Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Gregor Poročnik

Analiza in primerjava spletnih storitev SOAP in protokola gRPC v okolju mikrostoritev

MAGISTRSKO DELO

MAGISTRSKI PROGRAM DRUGE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Mentor: prof. dr. Matjaž Branko Jurič

Ljubljana, 2018

Avtorske pravice. Rezultati magistrskega dela so intelektualna lastnina avtorja in Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavljanje ali izkoriščanje rezultatov magistrskega dela je potrebno pisno soglasje avtorja, Fakultete za računalništvo in informatiko ter mentorja.

Pripadajoča izvorna koda je na voljo na https://github.com/kumuluz/kumuluzee in https://github.com/gpor89/masters-thesis, pod pogoji MIT licence.

©2018 Gregor Poročnik

ZAHVALA

Zahvaljujem se mentorju prof. dr. Matjažu Branku Juriču in laboratoriju za integracijo informacijskih sistemov za vodenje in usmerjanje pri izdelavi magistrskega dela. Hvala tudi izr. prof. dr. Poloni Oblak, družini, sodelavcem, kolegom in prijateljem, ki so prispevali k uspešnemu zaključku študija.

Gregor Poročnik, 2018

»The scientists of today think deeply instead of clearly. One must be sane to think clearly, but one can think deeply and be quite insane.«

— Nikola Tesla

Kazalo

Povzetek

Abstract

1 Uvod 1

1.1 Motivacija . . . 1

1.2 Povezane raziskave . . . 3

1.3 Hipoteza . . . 4

1.4 Cilji . . . 4

1.5 Struktura dela . . . 5

2 Pregled področja protokolov in tehnologij 7

2.1 RPC . . . 7

2.2 SOAP . . . 8

2.3 REST . . . 9

2.4 gRPC . . . 9

2.5 Oblak, mikrostoritve in arhitektura aplikacij . . . 10

2.5.1 Naravne oblačne aplikacije . . . 10

2.5.2 Koncept komunikacije . . . 11

2.5.3 Zgradba in lastnosti . . . 13

3 Postopki podpore spletnih storitev v programskih okoljih 17

3.1 Raziskava . . . 17

3.1.1 Spring Boot . . . 18

3.1.2 Thorntail 18
3.1.3 KumuluzEE 18
3.1.4 Primerjava ogrodij Spring Boot, Thorntail in KumuluzEE 19
3.2 Pristop 20
3.3 Načrtovanje rešitve 21
3.4 Rešitev 21
3.5 Sklep 23

4 Analiza hitrosti in učinkovitosti protokolov 25

4.1 Določitev okolja 26
4.1.1 Strojna oprema 26
4.1.2 Ogrodja 26
4.2 Določitev postopka 27
4.3 Primerjava protokolov glede na hitrostne metrike 28
4.3.1 Zagon storitev 28
4.3.2 Delovanje storitev 29
4.4 Sklep 33

5 Raziskava primernosti protokolov v okolju mikrostoritev 35

5.1 Opredelitev raziskave 35
5.2 Določitev evalvacijskih metrik 36
5.2.1 Interoperabilnost 36
5.2.2 Hitrost izvajanja 36
5.2.3 Kakovost 37
5.2.4 Zanesljivost 38
5.2.5 Enostavnost razvoja 38
5.2.6 Hitrost razvoja in strošek 39
5.2.7 Količina porabe sistemskih virov 39
5.2.8 Odpornost na napake v razvoju 39
5.2.9 Odpornost na napake v delovanju in samozdravljenje . 40
5.2.10 Število vrstic programske kode 41

5.2.11 Primernost za vzdrževanje 41
5.2.12 Varnost 41
5.2.13 Transakcijska podpora 42
5.2.14 Neodvisnost od transportnega protokola 42
5.2.15 Dostopnost storitev, odkrivanje in objavljanje 42
5.2.16 Spremljanje obremenjenosti in učinkovitosti sistema . . 43
5.2.17 Spremljanje vitalnosti sistema 43
5.2.18 Imunost protokola na nepravilnosti v komunikaciji . . . 44
5.2.19 Podpora orodij za razvoj, testiranje in razhroščevanje storitev 44
5.3 Metodologija 45
5.4 Evalvacija metrik in rezultati 46
5.4.1 Interoperabilnost 46
5.4.2 Hitrost izvajanja 47
5.4.3 Kakovost 47
5.4.4 Zanesljivost 49
5.4.5 Enostavnost razvoja 50
5.4.6 Hitrost razvoja in strošek 50
5.4.7 Količina porabe sistemskih virov 51
5.4.8 Odpornost na napake v razvoju 53
5.4.9 Odpornost na napake v delovanju in samozdravljenje . 54
5.4.10 Število vrstic programske kode 54
5.4.11 Primernost za vzdrževanje 55
5.4.12 Varnost 56
5.4.13 Transakcijska podpora 57
5.4.14 Neodvisnost od transportnega protokola 57
5.4.15 Dostopnost storitev, odkrivanje in objavljanje 58
5.4.16 Spremljanje obremenjenosti in učinkovitosti sistema . . 59
5.4.17 Spremljanje vitalnosti sistema 60
5.4.18 Imunost protokola na nepravilnosti v komunikaciji . . . 60

5.4.19 Podpora orodij za razvoj, testiranje in razhroščevanje storitev . . . 61
5.5 Ugotovitve . . . 63

6 Izboljšave protokolov in mehanizmov za delovanje v okolju mikrostoritev 69
6.1 Pomanjkljivosti . . . 69
6.2 Zahteve . . . 70
6.3 Predlog rešitve pomanjkljivosti . . . 71
6.3.1 Zagotavljanje informacij o vitalnosti storitve . . . 73
6.3.2 Diskusija . . . 73

7 Zaključek in sklepne ugotovitve 75

Seznam uporabljenih kratic

Kratica	Angleško	Slovensko
CDI	Context Dependency Injection	Mehanizem za upravljanje z objektnimi odvisnostmi
ESB	Enterprise Service Bus	Vodilo za komunikacijo poslovnih storitev
GPB	Google Protocol Buffers	Protokolni tokovi, Googlova oblika
gRPC	google Remote Procedure Call	Googlov protokol za oddaljeno klicanje metod
HTTP	Hyper Text Transfer Protocol	Protokol za prenos obogatenih besedil
HTTP/2	Hyper Text Transfer Protocol 2	Izboljšana različica protokola za prenos obogatenih besedil
IDL	Interface Description Language	Opisni jezik za vmesnike
JSON	JavaScript Object Notation	Oblika sporočila v jeziku JavaScript
MIME	Multipurpose Internet Mail Extensions	Večnamenski dodatek elektronskih sporočil
MIT	Massachusetts Institute of Technology	Organizacija, po kateri je imenovan licenčni model
MVC	Model View Controller	Arhitekturni vzorec za gradnjo aplikacij

PB	Protocol Buffers	Protokolni tokovi, oblika sporočil
PHP	PHP: Hypertext Preprocessor	Programski jezik
RAML	RESTful API Modeling Language	Modelirni jezik za grajenje REST vmesnikov
REST	REpresentational State Transfer	Predstavitvena arhitektura za prenos podatkov
RFC	Request For Comments	Zahtevek, na podlagi katerega skupnost poda mnenje
RMI	Remote Method Invocation	Oblika klica oddaljenih storitev (Java)
RPC	Remote Procedure Call	Oblika klica oddaljenih storitev
SOA	Service Oriented Architecture	Storitveno usmerjena arhitektura
SOAP	Simple Object Access Protocol	Enostaven protokol za izmenjavo sporočil
SSL	Secure Sockets Layer	Način komunikacije na podlagi certifikatov
TLS	Transport Layer Security	Način kriptiranja podatkov za prenos
UDDI	Universal Description, Discovery and Integration	Mehanizem za objavljanje spletnih storitev
UML	Unified Modeling Language	Jezik za modeliranje programskih in arhitekturnih rešitev
URI	Unique Resource Identifier	Enolični identifikator vira
VB	Visual Basic	Programski jezik

WSDL	Web Service Description Language	Opisni jezik sporočil spletnih storitev
XML	Extensible Markup Language	Razširljiva oblika sporočila

Povzetek

Naslov: Analiza in primerjava spletnih storitev SOAP in protokola gRPC v okolju mikrostoritev

Razvoj storitev v oblaku nam je omogočil hitrejši razvoj programske opreme in olajšal nameščanje programske opreme na platforme, vendar pa nam je zaradi lastnosti distribuiranih sistemov pripeljal nove izzive na področje storitvenih protokolov, predvsem v medsebojni komunikaciji, spremljanju in upravljanju delovanja mikrostoritev v distribuiranih sistemih. V tem magistrskem delu smo se osredotočili na področje mikrostoritev, lastnosti distribuiranih sistemov in storitvenih protokolov v povezavi z okoljem mikrostoritev, pri čemer smo raziskali povezavo med hitrostjo, učinkovitostjo in primernostjo za delovanje v distribuiranih sistemih najpomembnejših protokolov mikrostoritev. Ker nekateri izmed protokolov nimajo popolne podpore za delovanje v ogrodjih za izdelavo mikrostoritev, smo definirali postopek vpeljave novih funkcionalnosti in podpore protokolom skozi integracijo programskih knjižnic. Tekom nastajanja magistrskega dela smo ugotovili, da so smernice razvoja protokolov zelo podobne, ko govorimo o zapisu pravil, v nasprotnem primeru pa se zelo razlikujejo na področju učinkovitosti in funkcionalne podpore. Prav tako smo ugotovili mnoge pomanjkljivosti protokola SOAP, zato smo za učinkovito delovanje v mikrostoritvah predlagali tudi izboljšave.

Ključne besede

mikrostoritve, računalniški oblak, protokoli

Abstract

Title: Analysis and comparison of SOAP web services and gRPC protocol in microservice environment

Development of cloud services allowed us to speed up software development and make installing of software on platforms easier, but due to the characteristics of the distributed systems, this approach brought new challenges in the field of web service protocols, mostly intercommunication, monitoring and microservice operation management. In this master thesis we researched the field of microservices, the properties of distributed systems and web service protocols in relation to cloud-native applications. We studied the connection between speed, efficiency and suitability for the main protocols used by microservices. Since some of the protocols do not have complete support in frameworks that allow production of microservices, we defined the procedure to integrate new functionality and protocol support through the integration of program libraries. At the time of writing this thesis, we find that there are no major differences in principles of protocol description languages, despite the fact there are major differences in terms of effectiveness and functionalities. We assess that there are many downsides in the SOAP protocol, so we propose improvements by way of utilizing effective operations.

Keywords

microservice, protocols, cloud-native

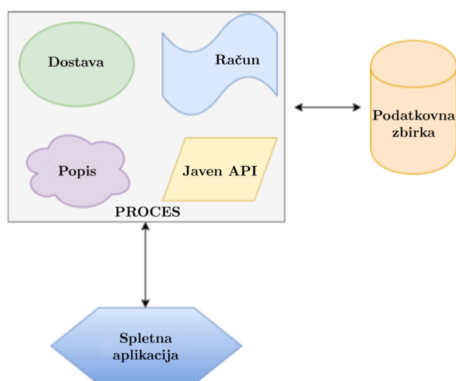
Poglavje 1

Uvod

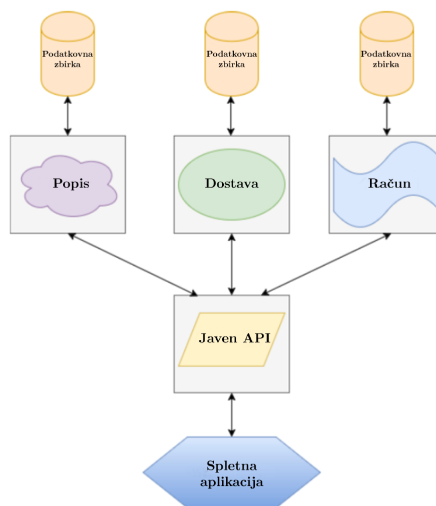
1.1 Motivacija

Zaradi zahtev po napredni programski opremi in zaradi potreb poslovnih procesov se čez čas pojavljajo novi arhitekturni slogi, ki zagotavljajo rešitve v poslovnih okoljih [1]. Pred pojavom mikrostoritev so bili sistemi sestavljeni iz tako imenovanih monolitnih aplikacij, kot je prikazano na sliki 1.1. V takšni arhitekturi se celotna poslovna logika izvaja znotraj enega izvajalnega okolja, ki teče znotraj enega strežnika. Takšna arhitektura med drugim zmanjšuje obvladljivost razvoja, povzroča kaotičen namestitveni proces, sistem je manj odporen na napake, zaradi omejitev strojne opreme pa je skalabilnost omejena. Za namestitev storitve v takšnem okolju potrebujemo več znanja, več časa, je dražje in neučinkovito.

Z arhitekturo mikrostoritev odpravljamo mnoge pomanjkljivosti arhitekture monolitnih aplikacij, kot so fleksibilnost, modularnost in odpornost na evolucijo [2], vendar z novo arhitekturo prinašamo nove izzive kot so porazdeljenost, zakasnjena doslednost in zapleteno vzdrževanje okolja [3]. Okolje mikrostoritev je distribuiran sistem, ki je zgrajen iz manjših aplikacij oziroma storitev, ki opravljajo točno in samo določeno opravilo, vsaka od teh pa je pripravljena za samostojno namestitev v izvajalno okolje, kot kaže slika 1.2. Podobno kot v starejši, storitveno usmerjeni arhitekturi (SOA), kjer kom-



Slika 1.1: Struktura monolitne aplikacije



Slika 1.2: Struktura mikrostoritev

ponenta ne predstavlja več tako imenovanega »middleware« in komunikacijskega vodila (tj. ESB), mikrostoritve ne vsebujejo več poslovne logike na vodilu, ampak je logika domena vsake od komponent [4]. Mikrostoritve lahko in je želeno graditi v različnih programskih jezikih in okoljih glede na primernost, zato je pomembno, da so protokoli, ki povezujejo storitve, interoperabilni.

Mikrostoritve gradijo šibko sklopljen sistem (angl. loosely coupled), saj arhitektura temelji na zasnovi, kjer sprememba delovanja ene storitve ne zahteva spremembe drugih storitev. Šibko sklopljena komponenta se v čim manjši meri zaveda ostalih storitev, s katerimi komunicira [5]. Komunikacija med storitvami poteka prek mrežnih komunikacijskih protokolov. Zaradi takega načina komunikacije od računalniškega omrežja in protokolov pričakujemo lastnosti, kot so zanesljivost, stabilnost, hitrost, odzivnost, odpornost na spremembe, interoperabilnost in nizko računsko zahtevnost. Slednje izzive rešujemo s pomočjo mnogih tehnik, vzorcev in mehanizmov.

1.2 Povezane raziskave

V tem poglavju opisujemo že raziskano področje, ki se navezuje na to delo in je pogoj za razumevanje tega dela.

Komunikacija med mikrostoritvami poteka po vnaprej znanih pravilih, imenovanih protokoli. Ko določimo protokol za prenos sporočil, s tem določimo tudi potrebno strukturo sporočila za prenos. V delu bomo natančneje uporabljali podatkovne strukture JSON, XML in nekoliko novejše podatkovne tokove (angl. protocol buffers) podjetja Google (v nadaljevanju GPB).

JSON je podatkovna struktura, ki je sestavljena iz ključev, ki predstavljajo niz, in vrednosti, ki predstavljajo množico vnaprej znanih tipov: primarni tip, objekt ali seznam. Primarne tipe delimo na niz (angl. string), število (angl. number), binarno vrednost (angl. boolean) in prazno (angl. null). Več podrobnosti tipov opišejo avtorji v viru [6]. XML je podatkovna struktura, ki predstavlja nekoliko natančnejši opis podatkov v spletnih storitvah. Jezik opisuje obliko in odvisnosti skozi drevesno strukturo, podatki so strukturirani skozi množico tipov in vrednosti elementov ter pripadajočih atributov, kot je natančneje opisano v viru [6]. Podatkovni tokovi so novejša, binarna podatkovna struktura, ki za razliko od JSON in XML niso v obliki, iz katere lahko človek razpoznava vsebino sporočila. V zakup prinaša manjšo velikost podatkov, potrebnih za prenos, in s tem hitrejšo hitrost prenosa. Podatkovna struktura zahteva posebna ogrodja in orodja, ki sporočilo pretvorijo v človeku prijazno obliko. Primer protokolnih tokov predstavljata rešitvi Apache Thrift in GPB [6]. Protokoli za komunikacijo SOAP, REST in gRPC za prenos sporočil med mikrostoritvami uporabljajo različne oziroma poljubne strukture podatkov. Tihomirovs idr. [7] izvajajo evalvacijo arhitekturnih slogov med protokoli REST-GPB, REST-XML in REST-JSON ter protokolom SOAP. Avtorja v članku opisujeta lastnosti, prednosti in slabosti načinov komunikacije skozi različne vidike in metrike. Prikazani rezultati v članku imajo pomanjkljivosti, kot so neznane podrobnosti analize in mnoge metrike, ki vplivajo na primernost protokola za delovanje v okolju mikrostoritev, ki jih s tem delom v sklopu analize protokolov odpravljamo, poleg tega

pa dodamo primerjavo protokola gRPC.

Skozi analizo raziskav oziroma literature opazimo, da mnogo razvijalcev piše o prednostih in slabostih razvoja storitev v različnih protokolih. Z vidika mikrostoritve je izbira protokola za prenos podatkov enostavna, z vidika okolja mikrostoritev pa lahko izbira protokola predstavlja resen izziv [8].

1.3 Hipoteza

Protokol SOAP je zaradi hitrosti manj primeren za okolje mikrostoritev, saj Tihomirovs idr. [7] opisujejo strukturo podatkov XML kot počasnejšo za obdelavo in prenos. Za razvoj in vzdrževanje odjemalcev, ki uporabljajo protokol REST, je treba več vloženega dela, saj je identifikacija sprememb in napak otežena zaradi nedefiniranega oziroma neobstoječega standarda za opis spletnih storitev. Standardi predstavljajo prvi pogoj za kakovost in široko uporabnost storitev. Protokol gRPC je primernejši za uporabo v okolju mikrostoritev zaradi uporabe učinkovitega binarnega protokola GPB. Definicija podatkovne strukture protokolnih tokov omogoča učinkovito integracijo, poleg tega pa vsebuje pomembne mehanizme, ki botrujejo k učinkovitemu delovanju mikrostoritev.

1.4 Cilji

Prvi cilj dela je analiza komunikacijskih protokolov s poudarkom na protokolih SOAP in gRPC. Ker se načini komunikacije razlikujejo po mnogih lastnostih, sta analiza protokolov in definicija metrik ključna cilja naloge, ki zagotavljata temelj za ocenitev primernosti za delovanje v okolju mikrostoritev. Drugi cilj dela je implementacija in integracija knjižnice CXF, ki omogoča komunikacijo med sistemi na podlagi komunikacijskega protokola SOAP. Ključen namen integracije je podpreti oziroma izboljšati mehanizem v ogrodju za gradnjo mikrostoritev KumuluzEE tako, da na enostaven način omogoča razvijalcu mikrostoritev izvedbo integracije med sistemi, ki teme-

ljijo na protokolu SOAP. Ogrodje KumuluzEE je načrtovano za grajenje mikrostoritev, kar pomeni, da integracija doprinese ključno prednost protokolu SOAP v primeru, da ga želimo uporabiti v okolju mikrostoritev.

Tretji cilj dela je ovrednotenje protokolov s poudarkom na SOAP in gRPC ter pokazati in dokazati primernost protokolov za delovanje v okolju mikrostoritev. Pogoj za potrditev domneve je analiza mikrostoritev in okolja, v katerem delujejo. Ker mikrostoritve prinašajo določene zahteve, prednosti in slabosti, to vpliva na zmožnost in primernost delovanja protokolov v takšnem okolju. S pomočjo integracije knjižnice CXF v ogrodje KumuluzEE bo opravljena funkcionalna in nefunkcionalna analiza protokola SOAP in drugih. Cilj dela je definirati metrike, določiti okolje, postopek in tako objektivno evalvirati vsakega izmed kandidatov v okoliščinah, ki so naravni del okolja, v katerem delujejo mikrostoritve.

Zadnji cilj raziskave predstavlja predloge, podaja smernice in konkretne rešitve za uporabo komunikacijskih protokolov v okolju mikrostoritev.

1.5 Struktura dela

V drugem poglavju obravnavamo področje komunikacijskih protokolov, različne oblike sporočil in zasnove komunikacij. Poleg pregleda področij protokolov za prenos podatkov pod drobnogled jemljemo tudi zahteve distribuiranega okolja oziroma lastnosti, ki so potrebne za uspešno in učinkovito delovanje storitev v okolju mikrostoritev. Tretje poglavje vsebuje rešitev pomanjkljive podpore za storitve SOAP v ogrodju za izdelavo mikrostoritev. Poglavje opisuje praktični primer integracije knjižnice, ki nudi podporo protokola SOAP v ogrodju KumuluzEE. Integracija predstavlja pogoj za naslednje poglavje, kjer analiziramo učinkovitost podane rešitve. Četrto poglavje vsebuje analizo hitrosti in učinkovitosti ter primerjavo komunikacijskih protokolov SOAP in gRPC. Zaradi razširjenosti in pomembnosti bomo vključili še arhitekturni slog REST. Skozi določitev okolja in metrik ter postopka izvedemo primerjavo glede na izbrane metrike. V petem poglavju izvedemo raziskavo primernosti

s pomočjo rezultatov analize iz četrtega poglavja in opredelimo posamezne protokole v prostor primernosti za delovanje v distribuiranih okoljih. V šestem poglavju povzamemo pomanjkljivosti in slabosti protokolov iz petega poglavja in predlagamo način izboljšave protokolov v smeri primernosti za delovanje v mikrostoritvah. Zadnje poglavje zajema zaključek in sklepne ugotovitve.

Poglavje 2

Pregled področja protokolov in tehnologij

V tem poglavju bomo predstavili področje protokolov, distribuiranih sistemov in mikrostoritev. Protokoli potrebujejo pravila za komunikacijo med storitvami, saj so storitve lahko zgrajene ob različnem času in z različno tehnologijo, zato je protokol edino pravilo, ki omogoča, da se sporazumevajo z enakim jezikom. V področju protokolov bomo omenili protokol RPC, ki predstavlja zgodovinski pogled na protokole. Opisali bomo tudi protokole SOAP, REST in gRPC ter značilnosti, na katere se bomo v delu natančneje osredotočili.

2.1 RPC

Klic za oddaljeni postopek (RPC) je način komunikacije v distribuiranem okolju. V programskem jeziku Java poznamo način izvajanja oddaljenih metod (RMI), ki izkorišča primaren način pretvorbe iz objektne v tekstovno obliko. Rešitev je ena izmed prvotnih načinov reševanja komunikacij med sistemi, pri katerih sistem izvede oziroma pokliče metodo na oddaljenem računalniku prek skupnega računalniškega omrežja. Takšen način omogoča programerju, da izvede metodo na oddaljenem sistemu na enak način kot

metodo, ki se kliče lokalno. Način komunikacije deluje v okviru enakih izvajalnih okolij, to je znotraj implementacij enega programskega jezika, in ne omogoča interoperabilnosti. Pri uporabi takšnega protokola se je pomembno držati pravil in struktur [5].

2.2 SOAP

SOAP je protokol, ki je med prvimi reševal zahteve s področja interoperabilnosti med sistemi. Protokol SOAP zahteva pogodbo v obliki opisnega jezika za spletne storitve WSDL. Datoteka WSDL s pomočjo oblike sporočila XML omogoča natančen opis storitev oziroma komunikacije med sistemi in tako ne določa specifičnih lastnosti nekega izvajalnega okolja. Podjetje Microsoft je razvijalo protokol z namenom, da bi bil popolnoma fleksibilen, tako zmožen komunikacije prek zasebnih omrežij, prek spleta ali celo elektronskega sporočila kot nosilca sporočil. Protokol je bil prvotno del specifikacije, ki je vsebovala tudi UDDI, ki omogoča repozitorij storitev. Tipično je za komunikacijski kanal uporabljan protokol HTTP, odprta pa je možnost uporabe tudi mnogih drugih protokolov. SOAP deluje kot vsebnik ali ovoj za prenos sporočil med sistemi. Sporočilo SOAP vsebuje obvezne elemente, kot so envelope, header, body in fault. Element envelope definira začetek in konec sporočila. Element header nosi dodane informacije strežniku za obdelavo, element body pa nosi potrebne podatke. V kolikor strežnik zavrne zahtevek, se odjemalcu vrne sporočilo z napako znotraj elementa fault. Varnost v protokolu zagotavlja standard WS-Security, ki realizira varnost s pomočjo elementa header v sporočilu XML. Zaradi načina izmenjevanja podatkov v obliki XML je količina potrebnih podatkov za prenos večja od primerljivih oblik, kar prinaša daljši odzivni čas in večjo procesorsko moč za obdelavo sporočil.

2.3 REST

REST je arhitekturni slog za gradnjo večjih in obširnejših distribuiranih sistemov. Ker temelj arhitekturnega sloga predstavlja protokol HTTP, je zelo primeren za skalabilne sisteme. Podatki so organizirani v obliki virov oziroma unikatnih identifikatorjev virov (angl. URI), na katere se z uporabo metod HTTP (POST, PUT, GET, DELETE...) ustvarja, posodablja, bere ali briše podatke. Za izmenjavo podatkov med sistemi se najpogosteje uporablja oblika sporočil JSON ali XML, redkeje pa tudi GPB. Izvedba storitve, ki temelji na protokolu REST, je praviloma preprosta in s tem učinkovita ter hitra, saj večina sistemov že izvorno zagotavlja in podpira aplikacijski protokol HTTP. Arhitekturni slog je eden izmed najbolj razširjenih zaradi enostavnosti sklada protokolov (HTTP, XML, URI, MIME), lastnosti požarnih zidov (vrata 80) in orodij, ki so izpopolnjena za delo s storitvami REST [9].

2.4 gRPC

gRPC (angl. Google Remote Procedure Call) [10] je izboljšana različica načina RPC in odpravlja mnoge pomanjkljivosti komunikacije, ki ima korenine še v prejšnjem stoletju. Protokol zagotavlja prednosti s področja interoperabilnosti, hitrosti pretvorbe sporočil, manjše količine prenesenih podatkov, sposobnosti upravljanja z verzijami sporočil in izboljšavami na področju nizko nivojske komunikacije. Protokol vsebuje podporo podatkovnih tokov, ki so manj procesorsko zahtevni in učinkovitejše upravljajo sistemske vire. Protokol vsebuje mnoge lastnosti, ki so pogoj za delovanje in migracijo zunaj okvirov izvajalnega okolja brez žrtvovanja stranskih učinkov [11] in tveganj.

2.5 Oblak, mikrostoritve in arhitektura aplikacij

Kot smo že pokazali v uvodu, so mikrostoritve posebna oblika aplikacij, ki delujejo v izvajalnem okolju, ki je fizično ali logično ločeno od ostalih. Te storitve med sabo komunicirajo s pomočjo komunikacijskih protokolov. S tem, ko vpeljemo protokole v takšen sistem, zapustimo okolje enega primerka storitve, vstopimo v divji svet nedeterminizma, v okolje distribuiranih sistemov, kjer se sistemi kvarijo na najrazličnejše in vedno bolj inovativne načine. V takšnih sistemih prihaja do situacij, kjer se informacije lahko izgubijo, pokvarijo ali pridejo na cilj v naključnem vrstnem redu. Dejstvo je, da je delovanje distribuiranih sistemov zahtevno. Komunikacijski protokoli igrajo pomembno vlogo v distribuiranih okoljih zaradi zahtev po hitrosti, odpornosti na napake, zmožnosti nadgradenj in sprememb sistemov, zmožnosti odkrivanja storitev in interoperabilnosti [12, 13].

2.5.1 Naravne oblačne aplikacije

Naravne oblačne aplikacije (angl. cloud-native applications) so aplikacije, ki so posebej prirejene za delovanje v oblaku. Od takšnih aplikacij se zahtevajo mnoge odlike, ki omogočajo lažji razvoj, lažjo namestitev, lažji nadzor, večjo neodvisnost in lažje vzdrževanje oziroma nadgradnjo storitev. Primarno je glavni namen takšnih storitev učinkovitost delovanja in hitrost v smislu razvoja [14].

Naravne oblačne aplikacije so zgrajene iz manjših delov okrog natančno določenih funkcionalnosti in imajo izolirano stanje. Velikost takšnih storitev opisujemo tudi kot granularnost sistema [15]. Zaradi homogenosti takšne storitve med seboj povezujemo z orkestracijskimi in skalirnimi sistemi, heterogenost storitev pa prinaša zahteve po odkrivanju storitev.

2.5.2 Koncept komunikacije

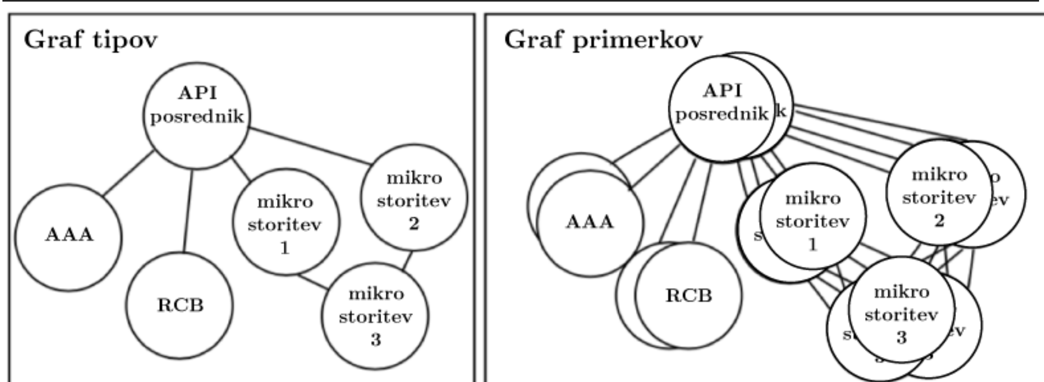
Načini komunikacije se s časom razvijajo zaradi različnih potreb s pomočjo razvoja infrastruktur, arhitektur in algoritmov. Začetki komunikacij so temeljili na trivialnih rešitvah, kot je prenos surovih podatkov po mediju, nato pa so se razvili v naprednejše rešitve, ki izkoriščajo prednosti od fizične ravni do zelo dovršene aplikacijske stopnje. Prihodnost vsekakor prinaša nove izzive na področju komunikacij. Te se bodo razvijale na podlagi odkrivanja novih in optimizaciji že znanih protokolov ter odkrivanja novih arhitekturnih slogov, ki obljubljajo napredne integracijske vzorce. Primera znanih vzorcev, ki preizkušata kakovost storitev, sta agregacija in kompozicija.

Agregacija vzpostavlja odnos, kjer lahko komponenta otrok obstaja brez starša. Lahko rečemo tudi, da storitev A uporablja storitev B. Kompozicija pa predstavlja odnos, kjer otrok ne obstaja brez starša. V tem primeru storitvi naročila in kupci predstavljata agregacijo, storitev vozlišče vmesnikov (angl. API gateway) pa kompozicijo. Lahko rečemo tudi, da si komponenta A lasti komponento B.

Okolje mikrostoritev lahko prikažemo z grafom, kjer vozlišča predstavljajo tip storitve. Slika 2.1 predstavlja primer takšnega heterogenega sistema. Graf, kjer vozlišče predstavlja primerek storitve, pa predstavlja homogen vidik postavitve storitev [15]. Glavni zahtevi v takšnem sistemu sta skalabilnost in imunost sistema na težave, ki jih s seboj prinaša distribuirano okolje.

Distribuirano računalništvo je zasnova, ki ima korenine že daleč v preteklosti. Osem glavnih zmot v distribuiranih okoljih je bilo opisanih že leta 1994 [16]. Te so:

1. omrežje je zanesljivo;
2. odzivnost je ničelna;
3. pasovna širina je neomejena;
4. omrežje je varno;



Slika 2.1: Graf tipov in primerkov mikrostoritev

5. topologija omrežja se ne spreminja;

6. obstaja le en skrbnik sistema;

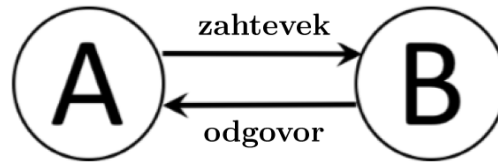
7. stroški prenosa so ničelni;

8. omrežje je homogeno.

Naštete izzive v svetu mikrostoritev rešujejo protokoli za komunikacijo in mehanizmi, ki izničijo ali minimalizirajo vpliv dejavnikov.

Skalabilnost je lastnost storitve, kjer je storitev zmožna in primerna za delovanje v mnogih primerkih. Takšne aplikacije popolnoma izkoriščajo lastnosti platforme ponudnikov storitev v oblaku, kjer se je zmogljivost strojne opreme zmožna prilagajati zahtevam v izvajalnem programskem okolju [17]. Drugi pomemben element sistema je neodvisnost izvajalnega okolja glede na strojno opremo in platformo. Aplikacije se v namestitvenem koraku zapakirajo v t. i. vsebnike (angl. containers), ki vsebujejo dodatne parametre in podatke ter služijo kot končna namestitvena enota v izvajalnem okolju.

Ena izmed lastnosti mikrostoritev je, da prek komunikacijskih vmesnikov komunicirajo z ostalimi storitvami, kot je prikazano na sliki 2.2. Zaradi tega je pomembno, da storitve vsebujejo detekcijski mehanizem, ki napake v komunikaciji zazna in pokvarljive enote samodejno izloči iz sistema. V mikrostoritvah vmesniki najpogosteje temeljijo na zasnovi protokola HTTP ali



Slika 2.2: Klic metod med mikrostoritvami

asinhronih sporočilnih mehanizmov. Upravljanje verzij protokoli rešujejo na različne načine, implicitno s pomočjo imenskih prostorov ali glav v sporočilih, najdemo pa tudi primere z eksplicitnim upravljanjem verzij znotraj imen metod [18].

2.5.3 Zgradba in lastnosti

V prejšnjem podpoglavju smo opisali mikrostoritve, kot so vidne s perspektive okolja in ostalih storitev. Poleg teh zahtev so pomembne tudi interne lastnosti posamezne storitve. Opisali smo koncept granularnosti kot pomemben vidik mikrostoritev [19]. Zasnova mikrostoritev za delovanje v večini primerov žrtvuje podatkovno doslednost (angl. consistency) na račun particijske tolerance (angl. partition tolerance) in razpoložljivosti (angl. availability) podatkov [20]. Particijska toleranca je pomembna zaradi nezanesljivih omrežij, razpoložljivost pa nudi hitrostni element sistema, ki je zelo pomemben v mikrostoritvah. Zaradi takšne zasnove mikrostoritve temeljijo na principu zakasnjene doslednosti (angl. eventual consistency), v kolikor poslovni model to omogoča. Temu pravimo tudi teorem CAP (angl. CAP theorem) [20]. Za homogene storitve, namenjene delovanju v mnogih primerkih, je praktično, da ne vsebujejo stanj, ampak delujejo neodvisno od ostalih primerkov storitev. Temu pravimo, da so storitve brez stanj (angl. stateless) [21]. Fehling idr. [22] predlagajo, da bi aplikacije v okolju mikrostoritev morale zadostiti naslednjim pogojem:

- izolacija (angl. isolated state):

storitve imajo ločeno stanje do katerega ima dostop, nadzor in odgovornost, le točno določena komponenta;

- porazdeljenost (angl. distributed in its nature):
  storitve so porazdeljene na različna izvajalna okolja, ki imajo lahko različne, prilagojene lastnosti za učinkovito delovanje storitev;

- elastičnost (angl. elastic in a horizontal scaling way):
  storitve so zasnovane na način, da omogočajo hkratno delovanje mnogih primerkov iste storitve in dinamično horizontalno skaliranje;

- avtomatsko upravljanje (angl. automated management system):
  storitve uporabljajo mehanizme, ki zagotavljajo avtomatsko upravljanje sistema, konfiguracijo, namestitev in nadzor;

- šibka sklopljenost:
  storitve so načrtovane tako, da delujejo neodvisno od ostalih komponent sistema.

Stine [14] opisuje glavne motivacije okolja mikrostoritev skozi naslednje točke:

- hitrejša dostava programske opreme:
  zaradi omejene velikosti storitve sta razvoj in vzdrževanje hitrejša;

- večja izolacija napak:
  napako znotraj poslovne logike sistema je lažje odkriti zaradi majhnosti storitve;

- zmožnost odkrivanja in samodejne odprave napak:
  storitev je zmožna odreagirati na nepredvidljive napake, ki se dogajajo znotraj distribuiranega sistema;

- zmožnost skalabilnosti (horizontalne):
  storitev je načrtovana tako, da se s povečevanjem števila primerkov storitve v distribuiranem okolju povečuje tudi zmogljivost okolja;

- zmožnost uporabe široke palete platform in uporabe obstoječih sistemov:
  storitve so zgrajene na platformi oziroma programskem jeziku, ki v določenem segmentu ponuja najprimernejšo in učinkovito realizacijo ciljev.

Mnogo literature navaja mikrostoritve kot evolucijo storitveno usmerjene arhitekture, saj sta si arhitekturi v določenih specifikah podobni [23]. Ena izmed razlik med arhitekturami je vidik varnosti, saj zaradi arhitekture storitve tečejo drugače. Varnost je eden izmed mnogih vidikov, ki so bili zastavljeni v arhitekturi SOA, ključnega pomena pa je funkcionalnost tudi v okolju mikrostoritev [24].

Zaradi pomembnosti drugih vidikov, kot so spremljanje delovanja, spremljanje vitalnosti sistema in odpornost na napake, predstavljajo prednost tisti komunikacijski protokoli, ki že vsebujejo potrebne zahteve brez implementacij dodatne programske kode. Zelo pomembno je, da protokoli temeljijo na standardih, zaradi katerih storitve vsebujejo poenoten način delovanja, le-ta predstavlja manj zahteven in nedvoumen razvoj storitev.

Poglavje 3

Postopki podpore spletnih storitev v programskih okoljih

Ogrodja, ki jih uporabljamo za razvoj mikrostoritev, omogočajo hitro gradnjo komponent, ki med sabo komunicirajo s pomočjo spletnih protokolov. Spletne storitve načeloma podpremo v ogrodjih s pomočjo knjižnic in programskih struktur, ki omogočajo zahtevane funkcionalnosti. Da se lažje osredotočimo na težavo, bomo raziskavo omejili na programsko okolje Java, v katerem je na voljo ducat ogrodij za gradnjo mikrostoritev. Med njimi so poleg ostalih najbolj znana Spring Boot, Thorntail in KumuluzEE. V tem poglavju se bomo osredotočili na analizo obstoječih in raziskavo pomanjkljivih funkcionalnosti v omenjenih ogrodjih. Med razvojem spletnih storitev lahko predpostavljamo, da ogrodja, ki omogočajo izgradnjo storitev, ne nudijo potrebne podpore protokolom spletnih storitev.

3.1 Raziskava

V tezi smo omenili znana ogrodja, v sklopu katerih se v grobem gradijo spletne storitve. V okviru ogrodij pa bomo raziskali podporo protokolov in knjižnic, ki omogočajo hitrejšo gradnjo spletnih storitev in temeljijo na protokolih, ki so temelj tega dela.

3.1.1 Spring Boot

Ogrodje Spring se nekoliko razlikuje od ostalih ogrodij, ki načeloma podpirajo specifikacijo JavaEE. Spring ima na voljo svoj sklad orodij in knjižnic, ki podpirajo želene funkcionalnosti. Ogrodje podpira spletne storitve REST s pomočjo knjižnice Spring MVC, mogoče pa je graditi tudi storitve na specifikaciji JAX-RS s pomočjo knjižnic Jersey in CXF. Spletne storitve SOAP so prav tako kot REST omogočene s pomočjo vgrajene podpore in integracije s knjižnico CXF, ta vsebuje mehanizme za delovanje v okolju Spring. Podpore spletnih storitev ogrodje gRPC še ne vsebuje, vendar se v skupnosti že pojavljajo primeri integracij.

3.1.2 Thorntail

Thorntail, znan tudi kot Wildfly Swarm, je ogrodje kot odgovor na primerno obliko aplikacijskega strežnika JBoss za gradnjo mikrostoritev. Ogrodje naravno podpira integracijo knjižnice RestEasy istoimenskega podjetja, ki omogoča gradnjo spletnih storitev REST. V odprtokodni skupnosti je mogoče najti tudi integracije s knjižnico Jersey, vendar ostale knjižnice razen RestEasy, sodeč po skupnosti Red Hat, ne bodo uradno podprte s strani podjetja. Podpora spletnih storitev SOAP prihaja s strani knjižnice CXF skozi integracijo z ogrodjem. Podpore spletnih storitev gRPC v ogrodju Thorntail, ki temelji na aplikacijskem strežniku Undertow, še ni.

3.1.3 KumuluzEE

Ogrodje KumuluzEE [25] nudi mnogo prednosti skozi gradnjo mikrostoritev, kot sta zagonski čas aplikacije in majhna poraba sistemskih virov [26]. KumuluzEE temelji na odprtokodni programski kodi, licencirani pod licenco MIT, kar pomeni, da se lahko spreminja in razvija skozi skupnost razvijalcev. Ogrodje KumuluzEE omogoča podporo arhitekturnemu slogu REST, ki temelji na protokolu HTTP, s pomočjo knjižnice Jersey. S knjižnico Jersey preprosto razvijemo spletne storitve REST, vendar moramo paziti na manjše

Standardi	Axis2	CXF	JAX-WS/Metro
WS-Addressing	X	X	X
WS-Coordination	X		X
WS-MetadataExchange			X
WS-Policy	X	X	X
WS-ReliableMessaging	X	X	X
Web Services Security	X	X	X
WS-SecureConversation	X	X	X
WS-SecurityPolicy	X	X	X
WS-Transaction	X		X
WS-Trust	X	X	X
WS-Federation			

Tabela 3.1: Podpora standardom WS v knjižnicah SOAP

pomanjkljivosti, kot je upravljanje z verzijami s pomočjo glave HTTP Accept, saj razločevanje verzij na podlagi parametrov Accept ne deluje. Ogrodje vsebuje tudi podporo za spletne storitve SOAP z integracijo knjižnice Apache Metro, ki jo lahko prosto vključimo. V integraciji je vključena podpora standardom WS, kot je prikazano v tabeli 3.1. Močna pomanjkljivost integracije knjižnice Metro je manjkajoča podpora knjižnice CDI, ki omogoča lažje upravljanje z odvisnostmi med objekti, na kateri temelji mnogo aplikacij [27]. Enako velja za knjižnico gRPC v kombinaciji z ogrodjem KumuluzEE, v katerem ni popolne podpore za funkcionalnosti CDI.

3.1.4 Primerjava ogrodij Spring Boot, Thorntail in KumuluzEE

Analizirana ogrodja temeljijo na aplikacijskih strežnikih, ki ne predstavljajo bistvenih razlik v komunikaciji med storitvami. Pomembno je, da spletni strežnik podpira prenosni protokol, ki je potreben za prenos sporočil, kar je

Storitve	Spring Boot	Thorntail	KumuluzEE
REST	Spring MVC, CXF, Jersey	RestEasy	Jersey
SOAP	CXF	CXF	Metro
gRPC			KumuluzEE-gRPC

Tabela 3.2: Podpora spletnim storitvam v analiziranih ogrodjih

v primerih REST in SOAP tipično HTTP. Protokol gRPC zahteva uporabo protokola HTTP/2, ki še ni podprt v vseh aplikacijskih strežnikih. V grobem prenosni protokol HTTP/2 podpira privzeto aplikacijski strežnik Netty, v katerem deluje protokol gRPC. Ogrodje Spring Boot za delovanje uporablja aplikacijski strežnik Tomcat, Thorntail prisega na Undertow, ogrodje KumuluzEE pa deluje na aplikacijskem strežniku Jetty.

V analizi smo ugotovili podprte protokole in pomembne pomanjkljivosti v integracijah knjižnic z ogrodji. Tabela 3.2 vsebuje povzetek analize vseh ogrodij in njihovih integracij. Kot smo omenili v poglavju 3.1.3, obstajajo bistvene prednosti ogrodja KumuluzEE, zato ta predstavlja enega izmed konkurenčnih ogrodij, primernih za izboljšavo, ki jo bomo opisali v naslednjem poglavju.

3.2 Pristop

V analizi ogrodja KumuluzEE smo ugotovili, da obstajajo pomembne pomanjkljivosti ogrodij, ko gre za podporo spletnih storitev. Opazimo, da ogrodje KumuluzEE ne omogoča ključne funkcionalnosti uporabe CDI v kombinaciji s spletnimi storitvami SOAP in knjižnico Metro. Nadaljnja analiza kaže, da ogrodje temelji na zasnovi osrednjega modula, ki vsebuje aplikacijski strežnik in zagotavlja živ sistem. Ostali moduli so praviloma izbirni in dani na izbiro razvijalcu, da jih vključi po želji, v kolikor je potreba po dodatnih funkcionalnostih. Da bi izboljšali podporo, smo ugotovili, da lahko

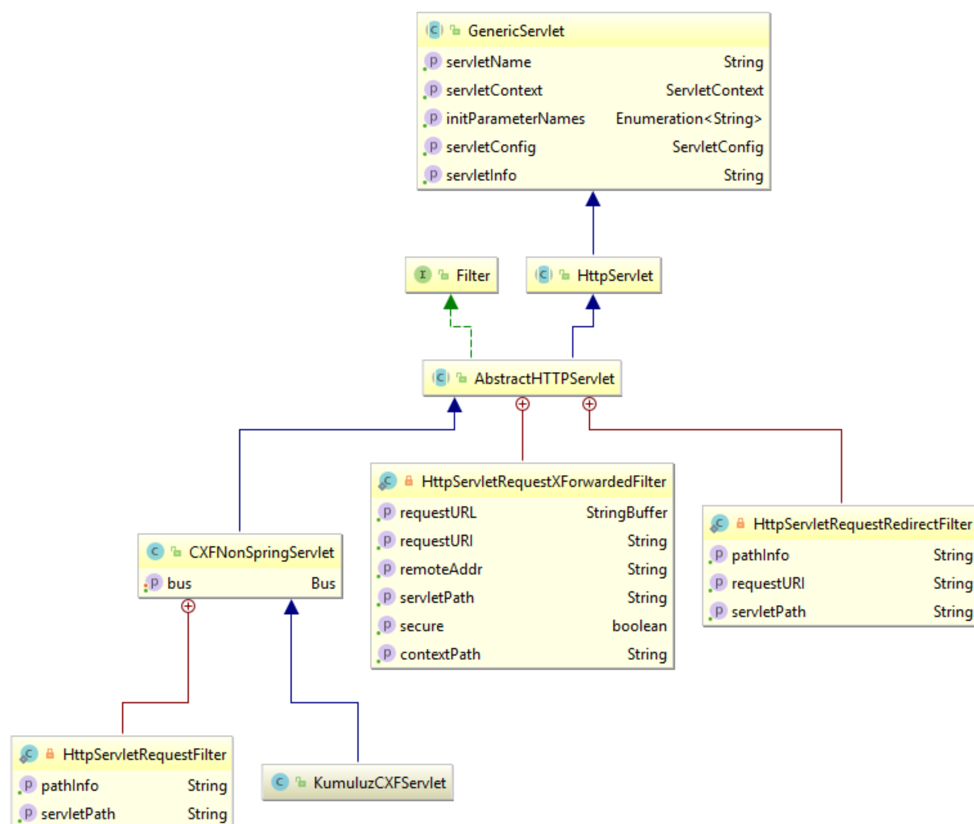
omogočimo podporo CDI tako, da uporabimo knjižnico Apache CXF, ki že vsebuje potrebno implementacijo in zagotavlja upravljanje s sporočili XML na ravni aplikacijskega protokola HTTP. Del pristopa k razvoju integracije zajema raziskavo delovanja ogrodja KumuluzEE ter raziskavo načina delovanja knjižnice Apache CXF in protokola SOAP [28]. Pristop zajema še raziskavo standardov WS (spletnih storitev) in pogostih programskih vzorcev, ki morajo biti nujno podprti, saj se od ogrodja pričakuje popolna kompatibilnost [29]. Skozi razvoj storitve v programskem okolju Java bomo upoštevali dobre prakse razvoja storitev [16].

3.3 Načrtovanje rešitve

Postopek integracij ogrodij temelji na analizi ogrodij in knjižnic, ki se uporabljajo za doseganje dodane vrednosti. V ogrodju KumuluzEE bomo zgradili neobvezen modul »jax-ws-cxf«, ki prinaša integracijo in odvisnosti iz knjižnice CXF. Dana izboljšava mora v okviru primerka spletne storitve omogočati funkcionalnosti, ki jih prinaša knjižnica CDI. V ogrodju je odvisnost CDI označena kot neobvezna, kar pomeni, da mora integracija ponujati možnost proste izbire, v kolikor želimo storitev razviti s pomočjo knjižnice CDI ali brez. Pomembno je, da je rešitev šibko sklopljena z osrednjim modulom ogrodja KumuluzEE, ta pa se doseže tako, da je odvisnost med »jax-ws-cxf« in osrednjim modulom KumuluzEE enosmerna. V tem primeru velja, da je neobvezen modul odvisen od obveznega in ne obratno.

3.4 Rešitev

Znotraj programskega okolja Java se rešitve storitev, ki temeljijo na protokolu HTTP, bistveno ne razlikujejo. Večina teh storitev je načrtovanih tako, da osnovo storitve predstavlja komponenta Servlet. Prednosti komponente Servlet so, da že vsebuje logiko, ki je potrebna za prevzem zahteve HTTP, kreiranje konteksta zahteve in branje lastnosti, potrebnih za implementacijo



Slika 3.1: Diagram UML integracije KumuluzEE s knjižnico CXF

storitve v nadaljevanju. Rešitev integracije bomo izvedli s komponento Servlet in knjižnico CXF, ki sprejme potrebne parametre za izvrševanje zahteve SOAP in temelji na protokolu HTTP, kot je prikazano na sliki 3.1. Diagram smo izvozili iz razvojnega orodja in temelji na jeziku UML, ki ga uporabljamo za načrtovanje in dokumentiranje implementacije. Načrtovanje diagramov v mikrostoritvah opisujejo avtorji v viru [30].

Za uspešno delovanje storitev SOAP moramo nastaviti parametre knjižnice CXF, ki temelji na zasnovi vodila (angl. Bus) in služi kot koncept za delitev virov, potrebnih za delovanje storitve. Parametri določajo, na katerem naslovu bo storitev SOAP na voljo in kateri primerek objekta bo služil za izvrševanje poslovne logike storitve. Parametre, ki jih določi razvijalec sto-

ritev, preberemo iz nastavitev ob zagonu sistema, med drugim se preberejo tudi parametri prek anotacije WsContext, ki določa globalno pot in URL do storitve. Če razvijalec anotacije WsContext ne uporabi, se nastavi privzeta pot do vmesnika SOAP, ki je sestavljena iz imena razreda in streže zahtevam SOAP. Ta vzorec je podoben in zasnovan po primeru integracije podpore SOAP v aplikacijskem strežniku JBoss [31].

Nadaljnji del rešitve predstavlja tudi projekt »kumuluzee-samples«, ki dopolnjuje rešitev s primeri uporabe storitev SOAP na podlagi integracije s knjižnico CXF. Primer storitve vsebuje tudi dobre prakse, npr. nastavitve projekta za delovanje in samodejno generiranje kode, primer datoteke WSDL in uporabe programskih vzorcev, kot sta interceptor vmesnika in interceptor zahtevka SOAP.

3.5 Sklep

Skozi postopek razvoja podpore spletnih storitev smo izkusili problematiko, s katero se soočajo glavni igralci razvoja programske opreme v podjetjih, ki dobavljajo rešitve v obliki ogrodij in aplikacijskih strežnikov. Z umestitvijo rešitve, pravilno implementacijo in zagotovitvijo delovanja smo dosegli funkcionalno in delujočo rešitev, ki ponuja uspešno komunikacijo SOAP med mikrostoritvami. Rešitev je razvita in dostopna na GitHubu v repozitoriju ogrodja KumuluzEE [32].

Poglavje 4

Analiza hitrosti in učinkovitosti protokolov

V tem poglavju bomo določili metodologijo in metrike, ki so primerne za proučevanje zmogljivosti delovanja protokolov spletnih storitev. Metrika je merljiva enota, ki natančno zajema, kar želimo meriti. Splošne definicije za zmogljivostno metriko ni, ker je odvisna od sistema, njen opis pa zahteva natančno razumevanje sistema in uporabe le-tega [33].

Osredotočili se bomo na kazalce, ki so v delovanju na zunaj nevidni, a predstavljajo pomemben vpliv na delujoč sistem. Metrike bomo aplicirali na področje komunikacijskih protokolov, podrobneje na lastnosti, ki prispevajo k učinkoviti in hitri komunikaciji med sistemi. Nekateri izmed storitvenih protokolov nosijo dodatne lastnosti in funkcionalnosti, zato bomo v takih primerih rezultate prikazali le za dotičen protokol. Osredotočili se bomo le na metrike, ki zagotavljajo hitrostno primerjavo protokolov in učinkovitosti njihovih ogrodij, in sicer:

- zagonski čas storitve;
- procesorski čas, potreben za zagon storitve;
- število ustvarjenih niti ob zagonu;
- poraba pomnilnika ob zagonu;

- količina potrebnega pomnilnika;
- branje enega objekta;
- pisanje enega objekta;
- hkratno branje mnogih objektov (zahtev);
- hkratno pisanje mnogih objektov (zahtev);
- hitrostna analiza pretvorbe sporočil;
- branje objektov različnih velikosti;
- vpliv metapodatkov na velikost sporočila.

4.1 Določitev okolja

4.1.1 Strojna oprema

Zmogljivostno analizo bomo izvedli na strojni opremi A, ki služi kot referenčni sistem za opravljanje zmogljivostne analize. Strojna oprema temelji na prenosnem računalniku Macbook Pro Early 2015 z operacijskim sistemom MacOS Mojave in procesorjem 2,7 GHz Intel Core i5, ki ima dve fizični, štiri logične procesorske enote in 16 GB 1867 MHz DDR3 sistemskega pomnilnika. Zaradi primerjave delovanja na različnih strojnih opremah bomo nekatere izmed metrik merili na strojni opremi B, ki temelji na namiznem računalniku HP Pro Desk 600 G3 z operacijskim sistemom Windows 10 in procesorjem 3,6 GHz Intel Core i7-7700, ki ima štiri fizične, osem logičnih procesorskih enot in 32 GB 2400 MHz DDR4 sistemskega pomnilnika.

4.1.2 Ogrodja

Analizo bomo izvedli v programskem okolju Java verzije 8. Za testiranje protokolov gRPC, SOAP in REST bomo uporabili ogrodja, v katerih bomo implementirali storitve na podlagi različnih protokolov. Da bi omogočili čim

objektivnejše rezultate, bomo za analizo protokolov SOAP in REST uporabili enako ogrodje, to je KumuluzEE. Zaradi dejstva, da gRPC na ogrodju KumuluzEE in aplikacijskem strežniku Jetty ne deluje, bomo za ta protokol uporabili sistem, ki temelji na aplikacijskem strežniku Netty.

4.2 Določitev postopka

V analizi se bomo osredotočili na pogoste vzorce, kot je na primer stresno testiranje, skozi postopek pa bomo skušali pridobiti objektivne rezultate za vse protokole spletnih storitev. Uporabili bomo enak postopek za vse kandidate ter pri tem uporabili enaka orodja in programsko okolje. V postopku bomo za vse protokole uporabili enako količino podatkov in čim bolj primerljive podatkovne strukture. Algoritem in podrobnosti podatkovnih struktur so objavljeni na spletni strani GitHub [32]. Sprva bomo pričeli z analizo porabe sistemskih virov in časovno analizo zagona vsakega od ogrodij.

```
$ python -c 'from time import time; print int(round(
    time() * 1000))'; java -jar masters-*/target/masters
    -*-1.0-SNAPSHOT.jar& jconsole -interval=1 $!
```

Primer 4.1: Ukaz, ki zažene storitev

Z ukazom 4.1, pri katerem nadomestimo »*« z imenom storitve, najprej izpišemo trenutno stanje milisekund od epohe, ki ga kasneje odštejemo od časa, ki nam ga storitev izpiše v trenutku, ko je na voljo. V tem koraku bomo analizirali zagonski čas aplikacije, ki ponuja analizirano storitev, pri tem pa bomo uporabili časovno enoto milisekunde. Naslednja metrika, ki nam bo pomembna v tem koraku, je procesorski čas, potreben za zagon storitve. Metrika je izražena v sekundah in pove, koliko sekund je procesor potreboval za postavitev storitve. Število potrebnih niti nam pove lahkoto oziroma enostavnost storitve, z njimi pa lahko merimo tudi kapaciteto. Količina pomnilnika označuje, koliko sistemskega pomnilnika je potrebovala storitev za

zagon, minimalna kopica pa, kolikšna je minimalna količina pomnilnika, ki je sploh mogoča za delovanje storitve.

Analizo protokolov za komunikacijo med mikrostoritvami bomo nadaljevali z merjenjem porabe sistemskih virov med delovanjem storitev. Zanimal nas bo odzivni čas storitev, ki je pogojen s porabo sistemskih virov, kot je procesorski čas, ter s porabo pomnilnika in največjim številom zahtev (obremenitvijo), ki jih je storitev zmožna izvesti na podani strojni opremi. V analizi delovanja bodo storitve opravljale funkcionalnosti, ki smo jih opisali v določitvi metrik. Zaradi narave delovanja ogrodij bomo prvi zahtevek izpustili iz hitrostne analize.

V hitrostni analizi delovanja storitev se bomo osredotočili na:

1. branje enega objekta;
2. branje mnogih objektov;
3. pisanje enega objekta;
4. pisanje mnogih objektov;
5. čas pretvorbe sporočil;
6. čas, potreben za prenos sporočil z uporabo novega kanala;
7. čas, potreben za prenos sporočil z uporabo obstoječega kanala.

Pri uporabi obstoječega kanala želimo pri vzpostavitvi povezave ponovno uporabiti čim manj programskih virov.

4.3 Primerjava protokolov glede na hitrostne metrike

4.3.1 Zagon storitev

V tabeli 4.1 so prikazani rezultati analize porabe virov ob zagonu storitev, ki implementirajo specifične protokole. Rezultati protokolov SOAP in REST

	gRPC	SOAP	REST
Zagonski čas	1054 ms	3675 ms	3302 ms
Procesorski čas	1,75 s	6,49 s	7,37 s
Število niti	19	29	29
Poraba pomnilnika	88,6 MB	189,7 MB	197,2 MB
Min. kopica pomnilnika po gc	5,9 MB	11 MB	13,5 MB

Tabela 4.1: Poraba sistemskih virov ob zagonu

so zelo primerljivi, saj temeljijo na enakem ogrodju, ki potrebuje nekoliko več virov za inicializacijo komponent in storitev. Vsi rezultati metrik v tem kontekstu kažejo v korist protokolu gRPC zaradi njegovega preprostega in lahkega ogrodja. V kolikor se odločimo za gRPC z aplikacijskim strežnikom Netty, žrtvujemo funkcionalnosti ogrodja KumuluzEE z aplikacijskim strežnikom Jetty v korist hitri postavitvi in manjši porabi virov.

4.3.2 Delovanje storitev

V tem poglavju želimo analizirati hitrosti protokolov v danih ogrodjih skozi delovanje storitve. Učinkovitost bomo merili skozi vnaprej dane metrike, ki temeljijo na učinkovitostnih analizah povezanih del. Muller idr. v svojem delu [34] raziskujejo vpliv oblike sporočil na velikost končne količine podatkov, potrebnih za prenos informacije med storitvami. Rezultat njihove raziskave je 75 odstotkov manjša velikost sporočila v prid binarni strukturi protokolnih tokov GPB proti obliki XML. Enajst odstotkov znaša razlika v potrebni procesorski moči za pretvorbo v prenosno obliko, prav tako v prid protokolnih tokov. Njihove rezultate lahko primerjamo z našimi, pridobljenimi iz analize. V tabeli 4.2 so rezultati analize spletnih storitev skozi delovanje. [1]

Iz tabel 4.2 in 4.3 je razvidno, da smo iz zmogljivejše strojne opreme

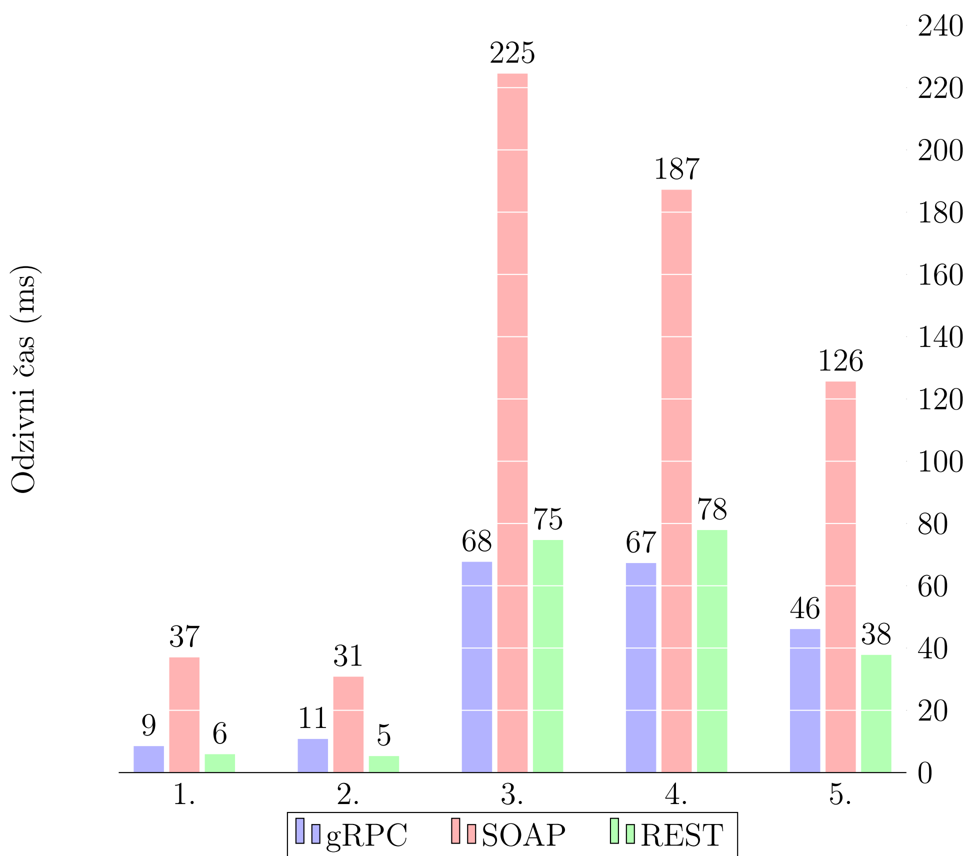
[1] n=10000, v zahtevku so upoštevani časi vzpostavitve povezave

	gRPC	SOAP	REST
1. Branje enega objekta	8,5 ms	37 ms	5,9 ms
2. Pisanje enega objekta	10,8 ms	30,8 ms	5,3 ms
3. Branje mnogih objektov[1]	67,7ms	224,5 ms	74,7 ms
4. Pisanje mnogih objektov[1]	67,3 ms	187,2 ms	77,9 ms
5. Čas pretvorbe sporočil[1]	46,1 ms	125,6 ms	37,8 ms
6. Fizična velikost sporočila za prenos[1]	1420 kB	4840 kB	3470 kB
7. Čas, potreben za prenos sporočil z uporabo novega kanala[1]	6137 ms	30310 ms	2320 ms
8. Čas, potreben za prenos sporočil z uporabo obstoječega kanala[1]	3860 ms	6250 ms	2158 ms

Tabela 4.2: Časovna analiza metrik glede na spletno storitev

	gRPC	SOAP	REST
1. Branje enega objekta	5,54 ms	21,9 ms	3,27 ms
2. Pisanje enega objekta	5,72 ms	23 ms	2,63 ms
3. Branje mnogih objektov[1]	37 ms	148 ms	48 ms
4. Pisanje mnogih objektov[1]	32 ms	138 ms	54 ms
5. Čas pretvorbe sporočil[1]	22 ms	66 ms	14 ms
6. Fizična velikost sporočila za prenos[1]	1420 kB	4840 kB	3470 kB
7. Čas potreben za prenos sporočil z uporabo novega kanala[1]	3333 ms	17700 ms	1354 ms
8. Čas potreben za prenos sporočil z uporabo obstoječega kanala[1]	2112 ms	3710 ms	832 ms

Tabela 4.3: Časovna analiza metrik glede na spletno storitev, uporaba strojne opreme B



Slika 4.1: Graf hitrosti in učinkovitosti, manj je bolje (po točkah, strojna oprema A)

B dobili boljše rezultate, vendar razmerje rezultatov med primerjanimi protokoli ostaja približno enako. Grafično predstavitev rezultatov najdemo na sliki 4.1.

Podrobno smo raziskali tudi časovno analizo prenosa sporočil glede na velikost sporočila, vendar rezultati niso predstavljeni zaradi nerelevantnosti. Do neprimerljivih rezultatov pride zaradi vpliva čistilca pomnilnika (angl. garbage collector), ki se pogosto začne vključevati ob večjih obremenitvah.

4.4 Sklep

V analizi hitrosti in učinkovitosti smo ugotovili, da novejši protokol gRPC za prenos sporočila porabi za kar trikrat manjšo količino prenosa, kar pomeni zelo veliko izboljšavo v primeru obremenjenosti omrežja med storitvami. Takšna izboljšava pomeni mnogo prednosti pri gradnji odjemalcev, ki se povezujejo na storitve prek manj kakovostnih omrežij z omejeno pasovno širino. Izmenjava in pretvorba podatkov sta bili v našem primeru in strojni opremi primerljivi s protokolom REST. Pričakovali smo boljši rezultat v pretvorbi sporočil s strani protokola gRPC, vendar boljših rezultatov na podani strojni opremi nismo pridobili. Če naredimo primerjavo med protokoloma REST in SOAP, opazimo, da je protokol REST nekoliko hitrejši. Ti rezultati služijo kot potrditev analize hitrosti v primeru uporabe video konference, ki so jo izvedli avtorji v članku [35]. Rezultati so prav tako primerljivi z analizo, ki so jo nekoliko podrobneje izvedli avtorji, kjer so primerjali protokola REST in SOAP [36].

V primerih, ko smo storitev pred vsakim klicem ponovno vzpostavili, smo pridobili mnogo slabše rezultate, zato je pomembno, da vzpostavitev in delovanje protokolov pravilno implementiramo. Slab vzpostavitveni čas protokolov vpliva na odzivni čas storitev ob prvem klicu storitve, tega bi bilo smiselno izboljšati z optimizacijo algoritmov ali samodejno vzpostavitvijo prvega klica, kar imenujemo tudi ogrevanje zagona storitev.

Poglavje 5

Raziskava primernosti protokolov v okolju mikrostoritev

V tem poglavju bomo opredelili metrike, s katerimi bomo analizirali lastnosti protokolov za komunikacijo med mikrostoritvami. Za posamezno metriko bomo opisali lastnosti storitev SOAP, gRPC in REST, po analizi pa bomo izvedli evalvacijo glede na prednosti oziroma slabosti posameznega protokola.

5.1 Opredelitev raziskave

Mikrostoritve predstavljajo dekompozicijo iz monolitnih izvajalnih okolij v neodvisno namestitvene storitve, ki so optimizirane za reševanje specifičnih problematik [5]. Pri transformaciji monolitne aplikacije moramo s procesom skozi več faz, ena izmed teh je vpeljava komunikacijskega mehanizma za komunikacijo med storitvami [37]. Glavni način komunikacije med mikrostoritvami poteka prek objavljenih in verzioniranih vmesnikov (angl. API). Da vmesniki učinkovito opravljajo svojo nalogo v komunikaciji, morajo temeljiti na komunikacijskih protokolih, ki so optimizirani za delovanje v okolju mikrostoritev. Mikrostoritve, prilagojene za delovanje v oblaku, moramo graditi tako, da zadovoljijo funkcionalne potrebe, kot sta spremljanje delovanja in učinkovitosti sistema ter zmožnost odkrivanja storitev [38]. Poleg funkcio-

nalnih zahtev so pomembne tudi nefunkcionalne lastnosti. Te so primernost in zahtevnost izvedbe storitve ter hitrost delovanja. V raziskavi bomo vključili in določili način meritev za doseganje rezultatov protokolov ter izvedli analizo primernosti glede na metrike, ki so ključne za učinkovito delovanje storitev znotraj distribuiranih okolij.

5.2 Določitev evalvacijskih metrik

Da lahko izvedemo primerjavo protokolov, moramo najprej določiti lastnosti, ki vplivajo na razvoj, delovanje in vzdrževanje protokolov, na katerih temeljijo storitve. Metrike bomo najprej opisali v okviru protokolov za komunikacijo med storitvami, nato pa za vsako izmed njih evalvirali komunikacijske protokole gRPC, SOAP in REST. Metrike, po katerih bomo analizirali storitve so:

5.2.1 Interoperabilnost

V okolju mikrostoritev je interoperabilnost pomembna lastnost protokolov, saj morajo biti storitve med sabo neodvisne. Vsaka izmed storitev je lahko zgrajena v poljubnem programskem okolju. Kot smo že opisali, je to ena izmed ključnih zahtev aplikacij, zgrajenih posebej za delovanje v oblaku. Odvisnost med storitvami predstavljajo komunikacijski protokoli, saj zagotavljajo kanal, prek katerega se storitve sporazumevajo.

5.2.2 Hitrost izvajanja

Ključna metrika za komunikacijski protokol, ki se uporablja v distribuiranem okolju mikrostoritev je učinkovitost, ki vpliva na hitrost izvajanja storitev. Poleg optimizacije oblike sporočil za prenos je pomemben algoritem, ki v izvajalnem okolju deluje tako, da v celoti izkorišča sistemske vire. Pomembno prednost v komunikaciji predstavljata zasnova asinhrone komunikacije in podpora podatkovnih tokov, ki so lahko eno- oziroma dvosmerni.

Ta lastnost je zelo pomembna, saj se uveljavlja kot zelo dobra strategija za delo s podatki [12]. Takšnim storitvam pravimo tudi, da so reaktivne (angl. reactive).

5.2.3 Kakovost

Razvoj spletnih storitev in vzdrževanje teh predstavljata mnoge dejavnosti, ki zahtevajo ogromno različnih virov in vključujejo različne skupine ljudi. Zato je za uporabnike in podjetja pomembno, da razumejo kakovost protokolov kot metriko, so jo sposobni meriti in predvideti. Kakovost vmesnikov povezujemo s pravilnostjo, razumljivostjo, dostopnostjo in konsistenco. Ti vidiki so predpogoj, da je rešitev sprejeta v skupnosti [39]. Rahman idr. [40] bolj podrobno označujejo kakovost storitev in jo definirajo na podlagi sledečih metrik:

- odzivnost,
- zanesljivost,
- cena na klic,
- razpoložljivost,
- zmogljivost,
- varnost,
- dostopnost,
- transakcijska sposobnost,
- kapacitivnost,
- integriteta,
- regulatornost,
- ugled.

Duan [41] v svojem članku dodaja poleg že omenjenih metrik še utilizacijo, odpornost na napake, skalabilnost in elastičnost. V okviru kakovosti bi dodali zelo pomemben vidik kvalitete API-jev, to je dokumentiranje storitev in vzdrževanje dokumentacije, ki je zelo pomembna in potrebna za uspešno integracijo in upravljanje. V to kategorijo spada tudi dokumentiranje razlik med programskimi vmesniki, v kolikor so ti v mnogih verzijah.

5.2.4 Zanesljivost

Zanesljivost komunikacijskih protokolov merimo v številu nedostavljenih glede na skupno število poslanih sporočil. V sklopu zanesljivosti moramo upoštevati tudi neprimeren odzivni čas storitev, ki predstavlja v okolju mikrostoritev zelo pomembno težavo, ki jo rešujemo z vnaprej pripravljenimi vzorci in orodji. V kolikor v zasnovi distribuiranega okolja opazimo izgubo sporočil, to pomeni nedoslednost podatkov med povezanimi storitvami, odprava takšnih neusklajenosti pa zahteva večjo kompleksnost implementacije storitev oziroma zagotavljanje večje odpornosti na napake.

5.2.5 Enostavnost razvoja

Na enostavnost razvoja vpliva več dejavnikov. Za določitev te metrike je treba določiti vzroke in oceniti vpliv na kompleksnost implementacije. Na težavnost razvoja pomembno vpliva programsko okolje, vendar se bomo zaradi poenostavitve primera tukaj omejili na programski jezik Java. V tem primeru so za razvoj storitev na voljo knjižnice, ki poenostavijo razvoj in preložijo na razvijalca le konfiguracijo in implementacijo poslovne logike storitve. Zimmermann idr. [9] omenjajo termin relativna težavnost, ki ga lahko izmerimo tako, da upoštevamo:

- število odločitev, ki jih moramo narediti med gradnjo storitve;
- število možnosti, ki je na voljo za implementacijo;
- strošek razvoja in velikost tehnološkega tveganja.

Strošek razvoja in velikost tehnološkega tveganja lahko izračunamo s pomočjo analize razširjenosti določenega protokola in analize postopka razvoja, ki vpliva na enostavnost, število napak in hitrost.

5.2.6 Hitrost razvoja in strošek

Da bi razvoj storitev učinkovito napredoval od zahteve do končne rešitve, je zelo pomemben dejavnik hitrost razvoja storitve. Hitrost razvoja je povezana s kompleksnostjo implementacije in prilagodljivostjo okolja, v katerem je storitev izvedena. Omenjeno metriko definirajo časovne enote, ki so potrebne za dosego enakega cilja, implementiranega v različnih komunikacijskih protokolih in okoljih. V tej metriki moramo upoštevati, da ključno vlogo pri hitrosti določajo ekipa, izkušnje in število razvijalcev.

5.2.7 Količina porabe sistemskih virov

Okolje mikrostoritev je sestavljeno iz majhnih in obvladljivih storitev, za katere je pričakovano, da je poraba sistemskih virov majhna. Majhna poraba sistemskih virov prinaša agilnost storitve in zmožnost namestitve na mnoge, različne ponudnike platform v oblaku. Količina porabe sistemskih virov je povezana s kompleksnostjo vzdrževanja in stroški storitev. Metrika definira enote, povezane s sistemskimi viri, to so procesorski čas, količina potrebnega pomnilnika in količina potrebnih podatkov za prenos po omrežju.

5.2.8 Odpornost na napake v razvoju

Skozi proces razvoja mikrostoritev prihaja do človeških napak, kjer se programska oprema obnaša na drugačen način, kakor je bilo zastavljeno v fazi arhitekture ali implementacije programske kode. Zaradi majhnosti storitev, zaradi katere arhitektura mikrostoritev prinaša prednosti, je odprava napak na storitvi preprostejša in potrebuje manjši obseg časa v okviru izdajanja programske opreme. Količina storjenih napak se meri v številu napravljenih napak glede na število vsebovanih vrstic programske kode. Pri tem moramo

vedeti, da je metrika pogojena s človeškim faktorjem in jo je težavno meriti zaradi različnih izkušenj razvoja programske opreme. Protokoli igrajo pomembno vlogo v delovanju mikrostoritev, kadar pride do napak v integraciji. Med sabo se razlikujejo v dovzetnosti za spremembe v izmenjanih sporočilih, protokoli lahko v tem primeru dopuščajo odstopanje od specifikacije brez zavrnitve sporočila.

5.2.9 Odpornost na napake v delovanju in samozdravljenje

Kot smo že omenili na začetku dela, obstaja osem glavnih zmot v distribuiranem računalništvu. Vsaka izmed teh vpliva na komunikacijo med storitvami tako, da jo prizadene v različnih in vnaprej nepredvidenih položajih. Če v tem primeru mislimo, da se ta problematika odkriva z uveljavitvijo mikrostoritev, se motimo. Iskanje rešitev za odpornost na napake v distribuiranem računalništvu sega že v rosno dobo spleta [42]. Odpornost na napake pomeni, da je storitev zmožna reagirati na napake v drugih storitvah tako, da je prvotna storitev v delujočem stanju, kljub temu da katera od kličočih storitev ni dosegljiva [43].

Odpornost na napake lahko rešujemo s slednjimi pristopi. O proaktivnem pristopu govorimo takrat, ko storitev poskuša predvideti negativne posledice in se odzove na način, da skuša le-te odpravljati, še preden težava nastopi. Druga možnost odziva je reaktivni odziv, kjer storitev ne predvideva napak in deluje do trenutka, ko težava nastopi. Lastnosti omenjenih pristopov ter več podrobnosti opisujejo avtorji [44]. Kot alternativa nastopi še tretja, adaptivna možnost, kjer se storitev prilagaja danemu trenutku in se vede situaciji primerno. Patra idr. [45] in Bala idr. [46] skozi modele, ki so primerljivi zahtevam kakovosti v tem delu, primerjajo z elementi metrik in analizirajo orodja za implementacijo odpornosti na napake. Metriko lahko definiramo kot sposobnost reagiranja v primeru napake v komunikaciji.

5.2.10 Število vrstic programske kode

Število vrstic kode je pomemben vidik v fazi razvoja mikrostoritev, saj se z večanjem števila vrstic povečuje možnost za napake, strošek, povečuje se kompleksnost storitve, ki lahko vodi izven zasnove mikrostoritev. Metrika je povezana s potrebnim časom za razvoj mikrostoritve, zato je pomemben dejavnik pri izbiri protokola za povezavo storitev v okolju mikrostoritev.

5.2.11 Primernost za vzdrževanje

Mikrostoritve s svojo majhnostjo pripomorejo k vzdrževanju storitev, zato je potrebno, da je zasnova komunikacijskih protokolov zastavljena na način, da je preprosta za vzdrževanje. Pomembno vlogo pri vzdrževanju nosi zmožnost dokumentiranja storitev. Dokumentacija mora biti jasna, podrobna, opisni jezik mora biti preprost, tvorjenje berljivih formatov pa mora biti podprto v več različnih človeku prijaznih oblikah. Storitve se skozi čas spreminjajo, prav tako pa se spreminjajo programski vmesniki do storitev, zato je pri protokolu pomembno, da podpira način vodenja verzij storitev oziroma razločevanje razlik v specifikaciji med verzijami storitev.

5.2.12 Varnost

Pomemben vidik v okolju mikrostoritev predstavlja varnost. Nekateri komunikacijski protokoli vsebujejo varnostne komponente že v samem protokolu, v takih primerih knjižnice oziroma ogrodja vsebujejo podporo, ki skrbi, da se podatki prenašajo v varnem načinu. Takšen primer varne komunikacije je preprostejši za vzdrževanje in razvoj, vendar prinaša pomanjkljivosti v podpori za različna programska okolja. Poleg vgrajene podpore lahko za varnost komunikacijskega kanala skrbi dodatna plast nad komunikacijskim protokolom, ki deluje kot vsebnik za zaščito komunikacije. Zelo razširjen primer takšnega protokola je TLS, ki deluje nad protokolom HTTP.

5.2.13 Transakcijska podpora

Zahteve po transakcijski podpori so v nekaterih poslovnih zahtevah nujne, okolje mikrostoritev pa zaradi tega občasno potrebuje podporo takšnim zahtevam. Transakcijska podpora v okolju mikrostoritev ni zaželena, ker ta pomeni žrtvovanje razpoložljivosti in particijske tolerance na račun doslednosti. Teorem CAP izvira iz podatkovnih zbirk, ki nudijo transakcijsko podporo. Na kratko se takšnim podatkovnim zbirkam pravi, da imajo podporo atomarnosti, doslednosti, izolacije in vzdržljivosti (angl. – ACID atomicity, consistency, isolation, durability). Mikrostoritve delujejo v distribuiranem okolju, kjer komunikacija poteka po omrežnih povezavah, ki so načeloma nezanesljive. Transakcijska podpora v takšnem primeru resno upočasni izvajanje sistemov, zato se v distribuiranih okoljih večinoma uporablja koncept zakasnjene doslednosti. Če je želja po transakcijski podpori storitev, so komunikacijski protokoli, ki imajo to zmožnost, v bistveni prednosti.

5.2.14 Neodvisnost od transportnega protokola

Nekatere zasnove komunikacij so odvisne od protokola, ki omogoča prenos podatkov, kar predstavlja tesno sklopljenost. V nekaterih primerih tesna sklopljenost transportnega protokola oziroma kanala pomeni prednost, vendar v večini primerov se izkaže, da je bolje, če so rešitve razčlenjene, ker te prinašajo prosto izbiro in lažje razumevanje delovanja.

5.2.15 Dostopnost storitev, odkrivanje in objavljanje

Distribuiran sistem predstavlja množico homogenih in heterogenih storitev, ki med sabo komunicirajo. Zaradi kardinalnosti storitev, ki se nenehno spreminja, in potreb po uporabi naključne storitve, zaradi izpada ali nedelovanja določenega primerka storitve se je treba v distribuiranem sistemu zavedati prisotnosti in lokacije storitev. Če lokacije storitev niso znane v fazi namestitve komponent, mora sistem vsebovati koncept za dinamično odkrivanje mikrostoritev, kjer se novo ustvarjene storitve same prijavijo in odjavijo iz

distribuiranega okolja mikrostoritev. Obstaja več vzorcev, ki realizirajo rešitev za dano težavo v takšnih okoljih [47]. Komunikacijski protokoli so tesno povezani z dostopnostjo storitev, saj morajo biti omenjeni vzorci podprti v programskih okoljih, kjer želimo objaviti storitve, natančneje vmesnike v distribuirano okolje.

5.2.16 Spremljanje obremenjenosti in učinkovitosti sistema

Zaradi množice storitev v distribuiranem okolju je zelo pomemben vidik nadzor nad sistemom in spremljanje določenih aspektov, ki dajejo dodano informacijo o stanju sistema. Najpogosteje nas v tem primeru zanima, kaj in kje je težava. Spremljanje metrik omogočimo navadno prek nadzornih plošč, ki pretvarjajo podatke iz posameznih storitev v obliko, prijazno za pregled sistemskega skrbnika sistema. Ti podatki so posebej pomembni, ker kažejo podrobne informacije o uporabi sistema, vsaka neobičajnost tako podaja dodano vrednost obnašanja sistema. Enega največjih izzivov v primeru spremljanja delovanja sistema predstavlja pisanje in umeščanje programske kode za spremljanje, saj ta predstavlja stroškovno težavo med količino metrik, ki jih želimo spremljati, ter ceno za realizacijo in umestitev le-teh [48]. V sklopu analize te metrike bomo evalvirali protokole glede na sposobnost spremljanja oziroma nadzorovanja v okolju.

5.2.17 Spremljanje vitalnosti sistema

Zaradi specifik arhitekture mikrostoritev in lastnosti distribuiranega okolja se poleg nefunkcionalnih zahtevajo tudi funkcionalne zahteve, ki zapolnjujejo vrzeli v pomanjkljivostih distribuiranega sistema pred monolitom. V okolju mikrostoritev se kaže velika potreba po nadzoru sistema. Ker je sistem raznovrsten oziroma heterogen, zgrajen iz mnogih storitev, je nujno, da vsaka izmed storitev podpira funkcionalnost za nadziranje vitalnosti sistema (angl. health check). Prednost je, če je storitev preverjanja vitalnosti določena na

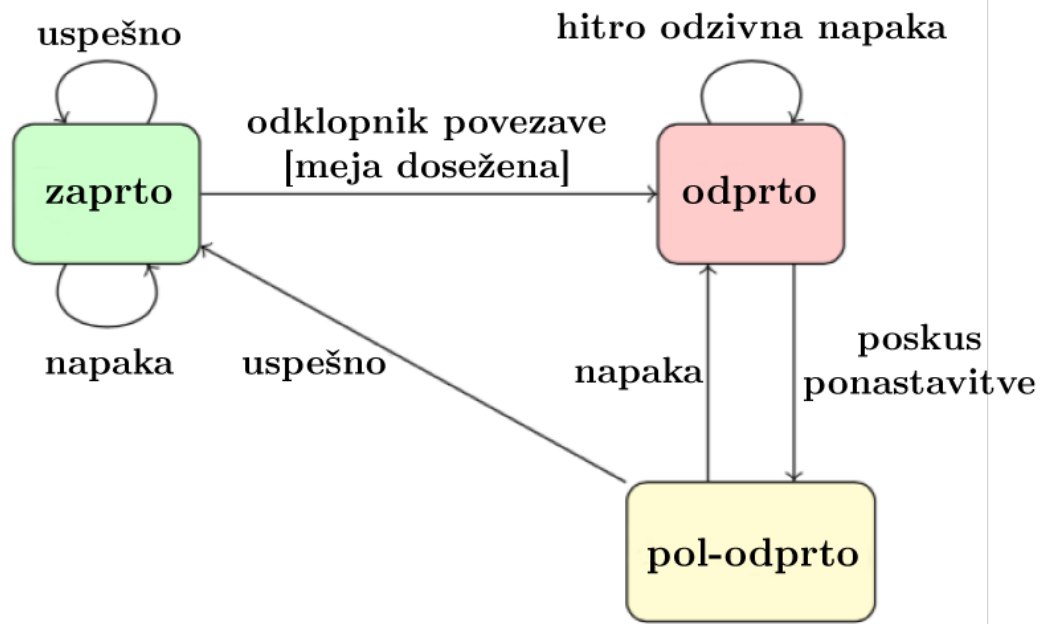
ravni protokola, s katerim se storitve sporazumevajo.

5.2.18 Imunost protokola na nepravilnosti v komunikaciji

Če katera izmed storitev prejme preveliko število zahtev v časovni enoti, se bo zagotovo v nekem trenutku pričela odzivati nenavadno, z veliko verjetnostjo pa bo postala neodzivna. V okolju mikrostoritev je odpoved komponent neizogibna, zato je treba ta pojav predvideti že v samem razvoju storitev. Zaradi heterogenosti storitev v okolju so storitve med seboj odvisne, zato lahko v primeru nedelovanja enega primerka storitve pripelje do kaskadne odpovedi sistema. V primeru nepričakovanega obnašanja storitev mora imeti sistem zmožnost reagirati na dane položaje tako, da odpoved enega primerka storitve vpliva na celoten sistem v čim manjšem obsegu. Sistemu, ki je zmožen obvladovati takšne pojave, pravimo, da je odporen na napake (angl. fault tolerant). Za obvladovanje takšnih pojavov v sistemih uporabljamo tehnike in mehanizme, kot je prekinjevalec tokov (angl. circuit breaker). Celotna zasnova temelji na zakonu hitre odpovedi (angl. fail fast). Slika 5.1 prikazuje mehanizem, ki deluje kot dodatni sloj komunikacijskega protokola, ki skrbi za spremljanje stanja povezav, in če zazna slabosti v eni izmed storitev, prekine komunikacijo za določen čas in s tem ponudi dani storitvi priložnost, da si opomore [47].

5.2.19 Podpora orodij za razvoj, testiranje in razhroščevanje storitev

Orodja za razvoj programske opreme nam olajšajo proces izdelovanja storitev in s svojimi funkcijami zmanjšajo verjetnost za napake med pisanjem specifikacij in programske kode. V programskem okolju Java se za razvoj storitev najpogosteje uporabljajo orodja Eclipse, Netbeans, IntelliJ IDEA in JDeveloper. Vsa izmed orodij podpirajo vmesnik za interaktivno razhroščevanje znotraj procesa izvrševanja programske kode. Vsako izmed storitev je



Slika 5.1: Diagram stanj prekinjevalca tokov

mogoče testirati na več načinov, z enotskimi testi, integracijskimi testi in okoljskimi testi, ki se glede na izbran protokol lahko razlikujejo v obliki.

5.3 Metodologija

V prejšnjem podpoglavju dela so postavljene metrike, na katerih temelji nadaljnja analiza posameznih komunikacijskih protokolov. V sklopu analize in rezultatov je uporabljen postopek v specifičnem okolju, znotraj katerega so vsi rezultati analize tudi ponovljivi. V okviru zmogljivostne primerjave smo uporabili strojno opremo, opisano v poglavju 4.1.1. Analiza vsebuje evalvacijo metrik, katere rezultat je prednost komunikacijskega protokola pred drugimi. Potti idr. [49] izvajajo zmogljivostno analizo protokolov SOAP in REST. To poglavje na novo zastavi okolje za izvedbo primerjave in kot prispevek dodaja primerjavo protokola gRPC. Tihomirovs in Grabis [7] snujeta primerjavo protokolov SOAP in REST z uporabo oblik sporočil JSON, XML

in GBP. Naveden vir ne vsebuje natančnega opisa okolja, ki je pogoj, iz katerega bi to delo lahko raziskavo nadaljevalo, zato smo v tem primeru okolje ponovno definirali.

5.4 Evalvacija metrik in rezultati

V tem poglavju povzamemo zmogljivostno analizo in v primerjavo dodamo metrike, ki predstavljajo funkcionalnosti v okolju mikrostoritev. V nadaljevanju interpretiramo rezultate skozi metrike, pomembne za primerjavo komunikacijskih protokolov. Primerjava temelji na podatkih in ugotovitvah iz literature tega dela ter na praktičnih izkušnjah ob implementaciji protokolov.

5.4.1 Interoperabilnost

- SOAP
  Pri protokolu SOAP za nastanek standardov oziroma bolj podrobno napotkov in dobrih praks skrbi organizacija Oasis. Interoperabilnost je tako zagotovljena s specifikacijo WS-I, ki določa postopke in teste, ki preverjajo interoperabilnost implementacij. Protokol SOAP s specifikacijo je podprt v programskih okoljih Java, C#, VB in številnih drugih.

- gRPC
  Kljub temu da gre za novejši protokol, ima gRPC zelo razširjeno podporo programskih jezikov, ki vključujejo Javo, Go, C++, Python, Ruby, C#, Node.js, Objective-C, PHP in Dart.

- REST
  Arhitekturni slog zaradi svoje oblike in prisotnosti protokola HTTP podpirajo skorajda vsa programska okolja, ki podpirajo izmenjavo tekstovnih sporočil prek protokola HTTP.

5.4.2 Hitrost izvajanja

V poglavju 4 smo izmerili hitrostne metrike protokolov.

- SOAP
  Če povzamemo rezultate hitrosti izvajanja, se je protokol SOAP odrezal kot najmanj učinkovit s strani odzivnosti storitve. Vzpostavitev delovanja storitve SOAP se izkaže kot zelo počasna. Pridobljeni rezultati ne kažejo v korist protokolu SOAP, vendar se vrednosti ne razlikujejo v tolikšni meri, da bi lahko označili storitev SOAP kot neprimerno za delovanje v okolju mikrostoritev.

- gRPC
  Analiza hitrosti postavlja protokol gRPC kot zelo konkurenčen REST-u s stališča hitrosti. V našem primeru se je gRPC zelo dobro odrezal pri pretvorbi sporočil v obliko, primerno za prenos, saj je prispevek v velikosti sporočila pri tem bil več kot tri kraten.

- REST
  V analizi hitrosti smo z rezultati pokazali, da so knjižnice, natančneje knjižnica JAXB, ki pretvarja binarno objektno obliko v obliko JSON, zelo učinkovite, saj binarni objekt pretvorijo oziroma serializirajo v zelo podobnem času kot mnogo novejši protokol gRPC.

5.4.3 Kakovost

- SOAP
  Protokol SOAP je starejši protokol in vsebuje mnoge prednosti, zaradi standardov in napotkov, ki določajo, kako morajo biti storitve grajene. Protokol se sicer zaradi hitrosti in težavnosti razvoja uporablja manj kakor konkurenčni, vendar še vedno vsebuje prednosti, zaradi katerih je uporaba tega protokola smiselna. Stabilnost protokola SOAP je odvisna od prenosnega protokola, ki služi kot kanal za prenos sporočil SOAP. Vodenje verzij protokola je v teoretičnem modelu preprosto,

saj je doseženo z menjavo imenskega področja, programsko pa z menjavo spletnega naslova, ki streže zahtevkom na določenem imenskem področju. Razvoj takšnih storitev je zaradi zasnove zahtevnejši, posledično zahteva slabe vzorce programske kode. Dokumentiranje storitev SOAP je mogoče s pomočjo orodij, ki pretvarjajo specifikacijo WSDL v berljiv dokument in primeren za prikaz. Ena izmed knjižnic, ki nudi pretvorbo vmesnika v dokumentacijo, je Enunciate, vodenje razlik med programskimi vmesniki pa nudi orodje Wsdldiff.

- gRPC
  gRPC je eden izmed novejših protokolov in še ni toliko razširjen. Vedno bolj se uveljavlja tam, kjer je potreba po hitrosti, vendar je zaradi načina kodiranja sporočil težavno razhroščevanje na prenosni poti. Standardov, ki določajo razvoj, ni, vendar obstajajo smernice za razvoj storitev, mnogo napotkov in smernic pa je na voljo na uradni spletni strani. Prednost protokola gRPC je koncept vodenja verzij programskih vmesnikov z dodatnim označevanjem definicije sporočil, zato lahko povzamemo, da je to ena izmed lastnosti, ki izboljšujejo kakovost storitev. V poglavju 5.2 smo v sklopu kakovosti omenili dokumentiranje, ki pa ni lastnost, ki ji dajejo navadno razvijalci protokolov velik pomen. Za ta namen tudi pri protokolu gRPC obstaja množica orodij, ki podpirajo dokumentiranje v obliki, primerni za razvijalce. Na spletu [50] je na voljo orodje Protoc-gen-doc, ki omogoča generiranje berljive dokumentacije.

- REST
  Arhitekturni slog uporablja protokol HTTP, ki je temelj za delovanje svetovnega spleta. Zaradi razširjenosti uporabe je implicitno podprt na vseh programskih okoljih. Standardov za arhitekturni slog REST ni, obstajajo pa smernice in dobre prakse za snovanje programskih vmesnikov, ki so omejeni na metode protokola HTTP in vire (URI), na katerih so podatki na voljo. Vodenje verzij programskih vmesnikov se rešuje

z uporabo glave protokola ali spremembe naslova vira, odločitev pa je odvisna od zasnove in dejavnikov distribuiranega okolja, v katerem storitve komunicirajo. Dokumentiranje storitev je v programskem okolju Java mogoče avtomatizirati s knjižnicami, kot je OpenAPI-annotations, in odprtokodnimi orodji, kot je Swagger-diff, ki omogočajo objavljanje razlik med programskimi vmesniki. Prav tako obstajajo mnoga orodja, ki omogočajo pretvorbo specifikacije v berljivo obliko.

5.4.4 Zanesljivost

- SOAP
  Kot del specifikacije v protokolu SOAP za zanesljivost dostave sporočil skrbi standard WS-RM (angl. Web Service Reliable Messaging), ki s pomočjo mehanizma sekvenc ustvari tok sporočil, ki zagotavlja uspešen prejem. Več podrobnosti najdemo v specifikaciji spletnih storitev [19].

- gRPC
  Protokol gRPC zagotavlja zanesljivost sporočil na ravni protokola, ki poskrbi za samodejno ponovno vzpostavitev povezave, razporeditev obremenjenosti, preverjanje vitalnosti, preklopa ob napakah in drugih.

- REST
  Arhitekturni slog REST ne vsebuje standardov, torej striktno ne predpisuje modela za zanesljivo dostavo sporočil. V ta namen uporabljamo prakso idempotentnosti klicev, saj protokol HTTP temelji na konceptu najboljšega truda (angl. best effort). Idempotentnost klica pomeni, da je zahteva uspešna tudi v primeru, ko je določena akcija že opravljena. Primer takšnega delovanja je brisanje vira, ki ne obstaja. Če uporabljamo v procesu sekvenco mnogih klicev REST, nastane težava, v kolikor kateri izmed njih ni dostavljen pravilno. Takrat se prične proces obnovitve modela REST, ki vzpostavi delujoče stanje [51, 52].

5.4.5 Enostavnost razvoja

- SOAP
  V programskem okolju Java in mnogih drugih je spletne storitve SOAP mogoče razviti na dva različna načina. Pri pristopu od spodaj navzgor razvijalec razvije programsko kodo in iz nje generira datoteko, ki opisuje spletno storitev. V drugem načinu od zgoraj navzdol, ki je bolj primeren, pa se najprej razvije datoteka, ki opiše storitev, iz nje pa avtomatika generira programsko kodo in s tem prihrani mnogo vrstic programske kode in možnosti za napake. Večina programskih okolij, ki podpirajo razvoj storitev SOAP, ima težave z razvojem, saj ne vsebujejo potrebnih orodij, ki razvijalcu olajšajo razvoj storitev SOAP.

- gRPC
  Razvoj storitev, ki temeljijo na protokolu gRPC, je preprostejši od razvoja storitev bolj razširjenega stila REST, kar ugotavljajo avtorji spletnega dnevnika [53]. V našem primeru razvoja mikrostoritev se je ravno tako izkazalo, da je razvoj storitev gRPC zaradi opisnega jezika protokolnih tokov hitrejši od konkurenčnih protokolov.

- REST
  Prednost arhitekturnega sloga za komunikacijo je enostavnost implementacije storitev, saj za razvoj ni potrebnih posebnih orodij. Sloj HTTP podpirajo v večini vsa programska okolja, razvijalec pa mora poskrbeti za pretvorbo podatkov iz programskega okolja v obliko, primerno za prenos. V programskem okolju Java za preprostejši razvoj obstaja knjižnica, ki poskrbi za prejem, izvršbo in odgovor zahtevka HTTP v pravilni obliki. Slaba lastnost je v tem primeru pomanjkanje podpore za opis storitev, kar poveča zapletenost razvoja.

5.4.6 Hitrost razvoja in strošek

- SOAP
  V kolikor je na voljo datoteka WSDL z opisom spletne storitve, je ra-

zvoj storitve s pomočjo orodij in pravega programskega okolja preprost in hiter. Opisna datoteka zahteva znanje označevalnega jezika XML, pravilnost datoteke pa je mogoče zagotoviti s sintaktičnim potrjevalcem. Zaradi opisne datoteke je poleg razvoja storitve hiter tudi razvoj odjemalcev, možnosti za napake med razvojem pa so minimalne.

- gRPC
  Kakor pri storitvah SOAP tudi v protokolu gRPC obstaja posebna datoteka PROTO, ki opisuje storitev gRPC v jeziku protokolnih tokov. Protokol sicer ni podprt v tolikšnem obsegu kot ostali protokoli, vendar je razvoj hiter v mnogih popularnih programskih okoljih zaradi podanih primerov, smernic in dobre dokumentiranosti protokola.

- REST
  REST ne vsebuje navodil za opisovanje in generiranje storitev, temveč prosto dopušča grajenje storitve s prosto izbranimi pristopi. Ker protokol REST temelji na osnovi dobrih praks in enostavnosti, se je s pomočjo skupnosti začel razvoj orodij, kot je OpenAPI oziroma nekdanji Swagger in RAML. Funkcionalnosti orodij, ki podpirajo generiranje storitev in pretvarjanje v dokumentacijo, so s pomočjo skupnosti vedno bolj obširne. Razvoj storitev REST je predvsem hiter v programskih okoljih, ki naravno podpirajo objektno strukturo JSON, kot je na primer JavaScript.

5.4.7 Količina porabe sistemskih virov

V poglavju 4 smo s pomočjo orodij, ki so na voljo v operacijskem okolju Mac OS, izmerili količino sistemskih virov, ki jih storitve potrebujejo za delovanje. Veliko težo rezultatom dajejo ogrodja, v katerih so storitve implementirane, zato je objektivno primerjati storitve, ki so bile analizirane v enakih ogrodjih, to sta SOAP in REST.

- SOAP
  Storitev SOAP je v ogrodju KumuluzEE za zagon potrebovala dobre

tri sekunde, kar je nekoliko več od primerljivega protokola REST. Pri merjenju procesorskega časa je potrebovala storitev dobrih šest sekund, kar pomeni dober rezultat v primerjavi s protokolom REST. Če analiziramo razmerje med zagonskim in procesorskim časom, ugotovimo, da rezultat kaže na slabšo izkoriščenost v primeru mnogih procesorskih enot, saj je bilo za zagon potrebno več časovnih enot in manj procesorskega časa. Ugotovimo, da je poraba pomnilnika v storitvi SOAP nekoliko manjša, vendar še vedno primerljiva.

- gRPC
  Protokol gRPC je zaradi različnih zmožnosti in funkcij nesmiselno primerjati s protokoloma REST in SOAP, ki sta zgrajena v drugačnem ogrodju in aplikacijskem strežniku. Skozi rezultate opazimo, da storitev zaradi svoje majhnosti in enostavnosti porabi bistveno manj sistemskih virov. Sam zagonski čas je hitrejši za več kot 300 %, prav tako procesorski čas, ki je manjši za podoben faktor. Na enostavnost storitve kažejo tudi metrike, kot sta število niti in poraba pomnilnika. Iz teh rezultatov je razvidno, kako pomembno je graditi manjše enote, ki so namenjene točno določenemu namenu, kar sovpada z mikrostoritvami.

- REST
  Storitev REST je v primeru porabe sistemskih virov primerljiva storitvi SOAP z nekoliko manjšim zagonskim časom in večjo porabo sistemskega pomnilnika. Ogrodje KumuluzEE z aplikacijskim strežnikom Jetty poskrbi za enakovredno število niti med storitvami, zato tukaj ni bistvenih razlik. Iz podanih rezultatov je razvidno, da vzpostavitev vmesnikov REST ni poceni, saj mora mehanizem ob zagonu poiskati vse definicije vmesnikov prek tako imenovanega iskalnika anotacij in registrirati poti v lasten notranji mehanizem.

5.4.8 Odpornost na napake v razvoju

- SOAP
  Komunikacijski protokol SOAP je zaradi natančnosti opisa storitev zelo odporen na napake v razvoju. Grajenje storitev in spreminjanje teh je zelo obvladljivo, predvsem velika prednost protokola je odpornost za napake pri integraciji. Opisni jezik WSDL natančno opisuje, kako se morajo odjemalci storitev povezati s ponudniki storitev, poleg tega pa je zaradi podrobnega opisa mogoče preveriti napake vsakega sporočila že na samem odjemalcu. S tem protokol SOAP ponuja minimalno odstopanje z vidika napak od specifikacije.

- gRPC
  Kakor pri protokolu SOAP tudi gRPC nudi prednosti grajenja storitev iz opisnega jezika. Opisni jezik, ki je določen z datoteko PROTO, sicer ni tako močan kot WSDL, ki ponuja veliko množico tipov in pravil, imenske prostore ter natančno definicijo lokacij storitev. V tem primeru govorimo o storitvi, ki sicer jasno specificira izmenjavo, vendar ne na tako podrobni ravni kakor konkurenčni protokol SOAP.

- REST
  Protokol REST ima omejeno podporo za usklajevanje vmesnikov oziroma API-jev na strani strežnika in odjemalca, zato je možnost za napake večja. Pobude, kot sta Swagger in OpenAPI, želijo na tem področju izboljšati podporo, zato je za ta namen na voljo ogromno orodij, ki omogočajo hitrejšo gradnjo storitev [54]. Delovanje storitev REST je manj striktno, protokol dopušča določene spremembe v strukturi sporočila, kar je navadno nastavljivo v implementaciji ponudnika JSON.

5.4.9 Odpornost na napake v delovanju in samozdravljenje

- SOAP
  Protokol SOAP je zmožen zagotavljati identifikacijo napak v izmenjavi sporočil s pomočjo validacije sporočil in shem XML. Tolerantnost do napak v protokolu SOAP zagotavljamo z replikacijo storitev in redundantnostjo, v literaturi pa zasledimo, da obstajajo tudi kompleksnejši modeli, ki rešujejo to problematiko, kot je upravljanje z napakami SOAP po dogovoru Byzantine [55], ta pa je na voljo kot modul v implementaciji Axis2.

- gRPC
  Protokol gRPC že vsebuje mehanizme za odpornost na napake v delovanju in samozdravljenje storitev s posebnim algoritmom [11], ki deluje na podlagi uravnavanja obremenitev.

- REST
  V protokolu REST ni vgrajenega mehanizma za samodejni odziv v primeru napak v storitvi. Kot rešitev je na voljo knjižnica Hystrix, ki ponuja lastnosti prekinjevalca tokov in v primeru napak v storitvi reagira z določenim mehanizmom. Knjižnica deluje ne glede na to, kakšno obliko sporočila določimo za prenos [56].

5.4.10 Število vrstic programske kode

- SOAP
  Število ustvarjenih vrstic kode je odvisno od uporabe ogrodja, ki služi za grajenje spletnih storitev SOAP. V začetni fazi grajenja storitev na način od zgoraj navzdol (najprej WSDL) je število vrstic kode večje zaradi opisnega jezika, vendar v kolikor storitev uporabljata vsaj dva odjemalca ali več, povprečno število vrstic kode strmo pade, saj v okviru odjemalcev pisanje dodatnih vrstic ni potrebno. Število vrstic

kode za poslovno logiko je v primeru programskega jezika Java skozi vse protokole podobno oziroma enako. V kolikor se za grajenje storitev uporabi način od spodaj navzgor, se število vrstic programske kode nekajkrat poveča, saj je v tem primeru potrebna implementacija struktur v programskem jeziku.

- gRPC
  Protokol gRPC izkorišča opisni jezik v datoteki PROTO za določitev podrobnosti protokola, programska koda objektov pa se generira samodejno s pomočjo programskih vtičnikov. Opisni jezik za razliko od jezika WSDL ni tako natančen, njegova struktura pa je bolj preprosta, tako da obsega manj vrstic programske kode.

- REST
  Protokol REST sam po sebi ne omogoča pretvorbe iz opisnega jezika v razredni model programskih jezikov. Zato je treba v tem primeru napisati mnogo vrstic kode, ki služijo v namen podatkovnega modela in implementaciji storitve. V tem primeru je število vrstic kode večje. Trenutno aktualna pobuda OpenAPI vsebuje nekaj projektov, ki podpirajo tovrstno generiranje kode, vendar ti projekti še niso popolnoma skladni in ustrezni.

5.4.11 Primernost za vzdrževanje

- SOAP
  V sklopu storitev SOAP je programska oprema primernejša za vzdrževanje na strani odjemalcev, na strani strežnika pa manj primernejša zaradi zasnove opisnega jezika WSDL in orodij za razvoj storitev. Ena izmed prednosti je zagotavljanje skladnosti implementacije storitve s pomočjo datoteke WSDL, s katero poskrbimo nespremenjeno delovanje storitve glede na specifikacijo.

- gRPC

Protokol gRPC zaradi zasnove nudi prednosti za vzdrževanje na strežniški kot na odjemalski strani, saj na ravni protokola nudi podporo spremembam v verzijah protokola, kjer spremembe na strežniški strani ne vplivajo neposredno na delovanje odjemalcev v okviru enake različice opisnega jezika PROTO.

- REST
  Programska oprema, ki nudi storitve REST, je manj primerna za vzdrževanje na strani odjemalcev, saj je potrebno veliko vložka za zagotavljanje doslednosti storitve s strežniško stranjo, v kolikor ne uporabljamo enega izmed orodij za dokumentiranje. Na strežniški strani je treba storitve in spremembe dokumentirati, tako da se minimalizira vložek, ki je potreben za implementacijo odjemalcev.

5.4.12 Varnost

- SOAP
  Storitve, zgrajene na protokolu SOAP, podpirajo standarde za kriptiranje in podpisovanje sporočil in s tem pomembno izboljšujejo varnostni vidik storitev. Varnost je enkapsulirana znotraj ovojnice sporočila SOAP, standard pa narekuje, kako so sporočila sestavljena in obravnavana.

- gRPC
  Protokol gRPC je mogoče uporabljati brez varnostnih elementov, v kolikor potrebujemo varnostno podporo, pa je na voljo varnostni mehanizem SSL/TLS. Za storitve, ki delujejo v okviru oblačnih storitev podjetja Google, je na voljo varnostni mehanizem na podlagi žetonov, vendar se ga sme in je smiselno uporabiti le, v kolikor storitve komunicirajo z ostalimi Googlovimi storitvami. V primeru protokola gRPC so vsi varnostni mehanizmi že vgrajeni v sam protokol in za to ni potrebne dodatne implementacije.

- REST
  Protokol REST sam po sebi ne določa varnostnih mehanizmov, kadar se izkaže potreba po varnostnih mehanizmih, se te zahteve rešujejo izven protokola z uporabo drugih protokolov in konceptov. V praksi se mnogokrat uporabljajo OAuth, OAuth2 in OpenID-Connect.

5.4.13 Transakcijska podpora

- SOAP
  Protokol SOAP določa obliko sporočil za prenos, od prenosnega protokola pa je odvisna zanesljivost dostave. Zanesljivost prenosa sporočila je možno zagotoviti s pomočjo standarda WS-Transaction, ki vsebuje navodila za razvoj storitve s transakcijsko podporo.

- gRPC
  gRPC za razliko od protokola SOAP, ki s specifikacijo WS-Transaction skrbi za zagotovitev uspešnosti klica, ne zagotavlja podpore v tej obliki. Protokol zaradi prisotnega mehanizma, ki vzdržuje listo zdravih storitev, nudi boljšo povezljivost in s tem manjšo verjetnost za napake, s tem pa izboljšuje zanesljivost protokola.

- REST
  Protokol ne vsebuje dodatne plasti za zagotavljanje zanesljivosti sporočil izven okvirjev protokola HTTP, kjer ni prisotnega mehanizma, ki bi nudil transakcijsko podporo na ravni sporočil. V tem področju tudi ni podpore ogrodij, ki bi zagotavljala takšne storitve v sklopu sloga REST.

5.4.14 Neodvisnost od transportnega protokola

- SOAP
  Prednost protokola SOAP je transparentnost in neodvisnost komunikacijskega protokola, ki je namenjen za izmenjavo sporočila SOAP [9].

Z uporabo datoteke WSDL določimo, kakšen protokol za prenos bomo uporabili za prenos vsebine SOAP. Zaradi enostavnosti in razširjenosti je ta protokol navadno HTTP, ni pa nujno, saj protokol podpira tudi izmenjavo podatkov na podlagi protokolov SMTP, FTP, MIME in drugih [57].

- gRPC
  Protokol gRPC predpisuje za izmenjavo podatkov prenosni protokol HTTP/2, ki je nadgradnja protokola HTTP in prinaša mnoge prednosti v komunikaciji oziroma izmenjavi sporočil. Ena izmed prednosti takšne izmenjave podatkov je istočasna izmenjava podatkov med odjemalcem in strežnikom brez ponovne vzpostavitve povezave.

- REST
  Zasnova REST je tesno sklopljena s protokolom HTTP zaradi načina razpolaganja virov, torej prosta izbira oziroma menjava prenosnega protokola ni mogoča.

5.4.15 Dostopnost storitev, odkrivanje in objavljanje

- SOAP
  Protokol SOAP je nudil podporo za odkrivanje storitev s pomočjo standarda UDDI (angl. Universal Description, Discovery and Integration), vendar je standard namenjen registraciji oziroma imeniku storitev, v okviru okolja mikrostoritev pa se pričakuje, da se lahko storitve prijavijo in odkrivajo dinamično. Standard je stabilen in je čez čas dozorel, vendar v industriji ni bil splošno sprejet in uveljavljen, zato je bil opuščen. Iz povezane literature je razvidno, da se kljub starosti protokola še vedno objavljajo dela, ki raziskujejo omenjeno problematiko [58, 59].

- gRPC
  Za protokol gRPC obstaja podpora za odkrivanje storitev v knjižnicah oziroma orodjih, kot je Etcd, ki nudi podporo tudi za ostale protokole.

Storitve se samodejno prijavijo v distribuirano okolje, ki temelji na distribuirani konfiguraciji, od koder je na voljo odkrivanje storitev drugih aplikacij.

- REST
  Protokol REST neposredno ne nudi podpore za odkrivanje in registracijo storitev, vendar obstajajo mnoge knjižnice in orodja, kot je na primer Etcd ali Consul, ki ponujajo odkrivanje in registracijo storitev znotraj namestitvenega okolja.

5.4.16 Spremljanje obremenjenosti in učinkovitosti sistema

- SOAP
  Protokol SOAP je starejši protokol. Zaradi manjše uporabnosti tako ni zaživel v okolju mikrostoritev v tolikšni meri, da bi se mehanizmi, ki so potrebni za delovanje sistema v distribuiranem okolju, razvili okoli omenjenega protokola. Manjkajoča je podpora programskega vmesnika za spremljanje učinkovitosti sistema.

- gRPC
  V sklopu spremljanja obremenjenosti storitve je podjetje Google pripravilo sklop knjižnic OpenCensus, ki omogočajo samodejno spremljanje obremenjenosti in učinkovitosti sistema oziroma metrik. Prav tako obstaja tudi implementacija »java-grpc-prometheus«, ki nudi integracijo orodja za spremljanje metrik s protokolom gRPC.

- REST
  Protokol REST v okolju mikrostoritev sobiva z mehanizmi, ki služijo za učinkovito delovanje storitev v distribuiranih okoljih. Za potrebe spremljanja storitev REST v programskem okolju Java igra ključno vlogo organizacija Eclipse. Ta specificira standard MicroProfile, ki določa način spremljanja metrik storitev. Ogrodja, kot sta KumuluzEE

in Thorntail, poskrbijo za spremljanje storitev s svojimi lastnimi implementacijami, ki podpirajo nadzor nad storitvami in spremljanje odzivnosti ter obremenjenosti storitve, funkcionalnosti, ki so zelo pomembne v takšnih okoljih.

5.4.17 Spremljanje vitalnosti sistema

- SOAP
  Protokolu manjka podpora za odkrivanje storitev, podpora za nadzor obremenjenosti sistema je mogoča z uporabo dodatnih orodij, storitev za nadzor vitalnosti sistema pa ni standardizirana za delovanje v okviru protokola SOAP.

- gRPC
  Protokol gRPC podpira preverjanje vitalnosti sistema s posebnim vmesnikom, definiranim v dokumentaciji protokola. Vmesnik vrača vitalnost sistema v sklopu stanj SERVING, NOT_SERVING in UNKNOWN.

- REST
  V okolju programskega jezika Java je za protokol REST na voljo specifikacija MicroProfile Health, ki narekuje, kako je zgrajen vmesnik, ki ponuja informacije o vitalnosti storitve.

5.4.18 Imunost protokola na nepravilnosti v komunikaciji

- SOAP
  Protokol SOAP v samem protokolu nima vgrajenega mehanizma za detektiranje nepravilnosti v komunikaciji in mehanizma, ki bi ugotovljene napake odpravljal oziroma minimaliziral vpliv napak na komunikacijski protokol. V okviru protokola HTTP kot prenosnega protokola za prenos sporočil SOAP je mogoče vpeljati mehanizem iz programske

knjižnice Hystrix, ki omogoča odkrivanje napak na prenosnem kanalu HTTP in minimalizacijo učinka napak na prizadeti sistem.

- gRPC
  Protokol gRPC vsebuje mehanizem za odkrivanje napak in hranjenje informacij o stanju ostalih primerkov storitev [11]. Mehanizem za vzdrževanje stanj povezav temelji na konceptu vzdrževanja bazena vseh povezav z ustreznimi stanji.

- REST
  Ker protokol REST temelji na prenosnem protokolu HTTP, je v sklopu programskega okolja Java na voljo knjižnica Hystrix podjetja Netflix, katere funkcionalnost je, da preprečuje širjenje napak v okolju, kjer se ena izmed storitev ne obnaša pravilno.

5.4.19 Podpora orodij za razvoj, testiranje in razhroščevanje storitev

- SOAP
  Storitve SOAP so uveljavljene skoraj dve desetletji in s tem tudi podpora orodij za razvoj le-teh. To, da so orodja za gradnjo storitev SOAP že del razvojnega okolja Java, kaže na to, kako uveljavljene so. Orodja, ki so del okolja, kot sta Wsgen in Wsimport, so na voljo za uporabo v ukazni vrstici, v kolikor pa želimo grafičen pogled na gradnjo storitev, je za to na voljo vgrajena podpora v okoljih IntelliJ, Eclipse, Netbeans, JDeveloper in drugih. Za oblikovanje strukture storitve v obliki datotek WSDL nudi zelo dobro podporo produkt XmlSpy podjetja Altova, prav tako pa najdemo podporo grajenju datotek WSDL tudi v omenjenih grafičnih okoljih. Za avtomatizirano testiranje vmesnikov lahko uporabimo orodja kot so SoapUI, JMeter in druga. Spremljanje sporočilnih tokov v protokolu SOAP ni tako razširjeno, vendar lahko uporabimo lastno implementacijo, ki jo vključimo kot SOAPHandler.

Dober primer je projekt Stagemonitor, ki daje enega izmed primerov implementacije.

- gRPC
  Grafična okolja, kot je IntelliJ, imajo vgrajeno podporo za lažje delo z datotekami PROTO prek lastnih vtičnikov, za gradnjo razrednega diagrama in predpripravljenih vmesnikov pa poskrbimo z vtičniki Maven. Sintakso PROTO datoteke si lahko ogledamo v primeru 5.1. Za avtomatizirano testiranje vmesnikov gRPC lahko uporabimo knjižnico Karate-grpc, obstaja pa tudi možnost testiranja delovanja s pomočjo testov enot. Pri protokolu gRPC lahko naletimo na težavo pri razhroščevanju v omrežju, saj se podatki prenašajo v binarni obliki, ki ni enostavno prepoznavna s pogleda človeka. Za spremljanje sporočilnih tokov je za protokol gRPC na voljo specifikacija OpenTracing, ki podpira mnoge implementacije, kot so Jaeger, LightStep in Instana.

```
service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply)
     {}
  rpc SayHelloAgain (HelloRequest) returns (
     HelloReply) {}
}

message HelloRequest {
  string name = 1;
}

message HelloReply {
  string message = 1;
}
```

Primer 5.1: Vsebina PROTO datoteke

- REST
  Za razvoj storitev REST lahko uporabimo več načinov razvoja. Če izberemo način od vrha navzdol, lahko izbiramo med mnogimi orodji, ki omogočajo specifikacijo vmesnikov. Razvoj v obliki RAML je eden izmed boljših, saj poleg opisljivosti vmesnika dobimo tudi vtičnik za urejevalnik Atom, ki omogoča vizualizirano gradnjo programskih vmesnikov. Podoben koncept je možen s pomočjo oblike OpenAPI, kjer s pomočjo spletnega urejevalnika urejamo in hkrati vizualiziramo želene programske vmesnike. Za okolje IntelliJ je podpora vgrajena v obliki vtičnikov, ki poskrbijo za dodatne funkcije za delo s specifikacijami v obliki RAML ali OpenAPI. Pogosto želimo iz specifikacije zgraditi objektni model tudi pri storitvah REST, zato za gradnjo razrednega modela izberemo enega izmed vtičnikov Maven. Za avtomatizirano testiranje vmesnikov REST lahko uporabimo Selenium, RestAssured, Postman, SoapUI in druge. Za spremljanje sporočilnih tokov v protokolu REST nimamo vgrajene podpore, temveč lahko uporabimo knjižnice in orodja, ki nam omogočajo spremljanje klicev na vmesnikih in centralizirano spremljanje obnašanje sistema. Na voljo so knjižnice, kot so Jaeger, LightStep, Instana in druge.

5.5 Ugotovitve

Skozi vse naštete metrike smo ugotovili, da obstaja mnogo lastnosti skozi katere lahko ocenjujemo in primerjamo protokole, vendar vse metrike nimajo enakega vpliva na primernost delovanja v okolju mikrostoritev, saj se kljub izbrani arhitekturi zahteve lahko razlikujejo glede na funkcionalne zahteve mikrostoritev. Skozi analizo najpomembnejših protokolov nismo upoštevali lastnosti podatkovnih središč, ponudnikov infrastrukture, ki so odvisni od protokolov in porazdeljenosti storitev kar vpliva na odzivnost storitev [60]. Iz podanih metrik lahko sklepamo, da imajo prednost tiste metrike, ki predstavljajo podporo procesu namestitve, upravljanja in nadzora. Lastnosti kot

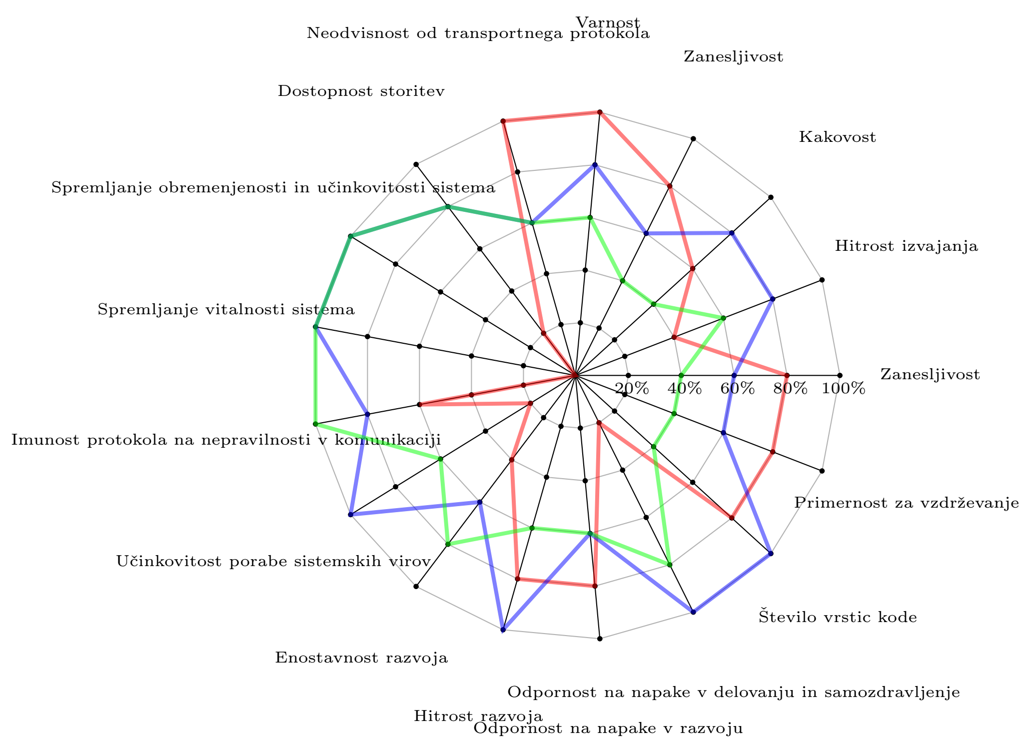
so učinkovitost, kakovost, hitrost razvoja, količina porabe sistemskih virov in število vrstic kode lahko rešujemo s pristopi kot sta povečanje sistemskih virov in izobraževanje razvojnikov.

V tabeli 5.1 smo prikazali rezultate analize protokolov glede na podane metrike, ki smo jih definirali skozi poglavje. Definirali smo tudi količino porabe sistemskih virov, vendar metrike nismo prikazali v tabeli rezultatov, saj so rezultati zaradi različnih ogrodij neprimerljivi. Rezultati metrik so zaokroženi na 20 odstotkov, ker rezultatov metrik ne moremo oceniti z veliko natančnostjo, lahko pa podamo grobo oceno glede na rezultate opravljene raziskave.

Uporabnost določenega protokola je določena z njegovimi lastnostmi, prednostmi in slabostmi [7]. Tabela 5.2 prikazuje lastnosti komunikacijskih protokolov in primernost uporabe glede na zahteve projekta. Graf 5.2 prikazuje primernost protokolov glede na lastnosti, ki so pomemben dejavnik razvoja aplikacij v okolju mikrostoritev.

	gRPC	SOAP	REST
Hitrost izvajanja	80%	40%	60%
Kakovost	80%	60%	40%
Zanesljivost	60%	80%	40%
Varnost	80%	100%	60%
Neodvisnost od transportnega protokola	60%	100%	60%
Dostopnost storitev	80%	20%	80%
Spremljanje obremenjenosti in učinkovitosti	100%	0%	100%
Spremljanje vitalnosti sistema	100%	0%	100%
Imunost protokola na nepravilnosti v komunikaciji	80%	60%	100%
Učinkovitost porabe sistemskih virov	100%	20%	60%
Enostavnost razvoja	60%	40%	80%
Hitrost razvoja	100%	80%	60%
Odpornost na napake v razvoju	60%	80%	60%
Odpornost na napake v delovanju in samozdravljenje	100%	20%	80%
Število vrstic kode	100%	80%	40%
Primernost za vzdrževanje	60%	80%	40%
Zanesljivost	60%	80%	40%
Podpora orodij	80%	60%	80%

Tabela 5.1: Tabela rezultatov primernosti protokolov glede na postavljene metrike



Slika 5.2: Diagram primernosti komunikacijskih protokolov za delovanje v okolju mikrostoritev (SOAP, gRPC in REST)

	Glavni kriterij	Primernost za projekte
REST	Boljša skalabilnost, kompatibilnost, hitrost, enostavnost, model strežnik odjemalec	Integracija mobilnih aplikacij, enostavna integracija odjemalec–strežnik
SOAP	Večja varnost in zanesljivost, manjša količina napak, asinhrona komunikacija, distribuirano računalništvo, podpora standardov (WSDL, WS)	Poslovni informacijski sistemi (B2B), bančni sistemi, plačilni sistemi
gRPC	Boljša skalabilnost, hitrost, omejena pasovna širina, manjša količina napak, integriran mehanizem za toleranco okvar	Integracija mobilnih aplikacij, enostavna integracija odjemalec–strežnik, sistemi z omejeno količino virov

Tabela 5.2: Lastnosti protokolov in primernost uporabe

Poglavje 6

Izboljšave protokolov in mehanizmov za delovanje v okolju mikrostoritev

Raziskava protokolov, ki smo jo naredili v prejšnjih poglavjih, kaže na večje pomanjkljivosti v lastnostih ogrodij in protokolih, ki so bistvene za delovanje storitev v distribuiranem okolju, zato bomo v tem poglavju načrtovali rešitev, ki rešuje pomanjkljivosti v protokolu SOAP.

6.1 Pomanjkljivosti

Novejši protokoli so učinkovitejši, vendar žrtvujejo mnogo funkcionalnosti, ki so jih nudili nekateri izmed starejših protokolov. Ena izmed problematik so starejše storitve, ki že implementirajo pomembne poslovne funkcionalnosti, vendar temeljijo na starejših protokolih. V kolikor je treba takšne storitve povezati v okolje distribuiranih sistemov, je smiselna implementacija pretvornika oziroma tako imenovanega vzorca adapter, ki služi kot prevajalec iz starejšega protokola v protokol, ki služi za komunikacijo med mikrostoritvami. Nekateri protokoli, kot je na primer SOAP, so zelo razširjeni, zato je smiselno dopolniti arhitekturo obstoječega protokola z manjšimi program-

skimi popravki tako, da kljubuje in zadostuje potrebam mikrostoritev.

Ena izmed najpomembnejših pomanjkljivosti protokola SOAP je pomanjkanje podpore za samodejno objavljanje in odkrivanje storitev. Protokol za prenos sporočil SOAP podpira množico prenosnih protokolov, zato je implementacija takšne funkcionalnosti zahtevna. Zaradi enostavnosti se bomo v tem poglavju osredotočili le na enega izmed protokolov za prenos sporočil SOAP. Problematiko bomo reševali v okviru prenosnega protokola HTTP, ki tvori s protokolom SOAP najpogostejšo izbiro pri prenosu podatkov med storitvami.

Pogoj rešitve za zagotavljanje odkrivanja storitev je analiza delovanja posamezne storitve in njenega okolja, poleg tega pa je pomembna notranja organizacija izmenjave podatkov, ki je opisana s pomočjo datoteke WSDL. V primeru datoteke WSDL, ki smo jo uporabili za hitrostno analizo, opazimo, da je datoteka sestavljena iz mnogih sklopov oziroma razdelkov. Datoteke WSDL vsebujejo shemo s sporočili, ki opisujejo obliko sporočil za izmenjevanje podatkov. V elementu »binding« najdemo listo operacij, ki omogočajo izvajanje oddaljenih metod. Vsaka izmed operacij opisuje metodologije izmenjave podatkov, pri katerih lahko izbiramo med interakcijami zahteva – odgovor, enosmerno sporočilo, obvestilo in zahtevan odgovor. Operacije vsebujejo tudi parameter SoapAction v klicu SOAP katerega je smiselno uporabiti za usmerjanje sporočil po omrežju, saj deluje na nivoju prenosnega protokola http.

6.2 Zahteve

Najprej bomo opredelili zahteve, ki so nujne za delovanje storitev SOAP in odkrivanje le-teh v distribuiranih okoljih. Ker se bomo osredotočili na prenosni protokol HTTP, nam bodo v pomoč dobre prakse odkrivanja storitev v arhitekturnem stilu REST. Temeljimo lahko na trditvi, da so storitve SOAP opisane v datoteki WSDL. Postavili bomo naslednje zahteve:

- arhitektura rešitve mora sovpadati z vzorci, preverjenimi v distribuira-

nih okoljih;

- odjemalec storitve mora odkriti prisotnost drugih storitev;
- storitev mora implementirati metodo za zagotavljanje vitalnosti;
- storitev se objavi na podlagi atributa »binding« v datoteki WSDL;
- strežnik mora imeti sposobnost objavljanja storitev.

6.3 Predlog rešitve pomanjkljivosti

V okolju mikrostoritev že obstajajo mnoge rešitve, ki rešujejo področje odkrivanja storitev, te pa se razlikujejo po arhitekturi in načinu delovanja. Sistemska komponenta, ki podpira odkrivanje storitev, lahko deluje kot ena storitev, na katero ostale storitve prijavljajo in odkrivajo storitve ter je namenjena le-temu. V takšnem načinu delujejo produkti, kot so Eureka, Consul, ZooKeeper in Etcd. Alternativni način implementacije je distribuiran, pri katerem vsaka izmed komponent skrbi za objavljanje in odkrivanje storitve s pomočjo protokolov, kot so DNS-SD, DHC, uPnP in NServiceBus, ki je licencirana storitev.

Težava v distribuiranih rešitvah za zagotavljanje odkrivanja storitev je pomanjkanje varnosti, saj je treba varnostne mehanizme naknadno integrirati. Kot primernejšo lahko označimo tisto rešitev, kjer je komponenta, ki zagotavlja odkrivanje, ločena, temu pravimo, da deluje kot register storitev.

V storitvah SOAP se za usmerjanje sporočil uporablja podatek SoapAction, saj je po standardu namenjen strežnikom in požarnim zidovom za pravilno usmerjanje na protokolu HTTP [61]. V primeru objavljanja in odkrivanja storitev, potrebujemo enolični identifikator storitve, zato vrednosti SoapAction ne moremo uporabiti, ker deluje znotraj množice operacij v storitvi SOAP.

Rešitve v programski opremi, ki temeljijo na specifikacijah, so bolje sprejete, zato je smiselno vpeljati odkrivanje storitev v specifikacijo, ki določa

vmesnik za implementacijo takšne funkcionalnosti. Po takšni specifikaciji integriramo implementacije oziroma knjižnice, s katerimi omogočamo rešitev odkrivanja in objavljanja storitev. Zelo pomembno je, da je specifikacija napisana tako, da je interoperabilna.

Specifikacija nam omogoča unificiran dostop do poljubnih implementacij, ki omogočajo iskano funkcionalnost, zato lahko pod njo zgradimo rešitev, ki omogoča registracijo vmesnikov SOAP v register storitev. Kot rešitev te integracije predlagamo izboljšavo knjižnic, ki podpirajo razvoj spletnih storitev ali rešitev, ki poveže specifikacijo s temi knjižnicami. Knjižnice, kot so Metro, CXF in Axis2 so najbolj pogoste, zato bi bila podpora specifikaciji za odkrivanje storitev tam najbolj zaželena.

Da lahko storitev SOAP objavimo v registru storitev, potrebujemo ime storitve, po katerem jo bodo ostale storitve odkrile. Najbolj smiselna je objava po imenu atributa Binding, ki ga najdemo v datoteki WSDL. Z objavo imena storitve, in naslova http, na katerem storitev posluša, pridobimo repozitorij storitev, kjer je na voljo množica primerkov z naslovi iskane storitve. Ko je neka storitev uspešno objavljena v repozitoriju storitev SOAP, moramo poskrbeti, da je status oziroma stanje storitve usklajeno. Za zagotovitev takšnih zahtev se lahko ravnamo po vzorcih, ki že obstajajo v rešitvah odkrivanja storitev v protokolu REST, na primer osveževanje zapisov s pomočjo periodičnih klicev repozitorija.

Razvoj storitev SOAP s funkcionalnostjo odkrivanja storitev je smiselno podpreti s tehnikami, ki omogočajo hitrejši razvoj. Ena izmed takšnih tehnik je razvoj s pomočjo anotacij, vendar takšne tehnike niso podprte v vseh programskih okoljih. Enega izmed takšnih primerov najdemo v ogrodju KumuluzEE, v modulu, namenjenemu odkrivanju storitev REST, kjer omogočamo razvoj odkrivanja storitev s pomočjo mehanizma anotacij. Razvijalec programske opreme s pomočjo mehanizma CDI vključi dostop do storitev, odkritih v repozitoriju storitev.

Odjemalce storitev SOAP implementiramo s tako imenovanimi Stubi. To so programski vmesniki, ki omogočajo lažji dostop do oddaljenih storitev, in

so praviloma samodejno generirani iz datoteke WSDL s pomočjo orodij, ki jih v programskem jeziku Java dobimo z namestitvenim paketom. Postopek generiranja ne vključuje poslovne logike, ki bi podpirala odkrivanje storitev, zato mora tudi na odjemalčevi strani za to poskrbeti rešitev, ki poveže Stub z vmesnikom, ki omogoča dostop do repozitorija spletnih storitev SOAP.

6.3.1 Zagotavljanje informacij o vitalnosti storitve

Za potrebe preverjanja stanja oziroma vitalnosti storitve je treba v protokol SOAP dodati novo operacijo, ki je namenjena vračanju informacij o delovanju storitve. V kolikor bi želeli postaviti pravila oziroma shemo za operacijo preverjanja vitalnosti, bi morala biti definicija del standarda JAX-WS. Le na ta način bi zagotovili, da vse storitve implementirajo operacijo na isti način.

6.3.2 Diskusija

V tem poglavju smo predlagali rešitev, ki bi prinesla bistvene prednosti protokolu SOAP za delovanje v okolju mikrostoritev. Predlagana rešitev je ena izmed možnosti reševanja pomanjkljivosti, vendar moramo vedeti, da obstajajo tudi alternativne poti. Ko načrtujemo rešitev je pomembno, da je ta zasnovana tako, da:

- je enostavna;
- je konsistentna;
- je razširljiva;
- je modularna;
- je sprejeta;
- jo lahko implementiramo v mnogih programskih okoljih.

Rešitve, ki upoštevajo pravkar naštete principe razvoja programske opreme vsebujejo večjo verjetnost, da bodo postale široko uporabne. Od naštetih lastnosti rešitev smo upoštevali enostavnost, konsistentnost z drugimi rešitvami, ki rešujejo problematiko v drugih programskih okoljih, razširljivost in modularnost. Pred izdajo programske opreme, pri kateri želimo doseči širšo množico je primerno, da objavimo publikacijo v obliki RFC (angl. Request For Comments). Na takšen način damo skupnosti možnost, da poda mnenje in izboljšave rešitev še preden je dana v uporabo.

Poglavje 7

Zaključek in sklepne ugotovitve

Skozi magistrsko delo smo obravnavali različna področja tehnologij, orodij, konceptov in praks, ki se prepletajo z arhitekturo mikrostoritev in distribuiranimi okolji. V delu smo opisali način integracij knjižnic, ki zagotavljajo komunikacijo med mikrostoritvami, z ogrodji, ki omogočajo razvoj storitev.

Skozi analizo protokolov smo ugotovili, da ni pravila o primernosti predlaganih protokolov za delovanje v naravnih oblačnih aplikacijah ravno zaradi poslovnih zahtev. Vsekakor igra dominantno vlogo na področju arhitekturni slog REST zaradi svoje razširjenosti in enostavnosti. Na konkretnem primeru smo dokazali, da se je protokol izkazal kot zelo hiter in učinkovit, vendar je gRPC zelo konkurenčen vsaj na področju učinkovite izmenjave podatkov in podpore opisljivosti, dokumentiranja in gradnje vmesnikov. Protokol SOAP ima svoje prednosti, vendar le-te v konceptu mikrostoritev niso tako izrazito potrebne, saj lastnosti, kot je transakcijska podpora, ne sovpadajo z lastnostmi mikrostoritev, kot sta hitrost in zakasnela doslednost.

V sklopu nadaljnega dela predlagamo podrobnejšo raziskavo izboljšav komunikacijskih protokolov in primerjavo arhitekturnega stila REST z uporabo protokolnih tokov. V poglavju 6 smo opisali predlog izboljšav protokola SOAP, zato bi bila smiselna praktična implementacija rešitve, ki podpre zahtevane funkcionalnosti. Glede na to, da se za mikrostoritve pričakuje, da so zgrajene v najprimernejšem programskem okolju glede na zahteve, je smi-

selno, da to delo nadaljujemo v smeri podrobnejše analize delovanja in primernosti protokolov v drugih programskih okoljih kjer ponovno analiziramo vpliv hitrosti in učinkovitosti.

Literatura

[1] J. Kress, H. Normann, D. S. et. al., Cloud computing and soa, Service technology magazine (LXXXII) (2014) 11.

[2] N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, L. Safina, Microservices: yesterday, today, and tomorrow, CoRR (2016) 1–3.

[3] Microservice trade-offs, `https://martinfowler.com/articles/microservice-trade-offs.html`, dostopano: 2018-08-16.

[4] T. Salah, M. J. Zemerly, C. Y. Yeun, M. Al-Qutayri, Y. Al-Hammadi, The evolution of distributed systems towards microservices architecture, in: 11th International Conference for Internet Technology and Secured Transactions (ICITST), Khalifa university, 2016, pp. 318–325.

[5] S. Newman, Building Microservices: Designing Fine-Grained Systems, O'Reilly Media, Sebastopol, CA, 2015.

[6] P. Johansson, Efficient communication with microservices, Master's thesis, Umea University, Sweden (2017).

[7] J. Tihomirovs, J. Grabis, Comparison of SOAP and REST Based Web Services Using Software Evaluation Metrics, Information Technology and Management Science (1) (2016) 92–97.

[8] Rest or soap in a cloud-native environment, `https://www.infoworld.com/article/3236391/cloud-computing/`

rest-or-soap-in-a-cloud-native-environment.html, dostopano: 2017-12-02.

[9] C. Pautasso, O. Zimmermann, F. Leymann, Restful web services vs. 'big' web services: Making the right architectural decision, in: Proceedings of the 17th International Conference on World Wide Web, WWW, ACM, New York, USA, 2008, pp. 805–814.

[10] Grpc sistem za oddaljeno komunikacijo, https://grpc.io/docs/guides/index.html, dostopano: 2017-12-02.

[11] All aboard the grpc train, http://lpan.io/migrating-to-grpc/, dostopano: 2018-08-16.

[12] The evolution of scalable microservices: From building microliths to designing reactive microsystems., https://www.oreilly.com/ideas/the-evolution-of-scalable-microservices, dostopano: 2018-08-16.

[13] V. Chang, M. Ramachandran, Y. Yao, Y.-H. Kuo, C.-S. Li, A resiliency framework for an enterprise cloud, International Journal of Information Management 36 (1) (2016) 155 – 166.

[14] M. Stine, Migrating to Cloud-Native Application Architectures, O'Reilly Media, Sebastopol, CA, 2015.

[15] D. Shadija, M. Rezai, R. Hill, Microservices: Granularity vs. performance, CoRR (2017) 1–2.

[16] M. Hofmann, E. Schnabel, K. Stanley, Microservices Best Practices for Java, IBM, New York, US, 2016.

[17] S. Brunner, M. Blöchlinger, G. Toffetti, J. Spillner, T. M. Bohnert, Experimental evaluation of the cloud-native application design, in: IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC), Zurich university, 2015, pp. 488–493.

[18] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, N. Josuttis, Microservices in practice, part 2: Service integration and sustainability, IEEE Software (2017) 97–104.

[19] Web services reliable messaging protocol (ws-reliablemessaging), `http://specs.xmlsoap.org/ws/2005/02/rm/ws-reliablemessaging.pdf`, dostopano: 2018-09-07.

[20] V. Andrikopoulos, C. Fehling, F. Leymann, Designing for cap - the effect of design decisions on the cap properties of cloud-native applications, in: Proceedings of the 2nd International Conference on Cloud Computing and Service Science, CLOSER, 2012.

[21] K. Hoffman, Beyond the Tvelwe-Factor App, O'Reilly Media, Sebastopol, CA, 2016.

[22] N. Kratzke, P. C. Quint, Understanding cloud-native applications after 10 years of cloud computing - a systematic mapping study, Journal of Systems and Software (2017) 1–16.

[23] M. Richards, Microservices vs. Service-Oriented Architecture, O'Reilly Media, Sebastopol, CA, 2015.

[24] M. Garriga, Towards a taxonomy of microservices architectures, Software Engineering Research Group (2016) 1–3.

[25] T. Faganel, Ogrodje za razvoj mikrostoritev v javi in njihovo skaliranje v oblaku, diplomsko delo (September 2015).

[26] Kumuluzee has the fastest start-up and smallest memory footprint, `https://blog.kumuluz.com/product/2017/10/29/kumuluzee-fastest-start-up-smallest-memory-footprint`, dostopano: 2018-11-20.

[27] Glassfish metro implementation, `https://javaee.github.io/metro/`, dostopano: 2017-12-02.

[28] N. Balani, R. Hathi, Apache CXF Web Service Development: Develope and deploy SOAP and RESTful web services, Packt Publishing, 32 Lincoln Road, 2009.

[29] The java api for xml-based web services (jax-ws) 2.2, https://jcp.org/en/jsr/detail?id=224, dostopano: 2017-12-02.

[30] F. Rademacher, S. Sachweh, A. Zündorf, Towards a uml profile for domain-driven design of microservice architectures, Software Engineering Research Group (2016) 9.

[31] Apache cxf integration, https://docs.jboss.org/author/display/JBWS/Apache+CXF+integration, dostopano: 2017-12-02.

[32] G. Poročnik, Master thesis repository, https://github.com/gpor89/masters-thesis, dostopano: 2018-09-02.

[33] J.-Y. L. Boudec, Performance Evaluation of Computer and Communication Systems, EFPL Press, Boca Raton, FL, 2011.

[34] J. Müller, M. Lorenz, F. Geller, A. Zeier, H. Plattner, Assessment of communication protocols in the epc network - replacing textual soap and xml with binary google protocol buffers encoding, in: IEEE 17Th International Conference on Industrial Engineering and Engineering Management, Institute for System IT engineering, Germany, 2010, pp. 404–409.

[35] F. Belqasmi, J. Singh, S. Y. B. Melhem, R. H. Glitho, Soap-based vs. restful web services: A case study for multimedia conferencing, IEEE Internet Computing (2012) 54–63.

[36] N. Nurseitov, M. Paulson, R. Reynolds, C. Izurieta, Comparison of json and xml data interchange formats: A case study, in: CAINE, 2009, pp. 157–162.

[37] J. Spillner, Y. Bogado, W. Benitez, F. Lopez-Pires, Co-transformation to cloud-native applications, in: Proceedings of the 8th International Conference on Cloud Computing and Service Science 19-21 March 2018, CLOSER, SciTePress, Madeira, 2018, pp. 1–12.

[38] M. Novak, S. N. Shirazi, A. Hudic, T. Hecht, M. Tauber, D. Hutchison, S. Maksuti, A. Bicaku, Towards resilience metrics for future cloud applications (01 2016).

[39] J. Tulach, Practical API Design: Confessions of a Java Framework Architect, Apress, Berkely, CA, USA, 2008.

[40] W. W. A. Rahman, F. Meziane, Challenges to describe qos requirements for web services quality prediction to support web services interoperability in electronic commerce (2018) 2–4.

[41] Q. Duan, Cloud service performance evaluation: status, challenges, and opportunities – a survey from the system modeling perspective, Digital Communications and Networks (2) (2017) 101 – 111.

[42] J. Laprie, Dependable computing and fault tolerance : Concepts and terminology, in: Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'., 1995, pp. 2–.

[43] E. Wolff, Microservices: Flexible Software Architecture, Addison-Wesley Professional, Boston, 2016.

[44] P. Kumari, P. Kaur, A survey of fault tolerance in cloud computing, Journal of King Saud University - Computer and Information Sciences.

[45] P. Patra, H. Singh, G. Singh, Fault tolerance techniques and comparative implementation in cloud computing 64 (2013) 37–41.

[46] A. Bala, I. Chana, Fault tolerance-challenges , techniques and implementation in cloud computing, in: International Journal of Computer Science Issues, IJCSI, 2012.

[47] F. Montesi, J. Weber, Circuit breakers, discovery, and API gateways in microservices, CoRR (2016) 3–4.

[48] M. L. Abbott, M. T. Fisher, The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise, Addison-Wesley Professional, Upper Sadle River, NJ, USA, 2015.

[49] P. K. Potti, S. Ahuja, K. Umapathy, Z. Prodanoff, Comparing performance of web service interaction styles: Soap vs. rest, in: 2012 Proceedings of the Conference on Information Systems Applied Research, University of North Florida, 2012, pp. 1–24.

[50] Awesome grpc - tools, `https://github.com/grpc-ecosystem/awesome-grpc`, note = Dostopano: 2018-10-20.

[51] A. Kobusińska, C.-H. Hsu, Towards increasing reliability of clouds environments with restful web services, Future Generation Computer Systems 87 (2018) 502 – 513.

[52] A. Marinos, A. R. Reza, S. Moschoyiannis, P. Krause, Retro: A consistent and recoverable restful transaction model (07 2009). `doi:10.1109/ICWS.2009.99`.

[53] Our experience designing and building grpc services, `https://blog.bugsnag.com/using-grpc-in-production/`, dostopano: 2018-08-16.

[54] Openapi tools, `https://openapi.tools/`, dostopano: 2018-10-01.

[55] S. Murugan, Byzantine fault tolerance model for soap faults, in: International Journal of Computer Science Issues, IJCSI, 2012.

[56] Pattern: Circuit breaker, `https://microservices.io/patterns/reliability/circuit-breaker.html`, dostopano: 2018-08-16.

[57] M. B. Juric, SOA Approach to Integration: XML, Web Services, ESB, and BPEL in Real-world SOA Projects, Packt Pub., 32 Lincoln Road, 2007.

[58] Z. Aljazzaf, Bootstrapping quality of web services, Journal of King Saud University - Computer and Information Sciences 27 (2015) 323 – 333.

[59] P. El-Kafrawy, E. Elabd, H. Fathi, A trustworthy reputation approach for web service discovery, Procedia Computer Science 65 (2015) 572 – 581.

[60] O. Tomanek, P. Mulinka, L. Kencl, Multidimensional cloud latency monitoring and evaluation, Computer Networks (2016) 104 – 120.

[61] Simple object access protocol (soap) w3 standard, `https://www.w3.org/TR/2000/NOTE-SOAP-20000508/#_Toc478383528`, dostopano: 2018-10-01.