Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Primož Hrovat

Mikrostoritve v decentraliziranem okolju

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: prof. dr. Matjaž Branko Jurič

Ljubljana, 2018

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Besedilo teme diplomskega dela študent prepiše iz študijskega informacijskega sistema, kamor ga je vnesel mentor. V nekaj stavkih bo opisal, kaj pričakuje od kandidatovega diplomskega dela. Kaj so cilji, kakšne metode uporabiti, morda bo zapisal tudi ključno literaturo.

TODO: dopolni z opisom, ko bo na voljo

Zahvaljujem se prof. Matjažu B. Juriču za podporo in usmeritve pri pripravi diplomskega dela. Posebna zahvala gre mojim staršem, bratom, sestram, starim staršem in prijateljem, ki mi vedno stojijo ob strani in me pri mojem delu podpirajo.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
API	application programming interface	aplikacijski programski vmesnik
PoW	Proof-of-Work	Dokaz o opravljenem delu
DBMS	Database Management System	Sistem za upravljanje podatkovne baze
IPFS	Interplanetary File System	Medplanetarni datotečni sistem
REST	Representational state transfer	-
GKE	Google Kontainer Engine	-
AWS	Amazon Web Services	-
DOS	Denial of Service	Zavrnitev storitve
CNCF	Cloud Native Computing Foundation	-

Povzetek

Naslov: Mikrostoritve v decentraliziranem okolju

Avtor: Primož Hrovat

Mikrostoritve danes počasi a vztrajno prevzemajo primat v svetu razvoja programske opreme. Nadomeščajo tradicionalne aplikacije, za čim učinkovitejše izvajanje pa obstaja vrsto orodij in tehnik, ki procese skaliranja, distribuiranja in odkrivanja rešujejo na učinkovite načine. Aplikacije, grajene v arhitekturi mikrostoritev se večinoma izvajajo v velikih računskih centrih - računalniških oblakih. Vsi ti mehanizmi omogočajo, da smo zmožni odgovoriti na tisoče hkratnih zahtevkov, pri tem pa ohranjamo dobre performančne lastnosti. Problemi nastopijo ob izpadu enega ali več računskih centrov, oziroma pri prenosu aplikacije iz enega v drug oblačni sistem. V diplomski nalogi sem raziskal tehnologijo veriženja podatkovnih blokov, ki predstavlja korak v smeri decentralizirane shrambe podatkov. Podobno stopnjo decentralizacije želimo doseči na ravni aplikacijske logike, kjer pa se pojavi veliko različnih problemov. Osredotočil sem se predvsem na decentralizirano odkrivanje storitev, ter razvil prvi prototip aplikacije ter razširitve za ogrodje KumuluzEE, ki demonstrirata decentraliziran način izvajanja.

Ključne besede: decentralizacija, distribuirane storitve, tehnologija veriženja podatkovnih blokov.

Abstract

Title: Microservices in decentralized environment

Author: Primož Hrovat

Microservices today are one of the leading design principles, when it comes to building modern applications. Monoliths are being replaced with this modern architecural style and multiple tools and techniques are being develop to support it. Applications are deployed to a remote computing center, often called simply as cloud. These principles allow us to effectively respond to thousands requests per second. The problems arrise when those cloud providers experience massive breakdowns or we wish to commute to different cloud provider. My thesis discusses blockchain technology that is currently one of the hot research topics. Blockchain provides decentralized and immutable data storage. Our vision is to enable that kind of decentralization to bussines logic (API). To achive that goal, we must solve multiple problems, one of which is decentralized service discovery. I have proposed a conceptual solution to service discovery in a decentralized environment and implemented first prototype, that enables decentralized execution of applications.

Keywords: decentralization, distributed services, blockchain.

Poglavje 1

Uvod

Poslovne storitve se danes selijo v oblak. S pojavom arhitekture mikrostoritev in podporne tehnologije, kot so vsebniki in okolja za orkestracijo vsebnikov, so, nekdaj ogromni, kosi programske opreme pričeli razpadati na manjše, logično ločene sestavne dele. Razmeroma majhne in neodvisne aplikacije, specializirane za opravljanje točno določenih nalog, omogočajo hiter vzpostavitveni čas in puščajo majhen odtis na porabi strojne opreme. Neodvisnost teh aplikacij nam omogoča tudi skaliranje teh posamenznih delov celotne storitve, ko je to potrebno. Vsebniki so razmeroma stara tehnologija, ki je s pojavom okolja Docker doživela pravi razcvet. Gre za „lahko“ obliko virtualizacije, virtualizacija tu poteka na nivoju procesov in ne operacijskega sistema. Pravi potencial vsebnikov izkoristimo šele z uporabo orkestratorjev, kot so Kubernetes, Amazon ECS, Google Container Engine (GKE), Docker Swarm, Azure Container Service in podobni. Ta orodja omogočajo monitoring, zaganjanje, zaustavljanje in preverjanje storitev, skladno s podanimi zahtevami. Storitve se izvajajo distribuirano (porazdeljeno) in replicirano, lahko tudi na fizično ločenih sistemih, kar zagotavlja visoko stopnjo odzivnosti in dosegljivosti. Stopnja dosegljivosti se danes meri na peti ali šesti decimalki („number of nines“).

S pojavom Bitcoina se je začel razvoj tehnologije, ki v decentraliziranem okolju omogoča varno in nespremenljivo hrambo podatkov. Veriženje

podatkovnih blokov je v zadnjih letih s pojavom različnih organizacij kot so Hyperledger in Ethereum doživelo razcvet. Nova tehnologija omogoča shranjevanje podatkov, repliciranih na vseh sodelujočih entitetah v omrežju, obenem pa zagotavlja njihovo nespremenljivost.

Tu se pojavlja vprašanje. Je možno tudi poslovno logiko prestaviti v decentralizirano okolje na način, da bo sodelujoči v omrežju ob klicu želene storitve vedno dobil odgovor, izvedla ga bo katerakoli entiteta, obenem pa zagotoviti pravilnost izvajana? Z uresničitvijo tega cilja bi miselnost decentraliziranih podatkov prestavili nivo višje, na nivo poslovne logike. Odzivnost in dosegljivost storitve bi s številom sodelujočih v omrežju dosegla 100%, prav tako bi bilo praktično nemogoče izvesti napade DOS.

V diplomski nalogi sem se osredotočil na reševanje problema odkrivanja storitev v decentraliziranem okolju.

Poglavje 2

Arhitekturni koncepti mikrostoritev

Razvoj monolitov je enostaven in danes dobro podprt v vseh danes prisotnih razvojnih okoljih. Prenos in namestitev teh storitev na strežniške sisteme je enostaven postopek, rešitev v obliki izvršljivih datotek ali s kopiranjem direktorijske strukture prenesemo v produkcijsko izvajalno okolje. Če želimo tako storitev skalirati, kot vstopno točko v naše zaledne sisteme nastavimo izenačevalnika obremenitev (ang. load balancer), ki poskrbi za enakomerno porazdeljevanje dela med posameznimi instancami storitve.

Slabosti te arhitekture se pojavijo, ko storitev postane kompleksnejša. Podaljša se zagonski čas (start up time), ob preobremenitvi le enega dela storitve je potrebno instancirati celotno aplikacijo. Prav tako je otežen proces sprotne dostave (Continouos Delivery) in sprotne integracije (Continouos Integration), za posodobitev enega dela sistema je namreč potrebno celotno storitev zaustaviti, namestiti novo različico in jo ponovno zagnati [14].

2.1 Arhitektura mikrostoritev

Gradnja aplikacij je do nedavnega potekala na način, da so razvijalci vse odvisnosti in logiko, potrebno za delovanje, zložili skupaj v veliko tvorbo -

monolit. Tak način gradnje ima svoje prednosti in slabosti. Prednosti na eni strani predstavljajo enostavna zgradba aplikacije, ker je vsa izvorna koda zbrana na enem mestu. Orodja za razvoj so prilagojena takemu načinu dela in razvijalcu ponujajo širok nabor funkcij, ki podpirajo celotno življenjko pot aplikacije, od razvoja, testiranja, namestitve v testna in produkcijska okolja ter vzdrževanje. Velikost projekta je na drugi strani ena od slabosti monolitov, majhne spremembe enega sestavnega dela potrebujejo ponovno namestitev celotne aplikacije. Skaliranje poteka vertikalno, le preko ustvarjanja novih instanc celotne aplikacije. Monolitna zasnova aplikacije razvijalce in vzdrževalce zaveže k dolgoročni uporabi tehnologij, ki so bile uporabljene na začetku. Kasnejše uveljavljanje novih tehnologij oziroma nadomeščanje obstoječih je časovno in finančno potratno, saj je potrebno celotno logiko aplikacije prepisovati [14].

Arhitektura mikrostoritev se problema gradnje aplikacij loti drugače. Celotno aplikacijo se logično razbije na posamezne sestavne dele, ki se izvajajo samostojno in se med seboj povezujejo preko lahkih komunikacijskih protokolov. Vsak sestavni del aplikacije je samostojen v smislu skaliranja, življenjske dobe posamezne instance, razvoja in podpornih tehnologij. Tak način gradnje sledi tistemu, ki ga v fizičnem svetu poznamo že stoletja, to je sestavljanje vnaprej pripravljenih delov v celoto. Lep primer tega je avtomobilska industrija. Na tekočem traku se na tisoče sestavnih delov različnih proizvajalcev zloži v polno funkcionalno enoto - avtomobil. Podobno želimo doseči pri razvoju programske opreme, kar bi pohitrilo in pocenilo razvoj novih aplikacij, krepko pa bi omejili tudi nepotrebno podvajanje izvorne kode [13, 10].

V povezavi z mikrostoritvami je tesno povezana tudi arhitektura „cloud-native“. Gre za novo paradigmo razvoja programske opreme, ki sledi načinom izdelave stvari v fizičnem svetu. Z namenom uveljavljanja in razvoja paradigme je bila ustanovljena „Cloud Native Computing Foundation“, ki bdi nad razvojem standardov in novih smernic. Oblačne sisteme sestavljajo [19]:

- Aplikacije in procesi, ki se izvajajo znotraj vsebnikov. Vsebniki so neodvisne in izoloirane izvajalne enote.

- Storitve, ki se dinamično upravljajo in nadzorujejo preko centralnega orkestracijskega procesa.

- Mikrostoritve, ki sos sestavni deli sistema in so med seboj šibko sklopljene.

Računalništvo v oblaku je uporaba in dostop do računskih virov (aplikacije, podatkovna skladišča, procesorski čas...) na zahtevo preko interneta. Plačilo se izvede skladno s količino porabljenih virov. Viri so elastični in zmožni hitrega in učinkovitega skaliranja ter prilagajanja trenutnim zahtevam. V grobem med seboj ločimo tri glavne komponente oblačnih sistemov: SaaS, PaaS in IaaS. SaaS (Software as a Service) - programska oprema kot storitev, je oblačna aplikacija, ki se izvaja na oddaljenih računalnikih, do katerih uporabnik storitve dostopa preko interneta. Prednosti teh aplikacij so dostopnost od koderkoli, za nemoteno delovanje aplikacije je odgovoren ponudnik. Storitev je zmožna dinamičnega skaliranja, da zadovolji trenutnim potrebam. PaaS (Platform as a Service) - platforma kot storitev, uporabnikom ponuja programsko platformo, brez stroškov nakupa in kompleksnosti upravljanja podporne strojne in programske opreme. IaaS (Infrastructure as a Service) - infrastruktura kot storitev, ponuja dostop do računskih virov (strežniki, omrežna oprema, shramba...). Uporabnikom ni potrebno investirati v lastno strojno opremo, podobno kot obe sestrski storitvi (SaaS in PaaS), je tudi ta zmožna samodejnega skaliranja [7].

2.2 Vzorci

2.2.1 Metrike

Posamezna storitev se lahko izvaja na različnih (fizičnih) lokacijah, na različni strojni opremi in v več instancah. Pojavi se potreba po spremljanju storitve in njenega obnašanja, kje in zakaj prihaja do izpadov oziroma morebitnih dolgih odzivnih časih. Vse želene metrike želimo shraniti na enem centralnem

mestu, jih po možnosti agregirati, in tako omogočiti enostavno odkrivanje napak in njihovo reševanje. V osnovi poznamo dva modela zbiranja metrik:

- mikrostoritev sama pošilja metrike centralni zbirki (push)
- centralna zbirka zahteva metrike od storitve (pull)

Zbiranje metrik ponuja dober vpogled v obnašanje posameznih storitev, s sabo pa prinese dodatno potrebno infrastrukturo in dodatne težave pri implementaciji [11].

Ena izmed bolj znanih storitev za zbiranje aplikacijsih metrik je Prometheus, odprtokodna zbirka orodij za zbiranje in agregiranje metrik ter obveščanje. Projekt je bil, takoj za orkestracijskim okoljem Kubernetes, pridružen fundaciji CNCF. Prometheus zahteva metrike od storitve (način pull), jih agregira in proži morebitna opozorila. Skupaj z ogrodjem Grafana omogoča celovit grafičen vpogled v zbrane metrike [16].

2.2.2 Preverjanje odzivnosti (Health Check)

Pogosto se zgodi, da se storitev še vedno odziva na posamezne zahteve, vendar so te neuspešne. V takšnih primerih je pričakovano obnašanje sistema, da nedelujočo storitev iz omrežja označi kot nedosegljivo in jo nadomesti z novo. Tu v igro vstopi koncept preverjanja odzivnosti storitev (ang. Health check), ki za vsako storitev pričakuje izpostavljeno dostopno točko, preko katere lahko sistem pridobi informacije o sposobnosti storitve, da odgovori na zahtevke. Vsaka storitev je odgovorna za preverjanje svojih zunanjih in notranjih virov in generiranja poročila o trenutnem statusu posamezne komponente. Tipični testi so preverjanje zmožnosti povezave na podatkovno bazo, dosegljivost ostalih odvisnih storitev, povezljivost z vrstami... Poleg zunanjih odvisnosti se preveri tudi stanje gostitelja: razpoložljiv prostor na disku, zasedenost CPE ipd. Nadzorna storitev periodično proži zahteve na vse registrirane storitve in preverja njihove odzive. Tipičen ukrep ob negativnem odzivu je zaustavitev in ponovni zagon nedelujoče storitve [12].

2.2.3 Odkrivanje storitev (Service discovery)

Ko želimo od zunanje entitete pridobiti podatke oziroma prožiti določeno akcijo, potrebujemo njen omrežni naslov (kombinacija naslova IP in številke vrat). Pri klasičnih aplikacijah, ki se izvajajo na fizični napravi, so naslovi relativno statični. Zelo malo je tudi klicev med posameznimi aplikacijami, saj se storitve med seboj kličejo preko programskih klicev. V arhitekturi mikrostoritev je omrežnih klicev bistveno več, ker je celotna programska logika aplikacije sestavljena iz več samostojnih storitev. Njihovo število se dinamično spreminja, skladno s tem tudi omrežni naslovi posameznih instanc. Posledično je potrebno za odjemalca vpeljati nov mehanizem, ki je zmožen dinamično odkriti naslove, na katerih se želena storitev nahaja.

Tipično za oblačne arhitekture je, da se odkrivanje storitev realizira s pomočjo centralnega registra. Storitve se ob pričetku izvajanja registrirajo. Odjemalska aplikacija ob potrebi po dostopu do zunanje storitve na register naslovi poizvedbo za lokacijo instanc želene mikrostoritve. Zakasnitev pri omrežnih klicih je bistveno večja kot pri programskih klicih znotraj aplikacije, zato želimo visoko učinkovitost poizvedb [15, 26].

2.3 Vsebniki in orkestracija

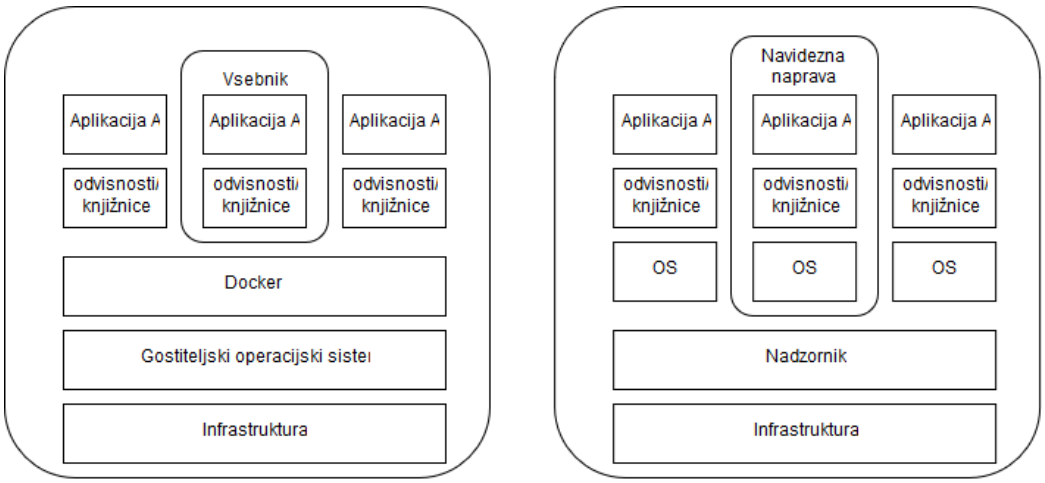
Vsebniki danes nadomeščajo virtualne naprave, predvsem na račun bolj učinkovite uporabe sistemskih virov in odpravljanjem nepotrebnih podvajanj programske opreme.

2.3.1 Tehnologija vsebnikov

Vsebnik je samostojen, izvršljiv skupek programske opreme, ki vsebuje vse potrebne odvisnosti, orodja, knjižnice in nastavitve, potrebne za izvajanje. Izvajanje kode znotraj vsebnika bo v vsakem gostiteljskem ogrodju (Windows, Linux, MacOS) enako. Notranji deli vsebnika so izolirani od okolice, kar odpravlja morebitne konflikte zaradi različnih verzij potrebnih odvisno-

sti. V primerjavi z navideznimi napravami, vsebniki virtualizirajo operacijski sistem in ne podporne strojne opreme. Z gostiteljskim sistemom in ostalimi vsebniki delijo jedro operacijskega sistema, vsak izmed njih v svojem naslovnem prostoru.

Shema 2.3.1 prikazuje bistveno razliko med vsebniško tehnologijo in tehnologijo virtualizacije [23].



Slika 2.1: Primerjava vsebnikov in navideznih naprav

2.3.2 Orkestracijska orodja

Proces prenosa in namestitve vsebnikov v izvajalno okolje je moč avtomatizirati. Proces postaja pomembnejši z rastjo števila vsebnikov in gostiteljskih sistemov. Ta tip avtomatizacije imenujemo orkestracija, ponuja pa nam vrsto funckionalnosti [1]:

- upravljanje z gostiteljskim sistemom
- instanciranje vsebnikov
- upravljanje z vsebniki
- povezovanje vsebnikov preko vmesnikov

- izpostavljanje storitev zunanjim napravam
- skaliranje gruče vsebnikov

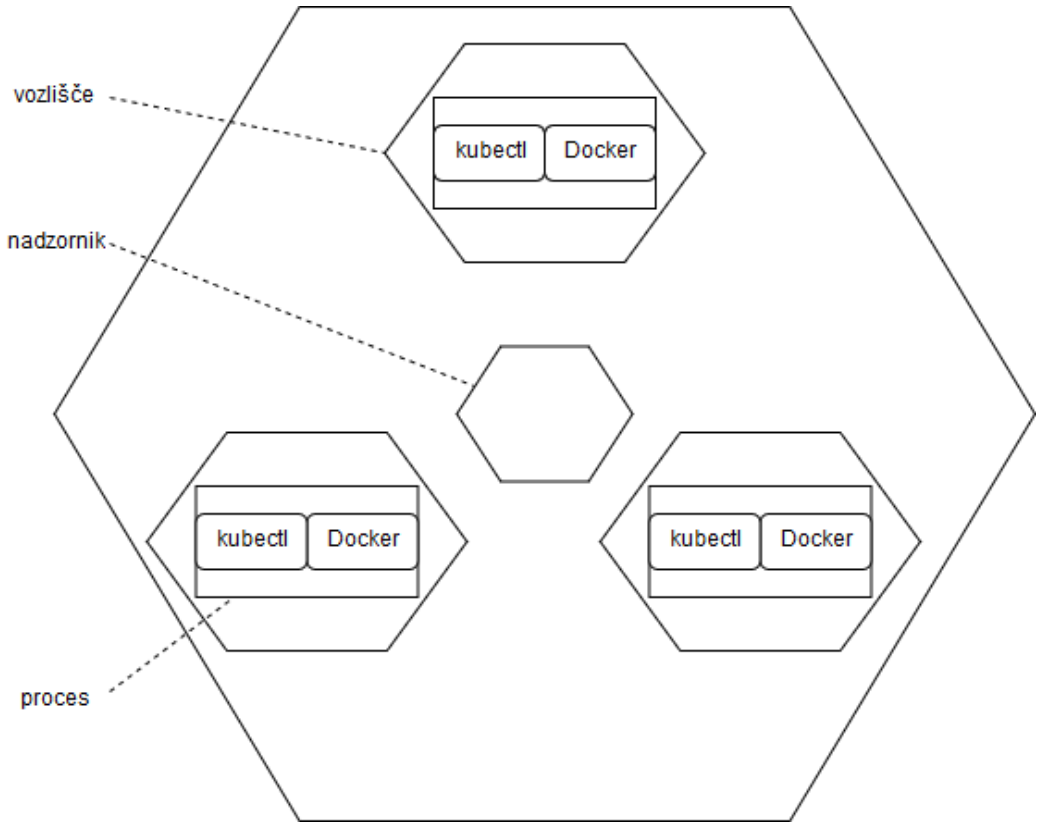
S pojavom in razširitvijo vsebniške tehnologije so se pojavila tudi številna orodja za orkestracijo. Med njimi najbolj poznana sta Docker Swarm in Kubernetes. Vsako izmed teh orodij orkestracijo rešuje na svoj način, Kubernetes pa je trenutno eno izmed najbolj razširjenih orodij. Njegove glavne funckionalnosti so:

- avtomatizirano nameščanje in repliciranje vsebnikov
- skaliranje
- porazdeljevanje dela (load balancing)
- nameščanje posodobitev
- odpornost na napake in odpovedi vsebnikov z avtomatskimi ponovnimi zagoni
- kontrolirano izpostavljanje notranjega omrežja zunanjim storitvam

Glavni sestavni deli Kubernetesa [1]:

- gruča: zbirka enega ali več strežnikov (ang. Node), ki svoje vire ponujajo nadzornemu procesu
- strok (pod): skupina vsebnikov in pripradajočih shramb, ki so dodeljene enemu gostitelju. Predstavljajo osnovno celico Kubernetesa, znotraj stroka si procesi delijo lokalni omrežni naslovni prostor.
- labele: dodeljene oznake posameznim entitetam (vsebnikom), ki omogočajo upravljanje z njimi v skupini
- storitve: skrbijo za osnovno dodeljevanje dela in izpostavljajo stroke zunanjemu svetu.

Slika 2.3.2 prikazuje osnovno shemo Kubernetes gruče.



Slika 2.2: Kubernetes gruča [21]

Poglavje 3

Tehnologija veriženja podatkovnih blokov

Veriženje podatkovnih blokov je peer-to-peer porazdeljena podatkovna shramba, dosežena s soglasjem, sistemom „pametnih“ pogodb ter drugih pomožnih tehnologij [4]. Osrednja komponenta sistema je glavna knjiga (ang. ledger), ki beleži vse akcije (transakcije), izvedene na omrežju [6]. Entiteta, ki transakcijo izvede, jo podpiše s svojim privatnim ključem. Skupek transakcij tvori podatkovni blok, bloki pa se med seboj povezujejo v podatkovno verigo. Posamezne člene verige med seboj povezuje zgoščevalna funkcija, na vhod katere postavimo zgoščeno vrednost trenutnega in prejšnjega bloka. Podatkovno verigo je moč vedno le podaljševati, trenutno veljavno in resnično stanje omrežja je trenutno najdaljša serija blokov. Celotna veriga blokov je replicirana na vsaki izmed sodelujočih entiteti.

Kombinacija teh pristopov omogoča, da nobena izmed sodelujočih entitet ne more spreminjati že zapisanih blokov. Napad na omrežje je možen le s pridobitvijo več kot polovice vseh sodelujočih entitet v omrežju, ki bi potrjevale resničnost ponarejenih transakcij in sčasoma sestavile daljšo podatkovno verigo, ki bi obveljala kot trenutna resnica. S temi mehanizmi se zagotovi veljavnost in nespremenljivost podatkov v okolju, ki mu apriori ni potrebno zaupati. Ni več potrebe po zunanji, zaupanja vredni, entiteti.

Za interakcijo z glavno knjigo in zapisovanje novih informacij, omrežje uporablja t.i. „pametne pogodbe“. To je del programske kode, ki se lahko odziva na dogodke v omrežju, izvede zapisano poslovno logiko ter ustvarja nove transakcije [6].

3.1 Razlaga osnovnih konceptov

3.1.1 Porazdeljena glavna knjiga

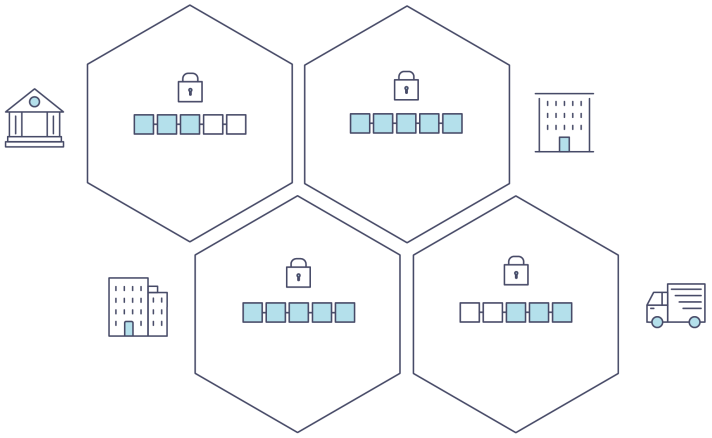
V osrčju tehnologije podatkovnih blokov je porazdljena glavna knjiga, ki vsebuje zapise o vseh transakcijah, izvedenih na omrežju. Glavna knjiga je replicirana na vseh entitetah v omrežju, ki med sabo sodelujejo in skrbijo za vzdrževanje. Informacije so v glavno knjigo le dodajajo z uporabo kriptografskih tehnik, ki zagotavljajo, da je vsaka zapisana transakcija trajna in nespremenljiva. To omogoča enostavno preverjanje izvora informacije, od tu tudi drugo poimenovanje za tehnologijo podatkovnih blokov kot sistem dokazovanja [6]. Na sliki 3.1.1 je prikazana osnovna shema omrežja.

3.1.2 Pametne pogodbe

Pametne pogodbe omogočajo kontroliran dostop in interakcijo z glavno knjigo. So osnovni mehanizem za enkapsulacijo informacij in njihovo preprosto vzdrževanje preko omrežja. Poleg tega omogočajo tudi določeno stopnjo avtomatizacije, kot izvrševanje transakcij brez človeškega posredovanja. V primerjavi s tradicionalnimi pogodbami, nam pametne pogodbe zagotavljajo višjo stopnjo varnosti in zmanjševanje dodatnih stroškov, ki so povezani z njihovim izvajanjem [24]. Na sliki 3.1.2 je prikazan osnovni potek interakcije pametne pogodbe z glavno knjigo.

3.1.3 Soglasje

Proces sinhronizacije glavne knjige v omrežju je imenovan soglasje. Zagotavlja, da se glavna knjiga posodobi le takrat, ko so transakcije in podatkovni



Slika 3.1: Porazdeljeno omrežje. Vsak uporabnik hrani kopijo glavne knjige [6]

.

bloki potrjeni s strani zaupanja vrednih udeležencev omrežja. Glavna knjiga se posodobi tako, da vsi udeleženci omrežja, izvedejo enake (zapisane) transakcije v istem vrstnem redu. Slika 3.1.3 grafično prikazuje primer doseženega soglasja.

3.1.4 Izvajalna okolja glede na zaupanje med udeleženci

V grobem lahko, glede na stopnjo zaupanja, ločimo dva tipa izvajalnih okolij, ki ga udeleženci delijo med seboj. Imamo omrežje, kjer so udeleženci vnaprej znani, identificirani s strani tretje osebe, ki ji zaupajo vsi sodelujoči. Interakcije med njimi so varne v smislu prevzemanja odgovornosti. Morebitna škodoželjnost udeleženca je enostavno kaznovana zaradi fizično overjenih oseb (pravnih ali fizičnih).

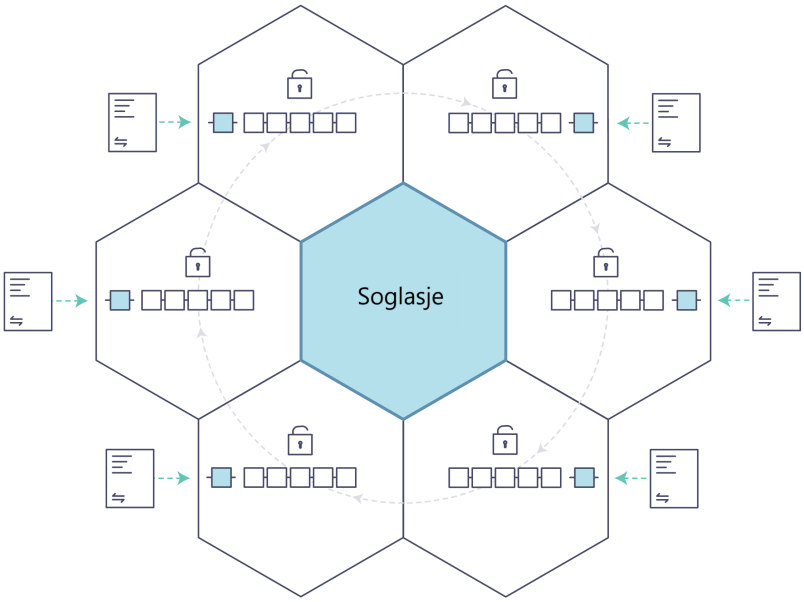
V javnih okoljih teh ugodnosti ne uživamo. V omrežju lahko sodeluje kdorkoli in to povsem anonimno, med udeleženci tako privzeto velja načelo

Slika 3.2: Interakcija pametne pogodbe z glavno knjigo [6].

nezaupanja. Zaupa se le stanju celotne podatkovne verige. Tipično so za potrjevanje blokov in novih transakcij uporabljene „kripto valute“, pridobljene s ti. postopkom rudarjenja. Ta omrežja večinoma temeljijo na Byzantine Fault-Tolerance (BFT) [6, 25].

3.1.5 Zasebnost in zaupnost

Javne podatkovne verige so replicirane na vseh sodelujočih entitetah, kar prinaša transparentnost, obenem pa poslovnim subjektom onemogoča učinkovito sklepanje dodatnih ugodnosti, aneksov ipd. s poslovnimi partnerji. Ena od



Slika 3.3: Doseženo soglasje v omrežju [6].

možnih rešitev problema je enkripcija podatkov, ki pa v tem primeru odpove. Vsak izmed sodelujočih ima dostop do celotne glavne knjige, kar omogoča enostavne napade s silo. V nadaljanju predstavljeno omrežje Fabric tu vpeljuje koncept kanalov. Ti predstavljajo logično grupiranje posameznih entitet in omejujejo dostop do pametnih pogodb in glavne knjige na posameznem kanalu [6].

3.2 Ethereum

Ethereum je decentralizirana platforma, ki izvaja pametne pogodbe (smart contracts) - aplikacije, ki se izvajajo natanko tako, kot so bile zapisane. Platforma je osnovana na verigi podatkovnih blokov, ki omogoča reprezentacijo in prenos vrednosti. Lahko si ga predstavljamo kot svetovni računalnik,

izvajanje programske kode pa poteka na vseh sodelujočih računalnikih. Pametne pogodbe ponujajo možnost interakcije s podatkovno verigo, določeni deli kode pa se izvajajo le pod točno določenimi pogoji [2].

Stanje v Ethereum omrežju določajo objekti, znani kot uporabniški računi (accounts). Vsak račun sestavlja 20 bajtov dolg naslov, prenos sredstev in informacij med računi pa predstavlja spremembo trenutnega stanja. Uporabniške račune sestavljajo štiri polja [2]:

- števec (nonce), ki preprečuje podvajanje transakcij
- trenutno stanje Ethra (ether balance) trenutna količina ethra v lasti računa
- pogodbena koda (contract code) opcijska
- shramba (storage) privzeto prazno

Ether je interno plačilno sredstvo v omrežju. Uporablja se kot nadomestilo za izvrševanje transakcij. Ethereum pozna dva tipa uporabniških entitet: zunanje (externally owned), določenih s privatnimi ključi in pogodbene (contract accounts), določenih s kodo. Zunanji računi ne obvladujejo kode, z ostalimi entitetami v omrežju pa lahko komunicirajo preko digitalno podpisanih transakcij. Pametne pogodbe so entitete, ki se v omrežju odzivajo na vnaprej določena sporočila: izvedejo del logike, berejo in pišejo v glavno knjigo oziroma pošljejo novo sporočilo v omrežje [2].

3.2.1 Komunikacija med entitetami

V omrežju obstajata dva načina komunikacije: sporočila (messages) in transakcije (transactions). Transakcije so podpisani podatkovni bloki, ki jih ustvarijo zunanji uporabniški računi. Sestavni deli transakcije so:

- prejemnik
- podpis pošiljatelja

- količina prenesenega ethra
- podatki (opcijsko)
- STARTGAS največje dovoljeno število izvedenih računskih operacij
- GASPRICE cena posamezne računske operacije

Sporočila so namenjena interni komunikaciji med pametnimi pogodbami. So le navidezni objekti, obstajajo izključno v izvajalnem okolju. Sestavlja jih [2]:

- pošiljatelj
- prejemnik
- količina prenesenega ethra
- STARTGAS

3.2.2 Navidezni stroj Ethereum

Navidezni stroj je glavna abstrakcija celotnega omrežja. Je izvajalno okolje za pametne pogodbe v Ethereum omrežju in služi kot „peskovnik“ za izvajanje kode. Celotni navidezni stroj lahko predstavimo s terko **(stanje blokov, transakcija, sporočilo, koda, spomin, sklad, programski števec, plin)**. *Stanje blokov* je predstavitev vseh računov s trenutnim stanjem ethra in shrambe. Vsaka izvedena operacija zmanjša vrednost preostale količine plina, glede na uteženost posamezne operacije. Transakcija se zaključi ob izvedbi zadnje operacije v programu oziroma s prekinitvijo, ko porabljena količina plina preseže največjo dovoljeno [2].

Potrjevanje in kreiranje blokov

Vsak blok v Ethereum verigi vsebuje kopijo vseh transakcij in zadnjega stanja omrežja. Poleg tega sta v bloku zapisani tudi zaporedna številka bloka in zahtevnost. Postopek validacije bloka poteka sledeče:

1. Preveri, če predhodni blok obstaja in je veljaven
2. Preveri časovni žig bloka - večji od prejšnejga bloka, vendar ne več kot 15 minut v prihodnosti
3. Preveri številko bloka, zahtevnost, izvor transakcije, izvor „strica“ in omejitev količine plina
4. Preveri veljavnost „Proof of Work“
5. Naj bo S[0] stanje na koncu predhodnega bloka
6. Naj bo TX seznam transakcij v bloku. Za vsak $\{i \mid 0, 1, \ldots, n-1\}$ je naslednje stanje $S[i+1] = APPLY(S[i], TX[i])$. V primeru napake ali presežene omejitve količine plina na blok (GASLIMIT), vrni napako.
7. Naj $S_FINAL = S[n]$. Nagrada za najden blok se izplača samo najditelju.
8. Preveri, da je vrhnje vozlišče Merklovega drevesa stanja S_FINAL enaka končnemu stanju v bloku. V tem primeru je blok veljaven.

Koda je izvedena s strani vseh sodelujočih entitet v omrežju [2].

3.3 Hyperledger

Hyperledger je družina odprtokodnih projektov, namenjenih razvoju tehnologije veriženja podatkovnih blokov. Projekt deluje pod okriljem organizacije The Linux Foundation, v sodelovanju s skupnostjo. Med prvimi in najbolj znamimi izmed Hyperledger projektov je Hyperledger Fabric, prvotno razvit v podjetju IBM in Digital Asset. Pod okrilje projekta Hyperledger spadajo še Sawtooth, Iroha, Burrow ter Indy. Vsak izmed projektov na svoj način rešuje izzive s področja podatkovnih verig ali pa naslavlja ozko problemsko domeno. Kot primer: projekt Indy se ukvarja s problematiko spletne identitete uporabnika [4]. Trenutno najbolj znana in razširjena platforma je Fabric,

trenutno v različici 1.1. Od ostalih podobnih projektov se loči predvsem v konceptu privatnih omrežjih, pri katerih je sodelovanje omejeno s sistemom dovoljenj. Omogoča modularno izbiro načina soglasja in ga je moč prilagajati zahtevam poslovnih uporabnikov [5].

3.3.1 Fabric

Hyperledger Fabric je v sami zasnovi namenjen poslovni uporabi. Omogoča modularno in prilagodljivo arhitekturo, podobno kot ostale implementacije tehnologije veriženja blokov pozna tudi pametne pogodbe, tu imenovane „chain code“. Pametne pogodbe se tu izvajajo znotraj vsebnikov Docker in omogočajo implementacijo v poljubnem splošnonamenskem programskem jeziku. Drugačen je tudi postopek izvedbe transakcije.

Celotno omrežje je zasnovano na predpostavki (delnega) zaupanja med sodelujočimi entitetami, za razliko od javnih omrežij. Enostavna je menjava implementacije protokola za doseganje soglasja, bodisi na osnovi reševanja napak ob odpovedi (Crash Fault Tolerant – CFT) ali bizantinske odpornosti na napake (Byzantine Fault Tolerance – BFT). Za samo delovanje ne potrebuje kriptovalute, potrjevanje transakcij in blokov pa ni nujno izvedeno s strani vseh sodelujočih, ampak le določene podmnožice, kar v teoriji omogoča paralelizacijo in posledično višjo zmogljivost [6].

Modularnost

Omrežje sestoji iz šestih osnovnih komponent, ki jih je moč poljubno menjati [6]:

1. urejevalnik (ordering service)
2. upravitelj članstva (membership service) - povezuje zunanje entitete z njihovimi kriptografskimi predstavitvami
3. P2P gossip protocol - opcijski

4. Pametne pogodbe (chaincode) - procesno izolacijo zagotavlja izvajanje znotraj vsebnikov Docker. Onemogočen je neposreden dostop do glavne knjige
5. SUPB (DBMS)
6. zamenljiva politika potrjevanja in validiranja

Pametne pogodbe

Pametne pogodbe so delčki programske kode, ki se izvajajo kot distribuirane aplikacije. Tri glavne značilnosti teh aplikacij so: veliko število sočasno izvajanih pametnih pogodb, dinamično dodajanje v omrežje in v osnovi nevredne zaupanja. Obstoječi načini izvajanja pogodb so umeščene v arhitekturo **uredi-izvedi**. Za njih je značilno, da transakcije validirajo in sekvečno uredijo, temu pa sledi propagacija potrjenih blokov po omrežju. Vsaka sodelujoča entitea nato transakcije izvede v tem vrstem redu. Za enoličen način sekvenčnega izvajanja tu nastane potreba po novem, determinističnem programskem jeziku. En izmed predstavnikov je programski jezik za programiranje pogodb v omrežju Ethereum, Solidity. Ker je vsaka izmed transakcij izvedena s strani vsake entitete, to predstavlja veliko porabo razpoložljivih virov ter omejuje skaliranje ter učinkovitost izvajanja [6].

Fabric pametne pogodbe izvaja po arhitekturi **izvedi-uredi-validiraj**. Vsaka transakcija je najprej izvedena s čimer se preveri njeno pravilnost. Nato je urejena, glede na protokol za doseganje soglasja. Ob koncu je transakcija validirana s strani za to pooblaščenih zunanjih entitet. Tu v igro vstopi domensko specifična politika potrjevanja. Slednje prinaša potencialno velike performančne prihranke [6].

Poglavje 4

Decentralizirano izvajanje

Izvajanje storitev se danes seli v oblak. Ta način izvajanja s seboj prinaša kar nekaj prednosti, kot so samodejno skaliranje in istanciranje posamezne storitve, glede na trenutne zahteve in potrebe. Računalniški oblak je abstrackija, ki za seboj skriva ogromne podatkovne in računske centre. Ti so v večini v lasti velikih korporacij, prednjačijo Amazon, Google in Microsoft. Odvisnost od ponudnika računalniškega oblaka zna biti problematična, napaka v sistemu lahko povzroči večurno nedostopnost naših storitev. Navkljub skrbi, ki jo ponudniki posvečajo vzdrževanju 100% dosegljivosti, od zadnjega odmevnejšega primera mineva le dobro leto in pol [20].

Izvajanje storitev želimo decentralizirati - vsaka sodelujoča entiteta v omrežju lahko, pod določenimi pogoji, izvaja katerokoli izmed nabora razpoložljivih storitev. Prednost, ki jo prinaša decentralizirano izvajanje poslovne logike je praktično nemogoč napad zavrnitev storitve (DOS) in porazdeljen napad zavrnitve storitve (DDOS). Napadalec je zmožen posamezno sodelujočo entiteto v omrežju obremeniti do te mere, da le ta preneha z izvajanjem določene storitve. Decentraliziran sistem bi v primeru prenehanja izvajanja storitve na eni entiteti izvajanje dodelil drugi. Postopek bi moral biti za končnega uporabnika storitve transparenten. S tehnologijo podatkovnih blokov bi bilo moč posamezne klice storitev tudi finančno ovrednotiti. Trenutno je potrebnih več klicev, ki kot prvo izvedejo samo poslovno logiko

aplikacije, temu pa sledi klic, ki izvede še finančno transakcijo. Podatkovna veriga nam omogoča, da klice storitev opremimo s finančnimi podatki in ob uspešni izvedbi izvajalca sistem samodejno nagradi.

V nadaljevanju diplomskega dela so predstavljeni osnovni koncepti rešitve registracije in odkrivanja storitev v decentraliziranem okolju.

4.1 Odkrivanje storitev v decentraliziranem okolju

Predlagana zasnova sistema za odkrivanje storitev v decentraliziranem okolju sestoji iz naslednjih komponent:

- Registracija nove aplikacije 4.1.1
- Registracija izvajalcev 4.1.2
- Registracija storitve 4.1.3
- Odkrivanje storitev 4.1.4

Registracija in odkrivanje storitve je uporaba že znanih konceptov, prenesenih v okolje, kjer vlogo registra storitev prevzema porazdeljena glavna knjiga. Dodatno smo uvedli še registracijo aplikacije, ki je pripravljena na izvajanje v decentraliziranem omrežju in registracijo izvajalcev. Slednja omogoča lastništvo več izvajalnih naprav eni posamezni identiteti.

4.1.1 Registracija nove aplikacije

Omrežje je javno, kdorkoli lahko objavi in ponudi novo aplikacijo, pripravljeno za izvajanje. Izvorno kodo oziroma izvršljivo datoteko aplikacije shranimo v decentralizirano shrambo. Temu sledi zapis podatkov (ime, verzija, izvajalno okolje...) o aplikaciji v glavno knjigo. Ob uspešni registraciji omrežje proži dogodek s podatki o novi aplikaciji. Posamezni izvajalec, ki se registrira na te dogodke, lahko takoj prične z izvajanjem aplikacije, če se za to odloči,

oziroma mu izvajanje dodeli omrežje. Razporejanje opravil in izvajanja po omrežju je stvar prihodnjih raziskav.

4.1.2 Registracija izvajalcev

Vsaka identiteta lahko upravlja z več izvajalnimi enotami. Ta pristop omogoča enemu uporabniškemu računu pripis vseh nagrad, ki si jih posamezni izvajalec prisluži. Podatki, ki se o posameznem izvajalcu zabeležijo so: lastnik (account), unikatna številka izvajalca (id) in naslov, preko katerega je izvajalec dosegljiv. Izvajalcu se lokalno konfigurira omejitve glede porabe sistemskih virov, ki jih lastnik nameni decentraliziranemu izvajanju storitev.

4.1.3 Registracija izvajanja storitve

Ob zahtevi za pričetek izvajanja aplikacije, ki je bila predhodno registrirana v omrežju, se prične postopek pridobivanja informacij o želeni storitvi. Iz registra aplikacije se pridobi podatke o imenu, verziji, lastniku in lokaciji shrambe. Izvorno kodo oz. izvršljivo datoteko storitve se nato pridobi iz omrežja, po potrebi jo nadzorni proces prevede in zažene. Storitev sama ob inicializaciji poskrbi za registracijo v registru (glavna knjiga). Zabeleži se podatke o izvajalcu in storitvi, ki je pričela z izvajanjem.

4.1.4 Odkrivanje storitve

Želeno storitev, pod pogojem, da se v omrežju izvaja, pridobimo z enostavno poizvedbo v glavni knjigi. Med razpoložljivimi storitvami odjemalec izbere eno, pridobi podatke o lokaciji izvajanja in izvede klic. Od tu naprej komunikacija med storitvami poteka preko izbranega protokola (REST, gRPC, Event-driven).

Deregistracija storitev predstavlja težji del naloge, ker ni centralnega procesa, ki bi bdel nad registriranimi storitvami in preverjal njihovo dosegljivost. Potrebna bo dodatno raziskovalno delo, kako učinkovito deregistrirati storitev, podobno kot je to rešeno pri trenutno uporabljenih registrih (etcd,

Consul, ZooKeeper). Za deregistracijo posamezne storitve je tako zaenkrat odgovoren nadzorni proces.

Poglavje 5

Implementacija predlagane rešitve

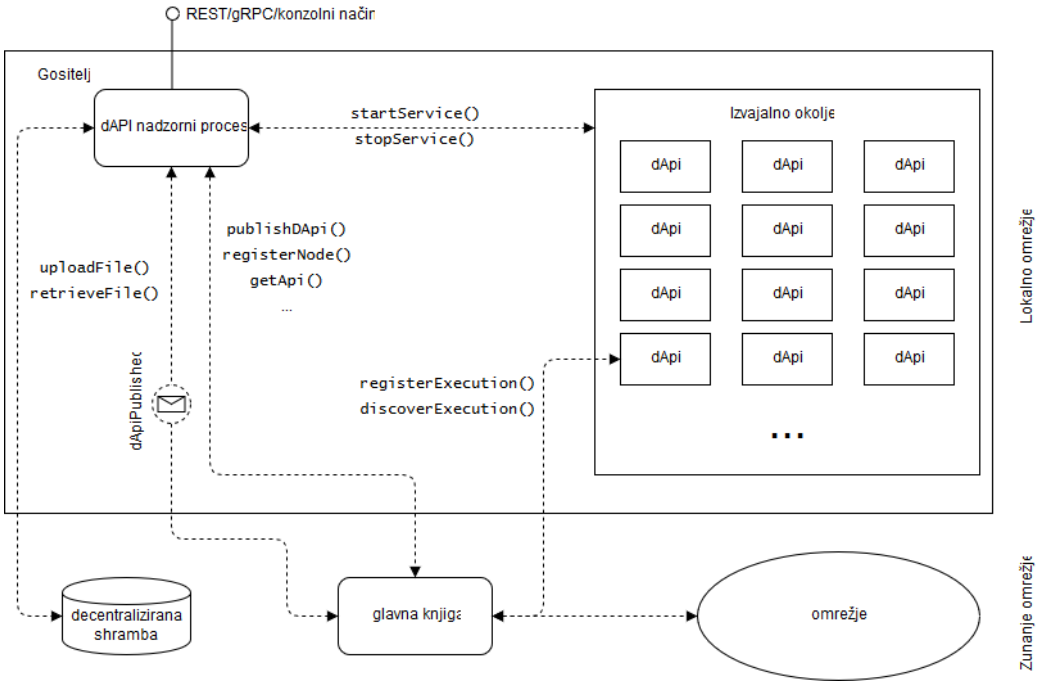
Pri implementaciji predstavljenih konceptov sem se osredotočil le na osnovne funkcionalnosti registra - registracijo aplikacije, registracijo izvajalcev in registracijo storitve.

Na sliki 5 je prikazana shema predlagane rešitve. Osrednji del omrežja je nadzorni proces - dApi manager, ki dostopa do glavne knjige, decentralizirane shrambe in upravlja s storitvami, ki jih gostitelj izvaja. Gostiteljski sistem v trenutni različici podpira upravljanje z vsebniki Docker. Aplikacije, ki so na voljo za izvajanje so shranjene na datotečni shrambi IPFS, podporna podatkovna veriga je na omrežju Ethereum. Posamezne komponente sistema so podrobneje opisane v nadaljevanju.

5.1 Nadzorni proces

Nadzorni proces za izvajanje decentraliziranih storitev (dAPIjev) skrbi za:

- registracijo novih storitev v omrežje,
- prenos izvršljivih datotek v in iz omrežja,
- zagon, zaustavitev in upravljanje storitev



Slika 5.1: Arhitekturna shema rešitve

Komunikacija z nadzornim procesom trenutno poteka preko REST aplikacijskega vmesnika, v načrtu pa je implementacija vmesnikov za konzolni nadzor ter podpora novejšim komunikacijsim protokolom kot so gRPC, Apache Trift in podobni.

Za registracijo nove storitve nadzornemu procesu podamo pot do slike vsebnika Docker. Sistem poskrbi za distribucijo slike v omrežje IPFS in podatke o storitvi doda v shrambo pogodbe na Ethereum omrežju. Ob uspešni registraciji se proži dogodek, ki sodelujoče entitete obvesti o novi storitvi, pripravljeni na izvajanje. Te se lahko na dogodek lahko odzovejo z zahtevo za izvajanje storitve. Trenutno je izvajanje nove storitve potrebno prožiti ročno, ko bo pripravljen modul za razporejanje opravil, bo postopek v celoti avtomatiziran. Pred pričetkom izvajanje se želeno storitev pridobi iz omrežja (IPFS), sliko vsebnika naloži v izvajalno okolje Docker in proži izvajanje. Storitev nato sama poskrbi za registracijo in odkrivanje ostalih storitev

v omrežju.

Razširitev obstoječe KumuluzEE konfiguracijske datoteke:

```
dapi-manager:
  storage:
    remote:
      type: ipfs
      location: /ip4/127.0.0.1/tcp/5001
    local:
      downloadFolder: download
      execution:
  managers:
  - type: docker
    connection: tcp://192.168.99.100:2376
    tls: true
    certificate-path: /path/to/certificate
    instance-limit: 10
  blockchain:
    provider: ethereum
    host: http://127.0.0.1:8545
    account: /path/to/wallet
    password: password
```

5.2 Decentralizirana shramba

Izvršljive datoteke oziroma slike vsebnikov storitve je potrebno shraniti na način in lokacijo, kjer bodo dostopne vsem sodelujočim entitetam v omrežju. Podobno, kot želimo izvajanje storitev decentralizirati, moramo poskrbeti tudi za decentralizirano shrambo. V svoji implementaciji sem uporabil decentralizirano shrambo IPFS. Gre za projekt, osnovan na omrežju Ethereum, namen projekta pa je shranjevanje datotek v porazdeljeni shrambi, dostop do njih pa preko omrežji P2P, podobno kot deluje protokol BitTorrent, po

katerem se projekt tudi zgleduje. Vsako datoteko, ki jo želimo shraniti v omrežje, odjemalec razbije na podatkovne bloke, izračuna zgoščeno vrednost posameznega bloka, te vrednosti pa nato sestavi v strukturo imenovano Merkle Tree. Datoteko pridobimo enostavno preko zgoščene vrednosti v korenu drevesa. Omrežje je sposobno poiskati posamezne koščke prvotne datoteke, vsebino posameznega pa hitro preveri z izračunom zgoščene vrednosti. V kolikor nam kdo želi podtakniti napačne bloke posamezne datoteke, sistem to prepozna in neveljavne bloke preprosto zavrže, ko od ostalih sodelujočih entitet prejme iste dele datoteke. Datoteka je veljavna, v kolikor sistem uspe sestaviti podatkovno strukturo Merkle Tree, katerega zgoščena vrednost v korenu drevesa je identična podani [9].

5.3 Odjemalec za Ethereum omrežje

Sistem za delovanje potrebuje odjemalca, ki se zna povezati v Ethereum omrežje. V moji testni postavitvi je mesto odjemalca prevzel program Geth, implementacija Ethereum protokola v programskem jeziku Go. [3] Nadzorni proces se preko JSON RPC povezuje na lokalno instanco odjemalca Geth, decentralizirane storitve v izvajalnem okolju Docker pa se povezujejo na proces, dostopen preko mreže Infura. Infura nam omogoča enostaven dostop do Ethereum omrežja, brez potrebe po lokalnem izvajanju Ehtereum protokola. Za sodelovanje v Ethereum omrežju nam tako lokalno ni potrebno namestiti ničesar, prav tako nam ni potrebno hraniti celotne zgodovine podatkovne verige. Na spletni strani se enostavno registriramo, s tem pridobimo unikaten ključ, ki nam omogoča dostop do oddaljenega izvajalca [8].

5.4 Implementacija pametne pogodbe

Pametna pogodba, ki predstavlja register storitev in osrednji del sistema, je napisana v programskem jeziku Solidity. Solidity je jezik, v katerem ustvarjalci Ethereum omrežja priporočajo implementacije pametnih pogodb. Je

visoko nivojski jezik, precej podoben JavaScriptu in namenjen izvajanju na navideznem stroju Ethereum (EVM). Med konstrukti jezika najdemo dedovanje, knjižnice, uporabniško določene tipe in ostale visokonivojske konstrukte. Namensko orodje za razvoj pametnih pogodb je trenutno le eno, poznano pod imenom Remix, dostopno pa je tudi v spletni različici [17].

Osnovni konstrukti zasnovane pogodbe so strukture User, dApi, Worker in Execution.

```
contract Registry {
        struct User {
                string friendlyName;
                mapping(bytes32 => Worker) workers;
                mapping(bytes32 => dApi) dApis;
        }

        struct dApi {
                string name;
                string version;
                string location;
                bool isValid;
        }

        struct Worker {
                string name;
                bytes32 workerId;
                bool isValid;
        }

        struct Execution {
                Worker worker;
                string location;
                bool active;
```

```
}

/// MAPPINGS
mapping (address => User) users;
mapping (bytes32 => Execution[]) dAPIExecutors;
...
```

Struktura **User** predstavlja ponudnika decentraliziranih aplikacij. Vsakemu ponudniku pripada seznam izvajalcev (workers) in objavljenih API-jev (dApis). O posamezni storitvi hranimo ime, verzijo ter lokacijo, ki v našem primeru predstavlja zgoščeno vrednost korena slike Docker, shranjene na IPFS. Vsak izvajalec je samostjona enota, ki je zadolžena za izvajanje aplikacije, struktura Execution, v navezavi z mapiranjem *dAPIExecutors* pa povezuje aplikacijo z njenimi izvajalci in naslovu, preko katerega je dostopna.

Registracija poteka v več korakih, najprej je potrebno registrirati ponudnika storitve, nato se samodejno, ob zagonu instance nadzorne storitve, izvede registracija izvajalca. Za registracijo storitve poskrbi storitev sama, preko razširitve *Kumuluz-dapi*, takoj ob zagonu.

```
function registerApi(string _name,
        string _version,
        string _location,
        bytes32 _hash, bool _isValid) public {
        dApi memory dapi = dApi(
        _name,
        _version,
        _location,
        _isValid
        );

        users[msg.sender].dApis[_hash] = dapi;
        emit dAPIPublished(_name, _version, msg.sender,
        _hash);
```

```
}
...
```

Proces deregistracije izvajanja storitve je kompleksnejši postopek, ki zahteva kar nekaj računskega dela. Dokler je v sistemu malo registriranih izvajalnih enot je deregistracija dokaj neopazna, tako z vidika časa in cene. Problem nastopi v primeru velikega števila izvajalnih enot. Na testnem omrežju *Rinkeby* sem ob testiranju večjega števila registrianih izvajalnih enot presegel omejitev računskih korakov na transakcijo (gas).

```
function deregisterExecution(bytes32 _apiHash,
uint _wIndex) public {
        if (_wIndex >= dAPIExecutors[_apiHash].length)
                return;

        for (uint i = _wIndex;
        i<dAPIExecutors[_apiHash].length-1; i++){
                dAPIExecutors[_apiHash][i]=
                        dAPIExecutors[_apiHash][i+1];
        }
        dAPIExecutors[_apiHash].length--;
}
```

Večji problem nastopi ob deregistraciji izvajalca, tu bi bilo potrebno iterirati preko vseh registriranih storitev in poiskati izvajalca za brisanje. Ta del registra še ni implementiran in potrebuje dodatno pozornost v prihodnje.

5.5 Razširitev KumuluzEE platforme za podporo decentraliziranim aplikacijam

5.5.1 Implementacija Kumuluz-dapi

Želja in cilj za učinkovit razvoj decentraliziranih aplikacij je priprava programskih vmesnikov, ki bodo razvijalcem omogočili enostavno nadgradnjo in predelavo obstoječih aplikacij v dApije. V razvojni fazi je razširitev za aplikacije, ki uporabljajo platformo KumuluzEE. Vmesnik mora biti intuitiven in enostaven za uporabo, zato smo za zgled vzeli podobno in že obstoječo rešitev *KumuluzEE Service Discovery*, ki rešuje problem odkrivanja storitev v oblačnih arhitekturah [26].

Anotacije

Razširitev vsebuje dve anotaciji, s katerima opremimo obstoječo aplikacijo. Prva služi registraciji storitve, z njo pa se opremi glavni aplikacijski razred REST storitve, ki je trenutno edini podprt način komunikacije.

```
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface RegisterDApi {
        String name() default "";
        String environment() default "";
        String version() default "";
        ApiType apiType() default ApiType.REST;
}
```

Podatki, ki jih beležimo o posamezni storitvi so: *ime (name), okolje (environment), verzija (version) in tip aplikacijskega vmesnika (apiType).*

Druga anotacija skrbi za označevanje parametrov, ki jih preko javanske tehnologije CDI, vrinemo v spremenljivke.

```
@Qualifier
```

```
@Target({ElementType.FIELD, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface DiscoverDApi {
        @Nonbinding String name() default "";

        @Nonbinding String environment() default "";

        @Nonbinding String version() default "";

        @Nonbinding ApiType apiType()
         default ApiType.REST;
}
```

Programska logika

Preko procesiranja anotacij pridobimo podatke o želeni storitvi, pri inicializaciji pa razširitev poskrbi za samodejno registracijo, oziroma odkrivanje storitve. Za delovanje razširitve potrebujemo omogočeno tehnologijo CDI. Metoda za registracijo storitve je povsem enostavna, uporabljamo pa knjižnico Web3j. Web3j je majhna knjižnica, namenjena aplikacijam Java in Android, ki se zna povezati na omrežje Ethereum. Klice na REST vmesnik odjemalca (geth, parity) ovije v razrede in metode, omogoča pa tudi pretvorbo pogodb v jeziku Solidity v javanske razrede [22].

```
private void registerService(
        ServiceExecution apiExecutionDetails) {

        logger.info("Registering new service execution");
        registry
        .registerService(apiExecutionDetails.getApiHash(),
         apiExecutionDetails.getWorker(),
         apiExecutionDetails.getLocation()).observable()
        .subscribe(tx -> {
```

```
            BigInteger gasUsed = tx.getGasUsed();
            logger.info("New dApi executor registered: "
             + apiExecutionDetails.toString() +
             ". Gas spent: " + gasUsed);
            logger.info("Block number: "
             + tx.getBlockNumber());
            });
}
```

5.5.2 Priprava aplikacije

V standardno KumuluzEE aplikacijo je potrebno vključiti razširitev *kumuluzdapi*. Razširitev ponuja nabor anotacij, s katerimi storitev bodisi registriramo v omrežje, bodisi določeno storitev poiščemo. Potrebna je še dopolnitev konfiguracijske datoteke, v kateri podamo dodatne informacije o naši storitvi. Ob zagonu storitve se preko anotacij poišče in registrira v omrežje. V glavno knjigo se zapišejo podatki o izvajani storitvi, izvajalcu ter naslov, na katerem je storitev dostopna.

Za odkrivanje storitev poskrbi razširitev, ki v glavni knjigi poišče izvajalce storitve, med njimi pa na podlagi izbranega algoritma za razporejanje bremena, izbere enega izmed izvajalcev in pridobi naslov, na katerem se storitev trenutno izvaja.

Razširitev obstoječe KumuluzEE konfiguracijske datoteke:

```
dapi:
  blockchain:
    host: naslov izvajalca
    account: /pot/do/datoteke
    password: geslo
  type: docker
  storage: ipfs
  api: rest
```

V konfiguraciji so predvidene tudi trenutno neuporabljene nastavitve za tip izvršljive datoteke, v kateri shrambi se storitev nahaja in tip aplikacijskega vmesnika. Trenutno sistem podpira le vsebnike Docker, ki jih je moč pridobiti preko omrežja IPFS, aplikacijski vmesnik pa je REST.

Poglavje 6

Delovanje in evalvacija

6.1 Testna aplikacija

Za namene testiranja delovanja sistema sem implementiral dve preprosti REST storitvi - uporabniki (users) in kupci (customers). Storitev *uporabniki* hrani seznam vseh uporabnikov v sistemu. Storitev *kupci* od prve storitve pridobi seznam uporabnikov in ga le posreduje naprej. Prva storitev testira uspešnost registracije izvajanja, medtem ko druga služi kot test odkrivanja uspešno registrirane storitve.

Implementirani storitvi sta zelo enostavni, sestavljata ju le po dva Javanska razreda, zato jih na tem mestu ne bi posebej predstavljal.

6.2 Testiranje

Celotnega sistema nisem uspel spraviti v popolnoma delujoče stanje, zaradi težav in pomankljivosti, ki so podrobno navedene v podpoglavju 6.3. Ob zagonu nadzornega procesa se uspešno le-ta uspešno registrira kot izvajalec. Zatem s klicem na */dapi/account/registerApi* uspešno prenese sliko Docker na shrambo IPFS in programski vmesnik registrira na omrežje. Uspešno se izvede tudi proženje dogodka, ki sodelujoče obvesti o novo objavljeni storitvi.

S klicem na */dapi/address/startService* se iz shrambe IPFS prenese slika

aplikacije, vendar je le ta poškodovana in je ni moč naložiti ter izvesti. Sistem sem ročno zaobšel in preko nadzornega procesa zagnal lokalno kopije slike. Storitev se uspešno zažene in registira v omrežju, tu pa nastopi naslednja težava. Storitev je moč odkriti, vendar ne takoj po registraciji. Vzroka za to nisem uspel najti, ko pa je storitev moč odkriti, se testna aplikacija *kupci* uspešno poveže na storitev *uporabniki* in ob zahtevi pridobi celoten seznam uporabnikov, ki ga le posreduje naprej.

6.3 Pomankljivosti trenutnega sistema in izboljšave v prihodnosti

Prva različica sistema ponuja kar nekaj možnosti za izboljšave. Prva izmed pomankljivosti je sama cena registracije in deregistracije storitev in njihovih izvajalcev. Vsaka registracija novega izvajalca pomeni nov zapis na podatkovno verigo, kar v dinamičnem sistemu in ob trenutnih cenah transakcije na Ethereum omrežju predstavlja potencialno veliko finančno breme za izvajalca. Zahtevnejša od prve je druga pomankljivost sistema in sicer deregistracija izvajalca. Brisanje podatkov iz podatkovne verige je nemogoče zaradi same zasnove tehnologije - nespremenljivost. S transakcijo je možno le navidezno brisati - razveljaviti preteklo transakcijo, zapis pa na podatkovni verigi ostane. Brisanje je prav tako razmeroma draga operacija, posebno velik problem predstavlja brisanje izvajalca in s tem posledično še deregistracijo vseh storitev, ki jih je ta izvajalec v trenutku prekinitve izvajanja izvajal.

Dodatni mehanizmi, ki jih poznajo obstoječi sistemi za odkrivanje storitev, so še samodejna deregistracija storitve ob vnaprej določenem pretečenem času neaktivnosti. Trenutna implementacija tega mehanizma še ne pozna, problem predstavlja predvsem decentralizacija. V tem sistemu ne poznamo centralne storitve, ki bi ob določenem času iz registra preprosto brisala vse neaktivne storitve. Sistem potrebuje nov mehanizem, ki bo znal med storitvami poiskati neaktivne in jih odstraniti iz registra. Kakšen bo mehanizem in princip delovanja sta v tem trenutku neznanka.

Sistem potrebuje tudi način prerazporejanja zahtev po omrežju. Kdo izmed trenutno registriranih izvajalcev bo lahko najhitreje odgovoril? Hiter odgovor je pogojen s fizično oddaljenostjo gostitelja od izvajalca, omrežnimi zakasnitvami, hitrost samega izvajalca. Reševanje teh izzivov je naslednja v vrsti, ki jih predstavlja celostna postavitev, v praksi uporabnega, sistema.

Vsakega izvajalca storitve želimo za uspešno izveden klic ustrezno finančno nagraditi. Tu se poraja več vrst odprtih vprašanj, izstopa predvsem vprašanje finančne vrednosti posameznega klica in način preverjanja pravilnosti izvedbe klica. Je izvajalec dejansko pravilno izvedel zahtevano dejanje, tako kot je predvidel razvijalec in naročnik? Potreben je mehanizem, ki ga zaenkrat imenujmo „Proof of Execution“. Podobno kot trenutni mehanizmi „Proof of Work“, „Proof of Stake“ ter sorodni, ki jih uporabjajo decentralizirane podatkovne verige, potrebujemo mehanizem, preko katerega se bodo sodelujoče entitete v omrežju sposobne odločiti, ali je določen izvajalec pravilno izvedel zahtevano dejanje. S tem področjem se trenutno aktivno ukvarjajo tudi pri startupu SONM, ki objublja računsko moč na zahtevo v decentraliziranem okolju. Iz oblačnega računalništva želijo preiti na t.i. „računalniške storitve v megli“ (fog computing) [18].

Za podporo načrtovanemu sistemu je v prihodnje potrebno razviti tudi lastno podatkovno verigo. Podatkovna veriga bi morala omogočati hitro potrjevanje transakcij, ki so podporna veja registra storitev. Optimizirati bi bilo potrebno računsko zahtevnost danih operacij, oziroma omejiti izvajanje pametnih pogodb le na podmnožico entitet v omrežju, funckionalnost, podobna tisti v sistemu HyperLedger Fabric. Omrežje najverjetneje potrebuje tudi učinkovitejši algoritem za dosego konsenza. Proof of Work tu odpove, oziroma predstavlja preveliko časovno in finančno potratnost.

Sistem bi za praktično uporabo potreboval tudi učinkovit način za izvedbo finančnih transakcij ob uspešno izvedenem klicu. V arhitekturi mikrostoritev se klice storitev največkrat preusmerja preko aplikacijsih prehodov (API gateway). Ti zbirajo določene statistične parametre o klicih in, na podlagi vnaprej definiranih finančnih politik, zaračunajo uporabo. Ideja pri decen-

traliziranem sistemu je ob uspešnem klicu samodejno izvesti finančno transakcijo, ter centraliziran prehod nadomestiti s popolnoma decentralizirano logiko. Način, kako to izvesti je zaenkrat še neznanka in bo stvar prihodnjih raziskav.

Poglavje 7

Zaključek

Mikrostoritve in oblačna arhitektura sta prinesli revolucijo v načinu razmišljanja in gradnje aplikacij. Približujemo se temu, kar v fizični proizvodnji poznamo že dolgo, in sicer hitre proizvodne linije za izdelke. Podobno lahko danes, tako kot kocke, sestavljamo tudi aplikacije. Ne zanima nas kako vsak posamezen košček celotnega sistema rešuje svoj del problema, ampak koristimo vnaprej definiran aplikacijski vmesnik, ki se načeloma ne spreminja. Aplikacije, ki so končni produkt takega načina zlaganja, so odporne na hitre spremembe v okolju, sposobne samodejnega odpravljanja napak in skaliranja. Sestavni deli so med seboj šibko sklopljeni in odgovorni le za svoje ozko usmerjene naloge. Problem, ki ga naslavljamo, je morda enostavno spregledati. Trenutni sistem je odporen na vse vrste programskih napak, s pomočjo replikacij preko fizično ločenih računskih centrov deloma tudi strojnih okvar, še vedno pa lahko trpi dosegljivost in odzivnost [20]. V kolikor nam uspe te vire računske moči decentralizirati, sistemom dodamo še dodatno dimenzijo odpornosti in praktično onemogočimo napade zavrnitve odzivnosti (DOS).

Na poti k temu cilju je potrebno rešiti veliko odprtih vprašanj, ki so v trenutnih oblačnih sistemih že rešeni, ter problemi, ki se pojavijo pri decentraliziaciji.

V diplomski nalogi sem se ukvarjal z registracijo in odkrivanjem storitev v decentraliziranem okolju. Namestno centralnega registra sem uporabil

tehnologijo podatkovnih blokov, ki s pomočjo kriptorgrafskih prijemov zagotavlja nespremenljivost in enostavno preverljivost zapisanih in porazdeljenih podatkov. Registra mi ni uspelo razviti v meri, ki sem si jo zamislil. Pri implementaciji konceptov sem naletel na kup nepredvidenih težav, povezanih predvsem s samo tehnologijo podatkovnih blokov. Rezultat je delno delujoč sistem, ki je osnovan na omrežju Ethereum. Omrežje ni primerno za izvajanje nalog, ki so bile zamišljene, zato za nadaljnje raziskovalno delo po vsej verjetnosti potrebujemo prilagojeno implementacijo celotne podatkovne verige.

Navkljub le delnemu uspehu bi, ob dodatnem raziskovalnem delu, lahko na področju decentraliziranega izvajanja dosegli nove mejnike in zakoličili pot naslednje generacije programske opreme.

Literatura

[1] Containers and Orchestration Explained. Dosegljivo: https://www.mongodb.com/containers-and-orchestration-explained. Dostopano: 15. 07. 2018.

[2] Ethereum whitepaper. Dosegljivo: https://github.com/ethereum/wiki/wiki/White-Paper#ethereum. Dostopano: 22. 05. 2018.

[3] Go ethereum. Dosegljivo: https://geth.ethereum.org/. Dostopano: 28. 06. 2018.

[4] Hyperledger. Dosegljivo: https://www.hyperledger.org. Dostopano: 22. 05. 2018.

[5] Hyperledger - open source blockchain for business. Dosegljivo: https://www.ibm.com/blockchain/hyperledger.html. Dostopano: 22. 05. 2018.

[6] Hyperledger Fabric documentation. Dosegljivo: https://hyperledger-fabric.readthedocs.io. Dostopano: 22. 05. 2018.

[7] IBM. What is cloud computing? Dosegljivo: https://www.ibm.com/cloud/learn/what-is-cloud-computing. Dostopano: 11. 07. 2018.

[8] Infura. Dosegljivo: https://infura.io/. Dostopano: 28. 06. 2018.

[9] Ipfs. Dosegljivo: https://ipfs.io/. Dostopano: 28. 06. 2018.

[10] Microservices: a definition of this new architectural term. Dosegljivo: https://martinfowler.com/articles/microservices.html. Dostopano: 11. 07. 2018.

[11] Pattern: Application metrics. Dosegljivo: http://microservices.io/patterns/observability/application-metrics.html. Dostopano: 28. 06. 2018.

[12] Pattern: Health check. Dosegljivo: http://microservices.io/patterns/observability/health-check-api.html. Dostopano: 04. 07. 2018.

[13] Pattern: Microservice Architevture. Dosegljivo: http://microservices.io/patterns/microservices.html. Dostopano: 11. 07. 2018.

[14] Pattern: Monolithic Architecture. Dosegljivo: http://microservices.io/patterns/monolithic.html. Dostopano: 22. 05. 2018.

[15] Pattern: Service discovery. Dosegljivo: https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/. Dostopano: 04. 07. 2018.

[16] Prometheus overview. Dosegljivo: https://prometheus.io/docs/introduction/overview/. Dostopano: 28. 06. 2018.

[17] Solidity Documentation. Dosegljivo: http://solidity.readthedocs.io/en/v0.4.24/. Dostopano: 12. 07. 2018.

[18] Sonm. Dosegljivo: https://sonm.com/. Dostopano: 28. 06. 2018.

[19] The Linux Foundation. Cloud native computing foundation („cncf“) charter. Dosegljivo: https://www.cncf.io/about/charter/. Dostopano: 11. 07. 2018.

[20] Update: 11-hour AWS failure hits websites and apps. Dosegljivo: https://www.computerworld.com/article/3175643/cloud-computing/aws-failure-hits-web-sites-and-apps.html. Dostopano: 16. 07. 2018.

[21] Using Minikube to Create a Cluster. Dosegljivo: https://kubernetes.io/docs/tutorials/kubernetes-basics/create-cluster/cluster-intro/. Dostopano: 15. 07. 2018.

[22] Web3J. Dosegljivo: https://github.com/web3j/web3j. Dostopano: 16. 07. 2018.

[23] What is a container. Dosegljivo: https://www.docker.com/what-container. Dostopano: 11. 07. 2018.

[24] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *Principles of Security and Trust*, pages 164–186. Springer, 2017.

[25] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[26] Urban Malc. Zasnova in razvoj rešitve za dinamično odkrivanje mikrostoritev v oblačnih arhitekturah. Diplomska naloga, Fakulteta za in računalništvo in informatiko, Univerza v Ljubljani, 2017.