Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Aljaž Blažej

Optimizacija skaliranja mikrostoritev v okolju Kubernetes na podlagi zbiranja metrik

DIPLOMSKO DELO

UNIVERZITETNI ŠTUDIJSKI PROGRAM
PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Mentor: prof. dr. Branko Matjaž 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.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Opišite pomen analiziranja metrik v okolju mikrostoritev. Izpostavite razloge za zbiranje metrik, opišite pomembne skupine metrik in identificirajte najpomembnejša orodja za zbiranje metrik. Podrobno proučite knjižnico metrik projekta Dropwizard in opišite ter ovrednotite posamezne komponente. Zasnujte in izdelajte rešitev za zbiranje metrik v okolju mikrostoritev v Javi. Opišite okolje za orkestracijo vsebnikov Kubernetes. Vzpostavite in opišite način zbiranja metrik v okolju Kubernetes s pomočjo Heapster in Prometheus ter z uporabo vaše rešitve. Prikažite, kako uporabimo zbrane metrike za samodejno skaliranje mikrostoritev.

Kazalo

Povzetek

Abstract

1 Uvod **1**

1.1 Motivacija 1

1.2 Sorodna dela 2

1.3 Prispevki diplomske naloge 3

2 Zbiranje metrik v mikrostoritvah **5**

2.1 Kaj so metrike? 5

2.2 Razlogi za zbiranje metrik 5

2.3 Kaj mora vsebovati sistem za zbiranje metrik 7

2.4 Katere metrike so pomembne 8

2.5 Orodja za zbiranje metrik 12

3 Metrike Dropwizard **15**

3.1 Register 15

3.2 Merilnik 16

3.3 Števec 16

3.4 Histogram 17

3.5 Meter 17

3.6 Časovnik 18

3.7 Preverjanje vitalnosti 18

3.8 JVM 19
3.9 Apache HttpClient 19
3.10 JDBI 19
3.11 Ehcache 20
3.12 Jersey 20
3.13 Jetty 20
3.14 Beleženje 21
3.15 Spletne aplikacije 21
3.16 Poročanje metrik 22
3.17 Druge knjižnice 22

4 Načrtovanje in razvoj rešitve za zbiranje metrik mikrostoritev z ogrodjem KumuluzEE 23
4.1 KumuluzEE 23
4.2 Razširitev za zbiranje metrik 24

5 Kubernetes 29
5.1 Komponente 30
5.2 Enote 33
5.3 Namestitev 36

6 Zbiranje metrik v Kubernetesu in samodejno skaliranje 39
6.1 Heapster 39
6.2 cAdvisor 40
6.3 Nadzorna plošča 41
6.4 Uporaba metrik pri samodejnem skaliranju 41
6.5 Prometheus 42
6.6 Strežnik API 44
6.7 Konfiguracija horizontalnega razmnoževalca strokov 44
6.8 Povzetek 46
6.9 Izbira ustreznih parametrov in njihove prednosti 46

7 Zaključek 49

Seznam uporabljenih kratic

kratica	angleško	slovensko
API	Application Programming Interface	Aplikacijski programski vmesnik
APM	Application Performance Management	Upravljanje aplikacijskih zmogljivosti
AWS	Amazon Web Services	Amazonove spletne storitve
CDI	Contexts and Dependency Injection	Vnos konteksta in odvisnosti
CPE	Central Processing Unit	Centralna procesna enota
EWMA	Exponentially Weighted Moving Average	Eksponentno uteženo premikajoče povprečje
GCP	Google Cloud Platform	Googlova oblačna platforma
HTTP	HyperText Transfer Protocol	Protokol za prenos nadbesedila
JMX	Java Management Extensions	Javanske razširitve upravljanja
JSON	JavaScript Object Notation	Objektni zapis Javascript
JVM	Java Virtual Machine	Javanski virtualni stroj
REST	Representational State Transfer	Predstavitveni prenos stanja
SQL	Structured Query Language	Strukturirani povpraševalni jezik
URL	Uniform Resource Locator	Enolični lokator vira

Povzetek

Naslov: Optimizacija skaliranja mikrostoritev v okolju Kubernetes na podlagi zbiranja metrik

Avtor: Aljaž Blažej

Z večanjem števila uporabnikov interneta in kompleksnostjo spletnih aplikacij raste tudi potreba po optimizaciji uporabe strojne opreme, na kateri je nameščena aplikacija. Pri tej optimizaciji ima veliko vlogo koncept mikrostoritev, saj ima ta pristop veliko prednosti pred monolitnim, kot so boljša izkoriščenost virov, skalabilnost in izoliranost različnih delov aplikacije. Prav tako je zelo pomembno zbiranje metrik, ki se ob iskanju napak in analizi delovanja sistemov velikokrat uporabijo tudi kot podlaga za avtomatsko skaliranje mikrostoritev.

V diplomskem delu bomo predstavili razloge za zbiranje metrik in navedli najbolj koristne. Opisali bomo javansko knjižnico Dropwizard Metrics, ki je eno izmed najbolj razširjenih orodij za zbiranje in izpostavljanje metrik. Integrirali jo bomo v ogrodje KumuluzEE in s tem poenostavili izdelavo in poganjanje mikrostoritve, ki zbira in izpostavlja metrike v vsebniških okoljih. Predstavili bomo tudi sistem za orkestracijo in upravljanje z vsebniki, Kubernetes, njegove osnovne komponente ter namestitev večvozliščne gruče. Na koncu bomo pokazali, kako namestiti mikrostoritev v gručo Kubernetes in uporabiti metrike, ki jih zbira, kot vir za replikacijo vsebnikov.

Ključne besede: Kubernetes, metrike, skaliranje, Dropwizard, KumuluzEE.

Abstract

Title: Microservice scaling optimization based on metric collection in Kubernetes

Author: Aljaž Blažej

As web applications become more complex and the number of internet users rises, so does the need to optimize the use of hardware supporting these applications. Optimization can be achieved with microservices, as they offer several advantages compared to the monolithic approach, such as better utilization of resources, scalability and isolation of different parts of an application. Another important part is collecting metrics, since they can be used for analysis and debugging as well as the basis for automatic scaling of microservices.

In our diploma work we describe the advantages of collecting metrics and identify the most important ones. We also do a detailed analysis of the Dropwizard Metrics library, which is one of the most used tool–kits for monitoring Java applications. We implement an extension for collecting metrics in microservices developed with the KumuluzEE microservice framework and simplify the development and deployment of microservices that collect and expose metrics. We describe the container orchestration and management system Kubernetes, its components and the steps needed to create a multinode cluster. Finally, we deploy the microservice and use the collected metrics to optimize the scaling process.

Keywords: Kubernetes, Metrics, Scaling, Dropwizard, KumuluzEE.

Poglavje 1

Uvod

Področje oblačnih arhitektur in visoke skalabilnosti se v zadnjih letih zelo hitro razvija [9]. Trenutno je ena izmed najhitreje rastočih arhitektur koncept mikrostoritev, ki pa so tesno povezane s tehnologijo vsebnikov. Zato se je v zadnjih letih pojavilo veliko ogrodij [1] namenjenih orkestraciji in upravljanju z vsebniki, kot so *Kubernetes*, *Docker Swarm*, *fleet* in *Apache Mesos*. Ta orodja so namenjena zagotavljanju odzivnosti aplikacije pod veliko obremenitvijo, pri čemer je ključno merjenje delovanja, zbiranje metrik in ukrepanje na spremembe teh meritev. V diplomskem delu bomo pregledali vse korake instrumentacije sistema, od zbiranja metrik do njihove vizualizacije in uporabe pri povečanju odzivnosti aplikacije.

1.1 Motivacija

S prehodom iz klasičnih monolitnih aplikacij v mikrostoritve [16] in pojavitvijo orodij, ki z njimi upravljajo, se je spremenilo tudi zbiranje metrik ter njihova uporabnost. To nas je motiviralo k raziskavi, katere metrike je potrebno zbirati v aplikaciji, zgrajeni na principu mikrostoritev. Metrike so v novi aritekturi še bolj pomembne, saj so podlaga za sprotno prilagajanje števila instanc mikrostoritev. Zanimalo nas je, kako lahko z zbiranjem pravih metrik izboljšamo skaliranje vsebnikov in s tem poskrbimo za višjo

dostopnost in odzivnost aplikacije. To nas je motiviralo k razvoju razširitve, ki razvijalcem nudi enostavno in celovito zbiranje metrik. Opisali smo tudi primer rešitve, ki prikazuje, kako lahko s temi metrikami izboljšamo skaliranje mikrostoritev v okolju, ki upravlja z vsebniki (v opisani rešitvi je to okolje Kubernetes).

1.2 Sorodna dela

Trenutno avtomatsko skaliranje na podlagi metrik podpirata Amazon in Google, pri čemer je Googlova rešitev omejena na metrike, ki jih zbira sistem sam [2], Amazonova pa ob tem nudi tudi replikacijo na podlagi lastnih metrik. Pri zasebnih oblakih to nudi okolje Kubernetes [19], pri katerem pa je konfiguracija takega sistema zahtevna in slabo dokumentirana. Amazon EC2 nudi tri opcije, na podlagi katerih replicira mikrostoritve [11]:

- obremenjenost CPE ali delovnega pomnilnika,
- *CloudWatch* metrike [28] (informacije o AWS virih kot so število EC2 instanc, obremenjenost relacijskih in NoSQL podatkovnih baz, ...),
- lastne metrike.

Podobne možnosti nudi tudi *Google Cloud Platform* [2]:

- obremenjenost CPE,
- metrike Stackdriver (podobno kot CloudWatch),
- kapaciteta obdelave poizvedb HTTP.

Okolje Kubernetes z dodatkom Heapster (podrobneje opisan v podpoglavju 6.1) nudi zbiranje naslednjih metrik [24]:

- obremenjenost CPE posameznega vozlišča,
- zasedenost delovnega pomnilnika posameznega vozlišča,

- koliko CPE trenutno uporablja posamezen strok,
- koliko delovnega pomnilnika uporablja posamezen strok.

Od naštetih metrik lahko zadnji dve uporabimo kot podlago za replikacijo mikrostoritev. S prilagoditvijo gruče (opisano v poglavju 6) lahko kot podlago uporabljamo tudi katerekoli metrike, ki jih zbira in izpostavlja posamezna mikrostoritev.

V zasebnih oblakih se velikokrat uporablja tudi orodje Prometheus (podrobneje opisano v poglavju 6.5), ki je namenjeno zbiranju in vizualizaciji metrik. Na voljo je tudi prilagoditev tega orodja za okolje Kubernetes, imenovana Prometheus Operator [39].

Pri zbiranju metrik sta zelo razširjeni tudi orodji Nagios [3] in Icinga [33]. Orodji sta si zelo podobni, saj je Icinga poskus izboljšave orodja Nagios. Nobeno od orodij ni dobro prilagojeno zbiranju metrik mikrostoritev [18] in se za ta namen skoraj nikoli ne uporabljata.

1.3 Prispevki diplomske naloge

Rezultat diplomske naloge je razširitev ogrodja KumuluzEE, ki omogoča enostavno zbiranje širokega spektra metrik v mikrostoritvah, razvitih v tem ogrodju. Prav tako je rezultat konfiguracija večvozliščne gruče Kubernetes, ki zbrane metrike uporablja pri avtomatskem repliciranju vsebnikov. V delu je vključen tudi podrobnejši opis metrik, ki so pomembnejše v okolju mikrostoritev ter opis orodja *Dropwizard Metrics*. Prikazano je tudi zbiranje in vizualizacija metrik v okolju Kubernetes ter podroben opis vseh njegovih komponent in enot. Celotna rešitev demonstrira vzpostavitev lastnega okolja, ki s pomočjo pravih metrik, zbranih v mikrostoritvi, optimizirano skalira posamezne dele aplikacije in poskrbi za višjo odzivnost ter manjšo porabo virov.

Poglavje 2

Zbiranje metrik v mikrostoritvah

2.1 Kaj so metrike?

Metrike [45, 46] so številčne meritve, ki se izvedejo ob določenem času. Podajo informacijo o trenutnem stanju določene komponente, kombinacija večih metrik pa nudi razumevanje delovanja celotnega sistema. Metrika je sestavljena iz izmerjene številčne vrednosti in časovne enote. Večinoma jih zbiramo na določenem intervalu, kot na primer enkrat na sekundo, minuto, itd. V diplomskem delu se bomo osredotočili na metrike izvajanja aplikacij, saj se druge ne navezujejo na temo naloge [45].

2.2 Razlogi za zbiranje metrik

Zbiranje metrik in spremljanje izvajanja sistema je zelo pomembno tako v monolitnih aplikacijah kot tudi v mikrostoritvah [8]. V slednjih so te meritve velikokrat še bolj koristne, ker omogočajo, da se ogrodje, ki upravlja s številom instanc mikrostoritev, odzove na spremembe vrednosti metrik in temu primerno poveča ali zmanjša število vsebnikov. A to je le eden od razlogov za zbiranje metrik. Ključne so tudi pri doseganju odpornosti na

izpade (ang. fault tolerance) in prožnosti (ang. resilience) aplikacije. Odpornost na izpade [32, 48] je lastnost sistema, ki v primeru nepričakovanega dogodka ali napake ne preneha delovati. Primer takega izpada je odpoved ene od komponent, od katere je odvisen le del aplikacije [40]. Z zbiranjem metrik lahko zaznamo nepričakovano delovanje posameznih komponent in jih onemogočimo ter tako preprečimo izpad celotne aplikacije. Aplikacija je v tem primeru sicer omejena, saj lahko opravlja le naloge, ki niso odvisne od nedelujoče komponente. Prožnost [6] je izboljšava odpornosti na izpade, saj prožen sistem v primeru napake nemoteno nadaljuje z izvajanjem. Prožnost velikokrat srečamo v aplikacijah, ki so zgrajene na principu mikrostoritev in se izvajajo v vsebniških okoljih. V takih aplikacijah, v primeru neodzivnosti ali nepričakovanega delovanja neke mikrostoritve, sistem za upravljanje z vsebniki uniči vsebnik, v katerem ta mikrostoritev teče, ter ustvari novo instanco. Zbiranje pravilnih metrik je tedaj zelo pomembno, saj so na njihovi podlagi sprejete odločitve, kdaj je potrebno povečati število instanc posamezne mikrostoritve in kdaj je ta neodzivna ter jo je potrebno uničiti.

Zelo pomemben vidik je tudi odkrivanje napak, saj lahko s pomočjo pravilnih metrik in zabeleženih dogodkov, kot na primer klic metode, reproduciramo dogodek, ki je vodil do določene napake. Vzroki za odpoved aplikacije so lahko na primer pomanjkanje delovnega ali swap polnilnika, zasedenost kopice ali nedosegljivost komponent, od katerih je mikrostoritev odvisna. Dobro beleženje stanja sistema ob določenem času nam torej prihrani veliko časa, ki bi ga porabili z iskanjem neobstoječe napake v kodi. Veliko napak ali izpadov pa lahko tudi preprečimo, če pravilno nastavimo meje za vrednosti posameznih metrik, ki ob prekoračitvi bodisi opozorijo sistemske skrbnike o nevarnosti bodisi se avtomatsko odzovejo s skaliranjem posamezne komponente. Primer prvega je opozorilo o zapolnjenosti kapacitete trajnega pomnilnika, ki je v zasebnih oblakih lahko hitro rešena z dodajanjem novih diskov. Primer avtomatskega odziva pa je zgoraj omenjeno prilagajanje števila instanc mikrostoritve glede na prekoračitev zgornje ali spodnje meje. To pa vodi v boljšo izkoriščenost virov in posledično nižje stroške s strojno

opremo. Za boljšo predstavo je zbrane vrednosti dobro vizualizirati na grafu, kar omogoči hitrejše odkrivanje ozkih grl in delov aplikacije, ki bi jih bilo potrebno optimizirati. Zbiranje podatkov na dolgi rok omogoča tudi analizo sistema in podatke o tem, kdaj je aplikacija najbolj obiskana, kje se pojavlja največ napak, povprečen odzivni čas posameznih komponent in mnogo drugih koristnih informacij.

2.3 Kaj mora vsebovati sistem za zbiranje metrik

Ogrodja, ki so namenjena zbiranju metrik, njihovi obdelavi in reševanju zgoraj omenjenih problemov, imenujemo sistem za upravljanje aplikacijskih zmogljivosti (ang. application performance management - *APM*) [20]. Za razvoj celovite rešitve, kot je sistem APM, moramo pomisliti na nekaj ključnih lastnosti [30, 36]:

- **Celovitost**; Zbira metrike vseh komponent mikrostoritve in okolja, na katerem se izvaja. Prav tako mora meriti vse storitve in orodja, od katerih je mikrostoritev odvisna. S tem se znebimo črnih lukenj, ki otežijo ali celo onemogočijo iskanje vzroka določene napake.

- **Razumljivost**; Za metrike mora biti hitro razvidno, katera komponenta ali metoda jih je zabeležila in kaj predstavljajo. Prav tako morajo imeti priložen čas meritve.

- **Granularnost**; Zbira najbolj nizkonivojske meritve, ki omogočajo točen izvor problema in ne le področja.

- **Konsolidiranost**; Vse metrike so združene na enem mestu. Ker so različna področja meritev med seboj povezana, agregiran vpogled omogoča odkrivanje vseh dejavnikov, ki so vodili do nepričakovanega delovanja aplikacije.

- **Konstantnost**; Meritve se morajo izvajati neprestano in na kratkih intervalih. APM mora biti robusten in odporen na izpade drugih komponent.

- **Učinkovitost**; Zbiranje metrik ne sme vplivati na izvajanje merjene komponente.

- **Sočasnost**; Meritve potekajo v realnem času in sistem nudi sprotno vizualizacijo stanja vseh komponent. Prav tako je nujno sprotno javljanje napak in takojšnja opozorila.

- **Arhiviranje**; Obvezno je tudi hranjenje meritev na dolgi rok. To omogoča vizualizacijo porabe virov in delovanja aplikacije, podrobnejšo analizo in obdelavo meritev ter odkrivanje področij, potrebnih optimizacije.

2.4 Katere metrike so pomembne

Metrike je najbolje razdeliti v štiri nivoje [27] glede na to, katere dele sistema merijo. Začeli bomo z opisom merjenja oblačnih ponudnikov, nato se bomo osredotočili na vozlišča, procese in na koncu na zbiranje metrik, specifičnih za aplikacijo. Kot zadnje podpoglavje je opisano merjenje posebnih dogodkov, ki ga moramo izvajati na vseh prej omenjenih nivojih.

2.4.1 Oblačni ponudnik ali zasebni oblak

Začeli bomo na najnižjem nivoju [27] z zbiranjem metrik oblačnega ponudnika. Pri tem so pomembne metrike, ki se navezujejo na delovanje strežnikov ponudnika in odzivnosti glede na geografsko lokacijo strežnikov. Pomembno je tudi merjenje delovanja podatkovnih baz, ki jih nudi ponudnik, in drugih komponent, od katerih je naša aplikacija odvisna.

V primeru zasebnega oblaka je potrebno nadzorovati ogrodje, na katerem teče naša aplikacija, in sicer delovanje izenačevalnikov obremenitve (ang.

load balancer), razporejevanja vsebnikov po vozliščih in drugih komponent ogrodja.

2.4.2 Vozlišče

Zelo pomembno je spremljanje posameznega vozlišča in njegovih fizičnih komponent, kot so CPE, pomnilnik, disk in omrežje [52]. Pri tem moramo meriti odstotek časa, ko je komponenta zasedena, ali odstotek trenutne porabe komponente. Spremljati moramo tudi količino dela, ki ga komponenta še nima časa obdelati in je v čakanju, ter število napak, ki se pojavijo.

Pri CPE je ob trenutni zasedenosti pomembno tudi merjenje povprečne in najvišje zasedenosti, temperature CPE ter delovanje hladilnih sistemov (ventilatorjev). Povprečna in najvišja zasedenost je pomembna tudi pri delovnemu pomnilniku in disku. Pri slednjem je potrebno meriti tudi porabo swap in zdravje diska. Pomemben podatek je tudi število procesov ali vsebnikov, ki tečejo na posameznem vozlišču.

Prav tako moramo spremljati tudi, koliko časa je vozlišče v aktivnem stanju, kar v kombinaciji s podatkom, kdaj je bil strežnik prvič vzpostavljen, poda informacijo o odstotku razpoložljivosti vozlišča.

2.4.3 Proces in okolje

Sem spadajo metrike [13, 52], ki se navezujejo na proces in okolje, v katerem teče naša aplikacija. Zanima nas, koliko CPE, pomnilnika, diska in omrežja proces zaseda, koliko časa teče ter v kakšnem stanju je (se zaganja, teče ali se ustavlja).

Spremljati moramo tudi izvajalno okolje, ki je v našem primeru JVM. Sem spadajo meritve zbiralca smeti (ang. garbage collector), števila niti in druge. Te metrike je v okolju JVM možno pridobiti s pomočjo tehnologije *JMX*, ki upravlja s posebnimi javanskimi zrni, imenovanimi *MXBeans*. Ta zrna so posebna vrsta zrn *MBean*, ki upravljajo samo s podatkovnimi tipi, ki so na voljo vsem odjemalcem. Zrna MX zbirajo in izpostavljajo metrike

JVM ter se tako kot zrna MBeans izvajajo v virtualnem stroju. V spodnjem seznamu [22] so naštete vrste zrn MX in pomembne metrike, ki jih lahko pridobimo iz njih.

- **BufferPoolMXBean** nudi podatek o zasedenosti medpomnilnika.
- **ClassLoadingMXBean** nudi število razredov, ki je trenutno naloženih v JVM, celotno število naloženih razredov, odkar je bil JVM zagnan, in število razredov, ki niso več naloženi.
- **CompilationMXBean** nudi podatek o času, ki je potreben za prevod kode.
- **GarbageCollectorMXBean** nudi število obhodov zbiralca smeti in čas, ki ga je pri tem porabil.
- **MemoryManagerMXBean** vrne imena pomnilniških bazenov, s katerimi upravlja upravljalec pomnilnika.
- **MemoryMXBean** nudi osnovne podatke o kopici in o drugem pomnilniku, ki ga zaseda JVM.
- **MemoryPoolMXBean** nudi podrobne podatke o kopici in drugem zasedenem pomnilniku, kot so trenutna poraba, največja poraba in začetna poraba.
- **OperatingSystemMXBean** nudi podatke o operacijskem sistemu, na katerem teče JVM, in številu procesorskih jeder, ki so na voljo.
- **PlatformLoggingMXBean** nudi podatke o beležnikih (ang. logger), na primer ime in stopnja beleženja.
- **RuntimeMXBean** nudi podatke o JVM izvajalnem okolju, kot so podani argumenti, verzija, čas izvajanja, pot do knjižnice, itd.
- **ThreadMXBean** nudi podatke o nitih, kot so število niti, njihove identifikatorje, čas, ki ga je CPE posvetil posamezni niti, itd.

2.4.4 Aplikacija

Na zadnjem in najvišjem nivoju je merjenje aplikacije [52]. Sem spadajo metrike, specifične za našo aplikacijo. Zbiralnike teh metrik večinoma implementiramo sami na določenih mestih v kodi, kjer vidimo verjetnost napak ali zakasnitev. Za boljše razumevanje je te metrike najlažje razdeliti po kategorijah [30]:

- **Meritve pretočnosti** merijo obremenitev neke komponente ali metode na časovno enoto. Primer take meritve so število klicev metode na minuto, število povpraševanj podatkovni bazi na sekundo, število zahtevkov na sekundo, itd. Implementiramo jih v metode, ki so velikokrat klicane oziroma predstavljajo ozko grlo, v metode, ki jih izpostavljamo preko REST-a, in kot merilce zahtevkov HTTP na različnih naslovih.

- **Meritve uspeha** merijo število klicev metod, povpraševanj in zahtevkov, ki so bili uspešno izvedeni, ter število odgovorov HTTP s statusno kodo 2xx.

- **Meritve napak** merijo število napak po posameznih komponentah, klicih metod in število odgovorov HTTP s statusno kodo 4xx ali 5xx. Dobro jih je kategorizirati glede na to, kako so resne.

- **Časovne meritve** merijo trajanje metod ali blokov kode. Uporabne so za ugotavljanje, kje v kodi se pojavljajo največje zakasnitve in katere dele bi bilo potrebno optimizirati.

- **Števci** se večinoma uporabljajo za spremljanje števila elementov v vrsti ali skladu.

- **Meritve odvisnih komponent** merijo komponente, od katerih je mikrostoritev odvisna. Sem spadajo ostale mikrostoritve, podatkovne baze, sporočilne vrste, servleti (*Tomcat*, *Jetty*), ogrodja REST (*Jersey*), itd.

2.5 Orodja za zbiranje metrik

Na voljo je veliko tako plačljivih kot odprtokodnih orodij za zbiranje metrik. V naslednjih podpoglavjih bomo na kratko opisali orodja, ki se uporabljajo za merjenje javanskih aplikacij. Opisali bomo tudi knjižnice, s katerimi lahko implementiramo lasten sistem, ki zbira metrike, specifične za našo aplikacijo.

2.5.1 Plačljiva orodja APM

Obstaja veliko plačljivih orodij APM, ki zbirajo široko paleto metrik in nudijo intuitivni uporabniški vmesnik za njihov prikaz. Zaradi njihove prepoznavnosti na tržišču jih uporabljajo tudi večja podjetja. Najbolj uporabljena orodja so [23, 53]:

- New Relic,
- AppDynamics,
- Dynatrace,
- Datadog.

2.5.2 Odprtokodna orodja APM

Na voljo je tudi kar nekaj odprtokodnih orodij, ki so v veliko primerih dobra alternativa plačljivim. Seznam boljših odprtokodnih orodij APN [21]:

- **Stagemonitor** je namenjen izvajanju meritev na aplikacijah, ki tečejo na številnih strežnikih.
- **Pinpoint** je namenjen merjenju velikih porazdeljenih sistemov.
- **MoSKito** vključuje orodje za merjenje aplikacije na več vozliščih in strežnik, ki shranjuje zbrane metrike.
- **Glowroot** je hitro in preprosto orodje APM.

- **Kamon** je namenjen merjenju aplikacij, ki so zgrajene na Typesafe Reactive Platformi.

2.5.3 Knjižnice

Na voljo je tudi nekaj knjižnic, ki vsebujejo merilce za zbiranje metrik podatkovnih baz, povezav HTTP, virtualnega stroja, itd. Prav tako poenostavijo oblikovanje lastnih merilcev, ki jih potrebujemo za zbiranje metrik, specifičnih za našo aplikacijo. Najbolj razširjene za okolje Java so [37]:

- **Dropwizard Metrics** nudi veliko merilnih orodij, podrobno opisana v naslednjem poglavju.
- **Parfait** nudi zbiranje metrik z orodjema števec in časovnik. Metrike izvaža preko JMX ali Performance Co-Pilot platforme.
- **JAMon** podpira merjenje povezav HTTP, Spring aplikacij, JDBC, SQL, Log4j, zrn EJB in zbiralca smeti.
- **Java Simon** je izboljšava orodja JAMon, nudi merjenje z orodjema števec in časovnik.

Pri implementaciji orodja za zbiranje metrik v mikrostoritvah smo za osnovo izbrali knjižnico Dropwizard Metrics. Za to knjižnico smo se odločili, ker nudi največje število merilnih orodij in največ merilcev, namenjenih zbiranju metrik v ogrodjih, od katerih je aplikacija odvisna. Pri knjižnici Dropwizard Metrics imamo na voljo merilna orodja števec, histogram, meter, merilnik in časovnik, v ostalih knjižnicah pa smo omejeni na števec in časovnik. Prav tako le knjižnica Dropwizard Metrics nudi zbiranje metrik Ehcache, Jersey, Jetty, Apache HttpClient in možnost preverjanja vitalnosti. Nudi tudi največ možnosti pri poročanju metrik. Ker smo knjižnico Dropwizard Metrics uporabili pri naši razširitvi, smo jo podrobneje opisali tudi v naslednjem poglavju.

Poglavje 3

Metrike Dropwizard

Dropwizard Metrics je javanska knjižnica, ki nudi enostavno zbiranje sistemskih metrik. Je ena izmed najbolj uporabljenih knjižnic za ta namen in pokriva zbiranje skoraj vseh zgoraj omenjenih metrik. V nadaljevanju bomo podrobno opisali orodja, s katerimi zbiramo metrike, kako jih združujemo v registre, katere sistemske metrike imamo na razpolago in na kakšne načine jih lahko izpostavljamo. V nadaljevanju se bomo zgledovali po dokumentaciji [50].

3.1 Register

Register je vsebnik za vse zbrane metrike. Shranjuje jih glede na njihov tip in nudi pridobivanje metrik glede na njihovo ime. Register definiramo takole:

```
final MetricRegistry registry = new MetricRegistry();
```

Ustvarimo lahko tudi več registrov, s katerimi meritve grupiramo in dosežemo boljšo preglednost. Evidenco o registrih vodi statičen razred “SharedMetricRegistries”, ki omogoča dostop do registrov tudi v razredih, kjer niso bili definirani. Primer uporabe:

```
final MetricRegistry registry =
   SharedMetricRegistries.getOrCreate("register");
```

3.2 Merilnik

Merilnik (ang. gauge) je preprosto orodje, ki periodično meri določeno vrednost. Ustvarimo ga takole:

```
registry.register("merilnik", new Gauge<Integer>() {
    @Override
    public Integer getValue() {
        return value;
    }
});
```

Zgoraj definirani merilnik meri celoštevilsko vrednost. Na voljo imamo tudi drugačne tipe merilnikov, kot so:

- **JMX merilnik** meri določeno vrednost MBeana. Kot atribut sprejme ime zrna in ime vrednosti.
- **Merilnik razmerja** meri razmerje med dvema vrednostma. Ustvarimo ga podobno kot navaden merilnik, le da mu podamo dve vrednosti
- **Predpomnjen merilnik** je namenjen pridobivanju vrednosti, ki so računsko zahtevne. Vrednost, ki jo bere, je hranjena za čas, ki ga določimo z atributom. V tistem času merilnik vrne shranjeno vrednost.
- **Izpeljan merilnik** nudi izpeljavo vrednosti iz drugih merilnikov.

3.3 Števec

Števec (ang. counter) je preprosto orodje za prištevanje in odštevanje 64-bitnih vrednosti. Uporabljamo ga lahko takole:

```
final Counter counter = registry.counter("stevec");
stevec.inc();
```

Večinoma se uporablja za sledenje številu elementov v vrsti ali skladu, štetje klicev metod, neuspešnih poskusov določene operacije ali števila povezav.

3.4 Histogram

Histogram je prav tako merilno orodje, ki iz zbranih podatkov izračuna statistične vrednosti, kot so minimum, maksimum, povprečje, standardni odklon in kvartile. Uporabno je, ko želimo podrobno analizirati posamezno vrednost in ko meritve zbiramo dovolj pogosto, da so statistične vrednosti točne. Primer uporabe histograma je pri merjenju velikosti zahtevkov in odgovorov, čas dostopa do podatkovne baze, čas obdelave zahtevka ter poraba sistemskih virov, kot so CPE, delovni pomnilnik, swap, itd. Uporabljamo ga lahko takole:

```
final Histogram histogram = registry.histogram("histogram");
histogram.update(value);
```

3.5 Meter

Meter je namenjen meritvam števila dogodkov v časovnem obdobju. Nudi celotno število dogodkov od začetka merjenja in povprečno število dogodkov na sekundo. Prav tako ponuja povprečja dogodkov iz zadnjih petnajst minut, zadnjih pet minut in zadnje minute, ki so izračunana s pomočjo pomikajočega povprečja (EWMA). Primer uporabe metra:

```
final Meter meter = registry.meter("meter");
meter.mark();
```

3.6 Časovnik

Časovnik (ang. timer) meri, koliko časa traja, da se blok kode izvede. Ob vsaki izvedbi kode je izmerjen čas in izračunano povprečje vseh do sedaj izmerjenjih meritev. Tako kot pri metru so tudi pri časovniku izmerjena povprečja v zadnjih minutah, pridobimo pa lahko tudi enake statistične podatke kot pri histogramu. Primer uporabe:

```
final Timer timer = registry.timer("casovnik");
final Timer.Context context = timer.time();
..
context.stop();
```

3.7 Preverjanje vitalnosti

Preverjanje vitalnosti (ang. health check) je merilnik, ki testira, če določena komponenta, npr. podatkovna baza, deluje pravilno. Od klasičnega zbiranja metrik se razlikuje tako, da preveri le, ali se komponenta odziva oziroma ali je aktivna. Preverjanje vitalnosti ne zbira številskih vrednosti, temveč le binarno vrednost in je zato primereno pri hitrem ugotavljanju, ali vse komponente delujejo ter kje je prišlo do izpada. Uporabimo ga tako, da podamo, katero metodo mora modul klicati in kakšen je pričakovan rezultat. V primeru nepričakovanega rezultata, torej nezdrave komponente, se zabeleži izpis napake. Spodaj je naveden primer registracije preverjanja vitalnosti, pri čemer je "TestVitalnosti" razred, ki implementira klic metode komponente, ki jo je potrebno preveriti.

```
registry.register("preverjanje-vitalnosi", new
   TestVitalnosti());
```

3.8 JVM

Modul za zbiranje JVM metrik vsebuje merilce, pridobljene iz zrn MX, opisanih v podpoglavju 2.4.3. Merilci so razdeljeni v tri skupine:

- **Zbiralec smeti** vsebuje podatke, pridobljene iz zrna GarbageCollectorMXBean,
- **poraba pomnilnika** vsebuje podatke, pridobljene iz zrn MemoryManagerMXBean, MemoryMXBean in MemoryPoolMXBean,
- **stanje niti** vsebuje podatke, pridobljene iz zrna ThreadMXBean.

3.9 Apache HttpClient

Modul za instrumentacijo *Apache HttpClienta* vsebuje merilce števila vseh povezav in tistih, ki so aktivne, ter stopnjo hitrosti odpiranja novih povezav. Vsebuje tudi časovnike, namenjene merjenu trajanja HTTP zahtevkov. Ker instrumentiran odjemalec deduje od HttpClient razreda, ga ustvarimo na podoben način:

```
HttpClient client =
    InstrumentedHttpClients.createDefault(registry);
```

3.10 JDBI

JDBI modul nudi zbiranje meritev za knjižnico JDBI. JDBI je javanska knjižnica, ki poenostavi dostop do relacijskih podatkovnih baz. Modul nudi časovnike za merjenje trajanja SQL ukazov. Vključuje tudi več načinov poimenovanja teh časovnikov. Primer inicializacije:

```
final DBI dbi = new DBI(dataSource);
dbi.setTimingCollector(new
    InstrumentedTimingCollector(registry));
```

3.11 Ehcache

Ehcache modul nudi merilnike za spremljanje statistike predpomnilnika Ehcache. Ehcache je odprtokodno orodje, ki nudi predpomnilnik za javanske aplikacije. Omogoča začasno shranjevanje pogosto dostopanih podatkov in s tem hitrejše delovanje aplikacije. Modul zbira veliko meritev, kot na primer število elementov v predpomnilniku, na disku in v pomnilniku, ter podatek o tem, kolikokrat je bil iskan element najden na teh lokacijah. Na voljo so tudi časovniki, ki merijo čas iskanja in pridobivanja elementov.

3.12 Jersey

Jersey modul omogoča instrumentacijo REST metod s pomočjo anotacij. Na voljo je 5 različnih anotacij:

- **@Timed** ustvari časovnik in meri čas trajanja metode ob vsakem klicu.
- **@Counted** ustvari števec in ga poveča ob klicu metode.
- **@Gauge** ustvari merilnik in zabeleži vrnjeno vrednost klicane metode.
- **@CachedGauge** ustvari predpomnjen merilnik in v primeru, da je čas hrambe stare vrednosti potekel, zabeleži vrnjeno vrednost metode.
- **@Metered** ustvari meter in označi dogodek ob vsakem klicu.

3.13 Jetty

Jetty modul omogoča instrumentacijo Jetty servletov. Na voljo je široka paleta metrik, ki so razdeljene v tri sklope:

- **Povezave**, kamor spadajo meritve o trajanju in številu povezav ter stopnji sprejemanja in zapiranja novih povezav.

- **Bazen niti**, kamor uvrščamo podatke o aktivnih in neaktivnih nitih ter razmerje med njimi.

- **Upravljalec povezav HTTP**, kamor spadajo podatki o številu zahtevkov, potečenih povezav, metrike odgovorov HTTP, razdeljene po statusnih kodah, in metrike o zahtevkih GET in POST.

3.14 Beleženje

Na voljo sta dva modula, *Log4j* in *Logback*, ki merita stopnjo zabeleženih dogodkov glede na njihovo pomembnost. Oba lahko registriramo programsko.

3.15 Spletne aplikacije

Modul za merjenje spletnih aplikacij nudi filter za servlet, ki meri število zahtevkov in odgovorov glede na statusno kodo ter časovnike za merjenje trajanja povezav. Filter je mogoče definirati v web.xml datoteki, kjer je potrebno določiti tudi URL, na katerem filter opravlja meritve. Primer inicializacije:

```
<filter>
    <filter-name>instrumentedFilter</filter-name>
    <filter-class>
    com.codahale.metrics.servlet.InstrumentedFilter
    </filter-class>
</filter>
<filter-mapping>
    <filter-name>instrumentedFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

3.16 Poročanje metrik

Na voljo je veliko načinov za izpis ali izvoz zbranih metrik v različnih formatih:

- **Konzola**; metrike lahko izpišemo v konzolo, pri čemer lahko uporabnik nastavi interval poročanja in časovno enoto.
- **JMX**; metrike lahko izpostavimo v obliki JMX MBeanov.
- **CSV**; metrike lahko periodično izvažamo v .csv datoteke. Konfiguriramo lahko interval poročanja, časovno enoto metrik in direktorij, kamor se metrike izvažajo.
- **SLF4J**; metrike lahko posredujemo v *SLF4J* beležnik.
- **Servlet**; metrike lahko izpostavljamo s servletom v JSON formatu. Na voljo je pet servletov, in sicer servlet, namenjen prikazu splošnih metrik določenega registra, servlet za prikaz testov vitalnosti, servlet za izpis stanja niti v JVM, servlet za preverjanje odzivnosti drugih komponent (ang. ping) in administratorski servlet, ki združuje vse druge.
- **Ganglia**; metrike lahko poročamo *Ganglia* strežniku v obliki toka podatkov.
- **Graphite**; metrike lahko poročamo *Graphite* strežniku v obliki toka podatkov.

3.17 Druge knjižnice

Razvitih je bilo tudi veliko zunanjih knjižnic. Namenjene so prilagajanju zbiranja metrik v drugih okoljih, kot so CDI, AspectJ in Spring, optimizaciji delovanja v drugih programskih jezikih, kot so Scala in Clojure, ter poročanju metrik v veliko različnih okolij, kot je na primer Kafka, InfluxDB, Cassandra, DataDog in mnogo drugih [49].

Poglavje 4

Načrtovanje in razvoj rešitve za zbiranje metrik mikrostoritev z ogrodjem KumuluzEE

Dropwizardov modul za zbiranje metrik vsebuje možnost zbiranja širokega spektra metrik in zadostuje pri doseganju končnega cilja diplomskega dela. Ker pa zbiramo metrike v svetu mikrostoritev, potrebujemo še ogrodje, ki nam pomaga pri razvoju in gradnji le–teh. Za ta namen uporabimo ogrodje KumuluzEE, ki vsebuje vse potrebne komponente za razvoj in gradnjo prostorsko nepotratnih mikrostoritev, prilagojenih za izvajanje v vsebnikih. Izdelamo tudi razširitev, ki integrira Dropwizardove metrike v okolje KumuluzEE.

4.1 KumuluzEE

KumuluzEE [34] je JavaEE ogrodje, namenjeno razvoju mikrostoritev in migraciji obstoječih aplikacij v mikrostoritve. Ogrodje zapakira mikrostoritve v JAR datoteke vključno z odvisnostmi. Vključuje tudi razširitve, ki so v pomoč pri razvoju mikrostoritev. Primer take razširitve je konfiguracijski modul, ki nastavitve beleži lokalno ter v etcd in Consul shrambi. Prav tako

je na voljo razširitev za beleženje dogodkov (ang. logging), odkrivanje storitev (ang. service discovery), prekinjevalci toka (ang. circuit–breakers) in druge [25].

4.2 Razširitev za zbiranje metrik

Razširitev vsebuje osnovno komponento Dropwizard Metrics modula, ki vsebuje vsa merilna orodja za zbiranje lastnih metrik, in komponento JVM, ki vsebuje merilce za zbiranje metrik JVM. Ob tem pa je v razširitev dodanih nekaj drugih funkcionalnosti, ki poenostavijo zbiranje in izpostavljanje meritev ter ga prilagodijo svetu mikrostoritev. Te funkcionalnosti so zbiranje metrik s pomočjo anotacij, možnost konfiguracije in izboljšano izpostavljanje metrik.

4.2.1 Anotacije

V razširitev vključimo možnost anotiranja metod, kar ob vsakem klicu metode sproži določeno meritev, in anotiranja spremenljivk, kar skrajša njihovo definiranje. Pri tem lahko uporabimo metode, opisane v podpoglavju 3.12, pri čemer nismo omejeni z uporabo izključno v razredih, namenjenih storitvam REST. Pri tem si pomagamo s knjižnico Metrics CDI, ki omogoča podporo za anotacije v okoljih CDI. CDI je funkcija v Javi, ki omogoča dostop do objektov iz drugih zrn, pri čemer ohranja šibko sklopljenost komponent. To knjižnici omogoča prestrezanje klicev anotiranih metod in konstruktorjev.

Primer uporabe je merjenje trajanja metode. To lahko po novem dosežemo s @Timed anotacijo, ki ji podamo ime meritve:

```
@Timed(name = "casovnik")
public void merjenaMetoda() {
    ...
}
```

Primer uporabe anotacij na spremenljivkah:

```
@Inject
@Metric(name = "requests")
Meter merilnik;
```

4.2.2 Konfiguracija

V razširitev dodamo tudi možnost konfiguracije posameznih komponent. Konfiguracijo lahko razvijalec definira v konfiguracijski datoteki. Parametri, ki jih lahko nastavi, so ime privzetega registra, naslov, na katerem so dostopne metrike v JSON-u, in naslov za Prometheusov format (podrobneje opisan v podpoglavju 6.5) ter ime registra, v katerem so zabeležene metrike JVM. Prav tako lahko razvijalec definira ime storitve, trenutno verzijo in ime instance. Ker metrike istočasno javlja mnogo mikrostoritev, je ime instance pomembno pri ugotavljanju, kam določene metrike spadajo. Primer take konfiguracijske datoteke:

```
kumuluzee:
  service-name: testna-storitev
  instance: instanca1
  version: 0.0.1
  metrics:
    genericregistryname: default
    jvm:
      enabled: true
      registry: jvm
    servlet:
      enabled: true
      mapping: /metrics
    prometheus:
      enabled: true
      mapping: /prometheus
```

4.2.3 Izpostavljanje metrik

Dropwizardov modul nudi izpostavljanje metrik, a ne vključuje vseh funkcionalnosti, ki bi jih potrebovali za izvajanje v okolju mikrostoritev in za doseganje naloge, zadane v tem diplomskem delu. V razširitvi dodamo možnost prikaza večih registrov naenkrat in opcijo filtriranja teh registrov s pomočjo poizvedb v naslovu URL. Prav tako dodamo možnost prikaza metrik iz vseh registrov v Prometheusovem formatu, ki je v našem primeru ključen, saj ga potrebujemo za doseganje avtomatskega skaliranja. V izpisu je vidno tudi ime instance, ki pove, pod katero mikrostoritev spadajo prebrane metrike.

Primer skrajšanega izpisa v formatu JSON:


```
{
  "service" : {
    "timestamp" : "2017-07-15T13:11:17.208Z",
    "environment" : "dev",
    "name" : "testna-storitev",
    "version" : "0.0.1",
    "instance" : "instanca1",
    "availableRegistries" : [ "jvm", "default" ]
  },
  "registries" : {
    "jvm" : {
      "version" : "3.1.3",
      "gauges" : {
        "GarbageCollector.PS-MarkSweep.count" : {
          "value" : 1
        },
        ...
      }
    },
    "default" : {
```

```
    "version" : "3.1.3",
    "gauges" : {
      "com.kumuluz.ee.kumuluzee_metrics.Resource.
      customer_count_gauge" : {
        "value" : 5
      }
    },
  }
 }
}
```

Primer skrajšanega izpisa v Prometheusovem formatu (podrobneje opisan v podpoglavju 6.5):

```
# HELP
com_kumuluz_ee_kumuluzee_metrics_Resource_counter
Generated from Dropwizard metric import
(metric=com.kumuluz.ee.kumuluzee_metrics.Resource.counter,
type=com.codahale.metrics.Counter)
# TYPE
com_kumuluz_ee_kumuluzee_metrics_Resource_counter gauge
KumuluzEE_com_kumuluz_ee_kumuluzee_metrics_Resource_counter
{environment="dev",serviceName="testna-storitev",
serviceVersion="0.0.1",instanceId="instanca1",} 5.0
```

V poglavju smo opisali integracijo knjižnice Dropwizard Metrics v ogrodje KumuluzEE v obliki razširitve. Opisali smo izboljšave, s pomočjo katerih je razširitev preprostejša za uporabo in bolj primerna za izvajanje v okolju mikrostoritev. Pod te izboljšave spadajo možnost konfiguracije preko namenske konfiguracijske datoteke, izboljšano izpostavljanje metrik in možnost zbiranja metrik s pomočjo anotacij.

Poglavje 5

Kubernetes

Kubernetes [4, 7] je odprtokodno ogrodje, namenjeno orkestraciji, upravljanju in skaliranju vsebnikov. Je fleksibilen in lahko teče tako na javnih kot tudi na zasebnih oblakih, neodvisno od strojne opreme ali operacijskega sistema ter s tem nudi nivo abstrakcije in boljšo izkoriščenost virov. Vgrajeno ima odkrivanje storitev (ang. service discovery), izenačevalnik obremenitve ter posodabljanje mikrosoritev brez prekinitev v delovanju aplikacije. Omogoča tudi avtomatsko zaznavanje neodzivnih vsebnikov in njihovo zamenjavo ter razporeditev novih vsebnikov glede na obremenjenost gostitelja. Na voljo je tudi avtomatsko skaliranje vsebnikov na podlagi utilizacije CPE-ja in z nekaj truda tudi drugih metrik, pridobljenih iz samih mikrostoritev. Več o tem bomo povedali v naslednjem poglavju. Kubernetes deli vozlišča na dve vrsti, in sicer *glavna* (ang. master) in *delovna* (ang. worker). Na glavnih vozliščih je nameščenih več komponent, ki so dostopna preko vmesnika API in upravljajo z gručo, medtem ko so delovna vozlišča namenjena izvajanju nalog, ki so jim bile določene. Glavna vozlišča lahko prilagodimo tudi tako, da opravljajo naloge delovnih. Diagram vozlišč, komponent in njihove interakcije lahko vidimo na sliki 5.1. V opisu komponent in enot se zgledujem po dokumentaciji [10] in [12].

5.1 Komponente

5.1.1 Etcd

CoreOS etcd je podatkovna baza tipa ključ-vrednost (ang. key-value), ki je porazdeljena po več vozliščih. Nudi izbiro voditelja (ang. leader election) in tolerira izpade vozlišč, tudi vodilnih. Vrednosti, shranjene v etcd pomnilniku, lahko tudi spremljamo in ob spremembi gruče rekonfiguriramo. Kljub temu, da je bil etcd razvit za CoreOS, deluje na številnih drugih operacijskih sistemih, kot so druge distribucije Linuxa, BSD in MacOS. V Kubernetesu se etcd uporablja za shrambo konfiguracijskih podatkov, ki so nato dosegljivi vsem vozliščem. Nameščen je lahko le na glavnih vozliščih. Čeprav je dovolj le eno, se v produkciji v večini uporablja večje (liho) število etcd vozlišč.

5.1.2 Strežnik API

Tako kot etcd je tudi *strežnik API* nameščen le na glavnih vozliščih. Nudi komunikacijo med komponentami gruče preko RESTful vmesnika. Prav tako nudi tudi centralno točko, kjer lahko uporabnik upravlja vse enote, kot so npr. *storitve* in *stroki* (opisani v naslednjih poglavjih). Za ta namen je na voljo tudi konzolni vmesnik kubectl, do katerega lahko dostopamo preko lokalnega računalnika.

5.1.3 Razporejevalec

Razporejevalec (ang. scheduler) skrbi za enakomerno razporeditev strokov po vozliščih. Pri določanju vozlišča upošteva njegovo zasedenost, ob tem pa enake stroke razporeja na različna vozlišča ter s tem poskrbi dosegljivost mikrostoritve v primeru izpada vozlišča. Razporejevalec nudi tudi napredne možnosti, kjer lahko uporabnik določi, na katerih vozliščih se mora določen strok izvajati in na katerih se ne sme. To je uporabno v primeru, ko lahko neka mikrostoritev teče le na določeni strojni opremi ali platformi. Tudi razporejevalec je nameščen le na glavnih vozliščih.

5.1.4 Horizontalni razmnoževalec strokov

Horizontalni razmnoževalec strokov (ang. horizontal pod autoscaler - HPA) [5, 19] je zadolžen za avtomatsko replikacijo strokov v določenem replikacijskem nizu (opisan v podpoglavju 5.2.2). Število replik prilagaja glede na zasedenost CPE ali poljubne metrike, ki jo izpostavlja mikrostoritev. Pri tem je način z uporabo CPE privzet in enostaven za konfiguracijo, način s poljubnimi metrikami pa v zgodnji različici in zato zahtevnejši za konfiguracijo (podrobneje opisano v podpoglavju 6.4). Pri inicializaciji moramo razmnoževalcu strokov podati mejo, ob kateri pride do replikacije, in največje možno število strokov.

Razmnoževanje strokov se izvaja periodično, na intervalu, ki ga sami določimo ob inicializaciji gruče. Vsako periodo razmnoževalec povpraša strežnik API po vrednosti, ki predstavlja zasedenost CPE. To vrednost strežnik API pridobi od Heapsterja (opisan v podpoglavju 6.1), ki izmeri porabo vsakega stroka, naredi povprečje in iz povprečja izračuna odstotek celotne porabe CPE. V primeru poljubnih metrik pa razmnoževalec povpraša posebno skupino API, ki jo moramo ustvariti sami. V obeh primerih razmnoževalec nato pridobljeno vrednost primerja z mejo ter v primeru prekoračene meje poveča število strokov v replikacijskem nizu, za katerega je zadolžen. Če je število strokov enako določenemu maksimumu, tudi ob preseženi meji do replikacije ne pride.

Primer konfiguracije razmnoževalca, ki kot vir uporablja CPE [5]:

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: testna-mikrostoritev
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1beta1
```

```
    kind: ReplicaSet
    name: testna-mikrostoritev
  minReplicas: 1
  maxReplicas: 5
  targetCPUUtilizationPercentage: 50
```

V opisanem primeru razmnoževalec upravlja z replikacijskim nizom "testna-mikrostoritev" in ga razmnoži, ko poraba CPE preseže 50%. Največje možno število strokov pa je 5.

5.1.5 Docker

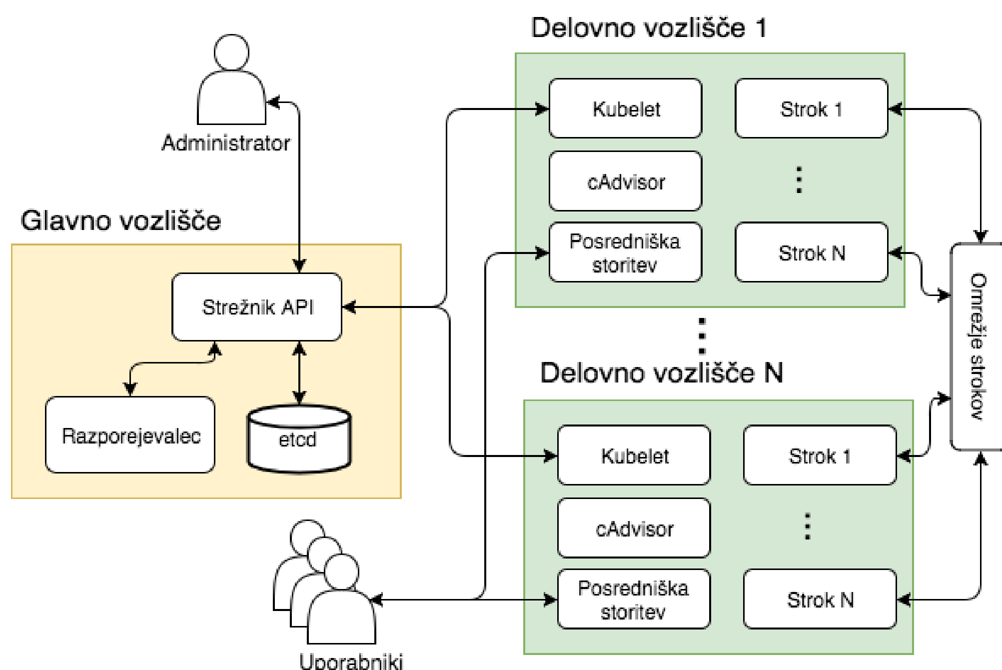
Docker [47] je tehnologija, ki nudi pakiranje storitev v vsebnike in okolje, v katerem se lahko izvajajo. S tem nudi tudi nivo abstrakcije in izolacije ter s tem možnost izvajanja mnogih mikrostoritev na enem gostitelju. Za ta namen jih uporablja Kubernetes kot osnoven konstrukt, ki ga razmnožuje, razporeja, nadzoruje in na splošno upravlja. Docker izvajalno okolje mora biti nameščeno na vseh vozliščih.

5.1.6 Kubelet

Kubelet je majhna storitev, ki se, prav tako kot Docker, izvaja na vsakem vozlišču in je kontaktna točka z gručo ter ostalimi vozlišči. Zadolžena je za prenašanje informacij med vozlišči in interakcijo z etcd strežnikom. Kubelet, ki teče na delovnih vozliščih, dobi od glavnih vozlišč ukaze o strokih in ostalih enotah ter je zadolžen za vzdrževanje predpisanega stanja.

5.1.7 Posredniška storitev

Posredniška storitev (ang. proxy service) je nameščena na vseh vozliščih in je zadolžena za posredovanje zahtevkov vsebnikom. Skrbi tudi za preprosto izenačevanje obremenitve (ang. load balancing) in izoliranost ter hkrati dostopnost storitev izven gruče.



Slika 5.1: Diagram komponent in vozlišč, ki sestavljajo gručo Kubernetes.

5.2 Enote

5.2.1 Strok

Strok (ang. pod) je osnovni gradnik v Kubernetesu, ki vsebuje enega ali več vsebnikov. Več vsebnikov zapakiramo takrat, ko so med seboj odvisni in je zato praktično, da si delijo lokalno omrežje in IP naslov. S tem zagotovimo, da so razporejeni na istem gostitelju. Tako kot vsebniki imajo tudi stroki kratko življensko dobo in se večinoma izvajajo v večih instancah. Strok lahko ustvarimo na dva različna načina, replikacijski strok (ang. replicated pod) ustvarimo preko replikacijskega niza, osnoven strok pa ustvarimo sami.

Primer koniguracijske datoteke za strok:

```
apiVersion: v1
kind: Pod
metadata:
```

```
  name: strok
  labels:
    app: metrics
spec:
  containers:
    - name: metrics
      image: aljazb/metrics
      ports:
        - containerPort: 8080
```

5.2.2 Replikacijski niz

Replikacijski niz (ang. replica set) je ogrodje, ki upravlja s stroki istega tipa ter skrbi, da so v predpisanem številu. V primeru neodzivnosti vsebnika ali ukazu po večjem številu vsebnikov, replikacijski niz avtomatsko ustvari nov vsebnik. Če neodzivni vsebnik ponovno postane funkcionalen, je eden od vsebnikov uničen.

Primer koniguracijske datoteke za replikacijski niz:

```
apiVersion: extensions/v1beta1
 kind: ReplicaSet
 metadata:
   name: niz
 spec:
   replicas: 3
   selector:
     matchLabels:
       app: metrics
     spec:
       containers:
         - name: metrics
           image: aljazb/metrics
```

```
      ports:
        - containerPort: 8080
```

5.2.3 Storitev

Storitev (ang. service) v Kubernetesu je abstrakcija, ki grupira podobne stroke, ki opravljajo enako funkcijo, in jih enovito izpostavlja. Zunanje aplikacije zato vidijo storitve kot eno dostopno točko in ne kot množico vsebnikov, ki se neprestano menjajo. Storitev sledi spremembam vsebnikov, kot so odpovedi, prerazporeditve in množenje, ter med njimi izvaja izenačevanje obremenitve, ob tem pa ohranja IP naslov, preko katerega je dostopna. Pri upravljanju s stroki si pomagajo z označbami (opisane v naslednjem poglavju).

Primer koniguracijske datoteke za storitev:

```
kind: Service
apiVersion: v1
metadata:
  name: storitev
spec:
  selector:
    app: metrics
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

5.2.4 Označba

Označbe (ang. labels) so namenjene organizaciji in določanju, katera enota spada v posamezno skupino. To koristijo storitve, ki glede na označbe usmerjajo promet pravilnim strokom, in replikacijski nizi, ki tako lažje spremljajo

število in stanje strokov, s katerimi upravljajo. Označbe so definirane v obliki ključ-vrednost, kar omogoča, da ima lahko posamezna enota več označb, posamezna označba pa mora imeti le eno vrednost. Enote, ki imajo določenih veliko označb, lahko lažje najdemo tako, da označbe kombiniramo.

5.3 Namestitev

Na voljo je veliko načinov za namestitev Kubernetesa glede na operacijski sistem, a trenutno najbolj razširjen in priporočen pristop je z orodjem Kubeadm [51]. Ta omogoča namestitev gruče na Ubuntu, CentOS in HypriotOS distribucijah ter deluje tako na javnih oblakih kot tudi na fizičnih strežnikih in virtualnih strojih. Namestitev gruče začnemo z ukazom "kubeadm init", ki najprej preveri kompatibilnost sistema ter prikaže morebitna opozorila in napake. Nato orodje generira žeton, preko katerega se bodo vozlišča lahko pridružila gruči, in certifikate, ki se bodo uporabljali pri avtentikaciji komponent in vozlišč. Po ustvarjenih certifikatih orodje ustvari konfiguracijsko datoteko, ki jo kubelet potrebuje za povezavo na strežnik API, in manifest, v katerem so zapisani stroki, ki jih mora kubelet ustvariti ob zagonu. Zaradi varnostnih razlogov kubeadm konfigurira vozlišče tako, da se na njem lahko zaženejo le sistemski stroki. To lahko v primeru, da nimamo na voljo veliko vozlišč, tudi spremenimo. Namestiti je treba tudi *omrežje strokov* (ang. pod network), ki skrbi za komunikacijo med stroki. Na voljo je več omrežij, najbolj razširjeni sta flannel in Weave Net. V gručo lahko zdaj dodamo delovna vozlišča tako, da na njih poženemo ukaz "kubeamd join" in podamo IP glavnega vozlišča ter prej generiran žeton.

Gručo smo vzpostavili na več načinov in z različnimi orodji, pri čemer smo iskali konfiguracijo, ki omogoča uporabo večih vozlišč in možnost skaliranja na podlagi lastnih metrik. Začeli smo z orodjem Minikube, ki ga je mogoče namestiti na Linux, MacOS in Windows operacijskih sistemih ter je namenjeno vzpostavitvi testnega okolja in učenju uporabe Kubernetesa. Orodje je omejeno le na eno vozlišče in ne omogoča vzpostavitve produkcijske

gruče in ga zato tudi nismo uporabili. Nato smo testirali orodje conjure-up, ki omogoča vzpostavitev večvozliščne gruče na sistemih Ubuntu. Z orodjem conjure-up smo konfigurirali gručo z dvemi vozlišči, a ga pri končnem izdelku nismo uporabili, saj ne omogoča prilagoditve strežnika API, ki je potrebna za možnost skaliranja na podlagi lastnih metrik. Preizkusili smo tudi ročno konfiguracijo gruče, ki je prav tako nismo uporabili pri končnem izdelku, saj je zastarela, zahteva veliko konfiguracije in je omejena na sisteme CentOS. Njena izboljšava in priporočen pristop je z orodjem Kubeadm [51], ki smo ga tudi sami uporabili za konfiguracijo gruče. Glavno vozlišče smo konfigurirali na virtualnem stroju Ubuntu. Nato smo mu dodali delovno vozlišče, ki je bilo konfigurirano na sistemu CentOS. Ker nismo imeli na voljo velikega števila virtualnih strojev, smo na glavnem vozlišču omogočili izvajanje vseh strokov in mu s tem dodali tudi naloge delovnih vozlišč. Namestili smo tudi omrežje strokov Weave Net. Ker smo želeli možnost prilagojenega skaliranja, smo pri inicializaciji gruče določili konfiguracijske parametre, ki so to omogočili. S temi parametri smo aktivirali poskusno različico razmnoževalca strokov (autoscaling/v2alpha1) in mu sporočili, da informacije o replikaciji povzema iz tuje skupine API (podrobneje opisana v podpoglavju 6.6). Prav tako smo zmanjšali interval izvajanja razmnoževalca strokov na 10 sekund, da smo lahko hitreje videli odziv na spremembo obremenitve. Konfiguracija, ki smo jo uporabili [26]:

```
kind: MasterConfiguration
apiVersion: kubeadm.k8s.io/v1alpha1
controllerManagerExtraArgs:
  horizontal-pod-autoscaler-use-rest-clients: "true"
  horizontal-pod-autoscaler-sync-period: "10s"
  node-monitor-grace-period: "10s"
apiServerExtraArgs:
  runtime-config: "api/all=true"
  feature-gates: "TaintBasedEvictions=true"
kubernetesVersion: "stable-1.7"
```

V poglavju smo opisali ogrodje Kubernetes, komponente, ki ga sestavljajo, in enote, s katerimi upravlja. Našteli smo tudi različne načine namestitve gruče Kubernetes in opisali postopek namestitve z uporabo orodja Kubeadm. Prav tako smo opisali namestitev naše gruče in konfiguracijo, ki smo jo pri tem uporabili.

Poglavje 6

Zbiranje metrik v Kubernetesu in samodejno skaliranje

V Kubernetesu je zelo pomembno spremljanje in razumevanje delovanja tako aplikacije kot tudi infrastrukture. Tukaj metrike razdelimo na tri nivoje, metrike aplikacije, strokov in vozlišč. Uporabnik mora imeti dostop do vseh treh nivojev metrik, saj se le tako lahko izogne črnim luknjam in tako lažje odkrije vzrok napake, zakasnitve ali preobremenitve. Za ta namen se uporablja dodatek Heapster.

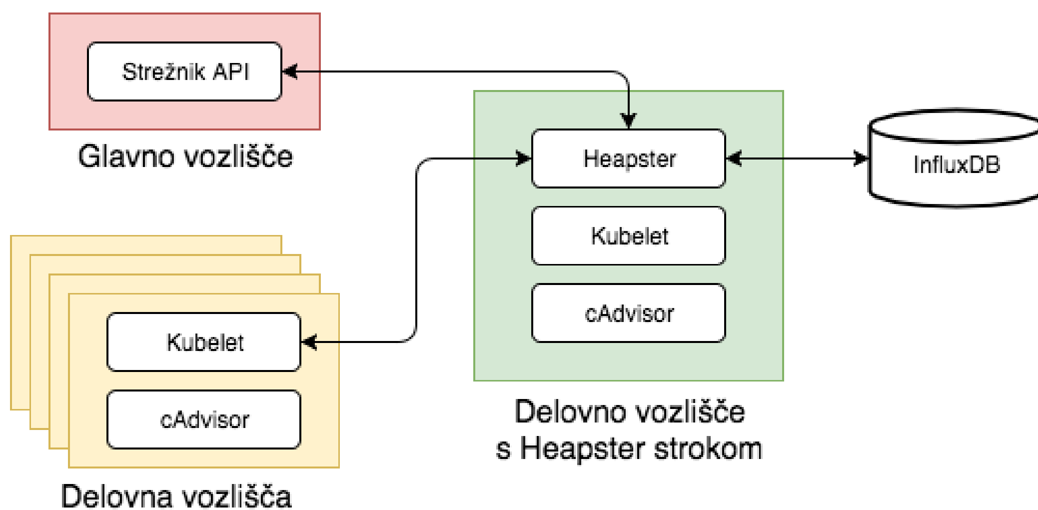
6.1 Heapster

Heapster [24, 43, 44] je agregator metrik, namenjen uporabi v okolju Kubernetes. Izvaja se v obliki stroka in avtomatsko najde vsa vozlišča. Iz kubeletov, ki tečejo na vozliščih, pridobi informacije o strokih in jih zabeleži. Na voljo je več zalednih sistemov, ki nudijo hranjenje in vizualizacijo zbranih podatkov, od katerih sta najbolj razširjena *Stackdriver Monitoring* in *InfluxDB*. Stackdriver Monitoring je Googlov produkt in deluje le na javnih oblakih GCP in AWS. Za zasebne oblake se večinoma uporablja kombinacija InfluxDB in *Grafane*. InfluxDB je odprtokodna podatkovna baza, ki je namenjena beleženju metrik, dogodkov in analitike izvajanja. Vgrajen ima

HTTP API, preko katerega lahko upravljamo s podatki. Grafana je orodje za vizualizacijo podatkov, shranjenih v eni ali več različnih podatkovnih bazah. Vsebuje tudi nadzorno ploščo, kjer so vsi ti podatki zbrani na enem mestu. Oba tečeta v strokih in se izdajata kot storitve Kubernetes, da ju lahko Heapster najde. Diagram zbiranja metrik s Heapsterjem lahko vidimo na sliki 6.1.

6.2 cAdvisor

cAdvisor [24, 43, 44] je orodje, namenjeno merjenju porabe virov v vsebnikih in izvajanju analitike nad temi podatki. Vgrajen je v kubelet, kateremu posreduje pridobljene metrike, ki jih ta nato pošilja Heapsterju. cAdvisor avtomatsko najde vse vsebnike na vozlišču in začne meriti porabo CPE, pomnilnika in omrežja. Do teh podatkov lahko dostopamo preko API-ja ali grafičnega vmesnika, kjer so na voljo informacije o celotnem vozlišču in posameznih strokih.



Slika 6.1: Diagram zbiranja metrik s Heapsterjem in cAdvisorjem.

6.3 Nadzorna plošča

Nadzorna plošča (ang. dashboard) [24] je dodatek h Kubernetesu, ki nudi osnovne informacije o enotah (strokih, storitvah, itd.), ki tečejo na posameznih vozliščih. V kombinaciji s Heapsterjem prikaže tudi podrobne metrike o vozliščih in strokih, kot so poraba delovnega pomnilnika in CPE. Nudi tudi možnost oblikovanja novih enot z nalaganjem datoteke YAML. Vse to je dostopno preko uporabniškega vmesnika.

6.4 Uporaba metrik pri samodejnem skaliranju

Za robustnost in hitro delovanje aplikacije je nujno zadostno število instanc mikrostoritve tudi ob nepričakovanem povečanju zahtevkov. Ker pa je število uporabnikov, ki dostopa do aplikacije, večinoma veliko nižje kot v vrhuncu prometa, je neprestano poganjanje števila instanc, ki zadostuje vrhuncu, potratno. Zato je najbolj učinkovit pristop avtomatsko skaliranje instanc glede na določen parameter. To imamo na voljo tudi v Kubernetesu, ki kot parameter uporablja zasedenost CPE. Konfiguracija takega skaliranja je opisana v podpoglavju 5.1.4. To zadostuje v veliko primerih manjših aplikacij, a pri večjih sistemih običajno potrebujemo več nadzora nad tem, kdaj se določena storitev replicira. To lahko dosežemo tako, da mikrostoritev sama zbira in izpostavlja metrike, ki so optimalni indikatorji obremenjenosti aplikacije, in so te nato uporabljene kot vir za skaliranje. Ker je ta pristop v Kubernetesu v času pisanja v zgodnji fazi razvoja, je konfiguracija malo bolj kompleksna in zahteva lastno skupino API, orodje Prometheus ter nekaj drugih prilagoditev, opisanih v naslednjih poglavjih. Predlog za tako implementacijo je opisan v [42], pri implementaciji smo se pa zgledovali po primeru, opisanem v [26].

6.5 Prometheus

Prometheus je odprtokodno orodje, namenjeno zbiranju metrik (več o Prometheusu lahko najdemo v [35]). Uporablja fleksibilen HTTP API, ki na določenem URL naslovu in glede na interval zbiranja prebere meritve v predpisanem formatu. Namenjen je zbiranju numeričnih vrednosti, ki jih shranjuje glede na čas meritve, ključ in samo vrednost. Ti podatki so nato dostopni preko grafičnega vmesnika, ki nudi vpogled o trenutnem in preteklem stanju sistema. Na voljo je tudi izris grafa, katerega parametre lahko sami konfiguriramo. Vsak Prometheus strežnik je samostojen in neodvisen od omrežja in pomnilnika. Zato je tudi zanesljiv in omogoča podatke tudi v primeru izpada, kar je koristno pri iskanju izvora problema.

Metrike, namenjene Prometheusu, morajo biti podane v predpisanem formatu [14]. Vsaka metrika mora imeti določen svoj tip, kar lahko naredimo z vrstico, ki se prične z oznako "# TYPE". Za to oznako sledi ime metrike in njen tip. Na voljo je pet različnih tipov:

- števec (ang. counter),
- merilnik (ang. gauge),
- histogram (ang. histogram),
- povzetek (ang. summary),
- nedoločen (ang. untyped).

V novi vrstici sledi opis metrike same, ki se prične z njenim imenom (enakim kot v opisu tipa). Nato znotraj zavitih oklepajev sledijo metapodatki metrike. Primer takih podatkov so metoda ali razred, iz katerega je bila metrika pridobljena, opozorilo, napaka, izvajalno okolje, verzija, itd. Za tem pa sledi sama vrednost metrike, ki je lahko v celoštevilski ali v decimalni obliki (pisana z decimalno piko). Metriki lahko dodamo tudi opis, in sicer pred vrstico z opisom tipa. Vrstica se mora začeti s "# HELP", čemur sledi ime

metrike in njen opis. Če se vrstica prične z "#" in se ne nadaljuje s "HELP" ali "TYPE", jo Prometheus interpretira kot komentar in jo ignorira.

Primer metrike v Prometheusovem formatu:

```
# HELP http_zahtevki Stevilo zahtevkov HTTP od zagona
   aplikacije.
# TYPE http_zahtevki counter
http_zahtevki {environment="dev",version="0.1"} 1039402
```

Na voljo je verzija Prometheusa, prilagojena zbiranju metrik v okolju Kubernetes, imenovana *Prometheus Operator* [38, 39]. Ta je enostaven za namestitev in omogoča uporabnikom oblikovanje, konfiguracijo in upravljanje z instancami, nad katerimi orodje izvaja meritve.

Osnovne lastnosti Prometheus Operatorja:

- **Oblikovanje in uničenje**; Prometheus Operator omogoča enostavno oblikovanje in uničenje instanc, ki merijo določen Kubernetesov imenski prostor (ang. namespace) ali posamezno storitev.

- **Konfiguracija**; Prometheus Operator omogoča konfiguracijo verzij, hrambe in replik direktno iz Kubernetesa.

- **Označbe**; Prometheus Operator omogoča izbiro virov za merjenje glede na Kubernetesove označbe.

Operator loči namestitev Prometheus instanc in konfiguracijo točk, ki jih merijo, ter za ta namen ustvari dva različna objekta, Prometheus in ServiceMonitor. Slednjih je več in vsak vsebuje informacije o tem, kako iz storitev pridobiti metrike, objekt Prometheus pa jih združuje in skrbi, da sistem teče tako, kot je bil konfiguriran.

Pri skaliranju na podlagi lastnih metrik je Prometheus Operator prvi korak za dosego tega cilja. Strok z mikrostoritvijo, ki izpostavlja metrike, namenjene Prometheusu, mora imeti določeno označbo, na podlagi katere jo bo ServiceMonitor našel. Prav tako morajo biti izpostavljene metrike v

Prometheus formatu. Ko v gručo Kubernetes namestimo Operator in ServiceMonitor, moramo v konfiguracijski datoteki določiti tudi prej omenjeno označbo. ServiceMonitor nato najde vse stroke s to označbo in zabeleži vse metrike, ki jih izpostavlja mikrostoritev. Te metrike lahko opazujemo in grafično izrišemo na spletnem uporabniškem vmesniku, ki ga nudi Prometheus.

6.6 Strežnik API

Naslednji korak je branje meritev, ki jih je zbral Prometheus, in posredovanje le–teh sistemu, ki je zadolžen za repliciranje strokov. Ta sistem se v Kubernetesu imenuje horizontalni razmnoževalec strokov. Parametre, ki jih uporablja kot vir pri skaliranju, pridobiva preko Kubernetes API-ja, zato je potrebno ustvariti *skupino API* (ang. API group), ki služi kot mediator, tako da bere metrike iz Prometheusa in jih posreduje razmnoževalcu strokov. To pa naredimo tako, da razširimo osnovni strežnik API z želenimi funkcionalnostmi. Ob tem se vsi podatki nove razširitve shranijo v ločeno etcd instanco. Ko ustvarimo razširitev, strežnik API avtomatsko odkrije novo skupino in omogoči ustvarjanje, izpis in brisanje na novo določenih objektov na enak način kot osnovne enote (stroki, storitve, itd.). Skupina API, namenjena posredovanju lastnih metrik, se mora imenovati "custom-metrics.metrics.k8s.io/v1alpha1", ker na tem naslovu horizontalni razmnoževalec dostopa do parametrov. Ustvariti je potrebno tudi storitev [41], ki teče na vseh delovnih vozliščih in posreduje podatke iz Prometheus instanc v ustvarjeno skupino API, ki se nahaja na glavnem vozlišču.

6.7 Konfiguracija horizontalnega razmnoževalca strokov

Zadnji korak je konfiguracija horizontalnega razmnoževalca strokov. Ob oblikovanju je potrebno podati označbo replikacijskega niza, ki nadzoruje število

strokov, ki bi jih radi replicirali. Prav tako moramo definirati, kateri parameter bo uporabljen kot vir za skaliranje in kakšno mejo mora preseči, da bo prišlo do replikacije. Uporabimo lahko tudi več parametrov, kjer moramo vsakemu določiti svojo mejo. V tem primeru bo prišlo do replikacije, ko bo vsaj eden od parametrov presegel mejo. Ker pa razmnoževalec v osnovni konfiguraciji pričakuje porabo CPE kot parameter pri replikaciji, moramo to spremeniti že pri oblikovanju gruče. Pri orodju Kubeadm lahko to naredimo s parametrom "horizontal-pod-autoscaler-use-rest-clients", ki razmnoževalcu pove, naj išče metrike preko API-ja. Prav tako moramo vklopiti "autoscaling/v2alpha1", ki je novejša skupina API, namenjena avtomatskemu skaliranju in omogoča vse te funkcionalnosti. Primer take konfiguracije Kubeadm lahko najdemo v podpoglavju 5.3.

Konfiguracijsko datoteko za razmnoževalec, ki iz strežnika API prebere vrednost "requests" in v primeru, da je večja od 100, sporoči replikacijskemu nizu z označbo "kumuluzee-metrics", naj poveča število replik (največje dovoljeno število replik je 5), zapišemo na naslednji način:

```
kind: HorizontalPodAutoscaler
apiVersion: autoscaling/v2alpha1
metadata:
  name: kumuluzee-metrics-hpa
spec:
  scaleTargetRef:
    kind: ReplicaSet
    name: kumuluzee-metrics
  minReplicas: 1
  maxReplicas: 5
  metrics:
  - type: Object
    object:
      target:
        kind: Service
```

```
      name: kumuluzee-metrics
    metricName: requests
    targetValue: 100
```

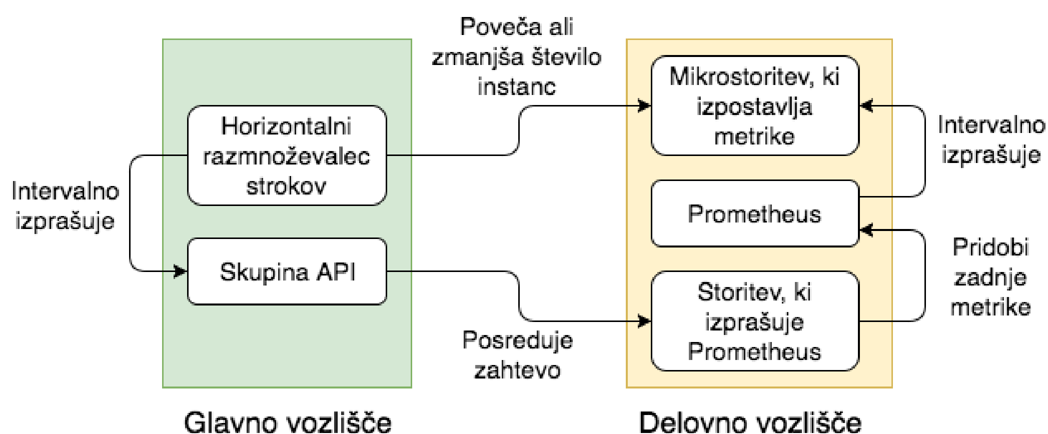
6.8 Povzetek

Replikacija na podlagi lastnih metrik je v zgodnji fazi, zato zahteva kar nekaj konfiguracije. Če povzamemo prejšnje korake (diagram je prikazan na sliki 6.2):

1. Razmnoževalec strokov intervalno povprašuje skupino API, zadolženo za lastne metrike, ta pa vrača zadnjo vrednost meritve.
2. Strežnik API posreduje zahtevo storitvi, ki teče na delovnih vozliščih, in direktno izprašuje Prometheus.
3. Prej omenjena storitev nato od Prometheusa pridobi zadnje pridobljene metrike. (Prometheus metrike zbira glede na svoj interval, neodvisno od povpraševanja razmnoževalca strokov).
4. Te metrike so nato posredovane nazaj do razmnoževalca, ta pa v primeru presežene vrednosti poveča število strokov, ki jih upravlja.

6.9 Izbira ustreznih parametrov in njihove prednosti

Prednost podajanja lastnih metrik je, da pri avtomatski replikaciji nismo omejeni le na porabo CPE in zato lahko dosežemo večji nadzor in fleksibilnost. Za skaliranje se uporabljajo metrike iz področij strojne opreme (poraba CPE, delovnega pomnilnika in diska), aplikacijskega strežnika (število zahtevkov, povezav in sej), podatkovne baze (število aktivnih niti in transakcij ter trajanje poizvedbe), sporočilnih vrst (število elementov v vrsti,



Slika 6.2: Diagram skaliranja na podlagi lastnih metrik.

čas obdelave elementa) in metrike procesa, v katerem se storitev izvaja (čas zasedenosti CPE in koliko delovnega pomnilnika zaseda proces) [15]. Najpogosteje se pri skaliranju uporablja eden ali kombinacija dveh parametrov, v nekaterih primerih pa tudi večih [29]. Najpogosteje uporabljeni so zasedenost CPE, čas obdelave zahtevka, število zahtevkov na sekundo njihove medsebojne kombinacije [29]. Prav tako se pri mikrostoritvah, ki so procesorsko in pomnilniško zahtevne, uporablja kombinacija zasedenosti CPE in delovnega pomnilnika [15, 31]. Kjub temu, da z uporabo pogostih parametrov dosežemo dokaj optimizirano prilagajanje obremenitvam, je priporočena podrobnejša analiza specifik posamezne mikrostoritve [17]. S tem lahko ugotovimo, kateri od parametrov je najboljši indikator za preobremenjenost, in s tem omogočimo hitrejši odziv na obremenitve in višjo raven prožnosti. Z izbiro pravega parametra torej dosežemo višjo odzivnost aplikacij in odpornost na izpade, ki so posledica večjih obremenitev, hkrati pa zmanjšamo porabo virov ob nižjem prometu.

Poglavje 7

Zaključek

V diplomskem delu smo razvili in opisali ogrodje, ki razvijalcu pomaga pri zbiranju pomembnih metrik, ter konfiguracijo gruče, ki te metrike uporablja pri replikaciji mikrostoritev. S tem smo optimizirali odzivnost aplikacije in porabo virov ter povečali odpornost pod velikimi obremenitvami. Opisali smo metrike, ki jih je potrebno zbirati v vseh okoljih, in tiste, ki so specifične za JavaEE okolje. Poglobili smo se tudi v knjižnico Dropwizard Metrics in predstavili merilna orodja, načine izpostavljanja metrik in sistemske meritve, ki jih modul zbira sam. Nato smo to knjižnico prilagodili okolju mikrostoritev s pomočjo ogrodja KumuluzEE, ki je aplikacijo pripravilo za izvajanje v vsebnikih. V razširitev smo dodali tudi nekaj izboljšav, ki so prišle prav v kasnejših korakih diplomskega dela. Z razširitvijo smo dosegli cilj implementacije enostavnega orodja za zbiranje metrik v mikrostoritvah. V mikrostoritvah, zgrajenih z ogrodjem KumuluzEE, lahko s pomočjo ene odvisnosti (ang. dependency) omogočimo zbiranje širokega spektra metrik in izpostavljanje le–teh v JSON ali Prometheusovem formatu. Z razširitvijo smo prav tako dosegli štetje klicev metod in merjenje njihovega trajanja s pomočjo anotacij, kar pripomore k enostavnejšemu in hitrejšemu dodajanju merilnikov ter k boljši berljivosti kode. Opisali smo tudi okolje Kubernetes, njegove komponente in enote, ki jih lahko oblikujemo, ter korake, ki so potrebni za namestitev gruče, sestavljene iz glavnih in delovnih vozlišč. Namestitev se

je ob izbiri orodja Kubeadm izkazala za dokaj enostavno, saj orodje poskrbi za večino konfiguracije. Prav tako smo predstavili orodje cAdvisor, ki zbira sistemske metrike, in razširitev Heapster, ki te metrike, pridobljene iz večih vozlišč, združuje ter jih poroča nadzorni plošči ali shrani v podatkovno bazo InfluxDB. Na koncu smo prikazali tudi enega od načinov uporabe zbranih metrik, in sicer kot podatek, kdaj je potrebno povečati ali zmanjšati število instanc določene mikrostoritve. To smo dosegli s konfiguracijo Prometheusa v kombinaciji s strežnikom API, ki posreduje informacije razmnoževalcu strokov. Podrobneje smo opisali vsak korak te konfiguracije in orodja, ki so za to potrebna. Končno konfiguracijo smo tudi testirali z mikrostoritvijo, zgrajeno v ogrodju KumuluzEE, ki je s pomočjo razširitve izpostavljala število zahtevkov na sekundo. To metriko smo nato v razmnoževalcu podali kot vir za skaliranje. V neskončni zanki smo dostopali do testne mikrostoritve, na kar se je odzval razmnoževalec in po kakšni minuti povečal število strokov. Replikacija na podlagi metrik, zbranih v mikrostoritvi, je v Kubernetesu verzije 1.7 v zgodnji fazi razvoja in je zato še razmeroma nezanesljiva in zahtevna za konfiguracijo. Za prihodnje verzije so že izdelani predlogi izboljšav, kjer ne bo potrebe po Prometheusu in drugih vmesnih komponentah.

Literatura

[1] Anand Akela. 4 Cluster Management Tools to Compare. http://blog.appdynamics.com/product/4-cluster-management-tools-to-compare/. [Dostopano: 8. 8. 2017].

[2] Autoscaling Groups of Instances. http://cloud.google.com/compute/docs/autoscaler/. [Dostopano: 8. 8. 2017].

[3] W. Barth. *Nagios, 2nd Edition: System and Network Monitoring*. No Starch Press Series. No Starch Press, 2008.

[4] D. Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, Sept 2014.

[5] Jonas Boner. Horizontal Pod Autoscaling Walkthrough. http://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. [Dostopano: 8. 8. 2017].

[6] Jonas Boner. Resilience is by design. http://zeroturnaround.com/rebellabs/resilience-is-by-design-by-jonas-boner/. [Dostopano: 8. 8. 2017].

[7] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Commun. ACM*, 59(5):50–57, April 2016.

[8] Augusto Ciuffoletti. Automated deployment of a microservice-based monitoring infrastructure. *Procedia Computer Science*, 68:163 – 172,

2015. 1st International Conference on Cloud Forward: From Distributed to Complete Computing.

[9] Louis Columbus. 2017 State Of Cloud Adoption And Security. http://forbes.com/sites/louiscolumbus/2017/04/23/2017-state-of-cloud-adoption-and-security/. [Dostopano: 8. 8. 2017].

[10] Concepts. http://kubernetes.io/docs/concepts/. [Dostopano: 29. 6. 2017].

[11] Dynamic Scaling. http://docs.aws.amazon.com/autoscaling/latest/userguide/as-scale-based-on-demand.html. [Dostopano: 5. 8. 2017].

[12] Justin Ellingwood. An Introduction to Kubernetes. http://digitalocean.com/community/tutorials/an-introduction-to-kubernetes. [Dostopano: 29. 6. 2017].

[13] Essential Server Performance Metrics you should know, but were reluctant to ask. http://monitis.com/blog/essential-server-performance-metrics-you-should-know-but-were-reluctant-to-ask/. [Dostopano: 1. 7. 2017].

[14] Exposition formats. http://prometheus.io/docs/instrumenting/exposition_formats/. [Dostopano: 8. 7. 2017].

[15] H. Ghanbari, B. Simmons, M. Litoiu, and G. Iszlai. Exploring alternative approaches to implement an elasticity policy. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 716–723, July 2011.

[16] J. P. Gouigoux and D. Tamzalit. From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 62–65, April 2017.

[17] M. Z. Hasan, E. Magana, A. Clemm, L. Tucker, and S. L. D. Gudreddi. Integrated and autonomic cloud resource scaling. In *2012 IEEE Network Operations and Management Symposium*, pages 1327–1334, April 2012.

[18] Emma Henderson. Why Scalability is Such a Big Concern When Using Nagios. `http://opsview.com/resources/nagios-alternative/blog/why-scalability-such-big-concern-when-using-nagios`. [Dostopano: 6. 8. 2017].

[19] Horizontal Pod Autoscaling. `http://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/`. [Dostopano: 5. 7. 2017].

[20] Tom Huston. What is Application Performance Management? `http://smartbear.com/learn/performance-monitoring/what-is-application-performance-management/`. [Dostopano: 20. 8. 2017].

[21] Henn Idan. Java Performance Monitoring: 5 Open Source Tools You Should Know. `http://dzone.com/articles/java-performance-monitoring-5-open-source-tools-you-should-know`. [Dostopano: 1. 7. 2017].

[22] Interface PlatformManagedObject. `http://docs.oracle.com/javase/7/docs/api/java/lang/management/PlatformManagedObject.html`. [Dostopano: 25. 6. 2017].

[23] Hayden James. 20 Top Server Monitoring and Application Performance Monitoring (APM) Solutions. `http://haydenjames.io/20-top-server-monitoring-application-performance-monitoring-apm-solutions/`. [Dostopano: 20. 8. 2017].

[24] Vishnu Kannan, Victor Marmol, and Google Software Engineers. Tools for Monitoring Compute, Storage, and Network Resources. `http://kubernetes.io/docs/tasks/debug-application-cluster/resource-usage-monitoring/`. [Dostopano: 1. 7. 2017].

[25] KumuluzEE extensions. http://github.com/kumuluz/kumuluzee/blob/master/README.md#kumuluzee-extensions. [Dostopano: 6. 8. 2017].

[26] Lucas Käldström. Building a multi-platform Kubernetes cluster on bare metal with kubeadm. http://github.com/luxas/kubeadm-workshop/blob/master/README.md. [Dostopano: 8. 7. 2017].

[27] Noah Lehmann-Haupt. Zen and the Art of System Monitoring. http://scalyr.com/community/guides/zen-and-the-art-of-system-monitoring. [Dostopano: 1. 7. 2017].

[28] List the Available CloudWatch Metrics for Your Instances. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/viewing_metrics_with_cloudwatch.html. [Dostopano: 5. 8. 2017].

[29] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A. Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing*, 12(4):559–592, Dec 2014.

[30] Alexis Lê-Quôc. Monitoring 101: Collecting the right data. http://datadoghq.com/blog/monitoring-101-collecting-data/. [Dostopano: 1. 7. 2017].

[31] M. Mao, J. Li, and M. Humphrey. Cloud auto-scaling with deadline and budget constraints. In *2010 11th IEEE/ACM International Conference on Grid Computing*, pages 41–48, Oct 2010.

[32] Kate Matsudaira. Distributed Systems Basics – Handling Failure: Fault Tolerance and Monitoring. http://katemats.com/distributed-systems-basics-handling-failure-fault-tolerance-and-monitoring/. [Dostopano: 8. 8. 2017].

[33] V. Mehta. *Icinga Network Monitoring*. Community experience distilled. Packt Publishing, 2013.

[34] Piotr Mińkowski. Java EE MicroProfile With KumuluzEE. http://dzone.com/articles/java-ee-microprofile-with-kumuluzee. [Dostopano: 6. 8. 2017].

[35] Overview. http://prometheus.io/docs/introduction/overview/. [Dostopano: 5. 7. 2017].

[36] Isuru Perera. Java Performance Monitoring Libraries. http://isuru-perera.blogspot.si/2014/11/java-performance-monitoring-libraries.html. [Dostopano: 6. 7. 2017].

[37] Isuru Perera. Java Performance Monitoring Libraries. http://isuru-perera.blogspot.si/2014/11/java-performance-monitoring-libraries.html. [Dostopano: 1. 7. 2017].

[38] Prometheus Operator. http://github.com/coreos/prometheus-operator/blob/master/README.md. [Dostopano: 6. 7. 2017].

[39] Fabian Reinartz. The Prometheus Operator: Managed Prometheus setups for Kubernetes. http://coreos.com/blog/the-prometheus-operator.html. [Dostopano: 6. 7. 2017].

[40] D. Richter, M. Konrad, K. Utecht, and A. Polze. Highly-available applications on unreliable infrastructure: Microservice architectures in practice. In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 130–137, July 2017.

[41] Solly Ross. Custom Metrics Adapter Server Boilerplate. http://github.com/kubernetes-incubator/custom-metrics-apiserver. [Dostopano: 8. 8. 2017].

[42] Solly Ross. Custom Metrics API. http://github.com/kubernetes/community/blob/master/contributors/design-proposals/custom-metrics-api.md. [Dostopano: 8. 7. 2017].

[43] Jean-Mathieu Saponaro. How to collect and graph Kubernetes metrics. http://datadoghq.com/blog/how-to-collect-and-graph-kubernetes-metrics/. [Dostopano: 1. 7. 2017].

[44] Jean-Mathieu Saponaro. Monitoring Kubernetes performance metrics. http://datadoghq.com/blog/monitoring-kubernetes-performance-metrics/. [Dostopano: 1. 7. 2017].

[45] Marco Scotto, Alberto Sillitti, Giancarlo Succi, and Tullio Vernazza. Non-invasive product metrics collection: An architecture. In *Proceedings of the 2004 Workshop on Quantitative Techniques for Software Agile Process*, QUTE-SWAP '04, pages 76–78. ACM, 2004.

[46] A. Sillitti, A. Janes, G. Succi, and T. Vernazza. Collecting, integrating and analyzing software metrics and personal software process data. In *2003 Proceedings 29th Euromicro Conference*, pages 336–342, Sept 2003.

[47] S. Singh and N. Singh. Containers docker: Emerging roles future of cloud technology. In *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*, pages 804–807, July 2016.

[48] Lorenzo Strigini. *Fault Tolerance and Resilience: Meanings, Measures and Assessment*, pages 3–24. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[49] Third Party Libraries. http://metrics.dropwizard.io/3.2.3/manual/third-party.html. [Dostopano: 20. 8. 2017].

[50] User Manual. http://metrics.dropwizard.io/3.2.3/manual/index.html. [Dostopano: 17. 6. 2017].

[51] Using kubeadm to Create a Cluster. http://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/. [Dostopano: 30. 6. 2017].

[52] Matt Watson. 8 Key Application Performance Metrics and How to Measure Them. http://stackify.com/application-performance-metrics/. [Dostopano: 1. 7. 2017].

[53] Matt Watson. Comparison of 18 Application Performance Management Tools. http://stackify.com/application-performance-management-tools/. [Dostopano: 20. 8. 2017].