Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Domen Jerič

Porazdeljeno spremljanje izvajanja v arhitekturi mikrostoritev

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: prof. dr. Matjaž Branko Jurič

Ljubljana, 2019

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Preučite področje porazdeljenega spremljanja oz. sledenja izvajanja v arhitekturi mikrostoritev. V ta namen preučite področje mikrostoritev in področje spremljanja izvajanja, pri čemer se osredotočite na porazdeljeno spremljanje izvajanja. Identificirajte in opišite načine za spremljanje izvajanja in specifike, ki jih srečamo pri porazdeljenem sledenju ter specifično pri arhitekturi mikrostoritev. Preglejte specifikacijo MicroProfile OpenTracing in dva najbolj znana sistema za porazdeljeno sledenje, Jaeger in Zipkin. Razvijte implementacijo modula za porazdeljeno sledenje mikrostoritvam, ki bo sledila zgoraj navedeni specifikaciji.

Zahvaljujem se mentorju prof. dr. Matjažu Branku Juriču, as. Janu Meznariču in ostalim članom Laboratorija za integracijo informacijskih sistemov za vodenje in strokovno pomoč pri izdelavi diplomske naloge. Zahvala gre tudi moji družini in prijateljem, ki so me pri tem spodbujali.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
SLA	service-level agreement	pogodba med ponudnikom storitve in odjemalcev o standardih razporožljivosti storitve, ki jih mora ponudnik dosegati
HTTP	Hypertext Transfer Protocol	metoda za prenos informacij po spletu
DevOps	Development Operations	sistemski inženiring
I/O	Input/Output	vhod/izhod
RAM	Random access memory	bralno-pisalni pomnilnik
JVM	Java virtual machine	javanski virtualni stroj
API	Application programming interface	aplikacijski programski vmesnik
UDP	User Datagram Protocol	nepovezovalni protokol za prenašanje paketov
REST	Representational State Transfer	reprezentativni prenos stanja
JAX-RS	Java API for RESTful Web Services	javanski programski vmesnik za gradnjo REST spletnih storitev
CDI	Contexts and dependency injection	vstavljanje konstekta in vključitev odvisnosti v Javi

CRUD	Create, read, update, delete	akcije ustvari, preberi, posodobi, pobriši
IaaS	Infrastructure as a Service	infrastruktura kot storitev
PaaS	Platform as a Service	platforma kot storitev
SaaS	Software as a Service	programska oprema kot storitev
CNCF	Cloud Native Computing Foundation	fundacija za oblačno računalništvo
APM	Application performance management	sistemi za spremljanje izvajanja
CI	Continuous integration	neprekinjena integracija
CD	Continuous delivery	neprekinjena dostava

Povzetek

Naslov: Porazdeljeno spremljanje izvajanja v arhitekturi mikrostoritev

Avtor: Domen Jerič

Razvoj aplikacij v arhitekturi mikrostoritev prinaša ogromno prednosti, hkrati pa tudi nekaj novih izzivov. Enega izmed glavnih izzivov predstavlja spremljanje izvajanja. S spremljanjem izvajanja lahko bolje razumemo delovanje sistema, odkrijemo razloge za napake in vzroke za počasno delovanje aplikacije. V diplomski nalogi smo preučili različne pristope za spremljanje izvajanja, pri čemer smo se osredotočili na porazdeljeno sledenje. Ogledali smo si različne sisteme in načine za instrumentacijo aplikacij s porazdeljenim sledenjem. Razvili smo razširitev za odprtokodno ogrodje KumuluzEE – KumuluzEE OpenTracing. Razširitev razvijalcem omogoča, da JAX-RS mikrostoritve enostavno opremijo s funkcionalnostjo porazdeljenega sledenja. Uporabnost razvite razširitve smo prikazali na praktičnem primeru.

Ključne besede: mikrostoritve, spremljanje izvajanja, porazdeljeno sledenje, Jaeger, Zipkin, MicroProfile, OpenTracing, KumuluzEE.

Abstract

Title: Distributed tracing in microservices architecture

Author: Domen Jerič

Microservices architecture brings many advantages over traditional monolithic architectures. Along with the advantages there come new challenges. One of the main challenges is observability. Observability enables us to better understand system behaviour, resolve errors and analyze performance bottlenecks. In the diploma thesis we studied different approaches for application observability. The main focus of this thesis was distributed tracing. We looked at different systems and ways to instrument microservices with distributed tracing. We developed an extension for the KumuluzEE framework – KumuluzEE OpenTracing. Extension enables us to instrument JAX-RS microservices with distributed tracing, which we later demonstrate on a practical example.

Keywords: microservices, observability, distributed tracing, Jaeger, Zipkin, MicroProfile, OpenTracing, KumuluzEE.

Poglavje 1

Uvod

1.1 Motivacija

V zadnjih letih se pri razvoju aplikacij pojavljajo zahteve po visoki skalabilnosti in agilnosti razvoja. Kot posledica teh zahtev se je v praksi uveljavil razvoj aplikacij v arhitekturi mikrostoritev. Gradnja aplikacij v tej arhitekturi prinaša ogromno prednosti, prinaša pa tudi nekaj novih izzivov. Enega izmed njih predstavlja spremljanje izvajanja. S spremljanjem izvajanja lahko bolje razumemo delovanje aplikacije, odkrijemo razloge za napake in vzroke za počasno delovanje aplikacije. Za učinkovito spremljanje izvajanja se v praksi uporablja več pristopov. Nekateri so se uporabljali že v tradicionalnih – monolitnih – arhitekturah, nekateri pristopi pa so se pojavili z uveljavitvijo arhitekture mikrostoritev.

V arhitekturi mikrostoritev se zahtevek izvaja preko več mikrostoritev po omrežju. Obstoječi pristopi za spremljanje izvajanja nam prikažejo le celoten čas izvajanja in število izvedb zahtevka, kar pa nam za učinkovit nadzor nad aplikacijami v arhitekturi mikrostoritev ne zadošča. Dnevniški zapisi, ki jih beležimo v aplikaciji, so porazdeljeni po več mestih – težko jih povežemo z izvedbo določenega zahtevka. V praksi se kot rešitev za naštete probleme uveljavlja porazdeljeno sledenje. Rešitev nam omogoča spremljanje poteka zahtevka preko posameznih mikrostoritev, analizo časa izvajanja posameznih

akcij ter analizo odvisnosti mikrostoritev. V diplomski nalogi bomo porazdeljeno sledenje podrobneje preučili in ga primerjali z ostalimi pristopi za spremljanje izvajanja.

1.2 Cilji

Prvi cilj diplomske naloge predstavlja pregled specifik spremljanja izvajanja v arhitekturi mikrostoritev, pri čemer se osredotočimo na pristop porazdeljenega sledenja. Drugi, glavni cilj pa predstavlja implementacija razširitve za porazdeljeno sledenje za ogrodje KumuluzEE ter izdelava praktičnega primera, ki bo prikazal delovanje razvite razširitve.

1.3 Struktura diplomske naloge

V drugem poglavju pregledamo tradicionalne pristope razvoja aplikacij in jih primerjamo z arhitekturo mikrostoritev. Pregledamo tudi prednosti in izzive, ki jih prinaša arhitektura mikrostoritev. Na koncu drugega poglavja si ogledamo arhitekturo cloud-native. V tretjem poglavju si ogledamo različne pristope spremljanja izvajanja. Podrobneje preučimo pristop porazdeljenega sledenja in ga primerjamo z ostalimi pristopi.

V četrtem poglavju si ogledamo sisteme za porazdeljeno sledenje. Podrobneje si ogledamo sistema Jaeger in Zipkin, dva izmed najbolj popularnih samo-gostujočih sistemov za porazdeljeno sledenje. V petem poglavju pregledamo specifikacijo MicroProfile OpenTracing, po kateri kateri smo razvili razširitev.

Sledi poglavje s pregledom razvoja in arhitekture razširitve KumuluzEE OpenTracing. V zadnjem poglavju pa si ogledamo predstavitev delovanja razširitve na primeru Java aplikacije.

Poglavje 2

Arhitektura mikrostoritev

V tem poglavju si bomo ogledali arhitekturo mikrostoritev ter primerjali način razvoja mikrostoritev s tradicionalnimi pristopi. Ogledali si bomo tudi, kakšne izzive nam prinaša razvoj aplikacij v arhitekturi mikrostoritev.

2.1 Tradicionalni pristopi razvoja aplikacij

Pod tradicionalne pristope razvoja aplikacij spada gradnja monolitnih aplikacij. Za aplikacijo rečemo, da je grajena v monolitni arhitekturi, ko se komponente – dostop do podatkov in uporabniški vmesnik – izvajajo v istem procesu, na istem sistemu [24]. Prednosti tega pristopa so enostavnost razvoja, testiranja in namestitve aplikacije. Aplikacije v tej arhitekturi lahko enostavno skaliramo tako, da dodajamo dodatne instance celotne aplikacije za iznačevalnik obremenitve (angl. load balancer).

Slabost tega pristopa je, da vzdrževanje aplikacije z ogromno količino kode postane oteženo. Ker so posamezne komponente velikokrat močno sklopljene, je potrebno več ročnega testiranja. Dodatna težava monolitnih aplikacij je zanesljivost – hrošč v enem modulu lahko poruši celotno aplikacijo. Pri monolitnih aplikacijah smo omejeni tudi pri vključevanju novih tehnologij – spremembe vplivajo na delovanje celotne aplikacije. V monolitni arhitekturi prav tako ne moremo skalirati vsake komponente posebej, kot je to mogoče

v arhitekturi mikrostoritev [34].

2.2 Mikrostoritve

Mikrostoritve so v zadnjih nekaj letih postale zelo priljubljen način razvoja aplikacij. Aplikacije, grajene v arhitekturi mikrostoritev, so sestavljene iz manjših, samostojnih gradnikov – mikrostoritev. Mikrostoritve običajno tečejo na različnih računalniških procesih, komunikacija med njimi poteka po omrežju. Pri razvoju aplikacije v arhitekturi mikrostoritev moramo biti pozorni na to, da so posamezne mikrostoritve šibko sklopljene – med njimi ni odvisnosti, vsaka mikrostoritev teče sama zase [15].

Vsaka mikrostoritev ima svojo podatkovno bazo. Prednost tega je, da pri vsaki mikrostoritvi lahko izberemo tip baze, ki najbolj ustreza potrebam določene mikrostoritve. Ta pristop sicer lahko privede do podvajanja podatkov, ampak nam omogoča šibko sklopljenost mikrostoritev.

Ker komunikacija med mikrostoritvami poteka po omrežju, prav tako nismo omejeni z uporabo tehnologij. Za vsako mikrostoritev torej lahko uporabimo tehnologije, ki najbolj ustrezajo delu, ki ga mikrostoritev opravlja. Ker so mikrostoritve šibko sklopljene, lahko na njih delajo ločene ekipe, mikrostoritve lahko med sabo neodvisno posodabljamo. Poveča se produktivnost razvoja, ker posamezna mikrostoritev izvaja samo eno samo nalogo, kar omogoča hitrejše razumevanje delovanja. Posamezne mikrostoritve se zaženejo hitreje, kot velika monolitna aplikacija, kar prav tako pohitri razvoj in namestitev aplikacije. Posamezne mikrostoritve lahko skaliramo po potrebi, neodvisno od ostalih mikrostoritev, lahko jih enostavno in hitro testiramo. Aplikacije v arhitekturi mikrostoritev so bolj zanesljive od monolitnih – hrošč v eni mikrostoritvi ne bo porušil vseh ostalih mikrostoritev [33].

2.3 Izzivi arhitekture mikrostoritev

Seveda z dodatnimi prednostmi pridejo tudi določeni izzivi. Pri razvoju moramo dodatno poskrbeti za ravnanje z napakami na mikrostoritvah. Če smo se v monolitni arhitekturi lahko zanesli na to, da bomo odgovor neke komponente zagotovo dobili, moramo tukaj poskrbeti za vse scenarije, ki se lahko zgodijo ob napaki na neki mikrostoritvi. Upoštevati moramo tudi, da lahko pride do latence in da smo pri velikosti zahtevkov omejeni s pasovno širino.

V primerih, ko imamo aplikacijo sestavljeno iz velikega števila mikrostoritev (lahko tudi sto in več), postane oteženo razumevanje delovanja aplikacije kot celote. Želimo imeti nadzor nad delovanjem posameznih mikrostoritev, kot tudi aplikacije kot celote, da lahko razumemo napake in razloge za počasno delovanje aplikacije. Te izzive rešujemo s spremljanjem izvajanja (angl. observability), ki se mu bomo podrobneje posvetili v naslednjih poglavjih diplomske naloge.

Dodaten izziv predstavljajo poslovne transakcije, ki morajo v arhitekturi mikrostoritev posodobiti zapise v več bazah v različnih mikrostoritvah. Ta problem v arhitekturi mikrostoritev običajno rešujemo z vzdrževanjem eventuelne konsistentnosti.

V arhitekturi mikrostoritev je prav tako bolj zahtevna implementacija sprememb v več mikrostoritvah hkrati. Pri monolitnih aplikacijah smo lahko preprosto naredili spremembe v več modulih in aplikacija je bila pripravljena za posodobitev. V arhitekturi mikrostoritev se je takšnih sprememb potrebno lotiti sistematično in natačno načrtovati vsako takšno posodobitev [19].

Dodatna kompleksnost se pojavi tudi pri namestitvi mikrostoritev. V arhitekturi mikrostoritev ima vsaka mikrostoritev poljubno število instanc, vsaka instanca mora biti posebej konfigurirana itd. Za uspešno delovanje pri takšni stopnji kompleksnosti moramo imeti avtomatizirano namestitev in skaliranje aplikacij [19, 16]. To nam omogočajo vsebniki (angl. containers) in sistemi za orkestracijo (Kubernetes, Docker Swarm).

2.4 Arhitektura cloud-native

Arhitektura cloud-native označuje način razvoja aplikacij, ki maksimalno izkorišča prednosti računalništva v oblaku. Namen arhitekture cloud-native je povečati hitrost razvoja (zmanjšati čas od ideje do aplikacije v produkciji), povečati skalabilnost ter zmanjšati stroške obratovanja aplikacij [29].

Glavni elementi razvoja aplikacij v arhitekturi cloud-native so [29, 13]:

- DevOps – Sodelovanje med razvijalci in sistemskimi inženirji z namenom neprekinjenega zagotavljanja programske opreme (CI, CD), ki rešuje izzive uporabnikov.
- Infrastruktura – Uporaba "vse kot storitev" (IaaS/PaaS/SaaS) omogoča avtomatsko skalabilnost in odpornost na napake.
- Mikrostoritve – Mikrostoritve kot način razvoja aplikacij. Pomembna je šibka sklopljenost, kar omogoča, da med seboj neodvisno namestimo, posodabljamo in skaliramo posamezne mikrostoritve.
- Vsebniki – Pakiranje mikrostoritev v vsebnike. Omogoča horizontalno skalabilnost, enostavno testiranje in namestitev mikrostoritev.

Za doseganje večje prepoznavnosti arhitekture cloud-native je bila ustanovljena fundacija *Cloud Native Computing Foundation* (CNCF). Fundacija gosti pomembne projekte za razvoj arhitektur cloud-native (kot npr. Kubernetes, Prometheus, OpenTracing, Jaeger itd.). CNCF predstavlja nevtralno središče za sodelovanje med razvijalci, končnimi uporabniki in ponudniki. Je del neprofitne organizacije *The Linux Foundation* [8].

Poglavje 3

Spremljanje izvajanja v arhitekturi mikrostoritev

V tem poglavju si bomo ogledali različne pristope spremljanja izvajanja aplikacij. Pri tem se bomo osredotočili na porazdeljeno sledenje in ga primerjali z ostalimi pristopi za spremljanje izvajanja.

3.1 Spremljanje izvajanja

Spremljanje izvajanja razvijalcem in sistemskim inženirjem (DevOps) omogoča nadzor nad izvajanjem aplikacij. Pomaga nam razumeti delovanje aplikacije ter omogoča dostop do kritičnih informacij, ko se v aplikaciji pojavijo napake ali je odzivnost aplikacije slabša [18].

3.1.1 Razlogi za spremljanje izvajanja

Za spremljanje izvajanja obstaja več razlogov. V aplikacijah se bodo tudi po obsežnih testiranjih še vedno pojavile napake. S spremljanjem izvajanja želimo izboljšati stabilnost aplikacije. Pomaga nam pri hitrejšem odkrivanju razlogov za napake in analizi počasnega delovanja aplikacije. Ob napaki, ali ko metrike padejo izven pričakovanih rangov, je potrebno čimprej nekoga obvestiti, da aplikacija ne deluje oz. da bo kmalu prenehala delovati [40].

V velikih aplikacijah, ki so lahko sestavljene tudi iz več kot sto mikrostoritev, je razumevanje aplikacije kot celote oteženo. V tem primeru nam spremljanje izvajanja pomaga pri razumevanju delovanja – npr. porazdeljeno sledenje nam prikaže potek zahtevka preko posameznih mikrostoritev in analizo odvisnosti med mikrostoritvami. Podrobneje se bomo temu pristopu posvetili v posebnem podpoglavju.

Eden izmed razlogov za spremljanje izvajanja je tudi izboljšanje stabilnosti aplikacije na dolgi rok. S spremljanjem izvajanja skozi čas pridobimo podatke, ki nam lahko pomagajo izboljšati stabilnost aplikacije. V podatkih lahko najdemo vzorce pojavljanja napak in jih povežemo z dogodki kot npr. namestitev ali posodobitev aplikacije. Kot primer bi lahko ugotovili, da se 30 % vseh napak zgodi v času nalaganja posodobitev aplikacije. Iz tega lahko sklepamo, da bo potrebno izboljšati proces posodabljanja aplikacije. S primerjavo podatkov skozi čas lahko delamo tudi analize kot npr. "Ali spletna stran deluje počasneje kot prejšnji teden?", "Kako velika je baza in kako hitro raste?" itd. Analiziramo lahko tudi dogodke, kot npr. "Kaj se je še zgodilo v času, ko se je povečala latenca na neki mikrostoritvi?" [35].

Spremljanje izvajanja je potrebno tudi iz vidika pogodbe SLA. Če ne spremljamo metrik sistema, ne moremo vedeti, ali smo ugodili zapisani dosegljivosti aplikacije v pogodbi ali ne [40].

Pomembno komponento spremljanja izvajanja predstavljajo nadzorne plošče. Na nadzornih ploščah se prikazujejo glavne metrike sistema. Glavne metrike sistema vključujejo latenco (čas za dokončanje zahtevka), promet (koliko zahtevkov trenutno sprejema mikrostoritev – npr. št HTTP zahtevkov na sekundo), procent napak (kako pogosto se dogajajo napake na mikrostoritvah) in saturacijo (koliko je mikrostoritev zasedena – I/O, CPE, RAM) [7, 23].

3.1.2 Načini spremljanja izvajanja

Poznamo dva načina za spremljanje izvajanja. Izvajanje lahko spremljamo od zunaj (angl. black-box monitoring) ali od znotraj (angl. white-box moni-

toring). Ko spremljamo izvajanje od zunaj, testiramo z namenom razumevanja, kako mikrostoritev vidi uporabnik. Izvajanje od zunaj opisuje simptome problemov, ne dejanskih vzrokov zanje. Vprašanja na katera nam odgovarja, so npr.:

- Ali je mikrostoritev dosegljiva?
- Ali se mikrostoritev na zahtevke odziva po pričakovanjih?
- Ali so mikrostoritve, ki jih trenutna mikrostoritev potrebuje za delovanje, dosegljive?

Ko spremljamo izvajanje od znotraj, spremljamo dnevniške zapise mikrostoritve, metrike, vmesnike kot npr. JVM profiling interface itd.

V praksi se je v preteklosti za obveščanje o napakah uporabljalo predvsem spremljanje od zunaj. Danes ta trend upada, ker iz spremljanja izvajanja od znotraj lahko dobimo bolj smiselna obvestila o napakah. Spremljanje izvajanja od zunaj pa se še vedno uporablja v primerih, ko spremljanje izvajanja od znotraj ni možno (npr. spremljanje platforme IaaS). [37, 35].

3.1.3 Spremljanje izvajanja v različnih arhitekturah

V monolitni arhitekturi za spremljanje izvajanja zadošča že spremljanje osnovnih merik ter pregled dnevniških zapisov. Iz beleženja dnevniških zapisov, ki jih razvijalci vstavijo v kodo, lahko določimo razlog za napako. Iz metrik lahko ugotovimo razloge za počasno delovanje aplikacije.

V arhitekturi mikrostoritev pa samo spremljanje dnevniških zapisov in osnovnih metrik ne zadošča več. Aplikacije so v tej arhitekturi zgrajene iz množice manjših mikrostoritev, med katerimi poteka komunikacija po omrežju. Tu pa naletimo na težavo. Z obstoječimi pristopi ne moremo spremljati izvajanja poteka zahtevka preko posameznih mikrostoritev. Posledica tega pa je, da ne moremo ugotoviti, katera mikrostoritev na poti zahtevka nam upočasnjuje delovanje aplikacije ali javlja napako.

Takšnim sistemom je torej potrebno prilagoditi način spremljanja izvajanja. Za spremljanje metrik moramo imeti način za agregacijo metrik instanc posamezne mikrostoritve (za izris na nadzornih ploščah mikrostoritev, obveščanje o napakah itd.). Za beleženje dnevniških zapisov potrebujemo centralno voden sistem, ki nam bo omogočal njihovo agregacijo. Tem in več ostalim pristopom se bomo bolj podrobno posvetili v poglavju 3.5. Za beleženje sledi izvajanja pa se pojavi nov koncept – porazdeljeno sledenje. [38, 20].

3.2 Sledenje

Sledenje (angl. tracing) je ena izmed temeljnih praks spremljanja izvajanja aplikacij. Cilj sledenja je, da iz uporabnikovih akcij ugotovimo, katere operacije se izvedejo znotraj aplikacije. Pri sledenju uporabnikove akcije povežemo s poslanimi zahtevki. Znotraj aplikacije beležimo potek izvajanja po operacijah in čase izvajanja posameznih operacij. Iz zapisov, povezanih z zahtevkom, zgradimo sled, ki hrani pot izvajanja zahtevka skozi komponente aplikacije.

Sled se ob končani izvedbi zahtevka pošlje v zunanji sistem za spremljanje izvajanja (sistemi APM). V uporabniškem vmesniku sistema je potek zahtevka znotraj aplikacije grafično prikazan. To nam omogoča, da v primeru napake ali počasnega delovanja lažje in hitreje identificiramo problematično komponento [11]. Implementacija opisanega pristopa je v monolitnih arhitekturah enostavna. Vse, kar moramo storiti je, da v aplikacijo vključimo knjižnice ponudnika sistema APM. Avtomatska instrumentacija bo poskrbela za sledenje zahtevkov po celotni aplikaciji. Popularni ponudniki te rešitve so New Relic, AppDynamics in Dynatrace.

V arhitekturi mikrostoritev pa je implementacija sledenja zahtevnejša. V naslednjem podpoglavju si bomo ogledali specifike in načine za implementacijo sledenja v arhitekturi mikrostoritev.

3.3 Porazdeljeno sledenje

Porazdeljeno sledenje (angl. distributed tracing) nam omogoča pregled poteka zahtevkov, ki se izvajajo preko več mikrostoritev. Potek izvajanja zahtevka opisuje struktura, imenovana sled (angl. trace). Zbiranje podatkov v obliki sledi nam omogoči boljše razumevanje življenjskega cikla zahtevka. Z razumevanjem življenjskega cikla zahtevka lahko razhroščujemo zahtevke, ki potekajo preko več mikrostoritev. Tako lahko hitreje odkrijemo, katera operacija v določeni mikrostoritvi na poti zahtevka je vzrok za počasen odziv oz. povečano porabljanje virov [39]. Z analiziranjem sledi lahko ugotovimo, kakšen je bil potek zahtevka, ki je privedel do napake.

Sled si lahko predstavljamo kot usmerjen aciklični graf. Posamezna točka v grafu se imenuje razpon (angl. span) in označuje osnovno enoto dela. V razpon lahko zapišemo oznake, ki nam ob napakah olajšajo filtriranje zahtevkov glede na vrsto napake. Dodamo lahko tudi dnevniške zapise, ki nam pomagajo pri iskanju razlogov za napake.

Dodatno lahko zbiramo še demografske podatke o uporabnikih in o uporabi aplikacije. Ti podatki so koristni tudi prodaji in podpori uporabnikom. Prodaja in produktni vodje lahko sprejemajo boljše odločitve, ker iz zbranih podatkov lahko analizirajo, na kakšne načine se aplikacija uporablja. Podpora uporabnikom lahko iz zbranih sledi ugotovi, kakšen postopek pripelje do napake in bolje pomaga uporabnikom. Eno izmed specifikacij za implementacijo porazdeljenega sledenja si podrobneje ogledamo v poglavju 3.4.

Porazdeljeno sledenje predstavlja pomemben del arhitekture cloud-native. V arhitekturi cloud-native za učinkovito spremljanje izvajanja potrebujemo visoko skalabilne sisteme za porazdeljeno sledenje. Ti sistemi nam omogočajo razumevanje delovanja aplikacij, ki so lahko sestavljene tudi iz več sto mikrostoritev [47]. Z razumevanjem delovanja aplikacij pridobimo na agilnosti razvoja, aplikacije je lažje vzdrževati. To je tudi eden izmed ciljev CNCF. V CNCF sta trenutno včlanjena dva odprtokodna projekta iz področja porazdeljenega sledenja – projekt OpenTracing in sistem za porazdeljeno sledenje Jaeger [22].

3.3.1 Instrumentacija mikrostoritev

Za beleženje sledi izvajanja v arhitekturi mikrostoritev je potrebno mikrostoritve opremiti z implementacijo porazdeljenega sledenja. Implementacija je sestavljena iz dveh delov. Prvi del predstavlja implementacija propagacije konteksta razpona med mikrostoritvami in instrumentacija kode mikrostoritve s porazdeljenim sledenjem. Drugi del implementacije pa predstavlja poročanje sledi v sistem za porazdeljeno sledenje.

V prvem delu implementacije vsak zunanji zahtevek opremimo z unikatnim identifikatorjem. Ta identifikator nato pošiljamo naprej vsem stroritvam, ki se izvedejo na poti izvajanja tega zahtevka [31]. Znotraj posamezne mikrostoritve identificiramo vstopno in izstopno točko. Točki predstavljata filter vhodnih oz. odhodnih zahtevkov. V vstopni točki ekstrahiramo podatke konteksta razpona iz prejetega zahtevka. Znotraj posamezne mikrostoritve lahko v sled izvajanja zapišemo oznake, ki nam ob napakah olajšajo filtriranje zahtevkov glede na vrsto napake. V sled lahko dodamo tudi dnevniške zapise, ki nam pomagajo pri iskanju razlogov za napake. V izstopni točki v zahtevek vključimo podatke konteksta razpona.

Tak način implementacije nam omogoča, da s porazdeljenim sledenjem opremimo mikrostoritve neodvisno od programskega jezika ali ogrodja, v katerem so mikrostoritve implementirane. Za Javo EE obstaja specifikacija MicroProfile OpenTracing, ki definira načine implementacije porazdeljenega sledenja v JAX-RS mikrostoritve. Specifikacijo si bomo podrobneje ogledali v poglavju 5.

V drugem delu implementacije odjemalca ponudnika zbrane podatke asinhrono v ozadju pošilja v zbiralnik sledi (angl. trace collector) sistema za porazdeljeno sledenje. Drugemu delu se bomo bolj natančno posvetili v poglavju 4, ker se implementacije različnih ponudnikov sistemov med seboj razlikujejo.

Za samo implementacijo porazdeljenega sledenja v aplikacije imajo ponudniki sistemov za porazdeljeno sledenje na voljo knjižnice za različne programske jezike. Težava nastane, ko bi želeli ponudnika zamenjati. Posamezni

API-ji namreč med seboj niso kompatibilni – ob menjavi ponudnika bi morali spreminjati kodo aplikacije.

Kot rešitev se pojavi projekt OpenTracing, ki omogoča, da v projekt vključimo implementacijo porazdeljenega sledenja, brez da bi se pri tem vezali na ponudnika. OpenTracing definira specifikacijo in API-je za sled, razpone in prenašanje konteksta razpona. Mikrostoritev lahko instrumentiramo z OpenTracing API-ji, od izbranega ponudnika v mikrostoritev vključimo le odjemalca, ki je kompatibilen z OpenTracing API-ji. Na ta način lahko ob morebitni menjavi ponudnika v mikrostoritvi zamenjamo samo odjemalca, instrumentacijske kode v mikrostoritvi nam ni potrebno spreminjati. Podrobneje se bomo OpenTracing specifikaciji posvetili v naslednjem podpoglavju.

Implementacija porazdeljenega sledenja v obstoječo infrastrukturo je lahko zahtevna. Vsako mikrostoritev na poti zahtevka moramo opremiti s kodo za propagacijo konteksta razpona. Za boljše razumevanje poteka zahtevka znotraj posamezne mikrostoritve je potrebno znotraj operacij ustvariti razpone ter jih opremiti z oznakami in dnevniškimi zapisi.

Dodatno težavo pri implementaciji porazdeljenega sledenja predstavljajo ogrodja oz. knjižnice, na katerih imamo zgrajeno mikrostoritev. Nekatera ogrodja oz. knjižnice imajo že implementirano funkcionalnost porazdeljenega sledenja. V primeru, da ogrodje oz. knjižnica tega še ne ponuja, bomo morali za implementacijo poskrbeti sami. Implementacija postane še posebej problematična, kjer so mikrostoritve zgrajene v različnih programskih jezikih. Najbolj uspešna pri implementaciji porazdeljenega sledenja so podjetja, ki za gradnjo mikrostoritev po celotnem podjetju uporabljajo enake programske jezike oz. ogrodja [39].

Z uveljavitvijo servisnih mrež (angl. service mesh) se je pojavil dodaten način za implementacijo porazdeljenega sledenja v obstoječe arhitekture. Funkcionalnost porazdeljenega sledenja lahko implementiramo že na nivoju servisne mreže. Pri tem posamezne mikrostoritve na poti zahtevka obravnavamo kot črno škatlo. Poskrbeti moramo samo, da mikrostoritve posredujejo glavo zahtevkov, v kateri se propagirajo kontekstni podatki za sledenje. Na

ta način lahko v obstoječo infrastrukturo vključimo funkcionalnost porazdeljenega sledenja z najmanj sprememinjanja kode [39].

3.4 Projekt OpenTracing

OpenTracing je projekt, ki specificira standardiziran API ter ogrodja (angl. frameworks) in knjižnice (angl. libraries) za integracijo porazdeljenega sledenja v aplikacije. Razvijalcem omogoča, da v projekt vključijo implementacijo porazdeljenega sledenja, brez da bi se pri tem vezali na določenega ponudnika. Ponudniki (sistemi za porazdeljeno sledenje), ki trenutno podpirajo OpenTracing API so Jaeger, LightStep, Instana, Apache SkyWalking, inspectIt, stagemonitor in datadog. OpenTracing API je trenutno na voljo za jezike Java, Go, JavaScript, Python, Ruby, PHP, Objective C, C++ in C# [11].

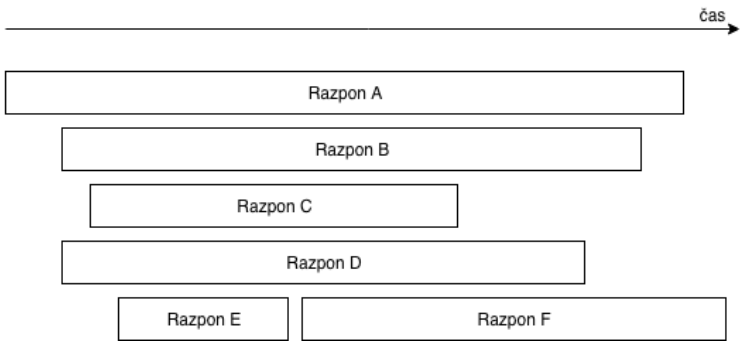
3.4.1 Povzetek specifikacije

Osnovno enoto v podatkovnem modelu projekta OpenTracing predstavlja razpon. Skupek razponov tvori strukturo, imenovano sled. Sled si lahko predstavljamo tudi kot usmerjen acikličen graf, sestavljen iz razponov, povezave med njimi se imenujejo reference (angl. references). Reference so lahko tipa "otrok od" (angl. ChildOf) ali tipa "sledi iz" (angl. FollowsFrom). Obe referenci se nanašata na direktno povezavo med staršem (angl. parent span) in otrokom (angl. child span) v grafu razponov [10].

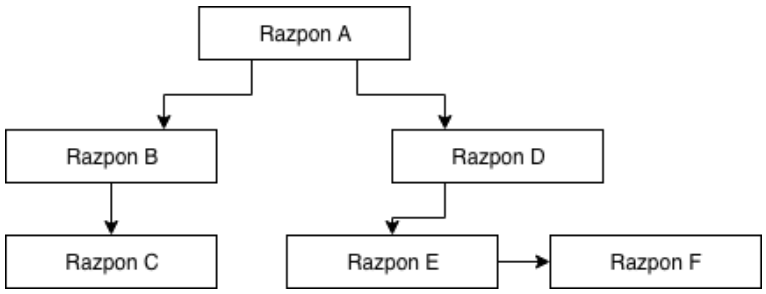
Na sliki 3.1 lahko vidimo primer sledi, prikazane glede na čas. Sled je sestavljena iz šestih razponov. Razpon A je korenski razpon (angl. root span). Ima dva otroka – referenca "otrok od A" – razpon B in razpon D. Razpon B ima enega otroka – razpon C. Razpon D ima prav tako enega otroka – razpon E. Razpon F ima referenco tipa "sledi iz" na razpon E. Prikaz sledi v grafu lahko vidimo na sliki 3.2.

Vsak razpon vključuje naslednje podatke [10]:

- ime operacije – Primer: "GET: /orders"



Slika 3.1: Primer sledi s šestimi razponi.



Slika 3.2: Prikaz sledi v grafu.

- začetno in končno časovno oznako (angl. start and finish timestamp)
- množica oznak (angl. tags) – Oznake so pari podatkov v obliki ključ:vrednost (angl. key:value). Omogočajo nam dodajanje podatkov, po katerih lahko filtriramo in v razpon dodamo dodatne informacije. Primer: {“component”: “jaxrs”}
- množica dnevniških zapisov (angl. logs) – Dnevniški zapisi so podatki oblike ključ:vrednost (angl. key:value). Omogočajo nam, da v razpon dodamo dodatne informacije, ki bi bile lahko uporabne pri razhroščevanju in za zapis ostalih informacij, vezanih na razpon. Primer: {“error”: “ Service unavailable.”}
- SpanContext – Kontekst razpona. Prenaša podatke med procesi.

Primer: { “trace_id”: “AAA123”, “span_id”: “BBB456”, “baggage_items”: { “custom_item”: “My custom item” } }.

- Prtljaga (angl. baggage) – V kontekst razpona lahko dodamo prtljago. Prtljaga je uporabna, ko želimo do podatkov dostopati po vsej sledi izvajanja. Pozorni pa moramo biti na velikost, saj se propagira z vsakim poslanim zahtevkom.

OpenTracing API omogoča, da je naenkrat v niti (angl. thread) aktiven samo en razpon, imenuje se aktiven razpon (angl. active span). Nadzor nad aktivnostjo razponov je prepuščen dosegu (angl. scope) [9].

3.4.2 Sledilniki

Kot smo zapisali že v začetku tega poglavja, nam OpenTracing ponuja, da v projekt vključimo implementacijo porazdeljenega sledenja, brez da bi se vezali na določenega ponudnika. Potrebujemo še rešitev, ki nam bo zbrane sledi pošiljala v zbiralnik sledi (angl. trace collector) izbranega ponudnika. To nam omogočajo t.i. sledilniki (angl. tracers). Sledilniki so del implementacije odjemalca sistema za porazdeljeno sledenje. Omogočajo nam ustvarjanje razponov, dostopanje do aktivnega razpona in pošiljanje konteksta preko različnih protokolov. Bolj natančno se jim bomo posvetili v praktičnem delu diplomske naloge.

3.5 Ostali pristopi za spremljanje izvajanja

V arhitekturi mikrostoritev se poleg porazdeljenega sledenja uporablja več ostalih pristopov za učinkovito spremljanje izvajanja. V tem podpoglavju si bomo ogledali agregacijo dnevniških zapisov (angl. log aggregation), preverjanje vitalnosti mikrostoritev (angl. health checks), metrike mikrostoritev (angl. service metrics) ter beleženje revizijske sledi (angl. audit logging).

3.5.1 Agregacija dnevniških zapisov

V arhitekturi mikrostoritev se dnevniški zapisi (angl. logs) beležijo pri vsaki instanci mikrostoritve posebej. Če bi želeli ugotoviti, kje se je določena napaka zgodila, bi morali pregledati ogromno dnevniških datotek (angl. log files). Želeli bi imeti pregled nad vsemi zapisi na enem mestu za lažjo analizo in iskanje po podatkih. To nam omogočajo sistemi za agregacijo dnevniških zapisov.

Takšni sistemi zapise iz več mest združijo in indeksirajo, kar nam omogoči hitro ter semantično smiselno iskanje. Omogočajo nam tako pregled zapisov v realnem času, kot tudi analiziranje trendov. Med najbolj popularnimi rešitvami danes je ELK – platforma, ki jo sestavljajo Elasticsearch, Logstash in Kibana. Vse tri rešitve razvija in vzdržuje Elastic. Logstash je sistem, ki sprejme podatke jih transformira in naloži v shrambo, kot npr. Elasticsearch. Elasticsearch je porazdeljen iskalni stroj (angl. search engine), Kibana pa orodje za vizualizacijo [38, 5, 14].

3.5.2 Metrike mikrostoritev

Metrike se uporabljajo za izris na nadzornih ploščah mikrostoritev in obveščanje o napakah ali upočasnjenem delovanju v realnem času. Obveščanje o napakah je kritičnega pomena za zagotavljanje dosegljivosti mikrostoritve in zmanjševanje časa okvare (angl. downtime) [38]. Delimo jih na infrastrukturne metrike in metrike mikrostoritev. Glavne infrastrukturne metrike so [17]:

- CPE,
- RAM,
- niti,
- datotečni deskriptorji,
- povezave na bazo.

Glavne metrike mikrostoritev so [17]:

- metrike, specifične za programski jezik,
- razporožljivost mikrostoritve,
- pogodba SLA,
- latenca,
- delovanje končnih točk,
- odzivi končnih točk,
- odzivni časi končnih točk,
- odjemalci,
- napake in izjeme,
- odvisnosti.

Če bi morali izbrati štiri izmed teh metrik, bi po poročanju Google SRE ekipe morali meriti latenco (čas za dokončanje zahtevka), promet (koliko zahtevkov trenutno sprejema mikrostoritev – npr. št. HTTP zahtevkov na sekundo), procent napak (kako pogosto se dogajajo napake na mikrostoritvah) in saturacijo (koliko je mikrostoritev zasedena – I/O, CPE, RAM) [35].

3.5.3 Preverjanje vitalnosti mikrostoritev

V arhitekturi mikrostoritev se včasih zgodi, da je neka mikrostoritev zagnana, vendar ni zmožna sprejemati zahtevkov. To lahko preverjamo tako, da v mikrostoritev dodamo končno točko za preverjanje vitalnosti, ki bo vračala vitalnost te mikrostoritve.
Na tej končni točki vračamo podatke, kot so [4]:

- Ali se mikrostoritev na zahtevke odziva po pričakovanjih?

- Ali so vse mikrostoritve, ki jih trenutna mikrostoritev potrebuje za delovanje, dosegljive?
- Preveri, če zaledne mikrostoritve zares delujejo z izvedbo "end-to-end" transakcije.

Končno točko za preverjanje vitalnosti periodično kliče npr. uravnalnik prometa (angl. load balancer), ki v primeru nedelovanja instance preusmeri promet na delujoče instance. Prav tako to končno točko lahko periodično kliče rešitev za spremljanje izvajanja (angl. monitoring service), ki v primeru nedelovanja pošlje opozorilo [4, 32].

3.5.4 Beleženje revizijske sledi

Beleženje revizijske sledi (angl. audit logging) nam pomaga pri razumevanju obnašanja uporabnikov. Ko se pri uporabniku pojavi napaka, je podpori uporabnikom, kot tudi razvijalcem zelo uporabno, če lahko ugotovijo, kakšen je bil postopek uporabe aplikacije, ki je privedel do te napake. Zelo zanesljiv način za beleženje revizijske sledi je pretakanje dogodkov (angl. event sourcing) [30]. S tem pristopom je možno vsako akcijo, ki jo je uporabnik naredil v aplikaciji, ponoviti kot zaporedje dogodkov.

3.6 Primerjava in ugotovitve

V arhitekturi mikrostoritev se pojavi ogromno novih izzivov, ki jih v tradicionalnih arhitekturah nismo poznali. Enega izmed večjih izzivov predstavlja spremljanje izvajanja, ki je kritičnega pomena za zagotavljanje dosegljivosti mikrostoritve. V tem poglavju smo pregledali rešitve, ki nam omogočajo učinkovito spremljanje izvajanja v arhitekturi mikrostoritev – porazdeljeno sledenje, agregacija dnevniških zapisov, metrike, preverjanje vitalnosti mikrostoritev ter beleženje revizijske sledi.

Porazdeljeno sledenje nam omogoča pregled poteka zahtevka preko posameznih mikrostoritev. Iz tega lahko ugotovimo, kakšne so odvisnosti med

posameznimi mikrostoritvami in katera mikrostoritev na poti zahtevka nam upočasnjuje delovanje aplikacije ali javlja napako. Z dodajanjem demografskih podatkov o uporabnikih lahko delamo analize, ki koristijo tako prodaji, produktnim vodjem kot tudi podpori uporabnikom.

Agregacija dnevniških zapisov nam omogoča, da imamo na enem mestu dostop do zapisov iz celotne aplikacije. Omogoča nam tako pregled zapisov v realnem času, kot tudi analiziranje trendov. Metrike mikrostoritev se uporabljajo predvsem za prikaz na nadzornih ploščah in obveščanje o napakah ali upočasnjenem delovanju. Preverjanje vitalnosti mikrostoritev se predvsem uporablja za periodično preverjanje dosegljivosti in delovanja mikrostoritve. Beleženje revizijske sledi nam omogoča, da lahko vsako akcijo, ki jo je izvedel uporabnik, reproduciramo – ugotovimo, kakšen je bil postopek uporabe aplikacije, ki je privedel do napake.

Vsi opisani način spremljanja izvajanja so komplementarni. Pomagajo nam lažje in hitreje odkriti in razrešiti napake, ko se pojavijo. Omogočajo nam, da so aplikacije bolj stabilne, da so izpadi čim krajši. Iz podatkov, ki jih zbiramo z različnimi pristopi, lahko odkrijemo nova spoznanja o uporabnikih, ki nam veliko pripomorejo k izboljšanju produkta.

Poglavje 4

Sistemi za porazdeljeno sledenje

V prejšnjem poglavju smo videli, da je implementacija porazdeljenega sledenja sestavljena iz dveh delov. Najprej moramo z implementacijo porazdeljenega sledenja opremiti mikrostoritve, te pa poročajo sledi v zunanji sistem. Sistemi za porazdeljeno sledenje se danes večinoma zgledujejo po sistemu Dapper, razvitem pri Googlu [36]. Sestavljeni so iz zalednega dela, ki shranjuje sledi, ki jih pošiljajo mikrostoritve, in uporabniškega vmesnika, ki sledi grafično prikazuje.

Sisteme delimo na samo-gostujoče (angl. self-hosted) in zunanje-storitvene (SaaS). Med najbolj prepoznavnimi samo-gostujočimi sistemi so Jaeger, Zipkin, AppDash, med najbolj prepoznavni zunanje-storitveni sistemi pa so Google Stackdriver Trace, AWS X-Ray, Datadog, New Relic, LightStep. V tem poglavju se bomo osredotočili na samo-gostujoče sisteme. Opisali in primerjali bomo sistema Jaeger in Zipkin, dva izmed najbolj popularnih odprtokodnih sistemov za porazdeljeno sledenje.

4.1 Jaeger

Jaeger je odprtokoden sistem za porazdeljeno sledenje, razvit pri podjetju Uber Technologies. Arhitekturno se zgleduje po sistemih Dapper in Zipkin. Sistem že od samega začetka podpira OpenTracing specifikacijo, enako kot OpenTracing je tudi Jaeger eden izmed projektov CNCF [22]. Sistem je visoko skalabilen, nima ene same točke odpovedi. Razvit je v programskem jeziku Go. Podpira več zalednih sistemov za shranjevanje podatkov – Cassandra, Elasticsearch, Apache Kafka in shranjevanje v spominu za testiranje. Sistem omogoča izvoz metrik v sisteme za spremljanje aplikacij kot npr. Prometheus. Združljiv je s sistemom Zipkin, sprejema sledi v Zipkin formatih, podpira tudi starejše Zipkin knjižnice [45].
Sistem rešuje naslednje probleme [45]:

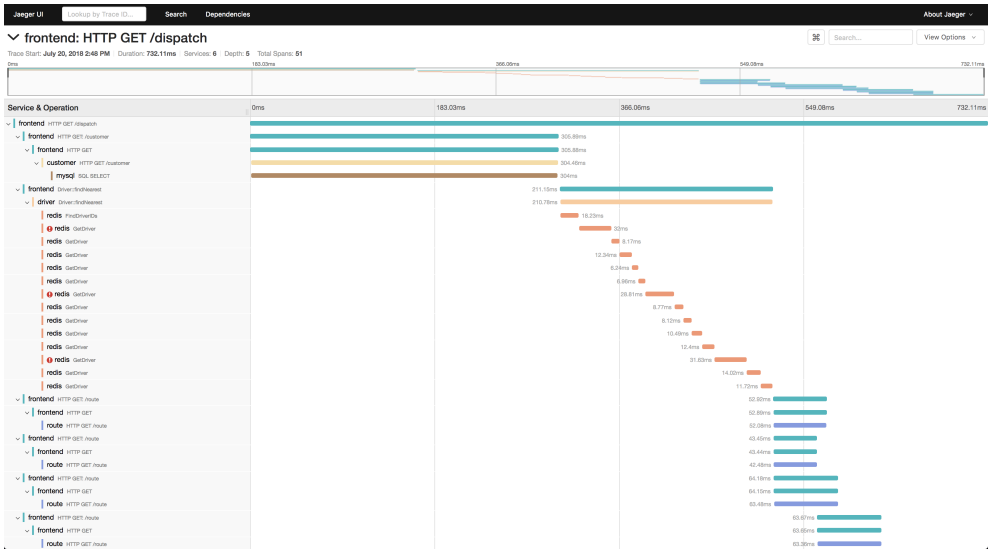
- spremljanje porazdeljenih transakcij,
- optimizacija latence in hitrosti delovanja,
- analiza glavnih vzrokov za napake in počasno delovanje,
- analiza odvisnosti med mikrostoritvami,
- prenašanje porazdeljenega konteksta.

4.1.1 Uporabniški vmesnik

Uporabniški vmesnik je zgrajen s popularnim ogrodjem React. Vmesnik je zgrajen tako, da lahko brez težav sprejme ogromne količine podatkov. V vmesniku lahko filtriramo glede na mikrostoritev, oznake, čas izvajanja itd. Na sliki 4.1 lahko vidimo primer prikaza sledi v sistemu Jaeger.

4.1.2 Arhitektura

V tem poglavju si bomo podrobneje ogledali arhitekturo sistema Jaeger. Na sliki 4.2 lahko vidimo posamezne komponente sistema. Integracija porazdeljenega sledenja v aplikacijo poteka tako, da najprej aplikacijo opremimo z



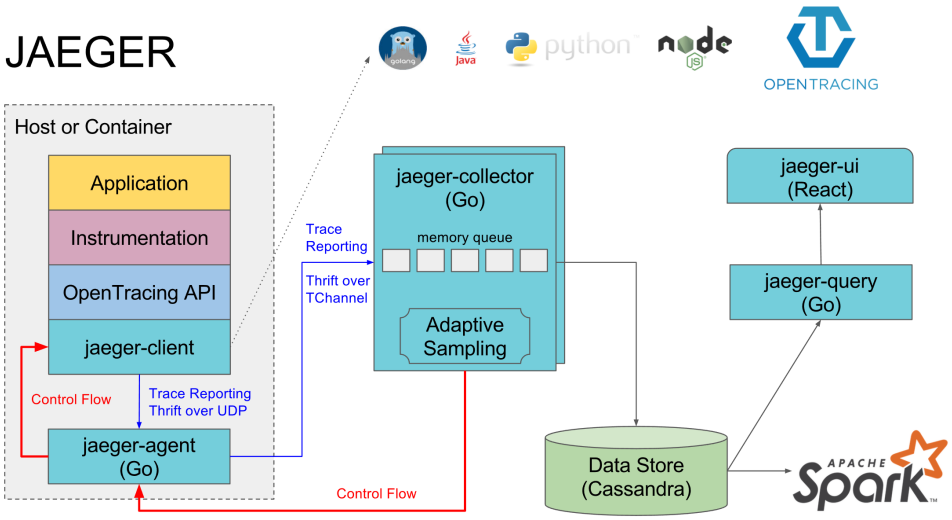
Slika 4.1: Primer prikaza sledi v sistemu Jaeger. Vir: [45]

OpenTracing knjižnicami za izbran programski jezik. Posamezna mikrostoritev na vstopni točki zahtevka ustvari nov razpon, v izstopni točki pa razpon zaključi in ustvari kontekst razpona (SpanContext) z identifikatorji za razpon, sled in morebitno prtljago. Kontekst se nato propagira z zahtevkom na naslednjo mikrostoritev na poti izvajanja zahtevkov.

Komponenta Jaeger odjemalec (jaeger-client) razpone s celotnimi podatki asinhrono v ozadju po protokolu UDP pošilja Jaeger agentu (jaeger-agent). Agent je skriti omrežni proces, izvaja se lokalno pri gostitelju oz. v vsebniku. Naloga agenta je, da sprejema razpone, jih združi po več skupaj in pošilja naprej v zbiralnik (angl. collector).

Zbiralnik sprejema sledi, ki jih pošiljajo agenti, in jih pošilja skozi procesni cevovod (angl. processing pipeline). Cevovod validira sledi, jih indeksira, izvede transformacije in jih shrani v izbran sistem za shranjevanje podatkov (Cassandra, Elasticsearch ali Apache Kafka).

Komponenta Ingester nam ob uporabi Apache Kafka za shranjevanje podatkov omogoča, da podatke beremo iz Kafke in jih zapisujemo v Cassandro ali Elasticsearch. To je uporabno za gradnjo poprocesnih podatkovnih cevo-



Slika 4.2: Komponente v sistemu jaeger. Vir: [42]

vodov.

Sistem nam omogoča visoko skalabilnost. Ker so zbiralniki brezstanjski (angl. stateless), jih lahko zaganjamo več hkrati. Agent se lahko poveže na iznačevalnik obremenitve (angl. load balancer) zbiralnikov ali na statičen seznam naslovov zbiralnikov [42, 44].

4.1.3 Vzorčenje

Vzorčenje pri porazdeljenem sledenju izvajamo, ker želimo zmanjšati stroške hrambe podatkov, zmanjšati količino hranjenih podatkov in zmanjšati količino podatkov, ki se prenašajo med agentom in zbiralnikom.

Jaeger odjemalec generira vse sledi, vzorči pa jih samo določen procent. Procent in način vzorčenja lahko nastavimo v konfiguraciji, privzeta vrednost pa je 0.1 %, kar pomeni, da je v vzorcu 1 na 1000 sledi.

Jaeger implementira vnaprejšnje konsistentno vzorčenje, kar pomeni, da bo prva mikrostoritev na poti zahtevka ustvarila sled, Jaeger odjemalec prve mikrostoritve pa bo odločil, ali se bo sled vzorčila, ali ne. Preostale mikrostoritve na poti zahtevka bodo upoštevale odločitev prve mikrostoritve. Na

ta način nam je zagotovljeno, da se bodo v sledi, ki jo vzorčimo zabeležili vsi razponi [46].
Za vzorčenje so nam na voljo naslednji načini [46]:

- konstantno – Vzorčevalnik (angl. sampler) se za vse sledi odloči enako. Lahko vzorči vse sledi ali nobene.
- probabilistično – Vzorčevalnik dela naključno vzorčenje z nastavljeno verjetnostjo.
- omejevalno – Vzorčevalnik uporablja omejitev, da zagotovi, da se sledi beležijo z določeno konstantno mero. Npr. lahko nastavimo omejitev 5 sledi na sekundo.
- oddaljeno – Odjemalec prepusti odločitev agentu, kakšen način se uporabi za določeno mikrostoritev. Način vzorčenja lahko določamo v centralni konfiguraciji, ali dinamično (prilagodljivo vzorčenje).

Prilagodljivo vzorčenje

Pri načinih vzorčenja, opisanih zgoraj, je težava, da imajo nekatere končne točke lahko veliko prometa, druge zelo malo. S pristopi vzorčenja, opisanimi zgoraj, je velika verjetnost, da končnih točk z majhnim prometom sploh ne bi dobili v vzorec. Druga težava je, da individualne mikrostoritve ne upoštevajo ostalih mikrostoritev, preko katerih gre zahtevek. Na neki mikrostoritvi lahko določimo zelo nizko probabilistično vrednost vzorčenja, vendar, če je na poti zahtevka kakšna mikrostoritev, na katero se sproži veliko zahtevkov, lahko že nizka vrednost vzorčenja poplavi sistem sledenja. Rešitev za opisani težavi je prilagodljivo vzorčenje.

Prilagodljivo vzorčenje je sestavljeno iz dveh delov. Prvi del rešitve je, da se vzorčenje izvaja glede na končno točko (angl. endpoint), ne samo na mikrostoritev. Drugi del rešitve pa je, da se lahko določi zagotovljeno minimalno mero vzorčenja. To pomeni, da bo v vzorec vedno prišlo N sledi na sekundo, vse sledi za tem pa se bo vzorčilo z določeno verjetnostjo. Rešitev je trenutno še v fazi razvoja [46].

4.2 Zipkin

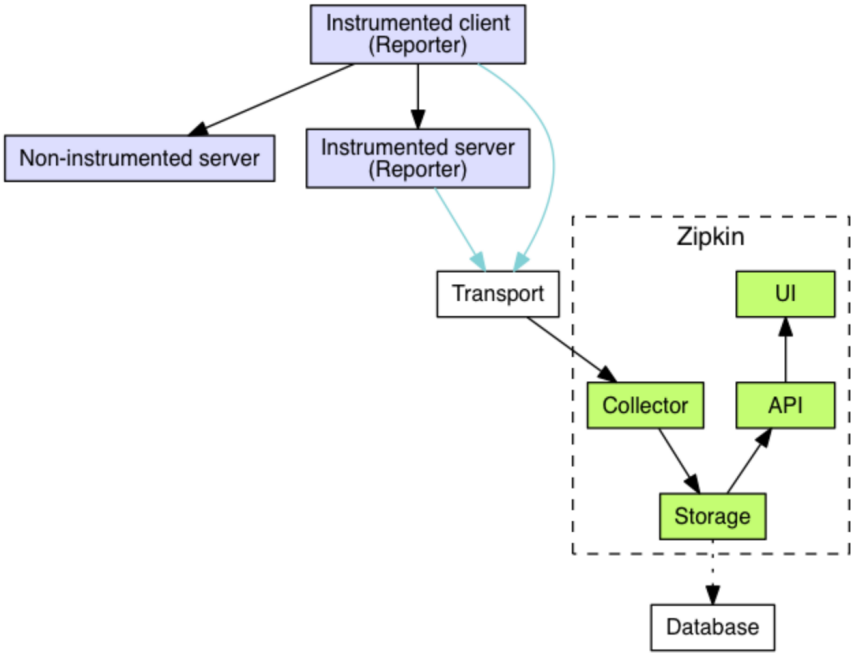
Zipkin je odprtokoden sistem za porazdeljeno sledenje, razvit pri podjetju Twitter. Arhitekturno sta si s sistemom Jaeger zelo podobna, saj se oba zgledujeta po sistemu Dapper. Zipkin je razvit v programskem jeziku Java. Za shrambo podatkov omogoča uporabo sistemov Cassandra, Elasticsearch, MySQL in shranjevanje v pomnilnik za testiranje [25].

4.2.1 Arhitektura

Integracija porazdeljenega sledenja v aplikacijo poteka podobno, kot pri sistemu Jaeger. V aplikacijo moramo torej dodati knjižnico, ki nam bo omogočila ustvarjanje razponov, sledi in poročanje v Zipkin. V Javi lahko uporabimo knjižnico Brave, ki jo razvijajo v ekipi OpenZipkin. Če pa želimo v prihodnosti enostavno zamenjati ponudnika, lahko aplikacijo opremimo z OpenTracing knjižnico v kombinaciji z Jaeger-Zipkin odjemalcem. Podrobneje se bomo posvetili OpenZipkin instrumentaciji, ker smo o OpenTracing knjižnici podrobno pisali že v prejšnjih poglavjih.

Na sliki 4.3 lahko vidimo komponente sistema Zipkin. Posamezna mikrostoritev na vstopni točki ustvari razpon, zabeleži oznake, začeten čas ter doda identifikatorje za razpon in sled v glavo zahtevka. Na izstopni točki se v razpon zabeleži končni čas. Po vrnjenem zahtevku poročevalec (angl. reporter) v ozadju asinhrono prenaša razpone v Zipkin zbiralnik (angl. collector). Sledi se v zbiralnik prenašajo po enem izmed protokolov (HTTP/Kafka/Scribe). Zbiralnik nato validira, shrani in indeksira podatke v izbranem podatkovnem sistemu (Cassandra/Elasticsearch/pomnilnik) [25].

V uporabniškem vmesniku lahko pregledamo sledi in odvisnosti med mikrostoritvami. API komponenta nam omogoča, da lahko v vmesniku filtriramo in sortiramo sledi glede na dolžino sledi, anotacije in čas izvedbe [27].



Slika 4.3: Komponente v sistemu Zipkin. Vir: [25]

4.2.2 Vzorčenje

Zipkin nam ponuja nastavitev probabilističnega vzorčenja glede na končno točko (angl. endpoint). Vzorčenje lahko nastavimo tudi na zbiralniku – lahko nastavimo procent sledi, ki naj jih zbiralnik shrani. Privzeto Zipkin instrumentacija zabeleži vsak zahtevek [26].

4.3 Primerjava

Sistema Jaeger in Zipkin sta visoko skalabilna in pripravljena za uporabo v produkciji. Oba sistema, kot tudi večina ostalih sistemov, ki smo jih našteli v začetku poglavja, sta kompatibilna z OpenTracing specifikacijo. Če aplikacijo opremimo z OpenTracing knjižnicami za beleženje sledi izvajanja, bomo v prihodnosti enostavno lahko zamenjali ponudnika. Vse, kar bo potrebno storiti, je, da bomo zamenjali odjemalca in popravili konfiguracijo. Koda v

aplikaciji bo ostala nespremenjena, sledi pa se bodo beležile v sistem novega ponudnika.

Jaeger nam ponuja več možnosti za izvoz podatkov za nadaljne procesiranje (Apache Kafka) in več možnosti vzorčenja. Poleg OpenTracing API-jev nam za enostaven prehod iz sistema Zipkin omogoča, da uporabimo kar OpenZipkin knjižnice. Primerjavo sistemov lahko vidimo tudi v tabeli 4.1.

Namesto sistemov Jaeger in Zipkin bi lahko za porazdeljeno sledenje uporabili katerega izmed zunanje-storitvenih sistemov. Zunanje-storitveni sistemi so večinoma APM sistemi, ki omogočajo celostno spremljanje izvajanja aplikacij v arhitekturi cloud-native. Omogočajo gradnjo nadzornih plošč s prikazom metrik, analizo poteka zahtevkov in transakcij, pregled dnevniških zapisov, pregled napak, opozarjanje ob napakah, profiliranje kode itd.

Glavna prednost uporabe celovitih sistemov je vsekakor ta, da imamo vse informacije o stanju aplikacije na enem mestu. Slabost tega pristopa pa je, da se vežemo na določenega ponudnika – menjava je lahko težavna. Ob izbiri ponudnika moramo biti pozorni, da uporablja standardne knjižnice (npr. OpenTracing za porazdeljeno sledenje), da bomo ponudnika lahko zamenjali brez večjih sprememb v kodi.

	Jaeger	**Zipkin**
Uradno podprti odjemalci za prog. jezike[1]	Go, Java, Node.js, Python, C++, C#	C#, Go, Java, Javascript, Ruby, Scala, PHP
Podprti podatkovni sistemi	Cassandra, Elasticsearch, Apache Kafka, pomnilnik	Cassandra, Elasticsearch, MySQL, pomnilnik

[1]Odjemalci v nekaterih programskih jezikih ne podpirajo vseh možnosti, ki jih specifikacija določa.

Podpora OpenTracing	Da. Tako Jaeger kot tudi OpenTracing sta CNCF projekta. Jaeger že od samega začetka podpira OpenTracing specifikacijo.	Da, z uporabo neuradno podprtih odjemalcev.
Vzorčenje	Konstantno, probabilistično, omejevalno, prilagodljivo.	Probabilistično.
Namestitev	Obstajajo predloge za namestitev v Kubernetes, OpenShift. Skalira se lahko vsaka komponenta sistema posebej – komponente se izvajajo v ločenih procesih.	Zbiralnik, API in uporabniški vmesnik v istem procesu. Manj dokumentacije glede namestitve.
Podpora za ogrodja in knjižnice	Podpira OpenTracing. Na GitHub repozitoriju opentracing-contrib je na voljo več instrumentacij za popularna ogrodja in knjižnice.	Z uradnimi odjemalci podpira popularna ogrodja. Podpora manjših knjižnic je prepuščena skupnosti.
Aktivna skupnost	Da. Jaeger je del CNCF, pripravljen za arhitekturo cloud-native – izvajanje v vsebnikih, vsebniška orkestracija, najboljša podpora za OpenTracing itd.	Ima aktivno skupnost, ampak ni del večjega ekosistema tako kot Jaeger.

Tabela 4.1: Primerjava sistemov Jaeger in Zipkin [6].

Poglavje 5

Specifikacija MicroProfile OpenTracing

V tem poglavju si bomo ogledali platformo Eclipse MicroProfile in specifikacijo MicroProfile OpenTracing.

5.1 Eclipse MicroProfile

V zadnjih nekaj letih je Java EE postala vse bolj stabilna, kar se je poznalo na manj pogostih izdajah. V tem času so se pojavili novi pristopi k razvoju programske opreme – arhitektura mikrostoritev. Z manj pogostimi izdajami Java EE ni več mogla zadostiti potrebam industrije. To je bil glavni razlog za nastanek platforme, optimizirane za razvoj aplikacij v arhitekturi mikrostoritev – Eclipse MicroProfile.

Platformo sestavlja več specifikacij za razvoj poslovnih Java aplikacij v arhitekturi mikrostoritev – metrike (angl. metrics), preverjanje vitalnosti (angl. health checks), dopustnost napak (angl. fault tolerance), porazdeljeno sledenje (angl. distributed tracing) itd. Z uporabo platforme lahko uporabljamo najnovejše cloud-native tehnologije, neodvisno od ponudnikov [41, 21]. V naslednjem poglavju se bomo podrobneje posvetili specifikaciji porazdeljenega sledenja.

5.2 Eclipse MicroProfile OpenTracing

Specifikacija MicroProfile OpenTracing definira obnašanje in API za dostopanje do OpenTracing sledilnika v JAX-RS aplikacijah. V specifikaciji je definiran način ustvarjanja razponov ob vstopu oz. izstopu iz aplikacije. Specifikacija definira tudi imena operacij razponov in več načinov, kako lahko določeno končno točko izločimo iz sledenja.

Specifikacija je narejena tako, da omogoča enostavno vključitev v že obstoječe arhitekture mikrostoritev. Definira dva načina delovanja – brez dodajanja dodatne kode v aplikacijo in z eksplicitno instrumentacijo [28].

5.2.1 Porazdeljeno sledenje brez dodajanja dodatne kode

Implementacija specifikacije mora omogočati, da se funkcionalnost porazdeljenega sledenja doda v aplikacijo brez dodajanja dodatne kode. Poleg tega naj omogoča, da razvijalci lahko dodajo funkcionalnost brez, da bi vedeli, kateri izmed sistemov za porazdeljeno sledenje se bo uporabljal v produkciji [28].

Zahteve za sledenje brez dodajanja dodatne kode [28]:

- Sledilniki – Vsaka aplikacija bo imela svojo instanco sledilnika (angl. tracer), ki mora biti dostopen po celotni aplikaciji. Med sledilniki lahko izbiramo v zunanji konfiguraciji, brez spreminjanja kode aplikacije.
- Ustvarjanje razponov za vhodne zahtevke – Ob vhodnem zahtevku na aplikacijo se iz konteksta razpona izvozijo podatki o sledi in razponu, če ta obstaja. Ustvari se nov razpon – otrok razpona, ki smo ga prejeli preko konteksta razpona.
- Ustvarjanje razponov za odhodne zahtevke – Ko iz mikrostoritve pošljemo zahtevek npr. na neko drugo mikrostoritev, se ustvari nov razpon, kot otrok trenutnega. V kontekst razpona se zapišejo podatki, razpon se konča ob prejetem odzivu.

Nazivi operacij vhodnih zahtevkov

Za nazive vhodnih zahtevkov sta na voljo dve možnosti za naziv operacij [28]:

1. razred-metoda (angl. class-method) – privzeti naziv

   `<HTTP metoda>:<naziv paketa>.<naziv razreda>.<naziv metode>`

2. HTTP pot (angl. HTTP path)

   `<HTTP metoda>:<vrednost anotacije @Path na razredu končne točke>/<vrednost anotacije @Path na metodi končne točke>`

Nastavitev je možno nastaviti v MicroProfile konfiguraciji pod ključem `mp.opentracing.server.operation-name-provider`.

Oznake vhodnih zahtevkov

Razponi, ki bodo ustvarjeni ob vhodnem zahtevku, bodo vsebovali naslednje oznake [28]:

- `Tags.SPAN_KIND = Tags.SPAN_KIND_SERVER`
- `Tags.HTTP_METHOD`
- `Tags.HTTP_URL`
- `Tags.HTTP_STATUS`
- `Tags.COMPONENT = "jaxrs"`
- `Tags.ERROR - Oznaka se doda, če se je zgodila napaka na strežniku (kode 5xx). Poleg tega se zapiše tudi dnevniški zapis oblike {event=error, error.object=<Instanca objekta napake>}`

Nazivi operacij odhodnih zahtevkov

Privzeti naziv izhodne operacije je:

`<HTTP metoda>`

Oznake odhodnih zahtevkov

Razponi, ki bodo ustvarjeni ob odhodnem zahtevku, bodo vsebovali naslednje oznake [28]:

- `Tags.SPAN_KIND = Tags.SPAN_KIND_CLIENT`
- `Tags.HTTP_METHOD`
- `Tags.HTTP_URL`
- `Tags.HTTP_STATUS`
- `Tags.COMPONENT = "jaxrs"`
- `Tags.ERROR - Oznaka se doda, če se je zgodila napaka na odjemalcu (kode 4xx). Poleg tega se zapiše tudi dnevniški zapis oblike {event=error, error.object=<Instanca objekta napake>}`

Onemogočanje strežniškega dela sledenja

Sledenje lahko onemogočimo za določene poti v aplikaciji. V MicroProfile konfiguraciji `mp.opentracing.server.skip-pattern` nastavimo vrednost ključa v obliki `java.util.regex.Pattern`. Končne točke specifikacij MicroProfile Health, MicroProfile Metrics, MicroProfile OpenAPI so vedno izključene iz sledenja. Ta konfiguracija ne onemogoči sledenja odhodnih zahtevkov, tudi, če je končna točka, na kateri se izvede zahtevek, onemogočena [28].

5.2.2 Porazdeljeno sledenje z eksplicitno instrumentacijo

Za eksplicitno kreiranje razponov je na voljo anotacija `@Traced`. Anotacijo lahko dodamo na razred ali metodo.

Anotacija @Traced

Anotacija `@Traced` ima dva argumenta [28]:

1. `value=[true|false]` – S tem argumentom lahko določimo, da se za določeno metodo ne ustvari razpon – nastavimo `value=false`.To ne onemogoči sledenja odhodnih zahtevkov, ki se izvedejo znotraj metode. Privzeto je nastavljena vrednost `value=true`.

2. `operationName="<NazivOperacije>"` – S tem argumentom lahko nastavimo naziv razpona. Za JAX-RS metode je privzeto nastavljen naziv opisan v poglavju 5.2.1.

Dostop do sledilnika

Po celotni aplikaciji naj bo možen dostop do konfiguriranega sledilnika preko vstavljanja odvisnosti (CDI). Preko sledilnika lahko v aktiven razpon dodajamo oznake, dnevniške zapise in prtljago.

V tem poglavju smo si podrobneje ogledali specifikacijo MicroProfile OpenTracing. V naslednjem poglavju bomo predstavili našo implementacijo predstavljene specifikacije.

Poglavje 6

Praktični del – razvoj modula za OpenTracing

V prejšnjih poglavjih smo pregledali porazdeljeno sledenje iz teoretičnega vidika. Glavni del diplomske naloge pa predstavlja praktični del – razvoj razširitve za porazdeljeno sledenje za ogrodje KumuluzEE, ki se mu bomo podrobneje posvetili v tem poglavju. Ogledali si bomo arhitekturo razširitve, kaj razširitev omogoča ter kako deluje.

6.1 Opis razširitve

V okviru diplomske naloge smo razvili OpenTracing razširitev za ogrodje KumuluzEE. KumuluzEE je odprtokodno ogrodje, ki omogoča razvoj mikrostoritev z uporabo programskega jezika Java in specifikacij Jave EE. Mikrostoritve, grajene z ogrodjem KumuluzEE, so optimizirane za hiter zagon in majhno velikost. Ogrodje prav tako ponuja več razširitev za razvoj cloud-native aplikacij kot npr. konfiguracija, odkrivanje mikrostoritev, beleženje dnevniških zapisov, metrike, varnost itd. [3].

OpenTracing razširitev smo implementirali po specifikaciji MicroProfile OpenTracing, uporabili smo OpenTracing knjižnice. Pri implementaciji smo se zgledovali po najboljših praksah za instrumentacijo ogrodja z OpenTracing

API-ji [12]. Razširitev je odprtokodna, dostopna je na Github repozitoriju [1].

Z dodajanjem razširitve v JAX-RS aplikacijo, razvito z ogrodjem KumuluzEE, lahko omogočimo funkcionalnost porazdeljenega sledenja. Razširitev podpira izvoz sledi v sistem Jaeger.

6.2 Arhitektura razširitve

Razširitev je sestavljena iz dveh Maven modulov – glavni (core) in Jaeger modul. Za takšno strukturo smo se odločili, ker želimo, da se v prihodnosti lahko enostavno doda podporo za dodatne sledilnike.

6.2.1 Glavni modul

Glavni modul predstavlja implementacijo specifikacije MicroProfile OpenTracing. Začeli bomo s pregledom implementacije strežniškega dela sledenja.

Strežniški del sledenja

Ustvarili smo JAX-RS filter na vstopni točki aplikacije in Servlet filter na izstopni točki. Na vstopni točki najprej preverimo, ali je vstopna točka onemogočena za sledenje v konfiguraciji in če je bila onemogočena v kodi z anotacijo `@Traced`. V tem primeru prekinemo z izvajanjem filtra – zahtevek se ne bo sledil.

V nadaljevanju v filtru ustvarimo nov razpon. Iz vhodnega zahtevka ekstrahiramo kontekst razpona (SpanContext), in če ti obstajajo, trenutni razpon nastavimo kot otroka vhodnemu. Nastavimo oznake po specifikaciji in začnemo z izvajanjem novo ustvarjenega razpona. Potek je viden v izseku 6.1.

```
1 SpanContext parentSpan = tracer.extract(Format.Builtin.
    HTTP_HEADERS,
2         new ServerHeaderExtractAdapter(requestContext.getHeaders
    ()));
```

```
3  spanBuilder = tracer.buildSpan(operationName).ignoreActiveSpan()
       ;
4
5  if (parentSpan != null) {
6      spanBuilder = spanBuilder.asChildOf(parentSpan);
7  }
8
9  spanBuilder = spanBuilder
10         .withTag(Tags.SPAN_KIND.getKey(), Tags.SPAN_KIND_SERVER)
11         .withTag(Tags.HTTP_METHOD.getKey(), requestContext.
       getMethod())
12         .withTag(Tags.HTTP_URL.getKey(),
13                 requestContext.getUriInfo().getRequestUri().
       toString())
14         .withTag(Tags.COMPONENT.getKey(), "jaxrs");
15
16 // Za dostop do razpona v filtru odziva (Servlet Response filter
       )
17 requestContext.setProperty(CommonUtil.OPENTRACING_SPAN_TITLE,
18         spanBuilder.startActive(true).span());
```

Izsek 6.1: Ustvarjanje razpona ob vhodnem zahtevku [1]

Drugi del strežniškega sledenja predstavlja Servlet filter. V tem filtru se razpon zaključi. Če je med izvajanjem prišlo do napake na strežniku (napake 5xx), dodamo potrebne oznake in ustvarimo dnevniški zapis (prikazano v izseku 6.2).

```
1  Span span = (Span) request.getAttribute(CommonUtil.
       OPENTRACING_SPAN_TITLE);
2
3  if (span == null) {
4      return;
5  }
6
7  HttpServletResponse httpResponse = (HttpServletResponse)
       response;
8
9  span.setTag(Tags.HTTP_STATUS.getKey(), httpResponse.getStatus())
```

```
    ;

if (httpResponse.getStatus() >= 500) {
    SpanErrorLogger.addExceptionLogs(span, e);
}

span.finish();
```

Izsek 6.2: Odzivni Servlet filter [1]

Sledenje odjemalca

Podprli smo sledenje zahtevkov JAX-RS odjemalca. Podobno kot pri strežniškem delu smo tudi tukaj ustvarili dva filtra. Prvi se izvede pred izvedbo zahtevka, drugi, ko dobimo odziv. V prvem filtru nastavimo oznake, in v zahtevek vstavimo kontekst razpona (SpanContext). To lahko vidimo v izseku 6.3.

```
Tracer.SpanBuilder spanBuilder = tracer.buildSpan(requestContext
    .getMethod())
        .ignoreActiveSpan()
        .asChildOf(tracer.activeSpan())
        .withTag(Tags.SPAN_KIND.getKey(), Tags.SPAN_KIND_CLIENT)
        .withTag(Tags.HTTP_METHOD.getKey(), requestContext.
    getMethod())
        .withTag(Tags.HTTP_URL.getKey(),  uri != null ? uri.
    toURL().toString() : "")
        .withTag(Tags.COMPONENT.getKey(), "jaxrs")
        .withTag(Tags.PEER_PORT.getKey(), uri != null ? uri.
    getPort() : 0)
        .withTag(Tags.PEER_HOSTNAME.getKey(), uri != null ? uri.
    getHost() : "");

Span span = spanBuilder.start();

tracer.inject(span.context(), Format.Builtin.HTTP_HEADERS, new
    ClientHeaderInjectAdapter(requestContext.getHeaders()));

```

```
15 // Za dostop do razpona v filtru odziva (Client Response filter)
16 requestContext.setProperty(CommonUtil.OPENTRACING_SPAN_TITLE,
       span);
```

Izsek 6.3: Ustvarjanje razpona ob odhodnem zahtevku [1]

V odzivnem filtru 6.4 zaključimo razpon in vanj zapišemo oznake in dnevniške zapise, če se je zgodila napaka na odjemalcu (napake 4xx).

```
1 Span span = (Span) requestContext.getProperty(CommonUtil.
      OPENTRACING_SPAN_TITLE);
2 span.setTag(Tags.HTTP_STATUS.getKey(), responseContext.getStatus
      ());
3
4 if (responseContext.getStatus() >= 400) {
5     SpanErrorLogger.addExceptionLogs(span, IOUtils.toString(
      responseContext.getEntityStream(), "UTF-8"));
6 }
7
8 span.finish();
```

Izsek 6.4: Odzivni filter odjemalca [1]

Sledenje z eksplicitno instrumentacijo

Za eksplicitno sledenje smo implementirali anotacijo `@Traced`, ki jo lahko dodamo na razred ali metodo. Naredili smo prestreznik Traced (TracedInterceptor), prikazan v izseku 6.5. V prestrezniku začnemo nov razpon in izvedemo metodo. Če se je v metodi zgodila napaka, to napako zabeležimo v razpon in ga zaključimo. V izseku lahko vidimo, kako lahko znotraj razširitve kot tudi v aplikaciji dostopamo do konfiguriranega sledilnika.

```
1 @Interceptor
2 @Traced
3 @Priority(Interceptor.Priority.APPLICATION)
4 public class TracedInterceptor {
5
6     @Inject
7     OpenTracingConfig tracerConfig;
```

```
8
9       @Inject
10      OperationNameUtil operationNameUtil;
11
12      @Inject
13      Tracer tracer; // dostopanje do sledilnika - CDI
14
15      private static final Logger LOG = Logger.getLogger(
    TracedInterceptor.class.getName());
16
17      @AroundInvoke
18      public Object trace(InvocationContext context) throws
    Exception {
19          Pattern skipPattern = tracerConfig.getSkipPattern();
20          ContainerRequestContext requestContext =
    RequestContextHolder.getRequestContext();
21
22          /** Ne izvedemo sledenja, ce je sledenje te koncne tocke
     onemogoceno v kodi
23          * (@Traced(false)) ali onemogoceno v konfiguraciji ali
    ce
24          * je metoda JAX-RS koncna tocka.
25          */
26          if (ExplicitTracingUtil.tracingDisabled(context) ||
27              requestContext != null && ExplicitTracingUtil.
    pathMatchesSkipPattern(requestContext.getUriInfo(),
    skipPattern) ||
28              ExplicitTracingUtil.isJaxRsResourceMethod(context.
    getMethod())) {
29              return context.proceed();
30          }
31
32          Span parentSpan = tracer.activeSpan();
33          String operationName = operationNameUtil.
    operationNameExplicitTracing(requestContext, context);
34
35          try (Scope scope = tracer.buildSpan(operationName).
    asChildOf(parentSpan).startActive(true)){
```

```
36
37              Object toReturn = null;
38              // izvedemo metodo, in ujamemo morebitne napake
39              try {
40                  toReturn = context.proceed();
41              } catch (Exception e) {
42                  SpanErrorLogger.addExceptionLogs(scope.span(), e
    );
43              }
44
45              return toReturn;
46          } catch(Exception exception) {
47              LOG.log(Level.SEVERE,"Exception occured when trying
    to create method span.", exception);
48          }
49
50          return context.proceed();
51      }
52  }
```

Izsek 6.5: Implementacija Traced anotacije (TracedInterceptor) [1]

6.2.2 Jaeger modul

V Jaeger modulu preberemo konfiguracijo in inicializiramo Jaeger sledilnik. Za konfiguracijo sledilnika uporabimo razširitev ogrodja KumuluzEE MicroProfile Config.

6.2.3 Konfiguracija

Modul KumuluzEE MicroProfile Config nam omogoča branje konfiguracije iz sistemskih lastnosti, okoljskih spremenljivk in konfiguracijskih datotek. V izseku 6.6 lahko vidimo primer konfiguracije razširitve kumuluzee-opentracing za Jaeger sledilnik. Nastavitve samega sledenja (iz MicroProfile specifikacije) nastavimo kot prikazano v izseku 6.7. V obeh primerih konfiguracije smo v komentarje poleg vredosti zapisali pomen posamezne nastavitve. Konfigu-

racija ni obvezna. Če razširitev vključimo v projekt brez konfiguracije, bo delovala s privzetimi vrednostmi, napisanimi spodaj.

```
kumuluzee:
  opentracing:
    jaeger:
      service-name: KumuluzEE project # Ime mikrostoritve. Ce ne nastavimo tega kljuca se bo uporabila vrednost, zapisana v kljucu kumuluzee.name
      agent-host: localhost # privzet gostitelj Jaeger agenta
      agent-port: 5775 # privzet vhod Jaeger agenta
      endpoint: /api/traces  # koncna tocka sledi
      auth-token: authToken # zeton, ki ga poslejmo kot "Bearer" na koncno tocko
      username: username # uporabnisko ime, ki se uporabi kot "Basic" avtentikacija na koncni tocki
      password: password # geslo, ki se uporabi kot "Basic" avtentikacija na koncni tocki
      reporter:
        log-spans: true # ali naj porocevalec belezi razpone
        max-queue-size: 10000 # maksimalna velikost vrste porocevalca
        flush-interval: 1000 # interval za splakovanje porocevalca (ms)
      tags: key1=val1, key2=val2 # oznake, ki se dodajo v vsak razpon
      sampler:
        type: const # privzet tip vzorcenja
        param: 1 # privzet parameter vzorcenja
        manager-host-port: http://localhost:5775 # Naslov do oddaljenega vzorcevalnika, ce ga uporabljamo
      propagation: jaeger # tip propagacije konteksta (jaeger ali b3)
      traceid-128bit: true # ali naj bo identifikator sledi 128 bitni. V prihodnosti bo to privzeta vrednost, trenutno je privzeto 64 bit.
```

Izsek 6.6: Primer konfiguracije sledilnika [1, 43].

```
mp:
  opentracing:
    server:
      operation-name-provider: http-path # nacin poimenovanja imen strezniskih operacij
      skip-pattern: /openapi.*|/health.*  # katere poti v JAX-RS aplikaciji naj se izlocijo iz sledenja
```

Izsek 6.7: Primer konfiguracije sledenja (nastavitve iz MicroProfile specifikacije [1, 28])

6.2.4 Inicializacija Jaeger sledilnika

Inicialicializacijo Jaeger sledilnika naredimo preko konfiguracijskega objekta `io.jaegertracing.Configuration`. V objekt zapišemo vrednosti, ki smo jih prebrali s konfiguracijskim modulom. S konfiguracijskim objektom lahko nato ustvarimo sledilnik tipa `io.opentracing.Tracer`. Na koncu še nastavimo globalni sledilnik (GlobalTracer), ki je uporaben v primeru, da uporabnik ne more uporabiti CDI za vključitev sledilnika. Primer inicializacije je viden v izseku 6.8.

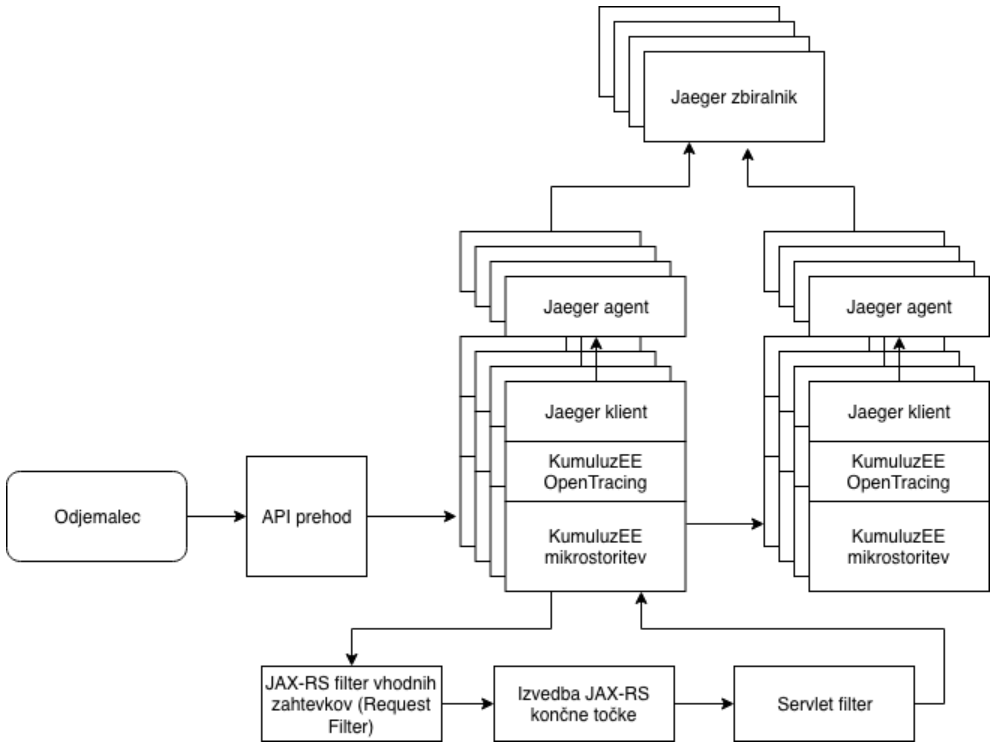
```
Configuration.SamplerConfiguration samplerConfig = new Configuration.SamplerConfiguration()
        .withType(tracingConfig.getSamplerType())
        .withParam(tracingConfig.getSamplerParam());

Configuration.SenderConfiguration senderConfig = new Configuration.SenderConfiguration()
        .withAgentHost(tracingConfig.getAgentHost())
        .withAgentPort(tracingConfig.getAgentPort());

Configuration.ReporterConfiguration reporterConfig = new Configuration.ReporterConfiguration()
        .withLogSpans(tracingConfig.reporterWithLogSpans())
        .withFlushInterval(tracingConfig.reporterWithFlushInterval())
        .withMaxQueueSize(tracingConfig.reporterWithQueueSize())
```

```
13          .withSender(tracingConfig.reporterWithSenderConfig());
14
15 Tracer tracer = new Configuration(tracingConfig.getServiceName()
       ).withSampler(samplerConfig).withReporter(reporterConfig).
       getTracer();
16
17 GlobalTracer.register(tracer);
```

Izsek 6.8: Primer inicializacije Jaeger sledilnika

6.3 Pregled delovanja razširitve

V tem poglavju bomo povzeli delovanje razširitve, ki smo jo podrobneje opisovali v prejšnjih podpoglavjih.



Slika 6.1: Grafični prikaz delovanja razširitve KumuluzEE OpenTracing.

Na sliki 6.1 je prikazana vključitev komponente v sistem z dvema KumuluzEE mikrostoritvama. V obeh mikrostoritvah imamo vključeno komponento kumuluzee-opentracing-jaeger[1], ki implementira specifikacijo MicroProfile OpenTracing ter poroča sledi v sistem Jaeger.

Odjemalec pošlje zahtevek na API prehod, ki ga usmeri na eno izmed razporožljivih instanc prve mikrostoritve. V prvi mikrostoritvi se začne izvajati filter vhodnih zahtevkov (JAX-RS Request filter). V filtru se začne izvajati nov razpon. Iz glave prejetega zahtevka se ekstrahirajo podatki o sledi, če ta obstaja. Trenutni razpon se nastavi kot otrok prejetega razpona. Po izvedbi filtra se zažene klicana JAX-RS končna točka. Med izvajanjem te točke lahko v JAX-RS metodi ali v zrnih, ki jih metoda kliče, dodamo anotacije `@Traced,` ki nam ustvarijo dodatne razpone znotraj posamezne mikrostoritve.

Po celotni aplikaciji lahko dostopamo do aktivnega sledilnika preko CDI. V metodah lahko preko aktivnega sledilnika v aktivni razpon dodajamo poljubne oznake, dnevniške zapise in prtljago. Prtljaga je lahko uporabna, če želimo do določenih podatkov imeti dostop kjerkoli na poti zahtevka. Pozorni pa moramo biti na njeno velikost, saj se propagira z vsemi poslanimi zahtevki.

Iz prve mikrostoritve nato kličemo naslednjo mikrostoritev, ki je prav tako opremljena z OpenTracing razširitvijo. Ob uporabi JAX-RS odjemalca se v glavo zahtevka avtomatsko dodajo podatki o sledi, aktivnem razponu in prtljaga. Nadaljnje izvajanje druge mikrostoritve poteka enako kot pri prvi mikrostoritvi.

Nazadnje se izvede še Servlet filter, v katerem se razpon zaključi. Dodajo se morebitne oznake, če je med izvedbo končne točke prišlo do napake. Razpon (imena operacij, oznake, pripadajoči dnevniški zapisi) se v ozadju asinhrono pošlje agentu sistema Jaeger.

Poglavje 7

Predstavitev porazdeljenega sledenja na primeru Java aplikacije

V tem poglavju si bomo delovanje komponente KumuluzEE OpenTracing ogledali še na praktičnem primeru. Postavili smo dve mikrostoritvi – kupci (customers) in naročila (orders). Mikrostoritvi uporabljata ogrodje KumuluzEE in implementirata enostaven REST API. Aplikacija je zasnovana, da z zahtevkom `GET /customers/{id}/orders` na mikrostoritev kupci lahko pridobimo njihova naročila iz mikrostoritve naročila. Celotna koda primera kot tudi navodila so objavljena na GitHub repozitoriju [2].

7.1 Postavitev JAX-RS mikrostoritev

Ustvarimo dva KumuluzEE projekta – kupci (customers) in naročila (orders). V projekta dodamo odvisnosti CDI, Jetty Servlet ter JAX-RS Jersey. V obeh mikrostoritvah implementiramo REST API – naredimo CRUD metode za upravljanje z kupci oz. naročili. Namesto uporabe podatkovne baze v našem primeru entitete shranujemo kar v ArrayList.

Sedaj lahko upravljamo s kupci in naročili posebej. Da bi prikazali

uporabnost porazdeljenega sledenja, v mikrostoritvi `customers` implementiramo zrno `CustomersBean`, v katerem bomo naredili `GET` zahtevek na mikrostoritev `orders`.Primer je prikazan v izseku 7.1.

```
@RequestScoped
public class CustomersBean {

    private Client httpClient;

    @PostConstruct
    private void init() {
        httpClient = ClientBuilder.newBuilder();
    }

    public Response getOrders() {
        try {
            return httpClient
            .target("http://localhost:3001/v1/orders")
            .request()
            .get();
        } catch (
        WebApplicationException | ProcessingException e) {
            throw new WebApplicationException(e);
        }
    }
}
```

Izsek 7.1: Zrno CustomersBean [2]

Dodamo tudi dodatno končno točko s potjo `{customerId}/orders` v razred `CustomersResource`. Končna točka samo kliče metodo v zrno `CustomersBean`.

7.2 Instrumentacija mikrostoritev

Do sedaj smo naredili dve mikrostoritvi, ki implementirata CRUD operacije nad kupci in naročili. Dodali smo tudi zrno, iz katerega naredimo zahtevek za pridobivanje naročil uporabnika. V tem poglavju bomo v mikrostoritvi dodali instrumentacijo porazdeljenega sledenja.

V `pom.xml` moramo dodati odvisnost za razširitev kumuluzee-opentracing-jaeger (izsek 7.2).

```
<dependency>
    <groupId>com.kumuluz.ee.opentracing</groupId>
    <artifactId>kumuluzee-opentracing-jaeger</artifactId>
    <version>trenutna_verzija</version>
</dependency>
```

Izsek 7.2: Vključitev kumuluzee-opentracing-jaeger razširitve v pom.xml

Dodamo še konfiguracijo za obe mikrostoritvi (prikazano v izsekih 7.3 in 7.4). Mikrostoritev `customers` zaganjamo na vhodu 3000, mikrostoritev `orders` pa na vhodu 3001.

```
kumuluzee:
  name: Customers Microservice
  server:
    http:
      port: 3000
  opentracing:
    jaeger:
      agent-host: localhost
      agent-port: 5775
mp:
  opentracing:
    server:
      operation-name-provider: http-path
```

Izsek 7.3: Konfiguracija mikrostoritve customers

```
kumuluzee:
  name: Orders Microservice
```

```
  server:
    http:
      port: 3001
  opentracing:
    jaeger:
      agent-host: localhost
      agent-port: 5775
mp:
  opentracing:
    server:
      operation-name-provider: class-method
```

Izsek 7.4: Konfiguracija mikrostoritve orders

Sedaj nam deluje poročanje strežniških razponov. Če želimo omogočiti še sledenje odhodnih zahtevkov, moramo v metodi `init()` v zrnu `CustomersBean` popraviti inicializacijo JAX-RS odjemalca (izsek 7.5).

```
@PostConstruct
private void init() {
    httpClient = ClientTracingRegistrar
        .configure(ClientBuilder.newBuilder())
        .build();
}
```

Izsek 7.5: Sledenje odhodnih zahtevkov [2]

Mikrostoritvi lahko opremimo še z anotacijami `@Traced`. V primeru smo anotacijo z nastavljenim imenom operacije dodali na metodo `getOrders()` v mikrostoritvi `customers` in na metodo `getAllOrders()` v mikrostoritvi `orders`. V teh dveh metodah smo zabeležili nekaj dnevniških zapisov in dodali oznake. To je prikazano v izseku 7.6.

```
@Inject
Tracer configuredTracer; //dostop do konfiguriranega sledilnika

@GET
@Traced(operationName = "GET all orders") // spremenimo privzeto
    ime razpona
public Response getAllOrders() {
```

```
7       configuredTracer.activeSpan().log("Getting all orders...");
    // dodamo dnevniski zapis v aktiven razpon
8       configuredTracer.activeSpan().setTag("test-tag", "Test tag
    value"); // dodamo oznako v aktiven razpon
9       List<Order> orders = Database.getOrders();
10      configuredTracer.activeSpan().log("Got "+ orders.size() + "
    orders.");
11      return Response.ok(orders).build();
12  }
```

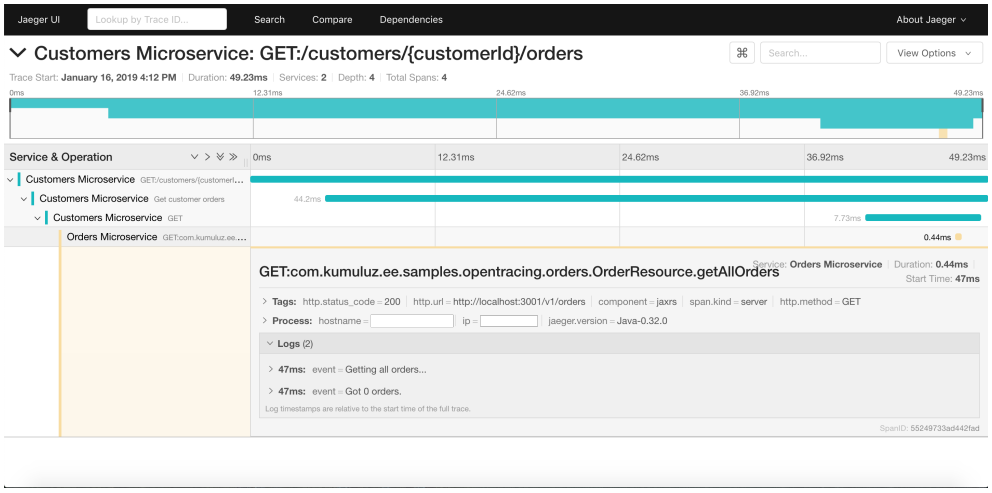
Izsek 7.6: Primer dodajanja oznak in dnevniških zapisov [2]

Sedaj lahko zaženemo sistem Jaeger. V našem primeru smo uporabili docker vsebnik (izsek 7.7).

```
1 docker run -d -p 5775:5775/udp -p 16686:16686 jaegertracing/all-
    in-one:latest
```

Izsek 7.7: Zagon sistema Jaeger kot Docker vsebnik

Izvedemo zahtevek `GET` http://localhost:3000/v1/customers/1/orders. Sled pa si lahko ogledamo v Jaeger konzoli, privzeto dostopni na http://localhost:16686/ (prikazano na sliki 7.1).



Slika 7.1: Prikaz sledi praktičnega primera v sistemu Jaeger.

7.3 Evalvacija

S praktičnim primerom smo prikazali, kako enostavno lahko mikrostoritve, narejene z ogrodjem KumuluzEE, opremimo s funkcionalnostjo porazdeljenega sledenja. Razširitev je razvita z OpenTracing knjižnicami, kar pomeni, da nismo vezani na ponudnika. Če bi želeli menjati ponudnika, ne potrebujemo spreminjati kode aplikacije, ampak samo popravimo konfiguracijo. Razširitev je razvita modularno, da se v prihodnosti lahko enostavno doda podporo za ostale sisteme porazdeljenega sledenja.

Poglavje 8

Zaključek

Arhitektura mikrostoritev nam prinaša določene prednosti, kot tudi nekaj novih izzivov. V okviru diplomske naloge smo se posvetili izzivu spremljanja izvajanja. Pregledali smo več komplementarnih pristopov, s katerimi lahko učinkovito nadzorujemo izvajanje aplikacije. Podrobneje smo preučili pristop porazdeljenega sledenja, ki nam omogoča, da sledimo izvajanju zahtevka preko posameznih mikrostoritev, povežemo dnevniške zapise in oznake z zahtevkom. Vse to nam omogoča učinkovito odkrivanje vzrokov za napake, pregled izvedbene poti zahtevka, analizo odvisnosti med mikrostoritvami itd.

Ogledali smo si sistema Jaeger in Zipkin, pregledali posamezne komponente, arhitekturo in načine vzorčenja obeh sistemov. Pregledali smo specifikacijo MicroProfile OpenTracing, ki definira obnašanje in API vmesnike za dostop do OpenTracing sledilnika v JAX-RS aplikacijah.

V okviru praktičnega dela diplomske naloge smo zasnovali in implementirali razširitev za porazdeljeno sledenje za ogrodje KumuluzEE. Razširitev je zgrajena modularno, kar omogoča enostaven nadaljni razvoj in vzdrževanje. Glavni del razširitve predstavlja implementacija specifikacije MicroProfile OpenTracing. Poleg glavnega modula smo implementirali še modul za poročanje sledi v sistem Jaeger. Razširitev je odprtokodna, objavljena je na Github repozitoriju [1].

Nazadnje smo razvili še praktični primer za namen predstavitve delova-

nja razvite razširitve. V ogrodju KumuluzEE smo postavili dve mikrostoritvi, med katerima smo pošiljali zahtevke. V obe mikrostoritvi smo vključili razvito razširitev in si ogledali sledi v sistemu Jaeger. S primerom smo prikazali uporabnost razširitve v praksi. Videli smo, kako enostavno je mogoče mikrostoritve, razvite z ogrodjem KumuluzEE, opremiti s funkcionalnostjo porazdeljenega sledenja.

Spremljanje izvajanja predstavlja nepogrešljiv sestavni del aplikacij. Omogoča nam, da bolje razumemo delovanje aplikacije kot celote, lažje odkrijemo vzroke za napake in počasnejše delovanje. V okviru diplomske naloge smo z razvojem razširitve razvijalcem KumuluzEE mikrostoritev omogočili, da mikrostoritve enostavno opremijo s funkcionalnostjo porazdeljenega sledenja.

Literatura

[1] KumuluzEE OpenTracing. Dosegljivo: https://github.com/kumuluz/kumuluzee-opentracing, 2019. [Dostopano 12. 1. 2019].

[2] KumuluzEE OpenTracing Sample. Dosegljivo: https://github.com/kumuluz/kumuluzee-samples/tree/master/kumuluzee-opentracing, 2019. [Dostopano 12. 1. 2019].

[3] What is KumuluzEE. Dosegljivo: https://ee.kumuluz.com/, 2019. [Dostopano 7. 1. 2019].

[4] Peter Arijs. Monitoring Microservices With Health Checks . Dosegljivo: https://dzone.com/articles/monitoring-microservices-with-health-checks, 2018. [Dostopano 3. 1. 2019].

[5] Dan Barker. 3 open source log aggregation tools. Dosegljivo: https://opensource.com/article/18/9/open-source-log-aggregation-tools, 2018. [Dostopano 3. 1. 2019].

[6] Daniel Berman. Zipkin vs. Jaeger: Getting Started With Tracing . Dosegljivo: https://dzone.com/articles/zipkin-vs-jaeger-getting-started-with-tracing, 2019. [Dostopano 30. 1. 2019].

[7] Stephen J. Bigelow. distributed tracing . Dosegljivo: https://searchitoperations.techtarget.com/definition/distributed-tracing, 2018. [Dostopano 29. 12. 2018].

[8] CNCF. Cloud Native Computing Foundation (CNCF). Dosegljivo: https://www.crunchbase.com/organization/cloud-native-computing-foundation, 2017. [Dostopano 6. 2. 2019].

[9] OpenTracing Specification Council. Scopes and Threading. Dosegljivo: https://opentracing.io/docs/overview/scopes-and-threading/, 2018. [Dostopano 29. 12. 2018].

[10] OpenTracing Specification Council. The OpenTracing Semantic Specification. Dosegljivo: https://opentracing.io/specification/, 2018. [Dostopano 29. 12. 2018].

[11] OpenTracing Specification Council. What is Distributed Tracing? Dosegljivo: https://opentracing.io/docs/overview/what-is-tracing/, 2018. [Dostopano 29. 12. 2018].

[12] OpenTracing Specification Council. Instrumenting frameworks. Dosegljivo: https://opentracing.io/docs/best-practices/instrumenting-frameworks, 2019. [Dostopano 8. 1. 2019].

[13] Anne Currie. What is Cloud Native? Dosegljivo: https://container-solutions.com/what-is-cloud-native/, 2017. [Dostopano 6. 2. 2019].

[14] Erik Dietrich. What Is Log Aggregation and How Does It Help You? Dosegljivo: https://blog.scalyr.com/2017/08/log-aggregation-help/, 2018. [Dostopano 3. 1. 2019].

[15] Namiot Dmitry and Sneps-Sneppe Manfred. On micro-services architecture. *International Journal of Open Information Technologies*, 2(9), 2014.

[16] Martin Fowler. Microservice Trade-Offs. Dosegljivo: https://martinfowler.com/articles/microservice-trade-offs.html, 2015. [Dostopano 27. 12. 2018].

[17] Susan J. Fowler. *Production-Ready Microservices.*, chapter 6. Monitoring - Key Metrics. O'Reilly Media, Inc., 2016.

[18] Zach Jory. Observability, or Knowing What Your Microservices Are Doing . Dosegljivo: `https://dzone.com/articles/observability-or-quotknowing-what-your-microservic`, 2019. [Dostopano 30. 1. 2019].

[19] Anton Kharenko. Monolithic vs. Microservices Architecture. Dosegljivo: `https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59`, 2015. [Dostopano 27. 12. 2018].

[20] Juraci Paixão Kröhling. Distributed tracing in a microservices world. Dosegljivo: `https://opensource.com/article/18/9/distributed-tracing-microservices-world`, 2018. [Dostopano 29. 12. 2018].

[21] Jean-Louis Monteiro. What is Eclipse MicroProfile? Dosegljivo: `https://www.tomitribe.com/blog/what-is-eclipse-microprofile/`, 2018. [Dostopano 5. 1. 2019].

[22] Cloud native computing foundation. Projects. Dosegljivo: `https://www.cncf.io/`, 2019. [Dostopano 4. 1. 2019].

[23] Netsil. The 4 Golden Signals of API Health and Performance in Cloud-Native Applications. Dosegljivo: `https://blog.netsil.com/the-4-golden-signals-of-api-health-and-performance-in-cloud-native-applications-a6e87526e74`, 2016. [Dostopano 29. 12. 2018].

[24] Kris Nova. What is a monolithic application? Dosegljivo: `https://blog.heptio.com/what-is-a-monolithic-application-e375f5ad5ecb`, 2015. [Dostopano 27. 12. 2018].

[25] Ekipa OpenZikin. Architecture. Dosegljivo: `https://zipkin.io/pages/architecture.html`, 2019. [Dostopano 4. 1. 2019].

[26] Ekipa OpenZikin. Sampling Policy. Dosegljivo: https://github.com/openzipkin/brave/tree/master/instrumentation/http#sampling-policy, 2019. [Dostopano 4. 1. 2019].

[27] Ekipa OpenZikin. Zipkin. Dosegljivo: https://zipkin.io, 2019. [Dostopano 4. 1. 2019].

[28] Steve Fontes Felix Wong Doychin Denkov Bondzhev Pavol Loffay, Kevin Grigorenko. Eclipse MicroProfile OpenTracing. Dosegljivo: https://github.com/eclipse/microprofile-opentracing/blob/master/spec/src/main/asciidoc/microprofile-opentracing.asciidoc, 2019. [Dostopano 7. 1. 2019].

[29] Pivotal. Cloud native apps. Dosegljivo: https://pivotal.io/cloud-native, 2019. [Dostopano 6. 2. 2019].

[30] Chris Richardson. Pattern: Audit logging. Dosegljivo: https://microservices.io/patterns/observability/audit-logging.html, 2018. [Dostopano 3. 1. 2019].

[31] Chris Richardson. Pattern: Distributed tracing. Dosegljivo: https://microservices.io/patterns/observability/distributed-tracing.html, 2018. [Dostopano 29. 12. 2018].

[32] Chris Richardson. Pattern: Health Check API. Dosegljivo: https://microservices.io/patterns/observability/health-check-api.html, 2018. [Dostopano 3. 1. 2019].

[33] Chris Richardson. Pattern: Microservice Architecture. Dosegljivo: https://microservices.io/patterns/microservices.html, 2018. [Dostopano 28. 12. 2018].

[34] Chris Richardson. Pattern: Monolithic Architecture. Dosegljivo: https://microservices.io/patterns/monolithic.html, 2018. [Dostopano 27. 12. 2018].

[35] Niall Richard Murphy David K. Rensin Kent Kawahara Rob Ewaschuk, Betsy Beyer and Stephen Thorne. *Site Reliability Engineering*, chapter Monitoring Distributed Systems. O'Reilly Media, Inc., 2016.

[36] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[37] Cindy Sridharan. *Distributed Systems Observability*, chapter 2. Monitoring and Observability - Blackbox and Whitebox Monitoring. O'Reilly Media, Inc., 2018.

[38] Cindy Sridharan. *Distributed Systems Observability*, chapter 4. The Three Pillars of Observability. O'Reilly Media, Inc., 2018.

[39] Cindy Sridharan. Monitoring in the time of Cloud Native. Dosegljivo: `https://medium.com/@copyconstruct/monitoring-in-the-time-of-cloud-native-c87c7a5bfa3e`, 2019. [Dostopano 30. 1. 2019].

[40] Dave Swersky. The Hows, Whys and Whats of Monitoring Microservices. Dosegljivo: `https://thenewstack.io/the-hows-whys-and-whats-of-monitoring-microservices/`, 2018. [Dostopano 29. 12. 2018].

[41] Alex Theedom. MicroProfile: 5 Things You Need to Know . Dosegljivo: `https://dzone.com/articles/microprofile-5-things-you-need-to-know`, 2017. [Dostopano 5. 1. 2019].

[42] Jaeger tracing. Architecture. Dosegljivo: `https://www.jaegertracing.io/docs/1.8/architecture/`, 2019. [Dostopano 4. 1. 2019].

[43] Jaeger tracing. Configuration via Environment. Dosegljivo: `https://github.com/jaegertracing/jaeger-client-java/blob/`

master/jaeger-core/README.md#configuration-via-environment, 2019. [Dostopano 12. 1. 2019].

[44] Jaeger tracing. Deployment. Dosegljivo: https://www.jaegertracing.io/docs/1.8/deployment/, 2019. [Dostopano 4. 1. 2019].

[45] Jaeger tracing. Jaeger documentation. Dosegljivo: https://www.jaegertracing.io/docs/1.8/, 2019. [Dostopano 4. 1. 2019].

[46] Jaeger tracing. Sampling. Dosegljivo: https://www.jaegertracing.io/docs/1.8/deployment/, 2019. [Dostopano 4. 1. 2019].

[47] Natasha Woods. CNCF Hosts Jaeger. Dosegljivo: https://www.cncf.io/blog/2017/09/13/cncf-hosts-jaeger/#, 2019. [Dostopano 30. 1. 2019].