Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Urban Malc

Zasnova in razvoj rešitve za dinamično odkrivanje mikrostoritev v oblačnih arhitekturah

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:

Proučite arhitekturo mikrostoritev in analizirajte pomen dinamičnega odkrivanja storitev. Umestite mikrostoritve v arhitekturo »cloud-native«. Opišite pomen odkrivanja storitev in najpomembnejše scenarije. Identificirajte, opišite in primerjajte najpomembnejše registre za odkrivanje storitev. Zasnujte in razvijte modul za odkrivanje mikrostoritev v Javi ter na praktičnem primeru prikažite in ovrednotite delovanje.

Zahvaljujem se prof. dr. Matjažu Branku Juriču za mentorstvo in strokovno pomoč pri izdelavi diplomske naloge. Zahvaljujem se tudi Janu Meznariču in ostalim članom Laboratorija za integracijo informacijskih sistemov za pomoč pri pisanju diplomske naloge. Na koncu se zahvaljujem še družini za podporo in motivacijo v času mojega študija.

Kazalo

Povzetek

Abstract

1 Uvod **1**

2 Arhitektura mikrostoritev in pomen odkrivanja storitev **3**

2.1 Mikrostoritve . . . 3

2.2 Arhitektura Cloud-Native . . . 6

2.3 Pomen odkrivanja storitev . . . 7

2.4 Scenariji odkrivanja storitev . . . 8

3 Registri za odkrivanje storitev **15**

3.1 etcd . . . 15

3.2 Consul . . . 19

3.3 ZooKeeper . . . 21

3.4 Eureka . . . 22

3.5 Primerjava . . . 23

4 Implementacija modula za odkrivanje mikrostoritev **27**

4.1 etcd implementacija . . . 30

4.2 Consul implementacija . . . 34

5 Prikaz delovanja **39**

5.1 Mikrostoritev Naročilo . . . 40

5.2 Mikrostoritev Stranka 41
5.3 Odjemalec 43
5.4 Postavitev aplikacije in primer uporabe 46

6 Zaključek **51**

Literatura **53**

Seznam uporabljenih kratic

kratica	angleško	slovensko
IaaS	Infrastructure as a Service	infrastruktura kot storitev
PaaS	Platform as a Service	platforma kot storitev
API	Application Program Interface	aplikacijski programski vmesnik
HTTP	Hyper-Text Transfer Protocol	protokol za izmenjavo hiperteksta
JSON	JavaScript Object Notation	objektni zapis JavaScript
TTL	Time-to-Live	življenjska doba
CAS	Compare-and-Swap	primerjaj in zamenjaj
CAD	Compare-and-Delete	primerjaj in izbriši
gRPC	gRPC Remote Procedure Calls	klici za oddaljene postopke gRPC
TCP	Transmission Control Protocol	protokol za nadzor prenosa
DNS	Domain Name System	sistem domenskih imen
XML	Extensible Markup Language	razširljivi označevalni jezik
CDI	Contexts and Dependency Injection	dodajanje konteksta in vključevanje odvisnosti platforme Java
URL	uniform resource locator	enotni naslov vira
JAX-RS	Java API for RESTful Services	vmesnik API za storitve REST platforme Java
NPM	Node Package Manager	upravitelj paketov Node
REST	Representational State Transfer	arhitekturni način prenašanja stanj

Povzetek

Naslov: Zasnova in razvoj rešitve za dinamično odkrivanje mikrostoritev v oblačnih arhitekturah

Avtor: Urban Malc

Arhitektura mikrostoritev poleg številnih prednosti prinaša tudi nove izzive, ki jih pri tradicionalnih arhitekturah ne srečujemo. Eden izmed njih je problem dinamičnega dodeljevanja naslovov. Moderne aplikacije, grajene v arhitekturi mikrostoritev se izvajajo v vsebniških okoljih, saj ta omogočajo enostavno postavitev in horizontalno skaliranje mikrostoritev. Vsebniška okolja zaradi poenostavitve skaliranja naslove mikrostoritev dodeljujejo dinamično. Problem nastane pri komunikaciji med mikrostoritvami, saj so njihovi naslovi znani šele ob njihovem zagonu. Potrebno je dinamično odkrivanje naslovov mikrostoritev med samim izvajanjem. V diplomskem delu smo raziskali pomen odkrivanja storitev v arhitekturi mikrostoritev. Zasnovali in razvili smo sistem, ki razvijalcem omogoča enostavno registracijo in odkrivanje storitev v ogrodju KumuluzEE in predstavili njegovo delovanje z vzorčno aplikacijo, ki ta sistem uporablja. Z dinamičnim odkrivanjem mikrostoritev smo omogočili uporabo dinamičnega dodeljevanja naslovov mikrostoritvam in s tem olajšali njihovo integracijo s sodobnimi oblačnimi okolji.

Ključne besede: mikrostoritve, cloud-native, odkrivanje storitev, etcd, Consul.

Abstract

Title: Design and implementation of dynamic microservice discovery solution in cloud architectures

Author: Urban Malc

Microservice architecture offers many advantages over monolithic application design, but at the same time presents challenges, not present in traditional architectures. One of the challenges is handling dynamic allocation of microservice addresses. Modern applications, built in microservice arhitecture typically run in containerized environments, which enable simple deployment and horizontal scaling of microservices. Containerized environments usually allocate microservice addresses dynamically, as this simplifies the scaling process. Problem arises when communication between microservices is required. Since addresses of microservices are allocated when microservices are deployed, dynamic discovery of microservice addresses during runtime is required. In our thesis, we discuss the importance of service discovery in microservice architectures. We design and develop a system, which enables simple service registration and discovery in the KumuluzEE framework. We demonstrate the usage of the developed solution through a reference application. Dynamic service discovery enables the usage of dynamic address allocation in microservices, which alleviates their integration with modern cloud environments.

Keywords: microservices, cloud-native, service discovery, etcd, Consul.

Poglavje 1

Uvod

Tradicionalne arhitekture gradnje aplikacij omogočajo hiter razvoj razmeroma enostavnih aplikacij, vendar se pri naraščanju kompleksnosti aplikacij pogosto izkažejo za nezadostne. S povečevanjem velikosti izvorne kode ta postane težko razumljiva, saj se meje modulov sčasoma zameglijo. Posledično se dodajanje novih funkcionalnosti upočasni. Zahtevna postane tudi zvezna integracija, saj vsaka sprememba v kodi zahteva ponovno postavitev celotne aplikacije. Horizontalno skaliranje je mogoče le s postavitvijo več instanc celotne aplikacije, kar se pogosto izkaže za neučinkovito, saj ozko grlo tipično predstavlja le del aplikacije. Razvijalci se z željo po lažjem obvladovanju kompleksnih aplikacij obračajo na arhitekturo mikrostoritev, ki z dekompozicijo aplikacije na izolirane storitve ponuja agilen razvoj aplikacij, lažjo implementacijo novih funkcionalnosti in granularno skalabilnost končnih storitev.

Arhitektura mikrostoritev izkorišča oblačna okolja, ki s svojo dinamičnostjo in možnostjo elastičnega dodeljevanja virov omogočajo abstrakcijo strojne opreme in s tem razvijalcem ponujajo večji fokus na programsko opremo. Poleg številnih prednosti, ki jih ponuja, pa arhitektura mikrostoritev prinaša tudi nove izzive, ki jih v tradicionalnih arhitekturah ni bilo. Eden izmed njih je potreba po dinamičnem odkrivanju mikrostoritev med njihovim izvajanjem.

Moderne aplikacije, ki so zasnovane v arhitekturi mikrostoritev, se izvajajo v vsebniških okoljih, saj ta ponujajo enostavno postavitev in horizontalno skaliranje mikrostoritev. V takih okoljih se število instanc mikrostoritev in njihovi naslovi spreminjajo dinamično, zato je za uspešno komunikacijo med mikrostoritvami potrebno odkrivanje njihovih naslovov med izvajanjem. Z uporabo dinamičnih naslovov mikrostoritev in sistema za odkrivanje storitev se poenostavi konfiguracija celotne aplikacije.

Cilj diplomske naloge je pregled in analiza načinov in pristopov k dinamičnem odkrivanju storitev ter pregled in primerjava registrov, ki omogočajo odkrivanje storitev. Poleg tega je cilj zasnova in razvoj rešitve modula za dinamično odkrivanje storitev v oblačnih arhitekturah za ogrodje mikrostoritev KumuluzEE ter razvoj aplikacije, ki skozi praktični primer prikaže delovanje razvitega modula.

V diplomskem delu bomo v drugem poglavju predstavili arhitekturo mikrostoritev in arhitekturo „cloud-native“. Predstavili bomo pomen odkrivanja storitev v omenjenih arhitekturah in podali pogoste scenarije, ki se pri odkrivanju storitev pojavijo. V tretjem poglavju bomo opisali in primerjali registre, ki se uporabljajo za potrebe odkrivanja storitev. Nato bomo v četrtem poglavju predstavili našo zasnovo in implementacijo modula za odkrivanje storitev, ki v ogrodje KumuluzEE prinaša enostavno uporabo registrov za odkrivanje storitev. V petem poglavju bomo prikazali delovanje razvitega modula skozi vzorčno aplikacijo.

Poglavje 2

Arhitektura mikrostoritev in pomen odkrivanja storitev

Arhitektura mikrostoritev je nov pristop h gradnji aplikacij, ki aplikacijo razdeli na več majhnih storitev - mikrostoritev. Mikrostoritve se povezujejo preko lahkih protokolov in s tem tvorijo celotno aplikacijo [16]. Arhitektura mikrostoritev je tesno povezana s celostno arhitekturo „cloud-native“. Arhitektura „cloud-native“ je nova paradigma razvoja programske opreme, ki je optimizirana za moderne porazdeljene sisteme, zmožna skaliranja na več deset tisoč samopopravljivih vozlišč. Z uporabo orkestracijskih orodij, ki dinamično dodeljujejo naslove mikrostoritvam, nastane potreba po lociranju mikrostoritev med samim delovanjem le-teh.

V tem poglavju je predstavljen koncept mikrostoritev in arhitektura „cloud-native“, ki ta koncept uporablja. Predstavljen je pomen odkrivanja storitev v arhitekturi mikrostoritev in tipični scenariji, ki jih srečujemo pri konceptu odkrivanja storitev.

2.1 Mikrostoritve

Mikrostoritev je samostojni in kohezivni proces. Vsaka mikrostoritev opravlja enostavno poslovno vlogo in je najpogosteje zadolžena za upravljanje ene

entitete v podatkovnem modelu aplikacije. Vsako mikrostoritev je mogoče zagnati samostojno [16, 26, 14].

Motivacija za uporabo novega pristopa prihaja iz problemov monolitne arhitekturne zasnove aplikacij. To je tradicionalni način za gradnjo aplikacij, saj je tak pristop najbolj naraven način za izdelavo aplikacij. Monolitne aplikacije predstavljajo rešitev enega problema, ki je dobro določen pred začetkom implementacije. Problem nastane pri dodajanju novih funkcionalnosti obstoječi aplikaciji. Kljub temu da dobri razvojni modeli zahtevajo ločitev aplikacje na posamezne module, se sčasoma meje modulov zameglijo, kar predstavlja problem pri dodajanju novih funkcionalnosti in popravljanju napak. Končni rezultat monolitne arhitekture je celostna aplikacija, ki je bodisi samostojno izvršljiva bodisi izvršljiva s pomočjo aplikacijskega strežnika. To pomeni, da vsaka sprememba zahteva ponovno postavitev celotne aplikacije, kar je pri velikih aplikacijah izguba dragocenega časa razvijalcev, ki bi bil lahko namenjen nadaljnjemu razvoju aplikacije. Ob povečevanju števila uporabnikov nastane potreba po skaliranju aplikacije. V monolitni arhitekturi ta problem po navadi rešujemo tako, da zaženemo več instanc aplikacije, pred njih pa postavimo sistem za uravnoteževanje obremenitve, ki razporeja promet med njimi. Pri takem skaliranju aplikacije prihaja do problemov pri sinhronizaciji med instancami. Poleg tega pogosto pridemo do spoznanja, da ozko grlo aplikacije nastane le pri enem delu aplikacije, kar pomeni neučinkovito porabo sredstev pri skaliranju celotne aplikacije. Kljub temu da monolitni razvoj aplikacij prinaša kar nekaj prednosti, se omenjeni problemi kopičijo, še posebej pri razvoju večjih aplikacij. Prav zaradi naštetih problemov popularizacija arhitekture mikrostoritev v zadnjih letih strmo narašča, saj ta ponuja elegantno rešitev omenjenih problemov [16].

Z delitvjo aplikacije na mikrostoritve so meje modulov aplikacije strogo začrtane. Ker so posamezne funkcionalnosti aplikacije ločene na mikrostoritve, je vzdrževanje in nadgrajevanje funkcionalnosti lažje. Vsako storitev je mogoče zagnati samostojno, kar pomeni, da je pri spremembah treba ponovno zagnati le tiste mikrostoritve, na katere se sprememba navezuje.

Možnost hitrega ponovnega postavljanja mikrostoritev vzpodbuja razvijalce k sprotnemu testiranju mikrostoritev, kar omogoča skoraj takojšnje okrevanje po morebitnih napakah. Arhitektura mikrostoritev ponuja enostavno rešitev horizontalnega skaliranja aplikacije, pri čemer se skalira samo tista mikrostoritev, ki predstavlja ozko grlo v aplikaciji. S tem je zagotovljena učinkovita poraba sredstev. Ker so mikrostoritve grajene z mislijo na enostavno skalabilnost, je potrebe po sinhronizaciji med posameznimi instancami mikrostoritev malo oziroma nič.

Šibka sklopljenost mikrostoritev omogoča mešanje tehnologij in jezikov in s tem razvijalcem ponuja svobodo pri izbiri načina implementacije posamezne mikrostoritve. Razvijalci v nasprotju z monolitno arhitekturo niso vezani le na eno tehnologijo, kar omogoča učinkovito uporabo tehnologij, ki so za posamezno mikrostoritev najbolj primerne.

Arhitektura mikrostoritev poleg mnogih prednosti prinaša tudi številne probleme, ki jih pri gradnji monolitnih aplikacij ni bilo. Prvi izmed njih je potreba po spremembi miselnosti razvijalcev. Neustrezna delitev aplikacije ne odpravlja problemov monolitnih aplikacij in s tem izničuje prednosti, ki jih prinaša arhitektura mikrostoritev. Poleg tega se z zamenjavo programskih klicev s klici preko omrežja povečuje zakasnitev klica ter obremenitev omrežja. Z uporabo arhitekture mikrostoritev se poveča tudi kompleksnost postavitve, saj je celotna aplikacija porazdeljene narave. Kompleksnost postavitve olajšujejo postopki „DevOps“, ki procese namestitve avtomatizirajo. Uporaba orodij za zvezno integracijo omogoča pogoste integracije sprememb v kodi v skupni repozitorij in avtomatizira prevajanje in testiranje mikrostoritev, kar povečuje možnost zgodnjega odkrivanja napak. Orkestracijska orodja skrbijo za avtomatsko postavitev in skaliranje mikrostoritev. Uporaba teh orodij nadalje poenostavlja namestitev mikrostoritev, kar pripomore k agilnem razvoju aplikacij [12].

Prednosti mikrostoritev tipično premagajo njihove slabosti šele pri kompleksnih sistemih, kjer je število uporabnikov veliko in je s tem potreba po skalabilnosti in odpornosti na napake večja.

2.2 Arhitektura Cloud-Native

Po definiciji fundacije za „cloud-native“ računalništvo (angl. Cloud Native Computing Foundation) imajo sistemi „cloud-native“ naslednje lastnosti [15]:

1. Aplikacije in procesi sistema tečejo v vsebniških tehnologijah, da dosežejo visoko izolacijo nad viri.

2. Procesi v sistemu se upravljajo dinamično s centralnim sistemom za orkestracijo, ki izboljšuje porabo virov in zmanjšuje stroške vzdrževanja.

3. Sistem je zgrajen v arhitekturi mikrostoritev, ki zaradi šibke sklopljenosti povečuje agilnost in poenostavlja vzdrževanje aplikacij.

S potrebo po procesiranju velikega števila uporabnikov se je populariziralo računalništvo v oblaku [22]. Sprva se je selitev v oblak začela s selitvijo strojne opreme v oblak, tako imenovano infrastrukturo kot storitev (angl. Infrastructure as a Service, IaaS). Ta je prinesla prednost hitrega skaliranja strojne opreme in selitev stroškov nakupa in vzdrževanja strojne opreme v stroške najema strojne opreme. Logično nadaljevanje infrastrukture IaaS je platforma kot storitev (angl. Platform as a Service, PaaS). Platforma PaaS ponuja pogosto uporabljene storitve (npr. podatkovno bazo) kot platformo. Prednost platforme PaaS je enostavnost, saj ponuja abstrakcijo operacijskega sistema storitve. Arhitektura „cloud-native“ nadaljuje z abstrakcijo z uporabo arhitekture mikrostoritev in izkoriščanjem vsebniških tehnologij. S poganjanjem mikrostoritev v vsebnikih dosežemo popolno izolacijo od sistema, na katerem mikrostoritev teče, orkestracijska orodja pa omogočajo abstrakcijo od samega poganjanja in skaliranja mikrostoritev. Šibka sklopljenost mikrostoritev prinaša enostavno zamenljivost storitev in zmanjšuje odvisnost od njihove implementacije [13, 20, 27, 28].

Arhitektura „cloud-native“ pa ne predstavlja le selitve aplikacij v oblak in uporabe arhitekture mikrostoritev, temveč prilagajanje celotnega razvojnega cikla za optimalno delo z oblačnimi infrastrukturami. To vključuje integracijo orkestracijskih in integracijskih orodij v razvojni cikel. V duhu mikrostoritev

se razvojne ekipe ne delijo več glede na specializacijo znanj, ampak glede na poslovne vloge [16].

Ena izmed poglavitnih prednosti, ki jih prinaša arhitektura „cloud-native“, je hitrost razvoja aplikacij. Arhitektura mikrostoritev z delitvijo aplikacije na poslovne vloge omogoča hitrejši razvoj novih funkcionalnosti, predvsem pa lažje testiranje in nadgradnjo posameznih mikrostoritev. Uporaba integracijskih in orkestracijskih orodij zmanjšuje čas objave in prevzame del vzdrževanja aplikacij z rok razvijalcev. S krajšim časom objave razvijalci hitreje pridobijo povratne informacije s strani strank, kar še dodatno pripomore k agilnem razvoju aplikacije [25].

2.3 Pomen odkrivanja storitev

Odkrivanje storitev igra ključno vlogo v arhitekturi mikrostoritev. Pri tradicionalnih monolitnih aplikacijah potrebe po odkrivanju storitev ni bilo, saj so se storitve klicale preko programskih klicev. Naslovi instanc aplikacije so bili statično določeni, saj je to poenostavilo samo administracijo le teh. Statično določanje naslovov mikrostoritev je v arhitekturi mikrostoritev nepraktično, saj se instance tipično izvajajo v vsebniških okoljih. Ta sicer podpirajo statično dodeljevanje naslovov, vendar s tem izgubimo možnost horizontalnega skaliranja, kar je ena izmed glavnih prednosti uporabe vsebniških okolij in ena izmed poglavitnih značilnosti arhitekture mikrostoritev. Zato se naslovi mikrostoritev tipično določajo dinamično, kar pa za sabo prinese potrebo po odkrivanju lokacij storitev med izvajanjem mikrostoritev.

Odkrivanje storitev je tipično realizirano s pomočjo registra storitev. Vanj se storitve ob začetku izvajanja registrirajo. Ko nastane potreba po dostopu do mikrostoritve, se registru pošlje poizvedba, na katero register odgovori z lokacijami instanc potrebne mikrostoritve.

Poizvedbe na register morajo biti realizirane učinkovito, saj je zakasnitev klicev pri arhitekturi mikrostoritev v primerjavi z monolitnimi aplikacijami že tako velika. Ker je število klicev, ki jih določena mikrostoritev opravi na

odkrito mikrostoritev tipično večja od števila sprememb v registru, je pri odkrivanju storitev pogost pristop naročanja na spremembe v registru. Ob prvi poizvedbi na register mikrostoritev od registra zahteva, naj jo obvesti o vseh spremembah instanc zahtevane storitve v registru. Ob spremembi instanc je register zadolžen za to, da spremembo sporoči vsem naročenim mikrostoritvam.

2.4 Scenariji odkrivanja storitev

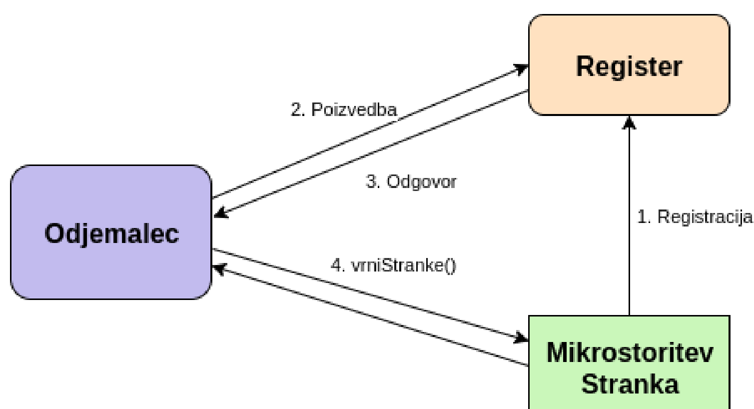
Osnovni scenarij odkrivanja storitev je ta, v katerem se mikrostoritev registrira v register, odjemalec, ki hoče to mikrostoritev odkriti, pa naredi poizvedbo na register, iz katerega pridobi naslov storitve. Naslednji primer predstavlja omenjeni scenarij:

1. Mikrostoritev Stranka se ob zagonu vpiše v register.
2. Odjemalec želi pridobiti seznam strank. Najprej izvede poizvedbo nad registrom.
3. Register odgovori z naslovom zahtevane mikrostoritve.
4. Odjemalec izvede klic vrniStranke() nad odkrito mikrostoritvijo in ta mu odgovori s seznamom strank.

Opisani primer prikazuje slika 2.1.

Za arhitekturo mikrostoritev je značilno horizontalno skaliranje mikrostoritev. V tem primeru lahko osnovni primer razširimo tako, da dodamo sistem za uravnoteževanje obremenitve na strani odjemalca. Razširjen primer sestavljajo naslednje operacije:

1. Vse instance mikrostoritve Stranka se ob zagonu vpišejo v register.
2. Odjemalec, ki želi prejeti seznam strank, najprej izvede poizvedbo nad registrom.



Slika 2.1: Osnovni scenarij odkrivanja storitev

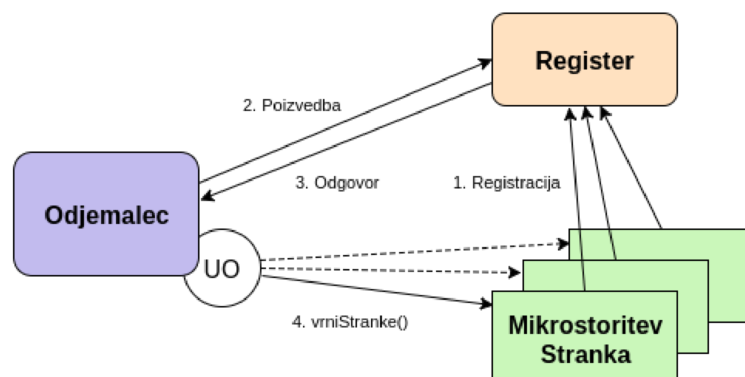
3. Register odgovori z množico naslovov vseh mikrostoritev, registriranih v registru.

4. Odjemalec izmed množice naslovov izbere enega in nad njim izvede klic vrniStranke().

Opisani primer prikazuje slika 2.2. Simbol z oznako „UO“ na sliki predstavlja sistem za uravnoteževanje obremenitve na strani odjemalca.

2.4.1 Uporaba prehoda API

Prehod aplikacijskih programskih vmesnikov (angl. Application Program Interface, API) je ključni del arhitekture mikrostoritev, saj predstavlja enotno vstopno točko do mikrostoritev. Modularna zasnova mikrostoritev in granulacija vmesnikov, ki jih te izpostavljajo, pogosto predstavljajo nezaželeno kompleksnost njihove uporabe s strani odjemalcev. Pogosto si želimo centralni dostop do vseh mikrostoritev. V tem primeru je pred mikrostoritve postavljen prehod API, ki prejete klice preusmerja na posamezne mikrostoritve. Z uporabo prehoda API se izognemo uravnoteževanju obremenitve na



Slika 2.2: Scenarij odkrivanja storitev z repliciranimi storitvami

strani odjemalca in s tem poenostavimo dostop do programskega vmesnika, ki ga mikrostoritve izpostavljajo. Prehod API omogoča večji nadzor nad klici in pogosto implementira kontrolo dostopa mikrostoritev. Obenem je z uporabo prehoda API možno zbiranje metrik klicev na samem prehodu [23].

V primeru uporabe prehoda API morata biti preko registra za vsako instanco registrirane mikrostoritve dostopna dva naslova. Prvi predstavlja dejanski naslov mikrostoritve, drugi pa predstavlja naslov prehoda API, preko katerega je mikrostoritev dostopna. Ta scenarij lahko ponazorimo z naslednjim primerom:

1. Vse instance mikrostoritve Stranka se ob zagonu vpišejo v register.

2. Odjemalec, ki želi prejeti seznam strank, izvede poizvedbo nad registrom, v kateri zahteva dostop do mikrostoritve Stranka preko prehoda API.

3. Register odgovori z naslovom prehoda API, preko katerega je dostopna zahtevana mikrostoritev.

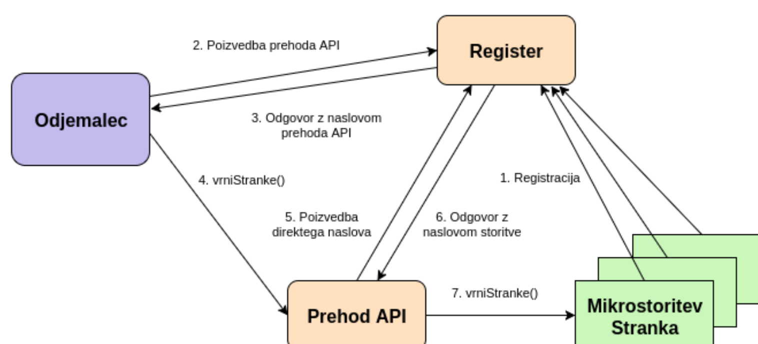
4. Odjemalec izvede klic vrniStranke() nad prehodom API.

5. Prehod API izvede poizvedbo nad registrom, v kateri zahteva direkten dostop do mikrostoritve.

6. Register odgovori z množico naslovov vseh mikrostoritev Stranka, registriranih v registru.

7. Prehod API izmed množice mikrostoritev izbere eno in ji posreduje klic, ki ga je prejel od odjemalca.

Opisani primer prikazuje slika 2.3.



Slika 2.3: Scenarij odkrivanja storitev s prehodom API

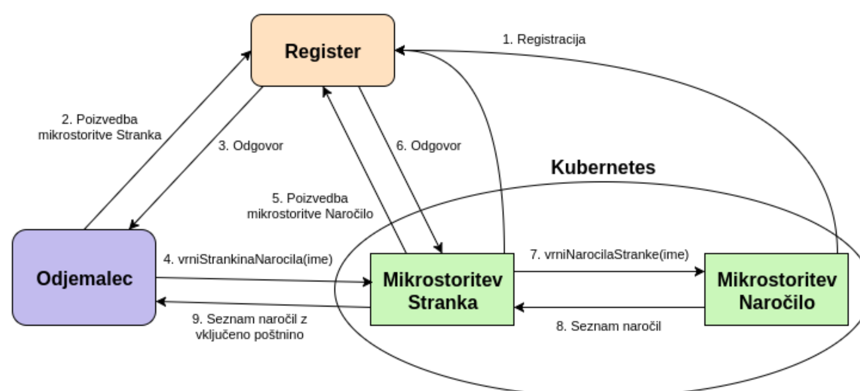
2.4.2 Uporaba orkestracijskih orodij

Za mikrostoritve je značilno, da jih poganjamo v obliki vsebnikov. V tem primeru je možna uporaba orkestracijskih orodij (npr. Kubernetes), ki olajšajo postavitev, nadzor in skaliranje mikrostoritev. Za ta orodja je značilno, da je dostop do storitev omogočen preko dveh naslovov. Notranji naslov je dosegljiv znotraj gruče, zunanji pa izpostavlja mikrostoritev izven nje. V primeru uporabe orkestracijskih orodij si ne želimo, da bi mikrostoritve v gruči ena do druge dostopale preko zunanjega naslova, saj s tem povzročimo dodaten pribitek pri zakasnitvi klica. Kljub temu pa mora biti mikrostoritev zunanjim

odjemalcem dostopna preko zunanjega naslova. Scenarij uporabe orkestracijskih orodij ponazarja naslednji primer, v katerem sta dve mikrostoritvi pognani v gruči Kubernetes, odjemalec pa do ene izmed njih dostopa izven gruče:

1. Mikrostoritvi Stranka in Naročilo, ki tečeta v gruči Kubernetes, se ob zagonu vpišeta v register.
2. Odjemalec izven gruče želi prejeti seznam naročil določene stranke. Začne z izvedbo poizvedbe nad registrom, kjer zahteva naslov mikrostoritve Stranka.
3. Register odgovori z zunanjim naslovom mikrostoritve Stranka.
4. Odjemalec izvede klic nad odkrito mikrostoritvijo.
5. Klicana mikrostoritev mora za uspešno izvedbo odjemalčevega klica izvesti klic na storitev Naročilo, zato izvede poizvedbo nad registrom, v kateri zahteva naslov potrebovane mikrostoritve.
6. Register odgovori z notranjim naslovom mikrostoritve Naročilo.
7. Mikrostoritev Stranka izvede klic vrniNaročilaStranke() nad mikrostoritvijo Naročilo, kjer zahteva seznam vseh naročil željene stranke.
8. Mikrostoritev Naročilo ji vrne zahtevan seznam.
9. Mikrostoritev Stranka vsem prejetim naročilom doda znesek poštnine (poštnina je vezana na stranko) in odjemalcu vrne rezultat.

Opisani primer prikazuje slika 2.4.



Slika 2.4: Scenarij odkrivanja storitev s storitvami v gruči Kubernetes

Poglavje 3

Registri za odkrivanje storitev

Pri konceptu odkrivanja storitev se pojavlja vprašanje, kam naslove storitev sploh shranjevati. Uporaba običajnih podatkovnih baz je neprimerna, saj te niso optimizirane za reševanje problema shranjevanja naslovov storitev in ne ponujajo ključnih funkcionalosti registra za odkrivanje storitev. Registri za odkrivanje storitev so tipično porazdeljene narave, saj le tako dosežemo odpornost na napake, ki je pri arhitekturi mikrostoritev bistvenega pomena. Podpirati morajo tudi mehanizem, ki v primeru padca instance njen naslov izbriše iz shrambe.

V tem poglavju so predstavljena orodja, ki so primerna za uporabo kot register za odkrivanje storitev, in njihova primerjava.

3.1 etcd

etcd je porazdeljena shramba parov ključ-vrednost. Je odprtokodno orodje, za razvoj katerega skrbi podjetje CoreOS. etcd uporablja protokol Raft za ohranjanje konsistentnosti med posameznimi vozlišči [2].

Raft je protokol, ki rešuje problem porazdeljenega soglasja. Algoritem porazdeljenega soglasja zagotavlja, da se v gruči nezanesljivih procesorjev, ki lahko predlagajo vrednost, izbere natanko ena izmed predlaganih vrednosti. Za doseg porazdeljenega soglasja je zahtevano, da vsi pravilno delujoči proce-

sorji izberejo enako vrednost, da je bila izbrana vrednost predlagana s strani pravilno delujočega procesorja ter da pravilo delujoči procesorji sčasoma izberejo vrednost. Protokol Raft omogoča doseg soglasja o vrstnem redu pojavitve dogodkov med vozlišči v nepredvidljivi porazdeljeni gruči [17, 18].

Gruča etcd vozlišč deluje pravilno, če pravilno deluje več kot polovica vozlišč v celotni gruči. To pomeni, da je gruča treh vozlišč odporna na odpoved enega vozlišča, gruča petih vozlišč na odpoved dveh vozlišč itd. Po priporočilih razvijalske ekipe etcd se gruče etcd v produkcijskih okoljih postavlja z lihim številom vozlišč.

etcd je popularna rešitev za konfiguracijo in koordinacijo mikrostoritev. Kot ključno komponento ga uporablja tudi orkestracijsko orodje Kubernetes, ki shrambo uporablja za shranjevanje podatkov o gruči strežnikov Kubernetes [7]. Zaradi odpornosti na napake, možnosti naročanja na spremembe ključev ter koncepta življenjske dobe, je etcd primeren za uporabo kot register pri odkrivanju storitev.

etcd ponuja dva vmesnika, preko katerih dostopamo do porazdeljene shrambe. To sta vmesnika etcd v2 in etcd v3, opisana v nadaljevanju.

3.1.1 etcd v2

Ključi so pri vmesniku etcd v2 hranjeni v hierarhični obliki. Nivoji v strukturi ključev se imenujejo direktoriji in so ločeni s poševnicami [3].

etcd v2 izpostavlja programski vmesnik protokola za izmenjavo hiperteksta (angl. Hyper-Text Transfer Protocol, HTTP). Preko njega so dosegljive možnosti za branje, pisanje in brisanje ključev. Omogočeno je ustvarjanje in brisanje direktorijev ter pridobivanje seznama vsebine direktorija. Vsa komunikacija preko vmesnika HTTP uporablja protokol objektnega zapisa JavaScript (angl. JavaScript Object Notation, JSON) za opis objektov.

Ključna funkcionalnost vmesnika etcd v2 je sledenje sprememb ključev. Preko vmesnika HTTP se klicatelj lahko prijavi na sledenje sprememb ključa ali direktorija. Ob spremembi ključa etcd sporoči klicatelju novo vrednost. Sledenje spremembam je implementirano s pomočjo indeksa, ki se ob vsakem

dogodku v shrambi poveča. Ob vsakem klicu za branje iz shrambe etcd poleg vrednosti vrne tudi indeks, ki označuja stanje, v katerem se je branje izvedlo. Pri prijavi na sledenje sprememb klicatelj priloži prejeti indeks novi zahtevi, kar pomeni, da želi prejeti spremembo ključa, ki je stopila v veljavo po poslanem indeksu.

etcd v2 omogoča specifikacijo življenjske dobe (angl. Time-to-Live, TTL) ključa. Življenjska doba je lahko nastavljena na posamezen ključ ali pa na celoten direktorij. Ob poteku življenjske dobe ključa se ključ in njegova pripadajoča vrednost v shrambi izbrišeta. V primeru, da je življenjska doba nastavljena na direktorij, se izbrišejo vse vrednosti, ki pripadajo temu direktoriju. Življenjska doba se preko vmesnika HTTP lahko osveži.

Vmesnik etcd v2 ne podpira transakcij, podpira pa enostavni atomarni operaciji „primerjaj in zamenjaj" (angl. Compare and Swap, CAS) in „primerjaj in izbriši" (angl. Compare and Delete, CAD). Operacija CAS omogoča pogojno menjavo ključa. Pogoj za menjavo je lahko ta, da ključ pred pisanjem ne obstaja, da je vrednost ključa pred pisanjem nastavljena na podano vrednost ali pa da je indeks, ki označuje, kdaj je bila vrednost nazadnje spremenjena, enak podani vrednosti. Operacija CAD omogoča pogojno brisanje ključa. Pri tej operaciji sta lahko pogoja preverjanje prejšnje vrednosti ključa ali pa preverjanje indeksa spremembe prejšnjega ključa. Vmesnik ponuja še atomarno ustvarjanje urejenih ključev. Tako ustvarjenemu ključu etcd določi ime, ki ima na zadnjem nivoju hierarhije števílsko vrednost. etcd zagotavlja, da ima tak ključ na zadnjem nivoju večjo vrednost kot ključi, ustvarjeni pred njim. Funkcionalnost ustvarjanja urejenih ključev je uporabna, ko je treba ustvariti vrsto ključev, ki morajo biti procesirani po vrstnem redu.

3.1.2 etcd v3

Vmesnik etcd v3 se je razvil zaradi potrebe po rešitvi problemov vmesnika etcd v2. Ti problemi vključujejo slabo učinkovitost protokola JSON in slabo učinkovitost naročanja na spremembe v registru [21, 4].

Vmesnik etcd v3 omogoča komunikacijo preko protokola klicev oddaljenih

postopkov gRPC (angl. gRPC Remote Procedure Calls, gRPC). Sporočila protokola gRPC so kodirana binarno in omogočajo dvakratno performančno izboljšavo procesiranja sporočil etcd v3 v primerjavi s protokolom JSON, ki ga uporablja vmesnik etcd v2. Poleg tega protokol gRPC omogoča multipleksiranje podatkovnih tokov preko ene povezave protokola za nadzor prenosa (angl. Transmission Control Protocol, TCP). etcd v3 to lastnost uporablja za multipleksiranje zahtev enega odjemalca preko ene same povezave. To predstavlja občutno izboljšavo v primerjavi z vmesnikom etcd v2, ki za vsako zahtevo potrebuje novo vzpostavitev povezave TCP.

Vmesnik etcd v3 implementira koncept življenjske dobe s pomočjo tako imenovanih žetonov „lease“. Vsak tak žeton ima življenjsko dobo, ki jo preko vmesnika lahko osvežujemo. Na žeton se lahko veže en ali več ključev. Ob izteku življenjske dobe žetona se vsi ključi, ki so vezani nanj, izbrišejo. Pri vmesniku etcd v2 je treba življenjsko dobo ključa osveževati za vsak ključ posebej. Vmesnik etcd v3 z uporabo žetona „lease“ zmanjšuje potreben promet v primeru, da en odjemalec vzdržuje življenjsko dobo več ključev.

Vmesnik etcd v3 uporablja prenovljen model za shrambo ključev, ki izboljšuje shrambo sprememb ključev. Nov podatkovni model zaradi sprememb v načinu shranjevanja ključev opušča hierarhično zgradbo ključev. Namesto tega ponuja iskanje ključev v intervalu, ki omogoča zahtevo vseh ključev s podano predpono. Ta način v kombinaciji z uporabo dobro zamišljenega modela za določanje imen ključev omogoča podobno funkcionalnost, kot jo ponuja vmesnik etcd v2 s pridobivanjem seznama vsebine direktorija.

Poleg omenjenih funkcionalnosti vmesnik etcd v3 razširja model sočasnega izvajanja vmesnika etcd v2 z uporabo transakcij. Transakcija vmesnika etcd v3 razširja funkcionalnost operacij CAS in CAD vmesnika etcd v2. Transakcija omogoča specifikacijo konjunkcije pogojev, seznama operacij, ki naj se izvede, če so vsi pogoji izponljeni, ter seznama operacij, ki naj se izvede, če je kateri izmed pogojev neizpolnjen.

3.2 Consul

Consul je orodje za odkrivanje in konfiguracijo storitev. Consul je odprtokodno orodje, za njegov razvoj pa skrbi podjetje Hashicorp. Poleg odkrivanja storitev ponuja še sisteme za preverjanje vitalnosti, ki se lahko navezujo na storitev ali pa na vozlišče samo. Za nameme konfiguracije storitev Consul ponuja shrambo ključ-vrednost. Poleg tega pozna koncept podatkovnih centrov in podpira komunikacijo med njimi brez potreb po dodatni abstrakciji. Consul svoje funkcionalnosti izpostavlja preko programskega vmesnika HTTP ali vmesnika sistema domenskih imen (angl. Domain Name System, DNS) [1].

Consul je porazdeljen in visoko dostopen sistem, sestavljen iz agentov Consul in strežnikov Consul. Agent Consul, ki tipično teče poleg posamezne storitve, skrbi za preverjanje vitalnosti storitev. Agenti komunicirajo s strežniki Consul, ki skrbijo za shrambo in replikacijo podatkov. Strežniki Consul so, po priporočilu razvijalcev, postavljeni v gručah od treh do petih vozlišč. Za komunikacijo med agenti Consul in strežniki Consul skrbi protokol „gossip“, ki je bil razvit pod istim podjetjem kot Consul.

Shramba parov ključ-vrednost je dostopna preko vmesnika HTTP, izpostavljenega na strežnikih Consul in agentih Consul. Shrambo je mogoče uporabljati kot tako, lahko pa uporabljamo funkcionalnosti, ki jih Consul gradi nad njo. Vmesnik ponuja klasične funkcionalnosti branja, pisanja in posodabljanja ključev, ponuja pa tudi naročanje na spremembe ključev. Shramba parov ključ-vrednost ponuja tudi transakcije, v katerih lahko atomarno spreminjamo več ključev. V transakcijah sta podprti tudi operaciji CAS in CAD. Funkcionalnost transakcij izpostavlja vmesnik HTTP.

Ena izmed funkcionalnosti, ki jih Consul gradi na svoji shrambi parov ključ-vrednost, je odkrivanje storitev. Storitve se v sistem Consul registrirajo z imenom storitve. Consul omogoča dodajanje informacij o storitvi s sistemom označb. Na vsako storitev se lahko veže več preizkusov vitalnosti storitve, ki določajo, ali storitev z vidika sistema Consul deluje pravilno ali ne.

Odkrivanje storitev poteka preko vmesnika HTTP, ki teče na strežnikih

Consul in agentih Consul. Vmesnik HTTP omogoča pridobivanje vitalnih storitev, s pomočjo brskanja po katalogu registriranih entitet pa omogoča tudi napredno filtriranje storitev. Poleg vmesnika HTTP je na agentih Consul omogočen tudi vmesnik DNS, preko katerega so storitve izpostavljene pod naslovom, ki vsebuje ime storitve ter ime podatkovnega centra, v katerem storitev teče. Poleg tega vmesnik DNS podpira tudi filtriranje storitev glede na njihove označbe. Sistem Consul poleg omenjenih načinov odkrivanja storitev podpira še predpripravljene poizvedbe, ki se izvedejo ob poizvedbi DNS, ki ustreza regularnemu izrazu. Ustvarjanje predpripravljenih poizvedb poteka preko vmesnika HTTP, definirane pa so z regularnim izrazom, ob katerem se prožijo, in z normalno poizvedbo HTTP. Predpripravljene poizvedbe v vseh poljih omogočajo tudi interpolacijo nizov, pridobljenih iz regularnega izraza. Predpripravljene poizvedbe izpostavljajo polno moč vmesnika HTTP preko vmesnika DNS.

Consul ponuja različne načine preverjanja vitalnosti storitev. Vitalnost storitve se lahko preverja s periodičnim preverjanjem rezultata skripte na sistemu, na katerem je zagnan agent Consul. Naslednji način preverjanja je periodično preverjanje rezultata poizvedbe HTTP ali pa preverjanje uspešnosti vzpostavljanja povezave TCP. Zadnji način je preverjanje s pomočjo življenjske dobe storitve. Pri tem načinu mora storitev sama poslati zahtevo HTTP agentu Consul, preden se življenjska doba izteče. Če storitev tega ne stori, se ta smatra kot nedelujočo. Ob zahtevi za branje registriranih storitev Consul skrbi za filtriranje nedelujočih storitev in vrača samo storitve, ki so uspešno opravile vse preizkuse vitalnosti.

Consul nad sistemi za preverjanje vitalnosti ponuja še koncept sej. Ustvarjena seja se poveže na obstoječe preizkuse vitalnosti in je aktivna, dokler so vsi povezani preizkusi uspešni. Z uporabo seje je omogočeno zaklepanje ključev v shrambi ključ-vrednost, s pomočjo katerega lahko implementiramo funkcionalnost porazdeljenega zaklepanja. Ob ustvarjanju seje je možno določiti, kaj se zgodi s ključi ob poteku seje. Ena izmed možnosti je, da se ključi izbrišejo, druga pa, da se sprosti zaklep vseh ključev.

Consul podpira postavitev sistema v več podatkovnih centrih. V vsakem podatkovnem centru mora biti prisotna gruča strežnikov Consul. Podatki se med podatkovnimi centri ne replicirajo. Ko strežnik Consul prejme zahtevo za podatek, ki se nahaja v drugem podatkovnem centru, jo ta posreduje strežniku Consul v podatkovnem centru, katerega podatek potrebuje. Ta zahtevo izvrši in rezultat vrne strežniku prvotnega podatkovnega centra, ta pa jo končno vrne odjemalcu.

Orodje Consul vključuje svoj spletni uporabniški vmesnik, do katerega lahko dostopamo s spletnim brskalnikom. Uporabniški vmesnik ponuja pregled nad stanjem vozlišč in registriranih storitev, pregled in spreminjanje shrambe parov ključ-vrednost ter nadzor nad politikami kontrole dostopa.

3.3 ZooKeeper

ZooKeeper je orodje za koordiniranje procesov porazdeljene aplikacije. Razvit je bil s strani fundacije Apache in je bil sprva podprojekt ogrodja Hadoop, zdaj pa se razvija kot samostojni projekt. ZooKeeper ponuja enostavno ogrodje za ustvarjanje kompleksnih koordinacijskih primitivov na strani odjemalca. S temi primitivi lahko rešujemo probleme porazdeljenih aplikacij, kot so skupinska komunikacija ter porazdeljeno zaklepanje in deljenje podatkov [11, 19].

ZooKeeper je porazdeljen sistem, sestavljen iz gruče strežnikov. Vsak strežnik implementira shrambo, podobno normalnemu datotečnemu sistemu. Vsak element v shrambi je definiran z enoličnim imenom, ki predstavlja pot do elementa. Pot je sestavljena iz elementov, ločenih s poševnico. Vsak element v shrambi lahko hrani podatek maksimalne velikosti enega megabajta. Strežnik ZooKeeper shrambo hrani v pomnilniku, ta pa je replicirana preko cele gruče.

Odjemalci do sistema dostopajo preko trajne povezave TCP do enega strežnika. Zahteve za branje se procesirajo direktno na strežniku, na katerega je povezan odjemalec. Zahteve za pisanje se posredujejo celotni gruči

strežnikov in se upoštevajo šele takrat, ko celotna gruča doseže soglasje o pisanju. Vse zahteve za pisanje so ob dosegu soglasja oštevilčene z enoličnim identifikatorjem „zxid“, s pomočjo katerega ZooKeeper ohranja konsistentnost podatkov. ZooKeeper z uporabo tega identifikatorja podpira tudi zahtevo za naročanje na spremembe elementov v shrambi. Ob spremembi elementa v shrambi strežnik ZooKeeper posreduje informacijo o spremembi vsem naročenim odjemalcem, ki so povezani nanj. Identifikator „zxid“ skrbi za to, da se spremembe med zahtevami ne izgubljajo.

ZooKeeper podpira tudi tako imenovane „Ephemeral“ elemente. Tak element ima življenjsko dobo seje odjemalca, ki ga je ustvaril. V primeru, da odjemalec zaključi sejo ali pa se seja zaključi zaradi prekinjene povezave TCP, se tak element iz shrambe izbriše.

3.4 Eureka

Eureka je odprtokodno orodje specifično namenjeno reševanju problema odkrivanja stortev. Razvito je bilo s strani podjetja Netflix [5, 24].

Eureka je porazdeljen sistem, katerega gruče so sestavljene iz strežnikov Eureka. Vsak strežnik ima svoj register, v katerega se storitve registrirajo, med strežniki v gruči pa je ta register repliciran. Vsak strežnik v gruči izpostavlja programski vmesnik HTTP, preko katerega so funkcionalnosti odkrivanja storitev dostopne. Vmesnik HTTP je oblikovan po principu REST, vsako sporočilo pa je kodirano kot objekt v obliki protokola JSON ali razširljivega označevalnega jezika (angl. Extensible Markup Language, XML).

Registracija storitev implementira koncept življenjske dobe. Storitev se ob zagonu registrira na strežnik Eureka, nato pa v konfigurabilnem intervalu registracijo osvežuje. Če storitev svoje registracije ne uspe osvežiti, jo strežnik Eureka po intervalu, ki je prav tako konfigurabilen, izbriše iz registra.

Za pomoč pri dostopanju do vmesnika HTTP Eureka ponuja implementacijo odjemalca v obliki modula Java. Ta poenostavlja registracijo in odkriva-

nje storitev. Odjemalec ponuja tudi možnost lokalnega repliciranja registra v internem pomnilniku, ki ga odjemalec uporabi v primeru nedostopnosti strežnikov Eureka. Odjemalec na določen interval osvežuje vsebino internega pomnilnika.

Eureka za potrebe povečane odpornosti na napake uporablja koncept samozaščite. Strežnik Eureka pozna število zahtev za osveževanje, ki jih mora prejeti od registriranih storitev v določenem intervalu. Če število zahtev v tem intervalu pade pod konfigurabilen prag, strežnik Eureka preide v stanje samozaščite. V tem stanju strežnik preneha z brisanjem pretečenih storitev v registru, da zaščiti trenutno informacijo o registriranih storitvah. Ko število zahtev spet preseže prag, strežnik Eureka preide iz stanja samozaščite nazaj v normalno stanje.

Eureka je bila primarno razvita za potrebe odkrivanja storitev v oblaku Amazon Web Services (AWS) in vključuje optimizacije, ki izboljšujejo učinkovitost delovanja v njem. Eureka pozna koncept regij AWS in pri odkrivanju storitev daje prednost tistim storitvam, ki tečejo v istih regijah.

3.5 Primerjava

Vsi opisani sistemi so porazdeljeni. S tem je dosežena odpornost na napake v primeru padcev strežnikov.

Skupna lastnost registrov etcd, Consul in ZooKeeper je ta, da implementirajo shrambo parov ključ-vrednost. Vsi imajo ključe razporejene v obliki hierarhije, opušča jo le etcd pri novem vmesniku etcd v3, ki pa namesto pridobivanja seznama direktorija omogoča pridobivanje seznama vseh ključev z isto predpono. Vsi omenjeni registri konsistentnost ohranjajo s pomočjo algoritma za dosego porazdeljenega soglasja. Soglasje je doseženo, če je dosegljiva več kot polovica strežnikov v gruči. Zato je za normalno delovanje omenjenih registrov potrebna dosegljivost več kot polovice strežnikov v gruči. Vsako pisanje v shrambo je oštevilčeno, sistemi pa hranijo tudi zgodovino sprememb ključev. Ta lastnost omogoča funkcionalnost naročanja na spre-

membe ključev.

ZooKeeper in etcd ne poznata koncepta storitve kot take, ponujata pa dobro izhodišče za izgradnjo sistema za odkrivanje storitev. Za razliko od njih, registra Consul in Eureka koncept storitve poznata, kar olajša implementacijo na strani odjemalca. Consul sistem za odkrivanje storitev gradi na svoji shrambi parov ključ-vrednost, doda pa ji bogato podporo za preverjanje vitalnosti storitev in povezovanja preko več podatkovnih centrov. Eureka koncept storitve pozna, saj je bila zgrajena izključno za reševanje problema odkrivanja storitev.

Izmed opisanih registrov najbolj izstopa sistem Eureka, ki namesto konsistentnosti poudarek daje dostopnosti. Strežniki Eureka ne uporabljajo porazdeljenega soglasja za repliciranje podatkov in omogočajo zapisovanje novih registracij tudi v primeru, da strežnik izgubi dostop do ostalih strežnikov v gruči. Eureka ne hrani zgodovine pisanja in zato ne podpira naročanja na spremembe. Odjemalec sicer lahko replicira register v svojem pomnilniku, a bo od strežnika ob zahtevi za posodobitev registra prejel samo spremembe med trenutnim stanjem na strežniku in stanjem na strežniku ob prejšnji zahtevi za posodobitev, ne pa tudi zgodovine teh sprememb.

Primerjavo registrov za odkrivanje storitev povzema tabela 3.1.

	etcd v2	etcd v3	Consul	ZooKeeper	Eureka
Vmesnik za dostop	HTTP + JSON	gRPC	HTTP + JSON, DNS	Trajna povezava TCP	HTTP + JSON/ XML
Shramba parov ključ-vrednost	Da	Da	Da	Da	Ne
Koncept storitve	Ne	Ne	Da	Ne	Da
Algoritem za doseg porazdeljenega soglasja	Da	Da	Da	Da	Ne
Sistem za preverjanje vitalnosti storitve	Življenjska doba ključa	Žeton z življenjsko dobo	Sistemi za preverjanje vitalnosti	„Ephemeral“ elementi	Življenjska doba storitve

Tabela 3.1: Podobnosti in razlike med registri za odkrivanje storitev

Poglavje 4

Implementacija modula za odkrivanje mikrostoritev

V okviru diplomskega dela smo razvili modul za ogrodje KumuluzEE, ki omogoča odkrivanje storitev. KumuluzEE je odprtokodna implementacija ogrodja za razvoj mikrostoritev v Javi. Uporablja specifikacijo Java EE in omogoča enostaven prehod iz monolitne arhitekture na arhitekturo mikrostoritev [9].

Cilj diplomskega dela je razvoj modula za dinamično odkrivanje mikrostoritev, ki bo razvijalcem omogočal enostavno uporabo registra za odkrivanje storitev preko javanskih anotacij. Razvijalcem mora biti omogočena enostavna izbira implementacije registra. Modul mora imeti podporo za uporabo prehoda API in omogočati enostavno integracijo z orkestracijskimi orodji, ki so za arhitekturo mikrostoritev značilna.

Osnovno delovanje modula za odkrivanje storitev je podprto z dvema anotacijama. Anotacija `DiscoverService` s pomočjo tehnologije za dodajanje konteksta in vključevanje odvisnosti platforme Java (angl. Contexts and Dependency Injection, CDI) v spremenljivko vrine naslov storitve, anotacija `RegisterService` pa registrira storitev v register. Anotacija za odkrivanje storitev omogoča vrinjanje v spremenljivke tipa String, URL ter WebTarget. Anotacija za registracijo storitev deluje na aplikacijskih razre-

dih `Application` vmesnika API za storitve REST platforme Java (angl. Java API for RESTful Services, JAX-RS).

Modul omogoča odkrivanje in registracijo storitev v več delovnih okoljih ter pod različnimi verzijami. Za registracijo in odkrivanje storitve potrebujemo ime storitve, okolje, v katerem se izvaja, ter njeno verzijo. Odkrivanje storitev podpira semantično verzioniranje v stilu upravitelja paketov Node (angl. Node Package Manager, NPM), ki omogoča izbiro največje prisotne verzije v podanem intervalu. S tem je omgočeno samodejno nadgrajevanje storitev.

Razvit modul ima podporo za prehode API, ki omogoča dva načina za dostop do storitve. Prvi način je direktni dostop do storitve, drugi način pa je dostop preko prehoda, katerega naslov je določen s ključem v registru. Če aplikacija zahteva dostop do storitve preko prehoda API, naslov tega pa v registru ni določen, modul vrne naslov storitve. Če je naslov prehoda API med delovanjem aplikacije dodan, odstranjen ali spremenjen, se odkrivanje storitev avtomatsko prilagodi novim spremembam.

Zasnova modula za odkrivanje storitev je modularna in omogoča podporo več produktov za odkrivanje storitev. V modulu za odkrivanje storitev smo podprli dva produkta za odkrivanje storitev - etcd in Consul. Izbira produkta za odkrivanje storitev je implementirana z izbiro odvisnosti. Za uporabo modula, ki za shrambo storitev uporablja etcd, je potrebna odvisnost na modul `com.kumuluz.ee.discovery.kumuluzee-discovery-etcd`, za uporabo modula, ki uporablja Consul, pa je potrebna odvisnost na modul `com.kumuluz.ee.discovery.kumuluzee-discovery-consul`. Obe implementaciji povezuje skupni modul, ki definira anotacije za registracijo in odkrivanje storitev ter skupne metode, ki jih uporabljate obe implementaciji.

Skupni modul definira vmesnik `DiscoveryUtil`, ki ga morajo podpreti vse implementacije. Ta definira naslednje metode:

- `register(String serviceName, String version, String environment, long ttl, long pingInterval, boolean singleton)`
  Ta metoda omogoča registracijo storitve. Ime storitve je podano s pa-

rametrom `serviceName`, njena verzija s parametrom `version` okolje, v katerem se storitev izvaja, pa s parametrom `environment`. Parameter `ttl` določa življenjsko dobo zapisa storitve, po izteku katere se storitev smatra za nedosegljivo, v primeru da storitev v tem intervalu svojega zapisa ne osveži. Parameter `pingInterval` določa interval, po katerem storitev znova osveži svoj zapis. Metoda ne vrne ničesar.

- Metoda `deregister()` iz registra izbriše vse registracije storitev, ki so bile registrirane z metodo `register()`. Metoda ne vrne ničesar.

- Metoda `getServiceInstances(String serviceName, String version, String environment, AccessType accessType)` se uporablja za odkrivanje vseh storitev z imenom `serviceName` in verzijo `version`, ki se izvajajo v okolju `environment`. Parameter `accessType` določa način dostopa do storitve. `AccessType.DIRECT` zahteva direktni dostop do storitve, `AccessType.GATEWAY` pa zahteva dostop preko prehoda API. Metoda vrne seznam naslovov URL.

- Metoda `getServiceInstance(String serviceName, String version, String environment, AccessType accessType)` je podobna prejšnji, le da ta vrne le eno instanco podane storitve.

- Metoda `getServiceVersions(String serviceName, String environment)` vrne seznam vseh verzij storitve z imenom `serviceName` v okolju `environment`, ki so na voljo.

- Metoda `disableServiceInstance(String serviceName, String version, String environment, URL url)` označi podano instanco storitve kot neaktivno. Neaktivne storitve se lahko še vedno normalno izvajajo, modul za odkrivanje storitev pa jih ne bo več vračal.

Skupni vmesnik `DiscoveryUtil` je implementiran kot zrno CDI in ga razvijalci lahko uporaljajo v namene uporabe registracije in odkrivanja storitev. Alternativa uporabi skupnega vmesnika je uporaba anotacij. Anotacija `RegisterService` uporablja skeniranje anotacij v času prevajanja.

Ob prevajanju programa prevajalnik preišče cel projekt in lokacije anotacij `RegisterService` zapiše v datoteko. Ta datoteka se ob izvajanju prebere, nato pa se nad vsemi odkritimi anotacijami pokliče metoda skupnega vmesnika, ki storitev registrira. Anotacija `DiscoverService` je implementirana kot proizvajalec CDI (angl. CDI producer) in podpira vstavljanje naslovov storitev v polja tipov URL, String ali WebTarget. Ob vstavljanju naslovov v polja se za izbiro naslova mikrostoritev uporabi metoda skupnega vmesnika, ki odkrije naslov storitve.

V nadaljevanju poglavja sta podrobno opisani implementaciji vmesnika z uporabo registra etcd in registra Consul.

4.1 etcd implementacija

etcd implementacija modula za odkrivanje storitev uporablja porazdeljeno shrambo parov ključ-vrednost etcd. Za dostop do shrambe je uporabljen vmesnik etcd v2. Implementacija je v naslednjem modulu:

```
com.kumuluz.ee.discovery.kumuluzee-discovery-etcd
```

Za dostop do shrambe etcd smo uporabili odprtokodno knjižnjico Etcd4j, ki uporablja vmesnik etcd v2.

Vsaka storitev se v shrambo etcd registrira kot direktorij, katerega ime ima naslednjo strukturo:

```
/environments/<delovno okolje>/services/<ime storitve>/
    <verzija storitve>/instances/<id instance>
```

Parametre, ki predstavljajo delovno okolje, ime storitve in verzijo storitve, določi uporabnik bodisi v anotaciji bodisi v konfiguracijskih ključih. Parameter, ki predstavlja id instance je čas registracije storitve v formatu časovnega žiga in se avtomatično ustvari ob registraciji storitve.

Direktorij storitve vsebuje ključe, ki vsebujejo informacije o registrirani storitvi. Ti ključi so naslednji:

- `/url` vsebuje naslov storitve, na katerem je ta dostopna. Če storitev teče v gruči, je to naslov, na katerem je storitev dostopna izven gruče. Naslov določi uporabnik s konfiguracijskim ključem. V primeru da uporabnik tega ne stori, skuša modul ta naslov pridobiti samodejno z uporabo naslovov vmesnikov omrežnih kartic v sistemu.

- `/containerId` vsebuje id gruče, v kateri storitev teče. Če storitev ne teče v gruči, ta ključ ni prisoten.

- `/containerUrl` vsebuje naslov storitve, na katerem je storitev dostopna znotraj gruče. Če storitev ne teče v gruči, ta ključ ni prisoten. V primeru da storitev teče v gruči, modul ta naslov pridobi samodejno z uporabo naslovov vmesnikov omrežnih kartic v sistemu, lahko pa ga uporabnik specificira s konfiguracijskim ključem.

- `/status` ključ ponazarja stanje registrirane storitve. Če je vrednost ključa enaka `disabled`, modul za odkrivanje storitev te storitve ne bo uporabljal. Če ključ ni prisoten, se storitev smatra za aktivno.

Ob registraciji storitve se v shrambi etcd ustvari direktorij storitve. Direktorij je ustvarjen z življenjsko dobo, ki jo uporabnik poda v anotaciji ali v konfiguracijskih ključih. Nato se v direktorij vpišejo omenjeni ključi. Zatem se zažene nit, ki na določen interval obnavlja življenjsko dobo direktorija storitve. Interval obnavljanja življenjske dobe ključa je konfigurabilen preko konfiguracijskih ključev mikrostoritve.

Ob odkrivanju registrirane storitve uporabnik poda ime storitve, delovno okolje, v katerem se storitev izvaja, in njeno verzijo. Modul najprej preveri, če je verzija podana kot interval v stilu NPM. V primeru da je, se izvede poizvedba na shrambo etcd, v kateri so zahtevani vsi ključi pod direktorijem naslednje oblike:

```
/environments/<delovno okolje>/services/<ime storitve>/
```

S tem modul pridobi vse registrirane instance vseh verzij storitve s podanim imenom. Te se shranijo v notranji pomnilnik. Nato modul pošlje

zahtevo za sledenje sprememb v omenjenem direktoriju. V primeru, da etcd obvesti modul, da je prišlo do spremembe, se interni pomnilnik osveži tako, da predstavlja novo stanje registriranih storitev. Modul nadaljuje odkrivanje storitev s tem, da izmed pridobljenih verzij izbere najvišjo, ki ustreza podanem intervalu. Izmed registriranih instanc po principu „round robin" izbere eno in jo vrne uporabniku.

V primeru da verzija ni podana kot interval v stilu NPM, se poizvedba na shrambo etcd ne naredi na nivoju verzij, ampak na nivoju instanc. Direktorij poizvedbe je torej naslednje oblike:

```
/environments/<delovno okolje>/services/<ime storitve>/
    <verzija storitve>/instances/
```

Modul po prejetem odgovoru shrani pridobljene instance v notranji pomnilnik in shrambi etcd pošlje zahtevo za sledenje sprememb direktorija. Podobno kot v primeru verzij se ob spremembah v shrambi etcd notranji pomnilnik osveži. Modul izmed prejetih instanc po principu „round robin" izbere eno in jo vrne uporabniku.

Ko modul prejme nadaljnje zahteve za odkrivanje storitve, najprej preveri, ali je zahtevana storitev v notranjem pomnilniku. Če je ta prisotna, modul ne pošilja novih zahtev na shrambo etcd, ampak izbere naslov instance iz pomnilnika in jo vrne uporabniku.

Modul implementira podporo za prehode API tako, da ob zahtevi za odkrivanje storitve v shrambi poišče ključ naslednje oblike:

```
/environments/<delovno okolje>/services/<ime storitve>/
    <verzija storitve>/gatewayUrl
```

V primeru da je ključ prisoten in zahtevan način dostopa preko prehoda API, modul uporabniku vrne pridobljen naslov prehoda API. V nasprotnem primeru modul vrne naslov storitve, tudi če je zahtevan način dostopa preko prehoda API. Pri pridobivanju naslova prehoda API s shrambe etcd, modul shrambi pošlje zahtevo za sledenje sprememb ključa, ki vsebuje ta naslov.

Naslov se tako v shrambi lahko doda, odstrani ali spremeni in zagnane storitve bodo spremembo upoštevale brez potrebe po ponovnem zagonu storitev.

Modul implementira podporo za poganjanje storitev v gručah s pomočjo identifikatorjev gruč. Uporabnik za vsako gručo v uporabi definira enoličen identifikator. Vsaka storitev, ki se izvaja v gruči, mora imeti definiran identifikator svoje gruče v konfiguracijskem ključu. Storitve, ki se izvajajo izven gruče, tega identifikatorja nimajo. Modul ob odkrivanju registrirane storite preveri, ali sta identifikatorja gruč odkrivajoče storitve in odkrite storitve enaka. Enakost identifikatorjev pomeni, da se obe storitvi izvajata v isti gruči, zato modul odkrivajoči storitvi vrne notranji naslov zahtevane storitve. V primeru neenakosti modul vrne zunanji naslov zahtevane storitve.

Implementacija logike agregacije naslovov mikrostoritev, ki jo modul za odkrivanje storitev uporablja, je prikazana v izseku 4.1. Pred predstavljenim izsekom modul poskrbi, da je notranji pomnilnik, ki vsebuje podatke o zahtevani storitvi, osvežen. Modul najprej preveri, če je v shrambi etcd oziroma notranjem pomnilniku prisoten naslov prehoda API. Če je ta prisoten in je zahtevan dostop preko prehoda API, poleg tega pa je v pomnilniku prisotna vsaj ena instanca zahtevane storitve, modul vrne naslov prehoda API. V nasprotnem primeru modul vrne seznam storitev, ki so prisotne v pomnilniku. Za vsako instanco mikrostoritve modul preveri, če se njen identifikator gruče ujema z identifikatorjem gruče storitve, ki odkrivanje izvaja. Če se ta identifikatorja ujemata, modul v seznam odkritih storitev doda interni naslov mikrostoritve, preko katerega je ta dostopna znotraj gruče. V nasprotnem primeru modul vrne zunanji naslov mikrostoritve.

```
List<URL> instances = new LinkedList<>();

URL gatewayUrl = getGatewayUrl(serviceName, version,
        environment);
if (accessType == AccessType.GATEWAY &&
        gatewayUrl != null &&
        this.serviceInstances.get(serviceName + "_"
```

```
        + version + "_" + environment).size() > 0) {
    instances.add(gatewayUrl);
} else {
    for (Etcd2Service service :
            this.serviceInstances.get(serviceName +
            "_" + version + "_" + environment)
            .values()) {
        if (this.clusterId != null &&
                this.clusterId.equals(
                service.getClusterId())) {
            instances.add(service.getContainerUrl());
        } else {
            instances.add(service.getBaseUrl());
        }
    }
}
return Optional.of(instances);
```

Izsek 4.1: Logika izbire naslova mikrostoritve implementacije etcd

4.2 Consul implementacija

Consul implementacija modula za odkrivanje storitev uporablja orodje za odkrivanje in konfiguracijo storitev Consul. Implementacija je v naslednjem modulu:

`com.kumuluz.ee.discovery.kumuluzee-discovery-consul`

Za dostopanje do programskega vmesnika, ki ga izpostavlja Consul, smo uporabili odprtokodno knjižnico `consul-client`.

Storitve se v sistem registrirajo z imenom, sestavljenim iz imena storitve in okolja, v katerem se ta izvaja. Ime ima naslednjo strukturo:

```
/<delovno okolje>/<ime storitve>
```

Verzija storitve se zapiše kot označba registrirane storitve. Po registraciji storitve modul v sistemu Consul ustvari preizkus vitalnosti storitve, ki zahteva, da storitev na določen interval obnavlja življenjsko dobo storitve. Ustvarjeni preizkus vitalnosti poveže z ustvarjeno storitvijo. Na koncu modul zažene nit, ki obnavlja življenjsko dobo storitve.

Naslov registrirane storitve je pri uporabi sistema Consul določen s strani agenta Consul, na katerem je storitev registrirana. Ker Consul predvideva, da je agent dostopen na istem naslovu kot storitev, ki se vanj registrira, ta ne ponuja možnosti specifikacije naslova storitve ob njeni registraciji. Ob odkrivanju storitve sistem Consul na poizvedbo odgovori z naslovom mikrostoritve, ne poda pa protokola, ki se za dostop do storitve uporablja. Modul za odkrivanje storitev zato v označbo registrirane storitve zapiše, ali je ta dostopna preko protokola HTTP ali HTTPS in to označbo uporabi pri odkrivanju storitev.

Ob odkrivanju storitve modul naredi poizvedbo na sistem Consul, v kateri zahteva vitalne instance storitev z želenim imenom in delovnim okoljem storitve. Zatem od sistema Consul zahteva, naj ga obvešča o spremembah instanc te storitve. Modul iz označb prejetih storitev prebere njihove verzije. Če uporabnik poda verzijo kot interval v stilu NPM, se izmed prejetih verzij izbere najvišja, ki ustreza podanemu intervalu. Po izbiri verzije modul po principu „round robin" izbere instanco storitve in jo vrne uporabniku. Protokol, preko katerega je storitev dostopna, se razbere iz označb storitve.

Implementacija naročanja na spremembe vitalnosti storitve je prikazana v izseku 4.2. Knjižnica, ki smo jo uporabili za dostop do sistema Consul ponuja enostavno naročanje na spremembe preko povratnih klicev. Na začetku implementacija inicializira objekt, ki skrbi za obveščanje sprememb vitalnosti mikrostoritve in specificira metodo, ki naj se ob spremembi izvede. Ta metoda počisti notranji pomnilnik, ki vsebuje podatke o instancah storitev, ki so se spremenile. Nato metoda prebere nove podatke o instancah storitve in jih vpiše v notranji pomnilnik. Po specifikaciji metode se obveščanje o

sprememba zažene.

```
ServiceHealthCache svHealth = ServiceHealthCache
        .newCache(healthClient, serviceKey);

svHealth.addListener(new ConsulCache
        .Listener<ServiceHealthKey, ServiceHealth>() {

   @Override
   public void notify(Map<ServiceHealthKey,
           ServiceHealth> newValues) {

       log.info("Service instances for service " +
               serviceKey + " refreshed.");

       serviceInstances.get(serviceKey).clear();
       serviceVersions.get(serviceKey).clear();

       for (Map.Entry<ServiceHealthKey, ServiceHealth>
               serviceHealthKey : newValues
               .entrySet()) {
           ConsulService consulService = ConsulService
                   .getInstanceFromServiceHealth(
                   serviceHealthKey.getValue());
           if (consulService != null) {
               serviceInstances.get(serviceKey)
                       .add(consulService);
               serviceVersions.get(serviceKey)
                       .add(consulService
                           .getVersion());
           }
       }
```

```
    }
});

try {
   svHealth.start();
} catch (Exception e) {
   log.severe("Cannot start service listener: " +
            e.getLocalizedMessage());
}
```

Izsek 4.2: Naročanje na spremembe vitalnosti storitve implementacije Consul

Modul implementira podporo za prehode API s pomočjo konfiguracijskega dela sistema Consul. Ob zahtevi za odkrivanje storitve modul iz shrambe ključ-vrednost preveri, ali je prisoten ključ naslednje oblike:

```
/environments/<delovno okolje>/services/<ime storitve>/
    <verzija storitve>/gatewayUrl
```

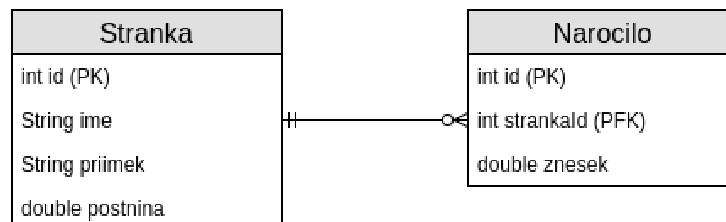
Podobno kot pri etcd implementaciji modul vrne naslov prehoda API v primeru, da je ta prisoten in je zahtevan način prehoda preko prehoda API. V nasprotnem primeru modul vrne naslov storitve.

V tem poglavju smo opisali modul za dinamično odkrivanje storitev, ki smo ga razvili v okviru diplomske naloge. Podprli smo registra etcd in Consul, z modularno zasnovo pa smo omogočili nadaljnji razvoj podpore za ostale registre. Razviti modul omogoča uporabo prehoda API in je optimiziran za uporabo z orkestracijskimi orodji. S tem je omogočena uporaba v sodobnih oblačnih okoljih.

Poglavje 5

Prikaz delovanja

V tem poglavju je predstavljen prikaz delovanja modula za odkrivanje storitev, ki smo ga razvili v okviru diplomske naloge. Delovanje je prikazano skozi aplikacijo, ki jo sestavljata dve mikrostoritvi - mikrostoritev Stranka in mikrostoritev Naročilo. Aplikacija uporablja dve entiteti. Entiteta, ki ponazarja stranko, ima svoj id, ki enolično določa stranko, poleg tega pa vsebuje informacijo o imenu in priimku stranke. Poštnina je za namen prikaza interakcije med mikrostoritvami vezana na stranko in je zato zapisana v atributu strankine entitete. Entiteta, ki ponazarja naročilo, ima id, ki enolično določa naročilo. Poleg tega vsebuje id stranke, na katero se določeno naročilo navezuje. Entiteta Naročilo vsebuje še informacijo, ki ponazarja znesek naročila. Entitetni diagram aplikacije prikazuje slika 5.1.



Slika 5.1: Entitetni diagram aplikacije

Za upravljanje entitete Stranka skrbi mikrostoritev Stranka, za upravljanje entitete Naročilo pa skrbi mikrostoritev Naročilo. Funkcionalnosti, ki jih mikrostoritvi ponujata, so izpostavljene preko vmesnika HTTP, oblikovanega po principu arhitekturnega načina prenašanja stanj (angl. Representational State Transfer, REST). Vsi objekti, ki se pošiljajo preko teles zahtev, so opisani s protokolom JSON. Funkcionalnosti posamezne mikrostoritve so podrobno opisane v nadaljevanju. Za namene prikaza delovanja aplikacije je implementiran preprost odjemalec. Ta uporablja vmesnika HTTP, ki ga mikrostoritvi izpostavljata. Odjemalec je prav tako podrobneje opisan v nadaljevanju. Obe mikrostoritvi in odjemalec so zgrajene nad ogrodjem KumuluzEE.

Aplikacija uporablja orkestracijsko orodje Kubernetes. Kubernetes je odprtokodno orodje za avtomatizacijo postavitve, skaliranja in upravljanja vsebniških aplikacij [8]. Obe mikrostoritvi sta zagnani v pripadajočih vsebovalnikih, upravljanih z orodjem Kubernetes. Odjemalec, ki dostopa do mikrostoritev, je zagnan izven orodja Kubernetes.

5.1 Mikrostoritev Naročilo

Vsaka stranka v aplikaciji ima lahko enega ali več pripadajočih naročil. Za upravljanje entitete naročila skrbi mikrostoritev Naročilo, ki izpostavlja naslednje funkcionalnosti:

- GET /v1/narocila - Vrne vse naročila, ki so vnešena v aplikacijo. Če je prisoten parameter poizvedbe {`strankaId`}, vrne vsa naročila, ki so vezana na stranko s podanim identifikatorjem.

- GET /v1/narocila/narociloId - Vrne naročilo, katerega id je enak podanemu parametru {`narociloId`}.

- POST /v1/narocila/ - V aplikacijo vstavi naročilo, podano v telesu zahteve POST.

- PUT /v1/narocila/narociloId - V aplikacijo vstavi ali prepiše naročilo, katerega id je enak parametru {`narociloId`}. Naročilo je podano v telesu zahteve PUT.
- DELETE /v1/narocila/narociloId - Iz aplikacije izbriše naročilo, katerega id je enak podanemu parametru {`narociloId`}.

Mikrostoritev Naročilo uporablja modul za odkrivanje storitev, s pomočjo katerega se registrira v register etcd. Mikrostoritev za registracijo uporabi anotacijo `RegisterService` nad aplikacijskim razredom JAX-RS. Uporaba je prikazana v izseku 5.1. Na podoben način se v register etcd vpiše tudi mikrostoritev Stranka.

```
@RegisterService
@ApplicationPath("v1")
public class NarociloAplikacija extends Application {
}
```

Izsek 5.1: Implementacija registracije mikrostoritve Naročilo

5.2 Mikrostoritev Stranka

Mikrostroritev Stranka skrbi za upravljanje entitete stranka v aplikaciji. Funkcionalnosti mikrostoritve so naslednje:

- GET /v1/stranke - Vrne vse stranke, ki so vnešene v aplikacijo.
- GET /v1/stranke/strankaId - Vrne stranko, katere id je enak podanemu parametru strankaId.
- POST /v1/stranke/ - V aplikacijo vstavi stranko, podano v telesu zahteve POST.
- PUT /v1/stranke/strankaId - V aplikacijo vstavi ali prepiše stranko, katere id je enak parametru strankaId. Stranka je podana v telesu zahteve PUT.

- DELETE /v1/stranke/strankaId - Iz aplikacije izbriše stranko, katere id je enak podanemu parametru strankaId.
- GET /v1/stranke/strankaId/narocila - Vrne vsa naročila, ki so vezana na stranko, podano v parametru {`strankaId`}. Vsem zneskom je dodan znesek poštnine, ki je vezan na stranko.

Pri mikrostoritvi Stranka je najbolj zanimiv klic, ki vrne vsa naročila stranke z dodanim zneskom poštnine, saj mora ta za uspešno izvedbo izvesti klic na mikrostoritev Naročilo. Implementacija tega klica je prikazana v izseku 5.2. Ob prejetem klicu modul za odkrivanje storitev v spremenljivko `narociloTarget` vstavi naslov mikrostoritve Naročilo. Metoda `vrniStrankinaNarocila` na začetku na standardni izhod izpiše naslov mikrostoritve Naročilo, ki ga je prejela od modula za odkrivanje storitev. Iz poti klica pridobi identifikator stranke, katere naročila so zahtevana. Nato opravi klic nad naslovom mikrostoritve Naročilo, katerega rezultat je seznam vseh naročil podane stranke v aplikaciji. Metoda nato znesku vsakega naročila doda znesek poštnine in vrne spremenjen seznam naročil.

```
@Inject
@DiscoverService(value = "narocilo",
        environment = "dev", version = "1.0.x")
private WebTarget narociloTarget;

...

@GET
@Path("{strankaId}/narocila")
public Response vrniStrankinaNarocila(
        @PathParam("strankaId") int strankaId) {

    System.out.println("Naslov mikrostoritve" +
            "Narocilo: " + narociloTarget.getUri());
```

```
    WebTarget narocilaStrankeTarget = narociloTarget
            .path("v1/narocila")
            .queryParam("strankaId", strankaId);

    List<Narocilo> strankinaNarocila =
            narocilaStrankeTarget
            .request(MediaType.APPLICATION_JSON).get()
            .readEntity(
                new GenericType<List<Narocilo>>() {});

    Stranka s = Baza.vrniStranko(strankaId);
    for(Narocilo n : strankinaNarocila) {
        n.setZnesek(n.getZnesek() +
                s.getPostnina());
    }

    return Response.ok(strankinaNarocila).build();
}
```

Izsek 5.2: Implementacija metode vrniStrankinaNarocila mikrostoritve Stranka

5.3 Odjemalec

Odjemalec, tako kot obe mikrostoritvi, uporablja modul za odkrivanje storitev. Za namen prikaza delovanja aplikacije odjemalec izpostavlja naslednji funkcionalnosti:

- GET /VstaviPodatke - V mikrostoritvi vstavi testne podatke. Podatki, ki jih odjemalec vstavi, so predstavljeni v tabelah 5.1 in 5.2.

- GET /IzvediPoizvedbo - Nad mikrostoritvijo Stranka izvede klic, ki vrne vsa naročila stranke z identifikatorjem 0 z dodano poštnino. Rezultat klica vrne kot rezultat poizvedbe.

id	ime	priimek	postnina
0	Janez	Novak	5
1	Miha	Kočevar	10
2	Ana	Golob	15

Tabela 5.1: Testni podatki, ki jih odjemalec vstavi v mikrostoritev Stranka

id	strankaId	znesek
0	0	10.5
1	0	18.1
2	1	2.3

Tabela 5.2: Testni podatki, ki jih odjemalec vstavi v mikrostoritev Naročilo

Obe funkcionalnosti sta implementirani kot preprosti servlet. Implementacija klica `/IzvediPoizvedbo` je predstavljena v izseku 5.3. Ob prejetem klicu modul za odkrivanje storitev v spremenljivko `strankaTarget` vstavi naslov mikrostoritve Stranka. Odjemalec najprej izpiše naslov mikrostoritve Stranka, ki ga je prejel od modula za odkrivanje mikrostoritev. Nato nad prejetim naslovom izvede klic, ki vrne naročila stranke z identifikatorjem 0 z vključeno poštnino in prejeti rezultat vrne kot rezultat klica.

```
@Inject
@DiscoverService(value = "stranka",
        environment = "dev", version = "1.0.0")
WebTarget strankaTarget;

@Override
```

```
protected void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
    response.setCharacterEncoding("UTF-8");

    System.out.println("Naslov mikrostoritve Stranka: "
            + strankaTarget.getUri());

    WebTarget vrniStrankinaNarocilaTarget =
            strankaTarget
            .path("v1/stranke/{strankaId}/narocila");

    PrintWriter responseWriter = response.getWriter();

    responseWriter.println("Naročila stranke 0 " +
            "(poštnina je vključena v znesek):");
    List<Narocilo> narocila =
            vrniStrankinaNarocilaTarget
            .resolveTemplate("strankaId", 0)
            .request(MediaType.APPLICATION_JSON)
            .get().readEntity(
                new GenericType<List<Narocilo>>() {});

    for(Narocilo n : narocila) {
        responseWriter.printf("ID naročila: %d\t" +
                "Znesek: %f\n", n.getId(),
                n.getZnesek());
    }
}
```

Izsek 5.3: Implementacija klica /IzvediPoizvedbo odjemalca

5.4 Postavitev aplikacije in primer uporabe

Obe mikrostoritvi zapakiramo kot vsebnika. Za pakiranje uporabimo orodje Docker. Opis vsebnika v obliki Dockerfile je enak za obe mikrostoritvi in je predstavljen v izseku 5.4.

```
FROM openjdk:8u131-jre-alpine



COPY target /usr/src/myapp
WORKDIR /usr/src/myapp

EXPOSE 8080

CMD ["java", "-server", "-cp", "classes:dependency/*",
        "com.kumuluz.ee.EeApplication"]
```

Izsek 5.4: Opis vsebnika mikrostoritve Dockerfile

Pri izdelavi vsebnika je uporabljena slika `openjdk:8u131-jre-alpine`. To je slika odprtokodne implementacije Javanskega virtualnega okolja (angl. Java Virtual Machine), zgrajena na sliki minimalnega operacijskega sistema Linux Alpine. Opis vsebnika se nadaljuje s kopiranjem izvorne kode mikrostoritve v vsebnik. Nato se izpostavijo vrata 8080, na katerih je izpostavljen vmesnik HTTP mikrostoritve. Na koncu se izvede ukaz, ki mikrostoritev zažene kot KumuluzEE aplikacijo.

Za nameme prikaza delovanja aplikacije smo uporabili implementacijo orodja Kubernetes Minikube. Minikube je orodje, ki omogoča enostavno postavitev gruče Kubernetes na lokalnem računalniku. Minikube zažene gručo Kubernetes z enim vozliščem v navideznem stroju lokalnega računalnika [10]. Za dostop do gruče Kubernetes smo uporabili orodje kubectl. kubectl je vmesnik za dostop do gruče Kubernetes preko ukazne vrstice [6].

V okolju Kubernetes pred zagonom mikrostoritev ustvarimo instanco registra etcd. To storimo tako, da z orodjem kubectl gruči Kubernetes pošljemo

objekt tipa Deployment, preko katerega gruča Kubernetes skrbi, da je v gruči prisotna natanko ena instanca registra etcd. Nato gruči Kubernetes pošljemo objekt tipa Service, ki gruči Kubernetes naroči, naj vrata vsebnika registra etcd izpostavi na vratih 2379 zunanjega naslova strežnikov v gruči. Vrata registra etcd na zunanjem naslovu izpostavimo zato, ker bo prek njih do registra dostopal odjemalec, ki bo zagnan izven gruče Kubernetes. Poleg tega objekt tipa Service v internem strežniku DNS gruče Kubernetes ustvari zapis, ki vsebuje naslov, preko katerega je register etcd dostopen znotraj gruče. V našem primeru ta zapis vsebuje naslov `http://etcd:2379`.

Pred zagonom mikrostoritev moramo poznati njihov zunanji naslov, preko katerega bodo te izpostavljene izven gruče. Zato gruči Kubernetes pošljemo dva objekta tipa Service, ki ustvarita zunanja naslova, na katerih bosta mikrostoritvi izpostavljeni. Primer omenjenega objekta za mikrostoritev Stranka je podan v izseku 5.5. Podan objekt naroči gruči Kubernetes, naj vrata 8080 vseh vsebnikov, ki imajo označbo `app` nastavljeno na vrednost `stranka`, izpostavi na vratih zunanjega naslova vseh strežnikov v gruči. Vrata, na katerih je orodje Kubernetes izpostavilo mikrostoritev Stranka, pridobimo z ukazom `kubectl describe service stranka`. V našem primeru so to vrata 32481. S podobnim objektom ustvarimo tudi zunanji naslov mikrostoritve Naročilo. V našem primeru je ta izpostavljena na vratih 32310.

```
apiVersion: v1
kind: Service
metadata:
  name: stranka
spec:
  type: NodePort
  ports:
  - port: 8080
    protocol: TCP
  selector:
```

```
    app: stranka
```

Izsek 5.5: Primer objekta Kubernetes tipa Service, ki ustvari zunanji naslov mikrostoritve Stranka

Za zagon mikrostoritev gruči Kubernetes pošljemo dva objekta tipa Deployment, preko katerih orodje Kubernetes zažene vsebnika naših mikrostoritev. Opis objekta je podan v izseku 5.6. Objekt opisuje, naj bo v gruči Kubernetes vedno prisoten en vsebnik mikrostoritve Stranka. Vsakemu vsebniku pred začetkom izvajanja dodamo tri okoljske spremenljivke. Okoljsko spremenljivko `KUMULUZEE_SERVER_BASEURL` modul za odkrivanje storitev uporabi ob registraciji zunanjega naslova mikrostoritve. V našem primeru je to naslov virtualne naprave, na kateri teče strežnik minikube, in vrata, na katerih gruča Kubernetes izpostavlja mikrostoritev Stranka. Okoljska spremenljivka `KUMULUZEE_DISCOVERY_CLUSTER` enolično identificira gručo, v kateri se mikrostoritev izvaja. V našem primeru smo za identifikator izbrali vrednost `minikube`. Okoljska spremenjivka `KUMULUZEE_DISCOVERY_ETCD_HOSTS` vsebuje naslov, na katerem je dostopen register etcd. V našem primeru smo uporabili naslov, ki je vpisan v interni strežnik DNS gruče Kubernetes. Opis objekta na koncu vsebuje informacijo, ki gruči Kubernetes naroči, naj odpre vrata 8080 vsebnika, na katerih je izpostavljen vmesnik HTTP mikrostoritve Stranka. S podobnim objektom tipa Deployment zaženemo tudi mikrostoritev Naročilo.

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: stranka-deployment
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: stranka
```

```
spec:
  containers:
  - image: um7316/stranka:1.0.0
    name: stranka
    env:
      - name: KUMULUZEE_SERVER_BASEURL
        value: http://192.168.99.100:32481
      - name: KUMULUZEE_DISCOVERY_CLUSTER
        value: minikube
      - name: KUMULUZEE_DISCOVERY_ETCD_HOSTS
        value: http://etcd:2379
    ports:
    - containerPort: 8080
      name: server
      protocol: TCP
```

Izsek 5.6: Primer objekta Kubernetes tipa Deployment, ki zažene vsebnik mikrostoritve Stranka

Odjemalca zaženemo na lokalnem računalniku izven gruče. Po inicializaciji odjemalca izvedemo poizvedbo GET nad naslovom `http://localhost:8080/VstaviPodatke`, ki vstavi testne podtake v mikrostoritvi Stranka in Naročilo. Zatem izvedemo poizvedbo GET nad naslovom `http://localhost:8080/IzvediPoizvedbo`, ki nad mikrostoritvijo Stranka izvede zahtevo o naročilih stranke z identifikatorjem 0 in vrne dobljeni rezultat. Rezultat poizvedbe, ki ga vrne odjemalec, prikazuje izsek 5.7.

```
Naročila stranke 0 (poštnina je vključena v znesek):
ID naročila: 0  Znesek: 15.500000
ID naročila: 1  Znesek: 23.100000
```

Izsek 5.7: Izpis poizvedbe `IzvediPoizvedbo` odjemalca

Odjemalec na standardni izhod izpiše naslednji zapis:

```
Naslov mikrostoritve Stranka: http://192.168.99.100:32481
```

Kot je razvidno iz zapisa, je modul za odkrivanje storitev mikrostoritev Stranka odkril na njenem zunanjem naslovu, na katerem je ta dostopna izven gruče. Ob izvedeni poizvedbi je mikrostoritev Stranka na standardni izhod izpisala naslednji zapis:

```
Naslov mikrostoritve Narocilo: http://172.17.0.3:8081
```

Ker sta tako mikrostoritev Stranka kot mikrostoritev Naročilo pognana znotraj iste gruče Kubernetes, je modul za odkrivanje storitev mikrostoritvi Stranka vrnil notranji naslov mikrostoritve Naročilo.

Izvedeni klici predstavljajo delovanje aplikacije in delovanje modula za odkrivanje storitev, ki ga ta uporablja. Modul za odkrivanje storitev odjemalcu ponudi zunanje naslove mikrostoritev, saj je odjemalec pognan izven gruče, mikrostoritve pa znotraj gruče. Pri dostopu mikrostoritve Stranka do mikrostoritve Naročilo modul za odkrivanje storitev vrne notranji naslov mikrostoritve, saj obe mikrostoritvi tečeta znotraj iste gruče.

Poglavje 6

Zaključek

Arhitektura mikrostoritev z izkoriščanjem oblačnih okoljih v razvoj aplikacij prinaša hitrost in agilnost. Med raziskovanjem smo spoznali ključne vidike arhitekture mikrostoritev. Spoznali smo, da je statično dodeljevanje naslovov mikrostoritvam neprimerno in s tem spoznali pomembnost odkrivanja storitev pri uporabi arhitekture mikrostoritev.

V diplomskem delu smo zasnovali in implementirali modul za odkrivanje storitev. Podprli smo dve implementaciji registrov za odkrivanje storitev - etcd in Consul. Modul za odkrivanje storitev podpira napredne scenarije uporabe s prehodom API in je prilagojen in optimiziran za uporabo poleg modernih orkestracijskih orodij, tipičnih za arhitekturo mikrostoritev. Omogočili smo enostavno avtomatsko nadgrajevanje verzij mikrostoritev z uporabo intervalov verzij v stilu NPM. Delovanje modula smo prikazali z implementacijo enostavne aplikacije, obenem pa smo predstavili napredne vidike uporabe modula v orkestracijskih orodjih.

Z implementacijo modula za odkrivanje storitev smo v ogrodju Kumuluz-EE podprli enostavno in intuitivno registracijo in odkrivanje storitev preko javanskih anotacij. Z uporabo dinamičnih naslovov mikrostoritev postane mikrostoritev neodvisna od lokacije izvajanja in s tem primerna za poganjanje v modernih vsebniških okoljih. Razvijalcem smo omogočili intuitivno organizacijo mikrostoritev glede na ime mikrostoritve, okolje, v katerem se

izvaja, ter njeno verzijo. Razvit produkt z modularno zasnovo in skupnim vmesnikom omogoča nadaljnji razvoj z možnostjo podpore ostalih implementacij registrov.

Dinamično odkrivanje storitev, ki smo ga podrobno predelali v diplomskem delu, postaja nepogrešljiv sestavni del mikrostoritev in arhitekture „cloud-native“, saj omogoča dinamično izvajanje storitev v okoljih za orkestracijo vsebnikov. V diplomskem delu smo z razvitim modulom doprinesli k razvoju javanskih ogrodij za mikrostoritve.

Literatura

[1] Consul orodje za odkrivanje in konfiguracijo storitev. Dosegljivo: https://www.consul.io/intro/index.html. [Dostopano: 4. 8. 2017].

[2] etcd porazdeljena shramba parov ključ-vrednost. Dosegljivo: https://github.com/coreos/etcd. [Dostopano: 4. 8. 2017].

[3] etcd2 api, specifikacija programskega vmesnika etcd v2. Dosegljivo: https://coreos.com/etcd/docs/latest/v2/api.html. [Dostopano: 26. 8. 2017].

[4] etcd3 api, specifikacija programskega vmesnika etcd v3. Dosegljivo: https://coreos.com/etcd/docs/latest/v2/api_v3.html. [Dostopano: 26. 8. 2017].

[5] Eureka, orodje za odkrivanje storitev. Dosegljivo: https://github.com/Netflix/eureka. [Dostopano: 23. 8. 2017].

[6] kubectl, vmesnik ukazne vrstice za dostop do gruče kubernetes. Dosegljivo: https://kubernetes.io/docs/user-guide/kubectl-overview/. [Dostopano: 30. 8. 2017].

[7] Kubernetes components, komponente orkestracijskega orodja kubernetes. Dosegljivo: https://kubernetes.io/docs/concepts/overview/components/. [Dostopano: 28. 8. 2017].

[8] Kubernetes orkestracijsko orodje. Dosegljivo: https://kubernetes.io/. [Dostopano: 30. 8. 2017].

[9] Kumuluzee ogrodje za razvoj mikrostoritev. Dosegljivo: `http://ee.kumuluz.com/`. [Dostopano: 9. 6. 2017].

[10] Minikube, orodje za enostavno postavitev gruče kubernetes na lokalnem računalniku. Dosegljivo: `https://github.com/kubernetes/minikube`. [Dostopano: 30. 8. 2017].

[11] Zookeeper, orodje za koordinacijo procesov porazdeljene aplikacije. Dosegljivo: `https://zookeeper.apache.org/`. [Dostopano: 24. 8. 2017].

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

[13] Sandro Brunner, Martin Blöchlinger, Giovanni Toffetti, Josef Spillner, and Thomas Michael Bohnert. Experimental evaluation of the cloud-native application design. In *Utility and Cloud Computing (UCC), 2015 IEEE/ACM 8th International Conference on*, pages 488–493. IEEE, 2015.

[14] Christof Fetzer. Building critical applications using microservices. *IEEE Security & Privacy*, 14(6):86–89, 2016.

[15] The Linux Foundation. Cloud native computing foundation ("cncf") charter. URL: `https://www.cncf.io/about/charter/`, 2015. [Dostopano: 17. 8. 2017].

[16] Martin Fowler and James Lewis. Microservices a definition of this new architectural term. *URL: http://martinfowler.com/articles/microservices.html*, 2014. [Dostopano: 11. 8. 2017].

[17] Heidi Howard. Arc: analysis of raft consensus. Technical report, University of Cambridge, Computer Laboratory, 2014.

[18] Heidi Howard, Malte Schwarzkopf, Anil Madhavapeddy, and Jon Crowcroft. Raft refloated: do we have consensus? *ACM SIGOPS Operating Systems Review*, 49(1):12–21, 2015.

[19] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8, page 9. Boston, MA, USA, 2010.

[20] Nane Kratzke and Peter-Christian Quint. Understanding cloud-native applications after 10 years of cloud computing-a systematic mapping study. *Journal of Systems and Software*, 126:1–16, 2017.

[21] Xiang Li. etcd3 — a new version of etcd from coreos. URL: `https://coreos.com/blog/etcd3-a-new-etcd.html`, 2016. [Dostopano: 26. 8. 2017].

[22] Vijay Rajagopal. Cloudna 2016 keynote: Cloud native application development. In *Local Computer Networks Workshops (LCN Workshops), 2016 IEEE 41st Conference on*, pages xiii–xiii. IEEE, 2016.

[23] Chris Richardson. Pattern: Api gateway / backend for front-end. URL: `http://microservices.io/patterns/apigateway.html`, 2014. [Dostopano: 8. 8. 2017].

[24] Abhijit Sarkar. Spring cloud netflix eureka - the hidden manual. URL: `http://blog.abhijitsarkar.org/technical/netflix-eureka/`, 2017. [Dostopano: 23. 8. 2017].

[25] Matt Stine. Migrating to cloud-native application architectures, 2015.

[26] Johannes Thönes. Microservices. *IEEE Software*, 32(1):116–116, 2015.

[27] 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.

[28] Rishi Yadav. What real cloud-native apps will look like. URL: `https://techcrunch.com/2016/08/03/what-real-cloud-native-apps-will-look-like/`, 2016. [Dostopano: 17. 6. 2017].