Anotacija za prejemanje sporočil

Za prejemanje Kafka Sporočil je na voljo anotacija `StreamListener`, s katero anotiramo metodo, ki se bo izvedla ob prejemu novega sporočila ali množice novih sporočil. Prikaz uporabe anotacije najdemo v izseku programske kode 5.3. V ozadju se v ločeni niti ustvari Kafka potrošnik, ki bere in povprašuje za nova sporočila, in ob novem sporočilu kliče anotirano metodo. Parameter anotirane metode je objekt iz Kafkine knjižnice Clients `ConsumerRecord` ali `List<ConsumerRecord>`, če prejemamo množico sporočil.

Anotacija ima tri parametre:

- `topics` – tabela imen tem, na katere je potrošnik naročen. V primeru, da parameter ni podan, se za ime teme vzame ime anotirane metode.

- `config` – oznaka konfiguracije potrošnika, privzeta vrednost je „consumer“.

- `batchListener` – logična vrednost, ki omogoča prejemanje množice sporočil v metodo namesto posameznih sporočil.

```
1 @StreamListener(topics = {"topic1"}, config = "customConsumer")
2 public void onMessage(ConsumerRecord<String, String> record) {
3  // predelaj prejeto sporocilo
4 }
```

Izsek programske kode 5.3: Primer uporabe anotacije *StreamListener*.

Anotacija omogoča tudi ročno potrjevanje sporočil, kar zahteva pravilno konfiguracijo potrošnika s poljem *enable.auto.commit* nastavljenim na *false* ter dodaten parameter v anotirani metodi, objekt `Acknowledgement`, ki ima dve metodi za potrjevanje sporočil:

- `acknowledge()`, ki potrdi odmik zadnjega prejetega sporočila za vse naročene teme in particije, in
- `acknowledge(Map<TopicPartition,OffsetAndMetadata> offsets)`, ki potrdi določen odmik za podan seznam tem in particij.

Primer uporabe anotacije `StreamListener` z ročnim potrjevanjem sporočil je v izseku kode 5.4.

```
1 @StreamListener(topics = {"topic"})
2 public void onMessage(ConsumerRecord<String, String> record,
     Acknowledgement ack) {
3  // predelaj prejeto sporocilo
4  ack.acknowledge(); // potrdi prejeto sporocilo
5 }
```

Izsek programske kode 5.4: Primer uporabe anotacije *StreamListener* z ročnim potrjevanjem sporočil.

5.1.3 Anotacija za procesiranje tokov

Razširitev KumuluzEE Event Streaming nudi tudi anotacije za enostavno uporabo Kafka Streams API, ki omogoča procesiranje tokov sporočil. Z ano-

tacijo `StreamProcessor` anotiramo metodo, ki ustvari in vrne graditelja procesorja toka sporočil `KStreamBuilder` ali `TopologyBuilder`, za primer glej izsek programske kode 5.5. V ozadju se iz podanega graditelja in konfiguracije ustvari procesor toka `KafkaStreams`. Anotacija ima tri parametre:

- `id` – enolična identifikacija, s katero se lahko kasneje sklicujemo na proizveden procesor toka.

- `config` – oznaka konfiguracije procesorja toka, privzeta vrednost je „streams“.

- `autoStart` – logična vrednost, ki omogoča samodejen zagon procesorja ob zagonu aplikacije, privzet je samodejen zagon.

```
1 @StreamProcessor(id = "processor", autoStart = false)
2 public KStreamBuilder streamProcessorBuilder() {
3  KStreamBuilder builder = new KStreamBuilder();
4
5  // izvedba procesiranja tokov
6
7  return builder;
8 }
```

Izsek programske kode 5.5: Primer uporabe anotacije *StreamListener*.

Do procesorja toka lahko dostopamo z uporabo anotacije `StreamProcessorController`, s katero anotiramo polje `StreamsController`. Anotaciji moramo podati obvezen parameter `id`, s katerim izberemo procesorja toka, ki smo ga predhodno ustvarili z anotacijo `StreamProcessor`. Primer uporabe anotacije je v izseku programske kode 5.6.

```
1 @StreamProcessorController(id = "processor")
2 private StreamsController streams;
3
4 public static void main(String[] args) {
5  streams.start();
```

```
6 }
```

Izsek programske kode 5.6: Primer uporabe anotacije *StreamListener* za dostop do kreiranega procesorja toka in zagon procesiranja.

5.2 Implementacija vzorcev dogodkovnih virov in CQRS s pomočjo razširitve KumuluzEE Event Streaming

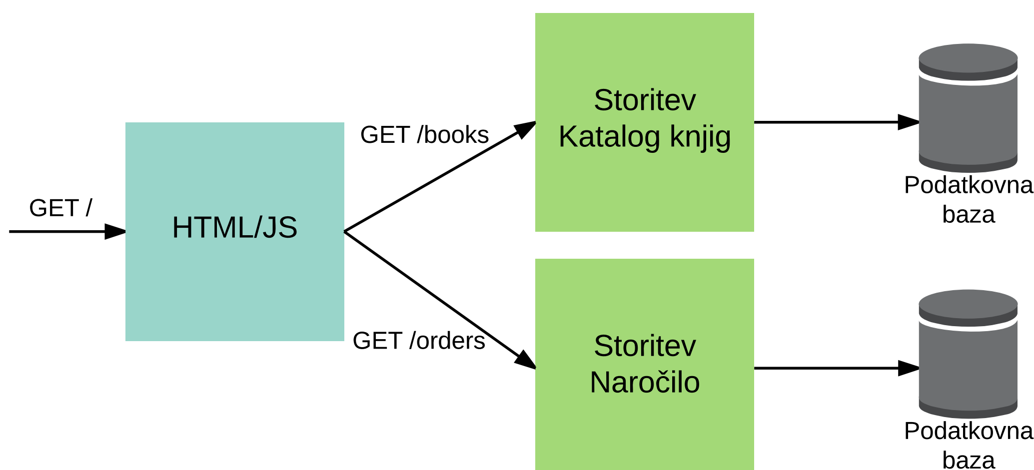
Kot primer uporabe razširitve KumuluzEE Event Streaming smo pripravili primer aplikacije za spletno knjigarno v arhitekturi mikrostoritev po vzorcih dogodkovnih virov in CQRS. Običajna zasnova aplikacije, kot je prikazana na sliki 5.2, kjer ima vsaka storitev ločeno podatkovno bazo, je problematična, saj uvaja porazdeljeno upravljanje s podatki. Temu se lahko izognemo z arhitekturnim vzorcem dogodkovnih virov, pri katerem nimamo običajne podatkovne baze. Z uporabo odprtokodnega ogrodja KumuluzEE in razširitve KumuluzEE Event Streaming smo zasnovali aplikacijo, ki ob vsaki spremembi entitete pošlje dogodek na platformo Kafka. Sledili smo tudi vzorcu CQRS ter med seboj ločili ukaze in poizvedbe. Kot prikaz zasnove aplikacije glej sliko 5.3.

Aplikacijo za spletno knjigarno smo zasnovali v treh mikrostoritvah: storitev za katalog knjig, naročila in dobavo knjig. V storitvi za katalog knjig procesiramo ukaze, kot so „nova knjiga“, „novo naročilo“ ali „nova dobava“, in glede na njihovo vsebino ustrezno posodobimo trenutno stanje entitet v shrambi stanj.

V Kafki uporabljamo pet tem, na katere pošiljamo dogodke, in sicer:

- *commandsTopic* – tema na katero pošiljamo ukaze (npr. kreiraj novo naročilo, dobavo).

- *bookEventsTopic* – tema z dogodki o knjigah.

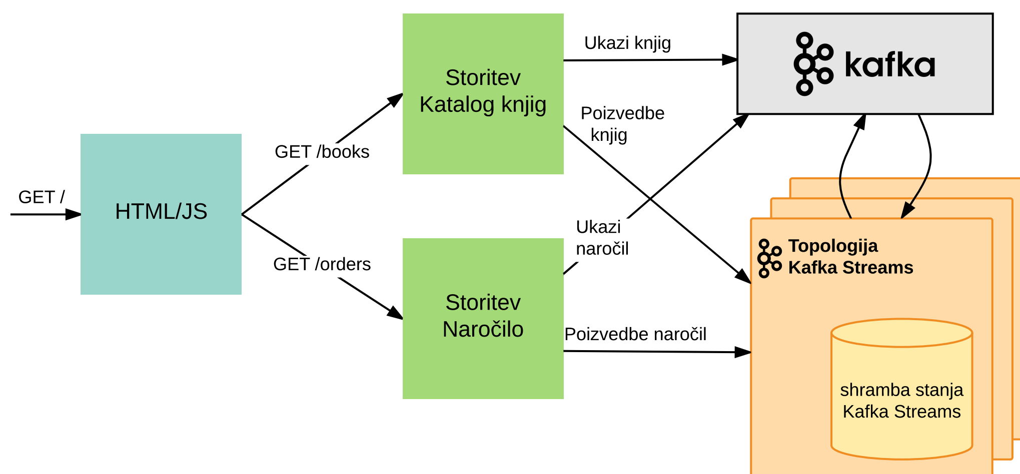


Slika 5.2: Zasnova primera aplikacije spletne knjigarne v običajni arhitekturi mikrostoritev z ločeno podatkovno bazo za vsako storitev.

- *shipmentsEventsTopic* – tema z dogodki o dobavi knjig.
- *ordersEventsTopic* – tema z dogodki o naročilih knjig.

Vsaka tema je porazdeljena na več particij, odvisno od števila instanc posameznih storitev, tako da lahko izkoristimo Kafkino uravnoteženje obremenitev in paralelno izvajanje procesiranja tokov.

Vzorec dogodkovnih virov smo dosegli s pomočjo Kafka Streams. Za lokalno hranjenje trenutnega stanja entitet smo uporabili Kafkino shrambo stanja (*state store*), ki je porazdeljena, obstojna in odporna na izpade. V storitvi za katalog knjig smo z anotacijo `StreamProcessor` ustvarili procesor toka, ki sprejema ukaze iz teme z ukazi, iz njih razbere podatke in posodobi entitete v shrambi stanja. Ob vsakem ukazu glede na vrsto ukaza objavi dogodek v specifično temo. Pri razvoju aplikacije smo sledili tudi vzorcu CQRS ter ločili procesiranje ukazov in poizvedb. Ukazi se pošiljajo na temo za ukaze, od koder se berejo in obdelajo v procesorju ukazov, poizvedbe pa opravljamo s t. i. interaktivnimi poizvedbami (*Interactive Queries*), ki jih ponuja Kafka Streams in s katerimi lahko izvajamo različne poizvedbe nad lokalno shrambo stanja.

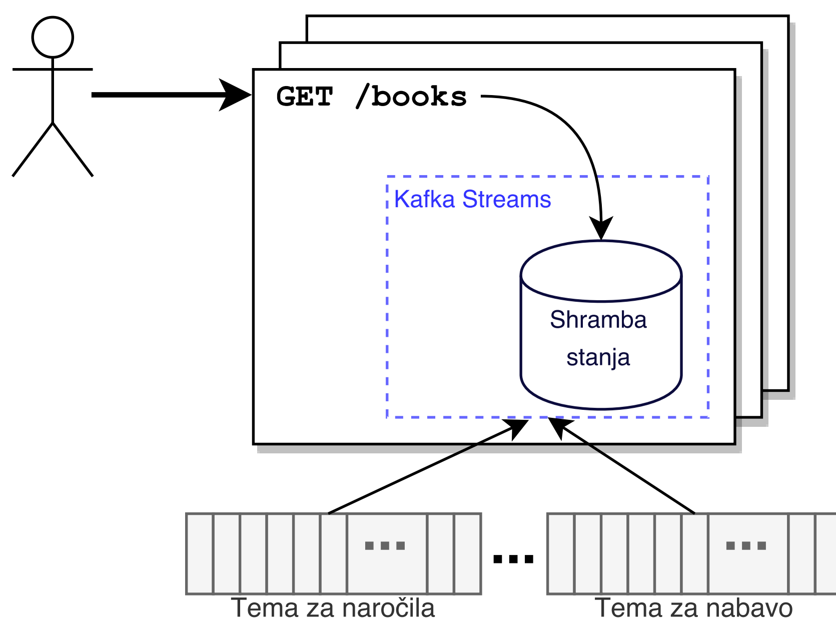


Slika 5.3: Zasnova primera aplikacije spletne knjigarne v arhitekturi mikrostoritev z vzorcema dogodkovnih virov in CQRS.

Potek izvedbe ukaza za naročilo knjig poteka tako, da se najprej s klicem POST na `/orders` na storitvi za naročila ustvari nov ukaz za novo naročilo. Ukaz se pošlje na temo z ukazi *commandsTopic*. Ukazi se procesirajo v storitvi za katalog, kjer se ustvari nov dogodek o novem naročilu na temi za naročila *ordersEventsTopic*. Nato se preveri zaloga knjige v shrambi stanja, v primeru, da je zaloge dovolj, se posodobi zaloga knjige v shrambi in objavi dogodek o uspešno opravljenem naročilu, v nasprotnem primeru pa se objavi dogodek o zavrnjenem naročilu. V storitvi za naročila je prav tako procesor toka, ki prejeme dogodke o naročilih in v shrambo stanja shranjuje trenutno stanje naročil; tako lahko s klicem GET na `/orders/orderId` poizvedujemo, kakšno je trenutno stanje naročila. Podobno delujejo tudi ukazi za novo dobavo in novo knjigo.

Seznam in zalogo knjig pridobimo s klicem GET na `/books`, s čimer izvedemo interaktivno poizvedbo nad shrambo stanja, kar je prikazano na sliki 5.4. V primeru več instanc storitve z procesiranjem toka se procesiranja toka in lokalna shramba stanja porazdeli po instancah. Kafka Streams omogoča pridobitev metapodatkov o instanci storitve, ki vsebuje del shrambe stanja z entiteto z določenim ključem. Kafka Streams trenutno še ne podpira sa-

modejnega preusmerjanja, zato moramo sami poskrbeti za višjenivojsko preusmeritev poizvedbe na instanco, ki hrani entiteto v svojem delu shrambe stanja.



Slika 5.4: Prikaz izvebe poizvedbe z klicem GET na `/books`.

Prednost takšne zasnove je šibka sklopljenost komponent, storitve so medseboj neodvisne. Poenostavljeno je vključevanje novih mikrostoritev v sistem, ker se te samo naročijo na dogodke v Kafki in delujejo neodvisno od ostalih storitev. V aplikacijo za spletno knjigarno bi tako lahko dodali novo mikrostoritev za odkrivanje prevar, ki bi procesirala dogodke o naročilih in odkrivala morebitne goljufije. V kompleksnejšem primeru, npr. produktnem sistemu velikega podjetja, bi se še bolje izrazila učinkovitost takšne zasnove sistema, saj so podatki o stanju entitet v spominu aplikacije ali na disku hitro dostopni, kar je zelo koristno, če dostopamo do velike količine podatkov. Izognemo se tudi podvajanju podatkov med shrambo za agregacijo in procesiranje toka ter shrambo, nad katero poizvedujemo [14].

Takšna zasnova omogoča tudi večjo skalabilnost. Storitve lahko glede na obremenitev posamezno skaliramo, s čimer izkoristimo porazdelitev bre-

mena procesiranja tokov s Kafka Streams. Z dogodkovno vodeno arhitekturo se izognemo težavam porazdeljenega upravljanja s podatki, ki so prisotne v sistemih mikrostoritev, kjer ima vsaka storitev svojo podatkovno bazo. Z vzorcem dogodkovnih virov pridobimo natančen revizijski dnevnik vseh sprememb v sistemu, za katerega potrebujemo ločeno orodje za beleženje sprememb v običajni zasnovi aplikacije. Dodatne prednosti pridobimo tudi z vzorcem CQRS. Z ločitvijo poizvedb in ukazov omogočimo večjo skalabilnost, prilagodljivost in boljšo optimizacijo branja oziroma pisanja. Glede na vrsto podatkov lahko prilagodimo shrambo stanj: tako lahko npr., kadar hranimo povezave med entitetami, uporabimo grafno bazo in tako optimiziramo poizvedbe nad podatki.

Poglavje 6

Sklepne ugotovitve

V diplomski nalogi smo obravnavali integracijo pretočnih dogodkov in mikrostoritev. Predelali smo področje mikrostoritev ter interakcijo med njimi. Predstavili smo najbolj znane tehnološke implementacije sinhronih in asinhronih načinov interakcije ter opisali njihove prednosti in slabosti. Predstavili smo arhitekturni pristop, ki temelji na pretakanju dogodkov, in opisali vzorca dogodkovnih virov in CQRS.

Natančneje smo opisali sporočilne vrste in platforme za pretakanje. Poudarili smo ključne razlike ter prednosti platform za pretakanje, kot so večja skalabilnost, pretočnost, učinkovita shramba podatkov, odpornost na izpade ter dodatne funkcionalnosti procesiranja tokov podatkov.

Analizirali smo arhitekturno zasnovo in delovanje platforme Apache Kafka, ki postaja vse bolj priljubljen način za prenašanje in procesiranje sporočil. Kafka je porazdeljena platforma za pretakanje. Njene ključne lastnosti so visoka horizontalna skalabilnost, pretočnost, hitrost in odpornost na izpade. Opisali smo proizvajalce in potrošnike sporočil ter procesorje toka Kafka Streams.

Kot del diplomske naloge smo razvili razširitev KumuluzEE Event Streaming za Javo za odprtokodno ogrodje KumuluzEE. Opisali smo njeno zasnovo in delovanje ter prikazali izseke kode, ki prikazujejo njeno uporabo. Razširitev je odprtokodna in javno dostopna na [3]. Kot prikaz uporabe

razširitve smo pripravili primer aplikacije spletne knjigarne, zasnovane na principu dogodkovno vodene arhitekture mikrostoritev in vzorcih dogodkovnih virov ter CQRS.

Razvita razširitev se je izkazala za uporabno in izpolnjuje naš cilj enostavne integracije pretočnih dogodkov in mikrostoritev. Možnost nadaljnjega dela vidimo v dodajanju podpore za druge platforme in implementacijo dodatnih funkcionalnosti Kafke.

Literatura

[1] Apache Kafka Documentation. Dosegljivo: https://kafka.apache.org/documentation/, 2017. [Dostopano 3. 8. 2017].

[2] KumuluzEE. Dosegljivo: https://ee.kumuluz.com/, 2017. [Dostopano 8. 8. 2017].

[3] KumuluzEE Event Streaming. Dosegljivo: https://github.com/kumuluz/kumuluzee-streaming, 2017. [Dostopano 13. 8. 2017].

[4] The HTTP/2 Protocol: Its Pros & Cons and How to Start Using It. Dosegljivo: https://www.upwork.com/hiring/development/the-http2-protocol-its-pros-cons-and-how-to-start-using-it/, 2017. [Dostopano 23. 8. 2017].

[5] Why event-driven microservices? Dosegljivo: https://github.com/cer/event-sourcing-examples/wiki/WhyEventDrivenArch, 2017. [Dostopano 23. 8. 2017].

[6] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, 2016.

[7] Ted Dunning and B. Ellen Friedman. *Streaming architecture: new designs using Apache Kafka and MapR streams.* OReilly, 2016.

[8] Martin Fowler. Focusing on Events. Dosegljivo: https://martinfowler.com/eaaDev/EventNarrative.html, 2006. [Dostopano 3. 8. 2017].

[9] Martin Fowler. Richardson Maturity Model. Dosegljivo: https://martinfowler.com/articles/richardsonMaturityModel.html, 2010. [Dostopano 14. 8. 2017].

[10] Martin Fowler. Polyglot Persistence. Dosegljivo: https://martinfowler.com/bliki/PolyglotPersistence.html, 2011. [Dostopano 2. 8. 2017].

[11] Martin Fowler. Microservices. Dosegljivo: https://martinfowler.com/articles/microservices.html, 2014. [Dostopano 2. 8. 2017].

[12] Martin Fowler. Microservice Trade-Offs. Dosegljivo: https://martinfowler.com/articles/microservice-trade-offs.html, 2015. [Dostopano 14. 8. 2017].

[13] Jay Kreps. Putting Apache Kafka To Use: A Practical Guide to Building a Streaming Platform (Part 1). Dosegljivo: https://www.confluent.io/blog/stream-data-platform-1/, 2015. [Dostopano 4. 8. 2017].

[14] Neha Narkhede. Event sourcing, CQRS, stream processing and Apache Kafka: What's the connection? Dosegljivo: https://www.confluent.io/blog/event-sourcing-cqrs-stream-processing-apache-kafka-whats-connection/, 2016. [Dostopano 8. 8. 2017].

[15] Neha Narkhede, Gwen Shapira, and Todd Palino. *Kafka: the definitive guide: Real-Time Data and Stream Processing at Scale*. Oreilly & Associates Inc, 2017.

[16] S. Newman. *Building Microservices*. O'Reilly Media, 2015.

[17] Christian Posta. What is Apache Kafka? Why is it so popular? Should you use it? Dosegljivo: `https://techbeacon.com/what-apache-kafka-why-it-so-popular-should-you-use-it`, 2016. [Dostopano 29. 8. 2017].

[18] Jun Rao. Confluence Apache Kafka Wiki - Clients. Dosegljivo: `https://cwiki.apache.org/confluence/display/KAFKA/Clients`, 2017. [Dostopano 23. 7. 2017].

[19] Chris Richardson. Pattern: Circuit Breaker. Dosegljivo: `http://microservices.io/patterns/reliability/circuit-breaker.html`, 2016. [Dostopano 6. 9. 2017].

[20] Chris Richardson and Floyd Earl Smith. *Microservices From Design to Deployment*. NGINX Inc, 2016.

[21] Matt Stine. *Migrating to cloud-native application architectures*. O'Reilly Media, Inc., 2015.

[22] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34(4):42–47, 2005.

[23] Johannes Thones. Microservices. *IEEE Software*, 32(1):116, 2015.

[24] Giovanni Toffetti, Sandro Brunner, Martin Blöchlinger, Josef Spillner, and Thomas Michael Bohnert. Self-managing cloud-native applications: Design, implementation, and experience. *Future Generation Computer Systems*, 72:165 – 179, 2017.