View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by ePrints.FRI

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Domen Balantič

Storitve REST in API-ji s tehnologijo verige blokov

DIPLOMSKO DELO

INTERDISCIPLINARNI UNIVERZITETNI
ŠTUDIJSKI PROGRAM PRVE STOPNJE
RAČUNALNIŠTVO IN MATEMATIKA

Mentor: prof. dr. Branko Matjaž Jurič

Ljubljana, 2017

COPYRIGHT. Rezultati diplomske naloge so intelektualna lastnina avtorja in Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavo in koriščenje rezultatov diplomske naloge je potrebno pisno privoljenje avtorja, Fakultete za računalništvo in informatiko ter mentorja.

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Proučite in analizirajte tehnologijo verige blokov. Opišite glavno knjigo, pametne pogodbe in ostale koncepte delovanja. Preglejte ključna področja uporabe tehnologije verige blokov in opišite novosti v verigi blokov 2.0. Identificirajte in primerjajte najpomembnejše platforme za razvoj rešitev z uporabo verige blokov. Podrobno opišite platformo Hyperledger Fabric. Izdelajte rešitev, ki bo omogočala zapisovanje dnevnika dostopov v verigo blokov z uporabo mikrostoritev ter demonstrirajte primernost pristopa in prednosti ter slabosti.

Zahvaljujem se mentorju prof. dr. Branku Matjažu Juriču za pomoč pri pisanju diplomske naloge. Posebna zahvala gre mojim staršem, dekletu, bratu, sestri in vsem prijateljem, ki so mi bili v oporo v času študija, predvsem pa v času nastajanja diplomske naloge, ker so verjeli vame, da mi bo uspelo in me spodbujali s pozitivnimi mislimi. Iskrena zahvala še lektorici Mojci, ki je zaslužna za slovnično pravilnost moje naloge.

Kazalo

Povzetek

Abstract

1 Uvod **1**

1.1 Motivacija 1

1.2 Cilji 1

1.3 Struktura diplomske naloge 2

2 Veriga blokov **3**

2.1 Ključne lastnosti verige blokov 3

2.1.1 Velikost blokov 4

2.1.2 Porazdeljena glavna knjiga 5

2.1.3 Pametne pogodbe 6

2.1.4 Soglasje 8

2.2 Uporaba verige blokov 9

2.2.1 Aktualni sistem beleženja zapisov 9

2.2.2 Sistem beleženja zapisov v prihodnosti z uporabo verige blokov 10

2.3 Zgodovina verige blokov 11

2.3.1 Razcepi verige blokov 13

2.4 Veriga blokov 2.0 14

3 Pregled in primerjava platform verige blokov 15
3.1 Bitcoin 15
3.1.1 Omrežje 16
3.1.2 Vrste transakcij 17
3.1.3 Način potrjevanja transakcij 17
3.1.3.1 Dokazilo o delu 18
3.1.3.2 Rudarjenje 18
3.1.4 Podpora za pametne pogodbe 19
3.2 Ethereum 19
3.2.1 Omrežje 20
3.2.2 Vrste transakcij 20
3.2.3 Način potrjevanja transakcij 21
3.2.3.1 Dokazilo o delu 21
3.2.3.2 Rudarjenje 21
3.2.4 Podpora za pametne pogodbe 22
3.3 Hyperledger Fabric 23
3.3.1 Omrežje 23
3.3.2 Vrste transakcij 24
3.3.3 Način potrjevanja transakcij 24
3.3.3.1 Praktična bizantinska odpornost na napake . . . 25
3.3.3.2 Postopek generiranja blokov 25
3.3.4 Podpora za pametne pogodbe 26
3.4 Primerjava platform verige blokov 26
3.4.1 Primerjava specifikacij 26
3.4.2 Primerjava omrežja 26
3.4.3 Primerjava vrste transakcij 27
3.4.4 Primerjava načinov potrjevanja transakcij 27
3.4.5 Primerjava podpore za pametne pogodbe 28
3.4.6 Tabelarična primerjava platform verige blokov 28

4 Hyperledger Fabric 30
4.1 Arhitektura 30

4.1.1 Transakcije 30
4.1.2 Podatkovni strukturi verige blokov 31
4.1.2.1 Stanje 31
4.1.2.2 Glavna knjiga 32
4.1.3 Vozlišča verige blokov 33
4.1.3.1 Odjemalec 33
4.1.3.2 Vrstnik 33
4.1.3.3 Naročnik 34
4.2 Potek transakcije 35
4.3 Odobritev in izvedba transakcije 39
4.3.1 Izoblikovanje transakcije 39
4.3.2 Odobritev transakcije 40
4.3.3 Posredovanje transakcije 42
4.3.4 Izvedba transakcije 42
4.4 Politika odobravanja transakcije 43
4.4.1 Specifikacije politike odobravanja 43
4.4.2 Ovrednotenje transakcije glede na politiko odobravanja 44
4.4.3 Primer politike odobravanja 44
4.5 Verižna koda 45
4.6 Ključne lastnostni platforme Hyperledger Fabric 46

5 Zapisovanje dnevnika dostopov v verigo blokov z uporabo mikrostoritev 48
5.1 Vsebinski opis aplikacije 48
5.2 Tehnični opis aplikacije 49
5.2.1 Zasnova aplikacije 49
5.2.2 Arhitektura aplikacije 50
5.2.2.1 Implementacija verižne kode 50
5.2.2.2 Priprava končne točke REST s pomočjo Node.js 57
5.2.2.3 Integracija beleženja dnevnikov dostopov v verigo blokov znotraj modula KumuluzEE Logs 61
5.2.3 Zagon in delovanje aplikacije 64

5.2.3.1 Priprava okolja za vzpostavitev lokalne verige blokov 64
5.2.3.2 Zagon omrežja verige blokov 68
5.2.3.3 Ustvarjanje in pridružitev kanalu 68
5.2.3.4 Namestitev in incializacija verižne kode 69
5.2.3.5 Zagon končne točke Node.js REST 70
5.2.3.6 Zapisovanje dnevnika dostopov v verigo blokov . 70
5.2.3.7 Branje informacij o dnevniku dostopov iz verige blokov 71
5.2.3.8 Samodejno beleženje dnevnikov dostopov v verigo blokov znotraj modula KumuluzEE Logs 73
5.3 Priprava aplikacije za produkcijsko okolje 73
5.4 Namestitev aplikacije v produkcijsko okolje 74

6 Zaključek 76

Literatura 78

Seznam uporabljenih kratic

kratica	angleško	slovensko
BTC	Bitcoin	Bitcoin
ETH	Ethereum	Ethereum
GB	gigabyte	gigabajt
ICO	Initial Coin Offering	začetna ponudba kriptokovancev
JSON	JavaScript Object Notation	notacija za označevanje JavaScript objektov
KVS	key/value store	shramba tipa ključ/vrednost
MB	megabyte	megabajt
nonce	number only used once	enkratno kriptografsko število
PBFT	Practical Byzantine Fault Tolerance	praktična bizantinska odpornost na napake
POW	proof-of-work	dokazilo o delu
REST	Representational State Transfer	način interoperabilnosti med računalniškimi sistemi
SDK	Software Development Kit	paket za razvoj programske opreme

Povzetek

Naslov: Storitve REST in API-ji s tehnologijo verige blokov

Avtor: Domen Balantič

V diplomski nalogi smo opisali osnovne koncepte in principe tehnologije verige blokov, ki omogoča decentraliziran način shranjevanja podatkov. Prednost decentraliziranega hranjenja podatkov je v visoki stopnji integritete shranjenih podatkov, saj jih po shranitvi v verigo blokov praktično nemogoče spreminjati. Opisali smo lastnosti treh najbolj priljubljenih in razširjenih platform verige blokov - platforme Bitcoin, platforme Ethereum in platforme Hyperledger Fabric - in jih med seboj podrobneje primerjali. V okviru diplomske naloge smo vzpostavili lokalno razvojno okolje verige blokov Hyperledger Fabric in izdelali praktičen primer zapisovanja dnevnika dostopov v verigo blokov prek končne točke Node.js REST. Rešitev omogoča ročno shranjevanje podatkov z naslovitvijo ustreznega zahtevka REST na naslov končne točke ali samodejno shranjevanje podatkov o dostopih znotraj modula KumuluzEE Logs.

Ključne besede: veriga blokov, zapisovanje dnevnika dostopov, Bitcoin, Ethereum, Hyperledger Fabric, Docker, Node.js, KumuluzEE Logs.

Abstract

Title: REST services and APIs using blockchain technology

Author: Domen Balantič

In the diploma thesis we described the basic concepts and principles of the blockchain technology, which enables a decentralized way of data storing. The advantage of decentralized data storage is in a high integrity of stored data since it is virtually impossible to change it after saving it into the blockchain. We described the features of the three most popular and well known platforms of the blockchain - the Bitcoin platform, the Ethereum platform, and the Hyperledger Fabric platform - and compared them in details. Within the diploma thesis we established the local development environment of the Hyperledger Fabric blockchain and created a practical example of logging access logs into the blockchain via the Node.js REST endpoint. The solution enables manual data storing by addressing the corresponding REST request to the endpoint address or automatically storing access data within the KumuluzEE Logs module.

Keywords: blockchain, logging, Bitcoin, Ethereum, Hyperledger Fabric, Docker, Node.js, KumuluzEE Logs.

Poglavje 1

Uvod

1.1 Motivacija

Veriga blokov je novodobna tehnologija, ki predstavlja podatkovni model prihodnosti, saj so vsi podatki podvojeni na vseh sodelujočih napravah v omrežju. Za končne uporabnike je veriga blokov primerna zaradi načina shranjevanja podatkov, ki se jih po shranitvi v verigo blokov ne da več spreminjati. Posledično je vsaka posodobitev podatkov ustrezno zabeležena. Uporabnikom je tako olajšan proces evidentiranja transakcij in sledenja premoženja v poslovni mreži.

Zaradi decentraliziranega shranjevanja zapisov je zato med drugim primerna za hrambo zdravniških kartotek in osebnih identitet, obdelavo transakcij in beleženje lastništva premičnin ter nepremičnin.

1.2 Cilji

Glavna cilja diplomske naloge sta vzpostavitev lokalnega razvojnega okolja in implementacija rešitve za zapisovanje dnevnika dostopov v verigo blokov z izbrano platformo verige blokov.

Glede na uspešnost vzpostavitve lokalnega razvojnega okolja in implementacije rešitve sta nadaljnja cilja preizkus delovanja izbrane platforme v

porazdeljenem omrežju in migracija rešitve za zapisovanje dnevnika dostopov v pripravljeno porazdeljeno okolje.

1.3 Struktura diplomske naloge

V drugem poglavju so opisani osnovni koncepti verige blokov, pomembni za nadaljnje razumevanje diplomskega dela. Poglavje primerja aktualni sistem beleženja zapisov in njegovo različico v bližnji prihodnosti. Za lažjo umestitev pionirske tehnologije verige blokov v zgodovino računalniških tehnologij poglavje vsebuje tudi kratko zgodovino verige blokov in kratek opis njene druge, posodobljene, različice.

Tretje poglavje opisuje in primerja tri najbolj priljubljene platforme verige blokov, pri čemer njihova primerjava temelji na arhitekturnih sposobnostih platform. Ker imajo platforme različne performančne specifikacije, kot so čas potrditve bloka, število transakcij in cena transakcije, je odločitev o izbiri platforme prepuščena končnemu uporabniku glede na izbrano infrastrukturo omrežja.

Za nadaljnje raziskovanje in pripravo praktičnega primera smo izbrali platformo Hyperledger Fabric, ki je podrobneje opisana v četrtem poglavju diplomske naloge.

Praktični primer zapisovanja dnevnika dostopa v verigo blokov z uporabo mikrostoritev je predstavljen v petem poglavju. Poglavje zaobjema pripravo lokalnega razvojnega okolja, podrobnejši opis rešitve danega problema in postopke migracije rešitve v produkcijsko okolje.

Poglavje 2

Veriga blokov

2.1 Ključne lastnosti verige blokov

Veriga blokov (ang. blockchain) je porazdeljena podatkovna baza, podvojena na vsaki napravi, ki se posodablja prek sistema pametnih pogodb in se nenehno sinhronizira, s sodelovanjem vseh udeležencev v omrežju, prek procesa imenovanega soglasje (ang. consensus) [24].

Blokovna baza je sestavljena iz dveh vrst zapisov - transakcij in blokov. Blok je nenehno rastoč seznam potrjenih transakcij, ki je zaradi lažjega pregleda kodiran v drevo Merkle [32]. Vsak blok vsebuje časovni žig in povezavo na prejšnji blok. Tako povezani bloki tvorijo verigo, imenovano veriga blokov [41], ki uporabnikom omogoča pregled vseh transakcij v omrežju vse od začetnega, t. i. genesis bloka, vključenega v samo programsko kodo. Z izvajanjem in hranjenjem novih transakcij veriga blokov raste [5].

Vsi računalniki v omrežju skrbijo za potrjevanje novih blokov v skladu z določenim protokolom, zato je blokovna baza odporna na spreminjanje podatkov. Njene podatke namreč hrani vsak računalnik v omrežju, zato je že enkrat zapisane podatke praktično nemogoče spreminjati, saj bi morali spremeniti vse nadaljnje bloke v verigi in poskrbeti za njihovo potrditev v celotnem omrežju, za kar bi potrebovali ogromno računsko moč. To uporabnikom omogoča enostavno preverjanje in revizijo transakcij. Zaradi oteženega

spreminjanja zapisov v blokovni bazi veriga blokov lahko služi kot javno porazdeljena glavna knjiga transakcij med uporabniki v omrežju [26].

Glavni cilj snovalcev verige blokov je bila varnost, zato je veriga blokov primer porazdeljenega računalniškega sistema z visoko stopnjo bizantinske odpornosti na napake (ang. Byzantine fault tolerance - BFT), ki določa odpornost računalniških sistemov na napake [36]. Problem dvojne porabe sredstev veriga blokov odpravlja z uporabo enotnega potrditvenega mehanizma in z vzdrževanjem univerzalne glavne knjige. Vsaka transakcija je namreč pred potrditvijo vključena v množico še nepotrjenih transakcij. Prva taka transakcija je v omrežju potrjena brez težav, saj sredstva še niso bila porabljena, medtem ko je druga transakcija označena kot neveljavna, saj zaradi ponovne porabe sredstev s strani omrežja ne prejme dovolj potrditev. Veriga blokov tako ne potrebuje zaupanja vrednega administratorja, ki bi preverjal enkratno porabo sredstev. [46, 10].

Zgoščevanje (ang. hashing) je ena izmed ključnih kriptografskih tehnik, vključenih v verigo blokov. Zgoščevalne funkcije kot vhodne podatke sprejmejo transakcije znotraj bloka, ki predstavljajo zapise, pripravljene za shranitev v verigo blokov in na podlagi katerih nato generirajo enolično zgoščeno vrednost. Če nekdo poskuša spremeniti podatke znotraj bloka, bi se sprememba odražala tudi v sami zgoščeni vrednosti. Vsak udeleženec v omrežju tako lahko hitro identificira spremenjene oz. pokvarjene bloke podatkov, saj se ob spremembi podatkov spremeni tudi zgoščena vrednost bloka. Zgoščene vrednosti blokov so med seboj povezane in tako tvorijo verigo blokov [28].

2.1.1 Velikost blokov

Zaradi zmanjšanja nevarnosti, povezanih s potencialnimi napadi zavračanja storitev (ang. denial-of-service attacks) na omrežje verige blokov, razvijalci platform verige blokov največkrat omejijo velikost enega bloka. Na ta način je omejeno maksimalno število transakcij vključenih v blok, katerih skupna velikost ne sme presegati postavljene omejitve. To ob razširitvi omrežja in povečanju števila transakcij predstavlja težavo, saj se čas potrditve tran-

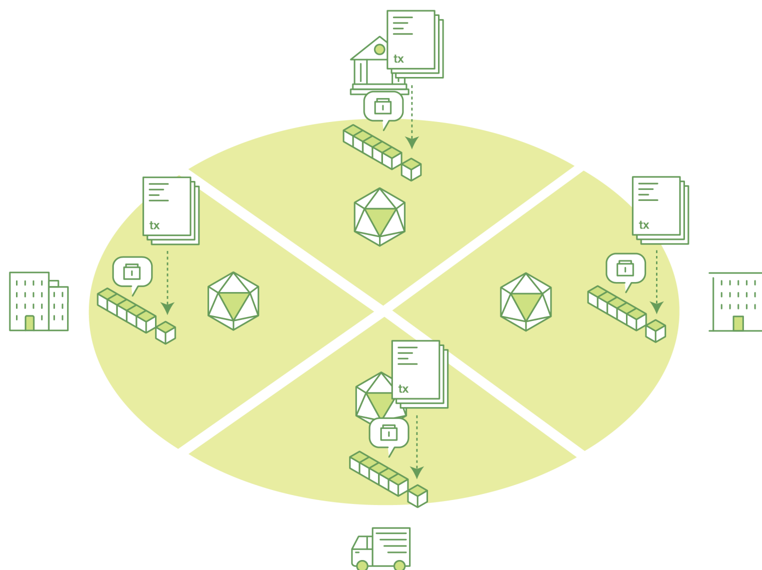
sakcij zaradi polnih blokov drastično poveča. Vozlišča, ki posredujejo transakcije v omrežje, bi lahko zaradi preobremenjenosti razširjanje določenih transakcij časovno zelo podaljšala ali kakšno izmed njih celo označila kot neveljavno. Sčasoma zato razvijalci stremijo k sistematičnemu povečevanju velikosti enega bloka glede na povečanje števila transakcij [47].

Uporabniki platforme Bitcoin so bili zaradi različnih mnenj razvijalcev o ustreznosti oz. neustreznosti povečanja števila blokov deležni popolne razcepitve verige blokov, ko sta 1. 8. 2017 nastali dve veji verige blokov Bitcoin - Bitcoin in Bitcoin Cash. Razvijalci prve platforme zagovarjajo podvojitev velikosti bloka z 1 MB na 2 MB, medtem ko razvijalci druge zagovarjajo povečavo velikosti z 1 MB na kar 8 MB.

2.1.2 Porazdeljena glavna knjiga

V osrčju omrežja verige blokov živi porazdeljena glavna knjiga (ang. ledger), ki beleži vse transakcije v omrežju. Glavna knjiga verige blokov je porazdeljena med vse udeležence v omrežju in podvojena na vseh vključenih napravah, saj je vsak sodelujoči odgovoren za njeno vzdrževanje. Porazdeljenost in sodelovanje med udeleženci sta ključni lastnosti, ki uporabnikom zagotavljata izmenjavo dobrin in storitev v realnem času.

Informacije, zapisane v verigo blokov, so s kriptografskimi tehnikami, zgoščevalnimi funkcijami, opisanimi v poglavju 2.1, zaščitene pred spreminjanjem podatkov. Ta lastnost uporabnikom omogoča enostavno preverjanje izvora in pristnosti informacij, saj te po zapisu v verigo blokov ne morejo biti spremenjene. Veriga blokov je zato včasih imenovana tudi sistem dokazovanja [24]. Slika 2.1 prikazuje porazdeljeno omrežje verige blokov.

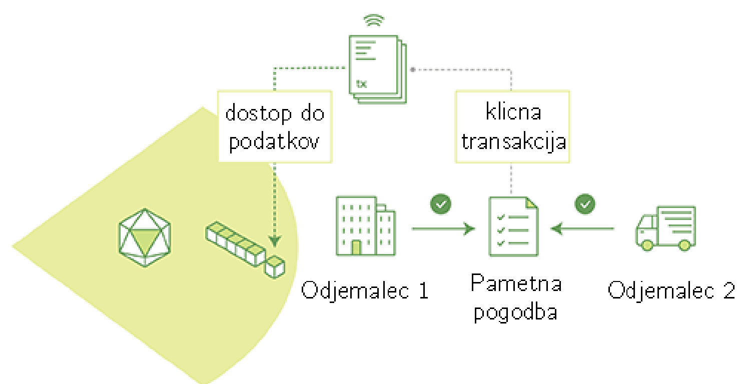


Slika 2.1: Porazdeljeno omrežje verige blokov, kjer vsak uporabnik hrani kopijo porazdeljene glavne knjige [24].

2.1.3 Pametne pogodbe

Veriga blokov vključuje podporo za pametne pogodbe, ki uporabnikom omogočajo nadzorovan dostop do glavne knjige. S tem podpira posodabljanje zapisanih informacij in omogoči dodatne funkcije za interakcijo s porazdeljeno glavno knjigo [24]. Pametne pogodbe so shranjene v verigi blokov in se lahko deloma ali v celoti izvršijo brez človeškega posredovanja. Leta 1996 je Nick Szabo prvič uporabil frazo "pametna pogodba" in jo opisal kot:

> Sam imenujem novodobne pogodbe "pametne" pogodbe, saj so veliko bolj uporabne kot njihove papirne različice. Pametna pogodba namreč vsebuje digitalno določena zagotovila in protokole, ki jim sledijo vse pogodbene stranke [42].

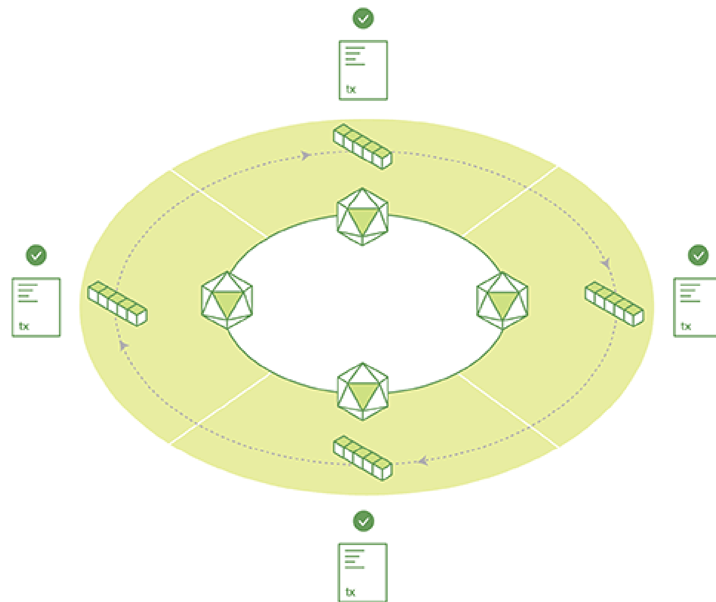


Slika 2.2: Način interakcije s porazdeljeno glavno knjigo prek pametne pogodbe [24].

Glavni cilj pametnih pogodb je zagotavljanje višje stopnje varnosti, kot jo zagotavljajo tradicionalne papirne pogodbe, in zmanjševanje dodatnih stroškov, povezanih z njihovim izvajanjem [3]. Pogodbe niso samo ključni mehanizem za enkapsuliranje informacij in njihovo preprosto vzdrževanje v omrežju, ampak udeležencem zagotavljajo tudi samodejno izvedbo določenih transakcij. Pametna pogodba lahko npr. samodejno določi stroške pošiljanja predmeta, ki se spreminja glede na prejemnikov naslov in želen datum prejetja pošiljke. S pogoji, o katerih se dogovorita obe stranki pogodbe in jih zapišeta v verigo blokov, se sredstva samodejno spreminjajo glede na prejemnikov naslov in želen datum prejetja [24]. Pametne pogodbe so največkrat uporabljene v povezavi s kriptovalutami, najbolj znan sistem za njihovo izvajanje je platforma Ethereum, ki razvijalcem s programskim jezikom Solidity omogoča njihovo implementacijo [3]. Slika 2.2 prikazuje način uporabniške interakcije s porazdeljeno knjigo.

2.1.4 Soglasje

Soglasje (ang. consensus) je proces sinhronizacije porazdeljene glavne knjige v omrežju, ki zagotavlja, da je glavna knjiga posodobljena samo takrat, ko so transakcije potrjene s strani zaupanja vrednih udeležencev omrežja. Porazdeljena glavna knjiga se posodobi tako, da vsi udeleženci v omrežju izvedejo iste transakcije v enakem vrstnemu redu [24]. Slika 2.3 prikazuje doseženo soglasje v omrežju verige blokov.

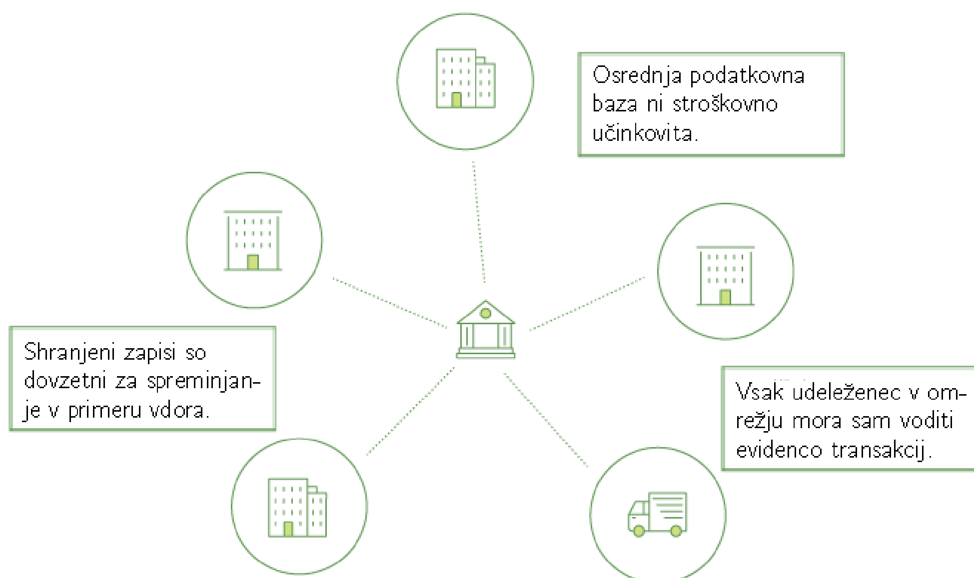


Slika 2.3: Primer doseženega soglasja in sinhronizacije zadnje različice glavne knjige med udeleženci v omrežju [24].

2.2 Uporaba verige blokov

2.2.1 Aktualni sistem beleženja zapisov

Današnja transakcijska omrežja so le malo posodobljene različice poslovnih omrežij, ki so nastala ob začetku shranjevanja in arhiviranja poslovnih zapisov. Člani poslovne mreže med seboj poslujejo preko transakcij, pri čemer vsak sam skrbi za vodenje evidenc o transakcijah. Člani lahko poslujejo s poljubnimi lastninami, vendar morajo ob uspešno zaključenem poslu poskrbeti za ustrezen prenos lastništva iz starega lastnika na novega. V nasprotnem primeru namreč preverjanje izvora in lastništva lastnine ni več mogoče.



Slika 2.4: Današnje transakcijsko omrežje [24].

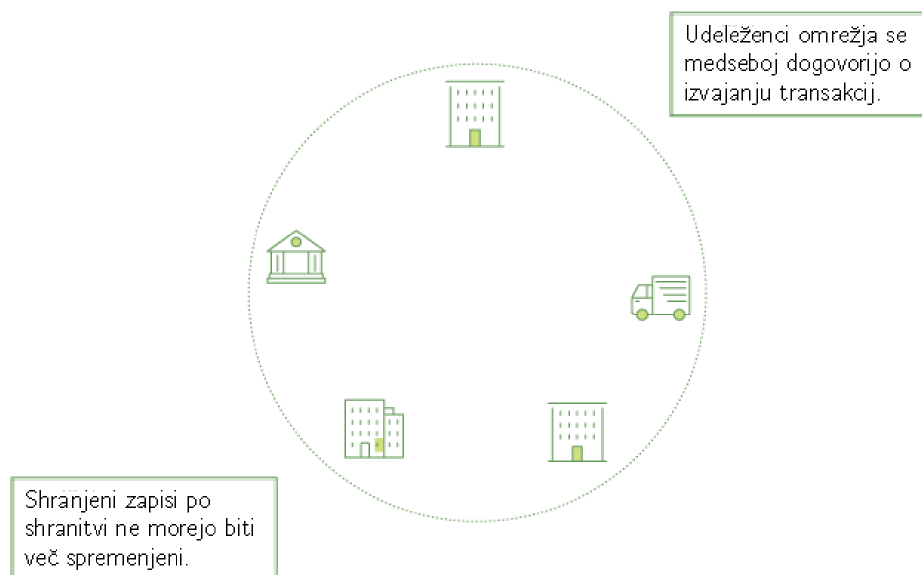
Sodobna tehnologija je ta proces prevzela iz prvih zapisov na kamnitih ploščicah in kasnejših na listih papirja ter poskrbela za njihovo digitalizacijo na diske in oblačne platforme, pri čemer je osnovna struktura procesa ostala

podobna. Ker v digitalnem svetu ne obstajajo enotni sistemi za upravljanje in ugotavljanje uporabniške identitete, je ugotavljanje porekla težavno, ki lahko obenem proces transakcije podaljša za nekaj dni. Pogodbe je kljub digitalizaciji še vedno treba podpisovati in večinoma izvrševati ročno, pri čemer vsaka podatkovna zbirka udeležencev hrani unikatne informacije o pogodbi in transakcijah, kar predstavlja enotno točko neuspeha. Z današnjim transakcijskim omrežjem je zato, kljub potrebi po transparentnosti in zaupanju, nemogoče izdelati sistem zapisov, ki bi obsegal celotno poslovno mrežo [24]. Slika 2.4 opisuje današnje transakcijsko omrežje.

2.2.2 Sistem beleženja zapisov v prihodnosti z uporabo verige blokov

Poslovno omrežje, ki vključuje standarne metode za upravljanje in ugotavljanje uporabniške identitete, izvajanje transakcij in hranjenje podatkov, bi lahko v prihodnosti nadomestilo aktualni sistem beleženja zapisov. Takšno omrežje je omrežje verige blokov.

Za razliko od današnjih sistemov, kjer vsak udeleženec z zasebnimi programi sam skrbi za vzdrževanje zapisov o transakcijah, si lahko v omrežju verige blokov udeleženci med seboj delijo programe za posodabljanje skupne glavne knjige transakcij. Vsak udeleženec v omrežju namreč hrani kopijo glavne knjige transakcij in procesov za njeno posodabljanje. Blokovna omrežja z možnostjo usklajevanja poslovnega omrežja prek deljene glavne knjige transakcij tako zmanjšajo čas, ceno in tveganja, povezana z zasebnim upravljanjem zapisov ter izboljšujejo zaupanje o podatkih med uporabniki [24]. Slika 2.5 opisuje omrežje verige blokov.



Slika 2.5: Omrežje verige blokov [24].

2.3 Zgodovina verige blokov

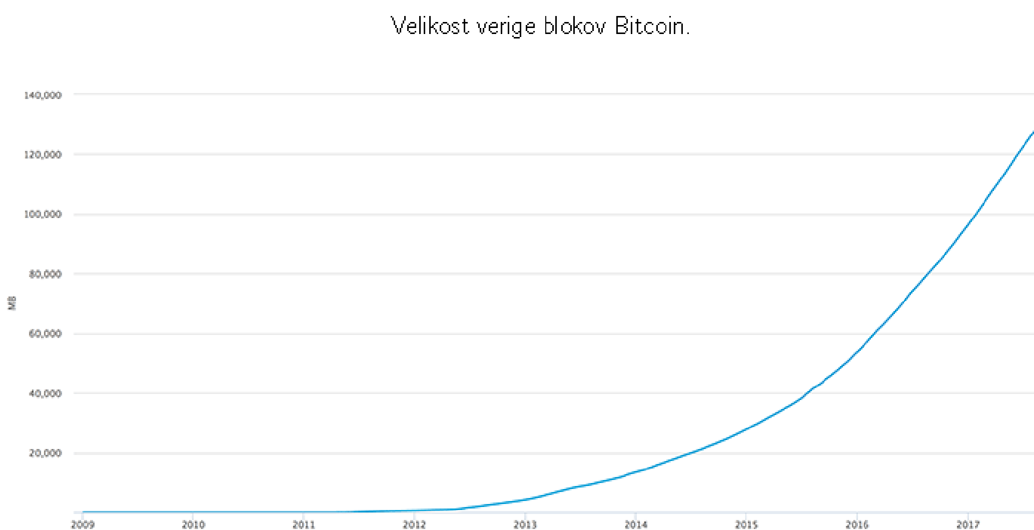
Leta 1991 sta Stuart Haber in W. Scott Stornett kot prva objavila besedilo o kriptografsko zavarovani verigi blokov, v katerem sta predstavila osnovne koncepte verige blokov in njeno teoretično ozadje [17]. Da bi v en blok lahko združili več zapisov, so leta 1992 znanstveniki Bayer, Haber in Stornett predlagali teoretično izboljšavo učinkovitosti verige blokov z vključitvijo transakcij v drevo Merkle [4].

Razvijalec ali skupina razvijalcev pod psevdonimom Satoshi Nakamoto je leta 2008 v članku Bitcoin: A Peer-to-Peer Electronic Cash System [33] opisala aplikacijo verige blokov in jo leto kasneje kot temeljno komponento vključila v digitalno valuto Bitcoin, kjer služi kot javna knjiga vseh transakcij. Distribucija javne knjige transakcij med vse uporabnike digitalne valute

Bitcoin in uporaba porazdeljenega strežnika za časovno žigosanje blokovni bazi omogoča samostojno upravljanje in vzdrževanje. Besedi blok in veriga sta bili v zapisih Satoshija Nakamota iz oktobra leta 2008 zapisani kot ločeni besedi, kasneje sta bili besedi združeni v besedno zvezo veriga blokov.

Avgusta leta 2014 je velikost verige blokov digitalne valute Bitcoin dosegla 20 gigabajtov [34]. Januarja 2015 je njena velikost dosegla 30 gigabajtov, s hitrim naraščanjem priljubljenosti digitalne valute Bitcoin med uporabniki pa se je velikost bazne datoteke v obdobju med januarjem 2016 in januarjem 2017 podvojila s 50 GB na 100 GB [8].

Z uporabo verige blokov je digitalna valuta Bitcoin postala prva, ki odpravlja problem dvojne porabe brez zahteve po zaupanja vrednem skrbniku. Vključitev blokovne baze v digitalno valuto Bitcoin je postala osnova za nadaljnje aplikacije verige blokov [41]. Slika 2.6 prikazuje naraščanje velikosti verige blokov digitalne valute Bitcoin.



Slika 2.6: Zgodovina velikosti verige blokov digitalne valute Bitcoin [8].

2.3.1 Razcepi verige blokov

Med delovanjem verige blokov lahko pride do pričakovane ali nepričakovane delitve verige blokov v dve različni veji. Poznamo dve vrsti delitev:

- **Popolna delitev verige blokov (ang. hard fork)** predstavlja trajno spremembo v verigi blokov, saj predhodno neveljavni bloki na točki delitve lahko postanejo veljavni ali veljavni bloki postanejo neveljavni. Veriga blokov se razcepi v dve popolnoma ločeni verigi, ki od točke razcepa sistem upravljata z uporabo različnih pravil. Delitev za nadaljnje delovanje verige blokov od vozlišč zahteva nadgradnjo na najnovejšo različico protokola, saj v nasprotnem primeru vozlišča s starejšo različico protokola v omrežju ne bodo več prejemala novo nastalih blokov. Ob delitvi tako ena veja verige blokov sledi novemu, posodobljenemu algoritmu, medtem ko druga še naprej uporablja starejšo različico protokola.

  Popolno delitev verige blokov se največkrat uporablja za odpravo pomembnejših varnostnih tveganj v starejših različicah protokola ali za dodajanje novih funkcionalnosti. Najbolj znan primer popolne delitve verige blokov za namene posodobitve protokola je delitev verige blokov Bitcoin, kjer je ob delitvi nastala nova veriga blokov imenovana Bitcoin Cash.

  Popolno delitev verige blokov se lahko uporablja tudi za razveljavitev določenega dela zgodovine transakcij. Najbolj znan primer razveljavitve zgodovine zaradi vdora v sistem in odtujitve sredstev je veriga blokov Ethereum, kjer so si ustvarjalci povrnili ukradena sredstva. Pri tem je nastala nova veriga blokov Ethereum Classic [19].

- **Delna delitev verige blokov (ang. soft fork)** v verigo blokov vpeljuje spremembo programske opreme, kjer samo prej veljavni bloki postanejo neveljavni. Ker vozlišča nove bloke razpoznajo kot veljavne, je delna delitev verige blokov združljiva s preteklo zgodovino blokov.

Takšen način delitve od rudarjev zahteva zgolj posodobitev programske opreme, ki vključuje nova pravila za potrjevanje blokov. Popolna delitev verige blokov na drugi strani od vseh vozlišč v omrežju zahteva nadgradnjo in dogovor o novi različici programske opreme.

Delna delitev verige blokov je bila v platformah Bitcoin in Ethereum v zgodovini že uporabljena z namenom dodajanja novih, nadgrajenih funkcionalnosti [40].

2.4 Veriga blokov 2.0

Termin veriga blokov 2.0 določa novo generacijo programabilne verige blokov, ki vključuje programski jezik, s pomočjo katerega lahko uporabniki pripravijo bolj izpopolnjene pametne pogodbe za dogovor med podpisnikoma pogodbe, npr. o samodejnem plačilu blaga ob potrditvi prejetja pošiljke ali samodejnem izplačilu dividend v primeru doseganja določene stopnje dobička.

S pomočjo verige blokov 2.0 lahko uporabniki vzpostavijo tudi zasebno verigo blokov na lastni infrastrukturi. V takšnem omrežju lahko sami nadzirajo, kdo ima dostop do verige blokov in posledično do podatkov, shranjenih v njej.

Tehnologije verige blokov 2.0 poleg transakcij omogočajo neposredno izmenjavo vrednosti brez posrednikov, kar posameznikom omogoča vstop na trg globalne ekonomije, in pridobivanje podatkov iz zanesljivih virov za nadaljnje proženje transakcij pametnih pogodb. Druga generacija verige blokov dovoljuje pripravo sistemov za hrambo digitalne identitete uporabnika, kar razvijalcem omogoča pripravo sistemov digitalnih volitev in sistemov za prenos lastništva med uporabniki [29].

Poglavje 3

Pregled in primerjava platform verige blokov

Za pregled in primerjavo lastnosti posameznih platform verige blokov smo izbrali tri platforme, ki so med razvijalci in uporabniki v tem trenutku najbolj priljubljene. Njihova primerjava temelji na arhitekturnih zasnovah. Med preostale bolj znane platforme verige blokov spadajo Ripple, Blackcoin, Namecoin, Quorum in druge.

3.1 Bitcoin

Bitcoin (BTC) je platforma verige blokov in kriptovaluta [10], ki jo je leta 2009 kot odprtokodno programsko opremo izdal programer ali skupina programerjev pod psevdonimom Satoshi Nakamoto [11, 48]. Zasnovan je kot sistem vsak z vsakim (ang. peer-to-peer), katerega transakcije potekajo neposredno med uporabniki brez posrednika [10]. Te transakcije preverjajo in potrjujejo omrežna vozlišča, ki ob potrditvi zapis shranijo v javno porazdeljeno knjigo transakcij, imenovano veriga blokov. Ker sistem deluje brez centralnega repozitorija in brez administratorjev sistema, je Bitcoin imenovan tudi prva decentralizirana digitalna valuta [38, 10].

Kriptovaluta Bitcoin je lahko zamenjana za preostale valute, produkte

ali storitve [45]. Končno število vseh kovancev kriptovalute Bitcoin znaša 21.000.000 in bo zaradi nenehnega spreminjana težavnostnega praga rudarjenja in zmanjševanja nagrade za rudarje predvidoma doseženo leta 2140 [49]. Podrobnejše specifikacije platforme Bitcoin so navedene v tabeli 3.1.

Kriptovaluta:	BTC
Leto objave:	2009
Način potrjevanja blokov:	dokazilo o delu
Čas potrditve bloka:	10 minut
Število transakcij:	povprečno 11.407 na uro [7]
Cena transakcije:	povprečno 7,35 USD [7]
Pametne pogodbe:	ne

Tabela 3.1: Specifikacije platforme Bitcoin.

3.1.1 Omrežje

Omrežje Bitcoin je javno, zato vsa vozlišča v omrežju sledijo spodnjim pravilom:

1. Vsaka transakcija je razposlana vsem vozliščem.
2. Vsako vozlišče samo zbira transakcije za združitev v blok.
3. Vsako vozlišče z zbranimi transakcijami poskuša ustvariti nov blok tako, da samo prične z izračunavanjem števila nonce.
4. Ko vozlišče pravilno izračuna število nonce, blok razpošlje vsem preostalim vozliščem.
5. Vozlišče prejeti blok sprejme, če ta vsebuje same veljavne transakcije, ki ne vključujejo že porabljenih sredstev.
6. Vozlišče potrdi sprejetje prejetega bloka z nadaljnjim postopkom ustvarjanja novega bloka v verigi blokov, v katerega vključi enolično zgoščeno vrednost predhodno sprejetega bloka.

Vozlišča v omrežju kot veljavno in aktualno verigo blokov vedno izberejo najdaljše zaporedje blokov, ki ga podaljšujejo z dodajanjem novih blokov. Če dve vozlišči hkrati odkrijeta število nonce in blok razpošljeta preostalim vozliščem, bo neodvisno tretje vozlišče za nadaljnje ustvarjanje blokov izbralo prvi prejeti blok transakcij. Pri tem bo poskrbelo za shranitev preostalih vej verige, če se bo v prihodnosti izkazalo, da bo ena izmed vej postala najdaljša in hkrati aktualna veriga blokov. Vozlišča, ki so bloke dodajala v eno izmed vej verige blokov, bodo ob odkritju nove, daljše verige blokov, s svojim delom nadaljevala na tej verigi. Če vozlišče spozna, da mu ob prejetju novega bloka v zgodovini verige blokov manjkajo vmesni bloki, bo od ostalih vozlišč zahtevalo prenos manjkajočih členov in na ta način poskrbelo za sinhronizacijo verige blokov [33].

3.1.2 Vrste transakcij

Ker je omrežje Bitcoin javno, so vse transakcije v vsakem trenutku vidne vsem uporabnikom omrežja kot tudi naključnim osebam, ki do podatkov dostopajo prek različnih spletnih raziskovalcev verige blokov - takšno najbolj poznano odprtokodno spletno orodje je Blockexplorer [9].

Posledično je vsaka transakcija v omrežju javno dostopna, vendar še vedno anonimna, saj se uporabniki znotraj omrežja ne identificirajo z uporabniškimi imeni, ampak s pomočjo naslovov, predstavljenih kot javni ključi. Za povečanje stopnje zasebnosti lahko uporabniki pred vsako transakcijo ustvarijo nov naslov in tako poskrbijo za novo povezavo med njihovim naslovom in identiteto v omrežju [31, 39].

3.1.3 Način potrjevanja transakcij

Platforma Bitcoin za potrjevanje predlaganih zapisov za shranitev v verigo blokov uporablja sistem rudarjenja in t. i. način dokazovanja skladnosti zapisa z dokazilom o delu ali POW (ang. proof-of-work). Podatke, shranjene v verigi blokov, ki uporablja sistem dokazila o delu, je izjemno težko spre-

meniti, saj bi moral napadalec spremeniti vse naslednike bloka, ki ga želi spremeniti [18]. Ker ves čas nastajajo tudi novi bloki, zahtevnost spreminjanja podatkov narašča z nastajanjem novih blokov in povečevanjem števila potrditev preteklih blokov [43].

3.1.3.1 Dokazilo o delu

Dokazilo o delu je številčni podatek, ki ga je z današnjo računsko močjo računalnikov izjemno težko izračunati, vendar po drugi strani omogoča preprosto preverjanje njegove pravilnosti. Izračun števila je proces, v katerem imajo računalniki majhno verjetnost za pridobitev pravilnega števila, zato ta proces v povprečju zahteva veliko število poskusov, preden se generira veljaven dokaz o delu.

Bitcoin za generiranje blokov in dokazovanje njihove integritete uporablja algoritem Hashcash z dvojno zgoščevalno funkcijo SHA256, ki od uporabnikov omrežja zahteva, da za potrditev bloka zapisov izračunajo število, imenovano nonce, ki enolično določa podatke v bloku [6].

Algoritem Hashcash implementira enosmerno dvojno zgoščevalno funkcijo SHA256, definirano kot $y = H(x)$, ki uporabniku omogoča enostaven izračun števila y in zelo težaven izračun števila x, ki bi ustrezal izbranemu številu y. Enosmerna funkcija tako zadostuje vsem trem želenim kriterijem odpornosti dobrih zgoščevalnih funkcij [20].

3.1.3.2 Rudarjenje

Rudarjenje je način zbiranja novo nastalih transakcij in tvorjenje bloka z vsemi zbranimi, še nepotrjenimi, transakcijami, s katerim rudarji potrjujejo bloke in transakcije ter na ta način poskrbijo, da je veriga blokov v vsakem trenutku dosledna in popolna [43].

Da bi udeleženci omrežja sprejeli blok, morajo rudarji poskrbeti za izračun števila nonce. Vsak rudar mora preveriti veliko različnih vrednosti za število, dokler ne doseže težavnostnega praga ali najde pravega števila. Rudar, ki odkrije pravo število, je za njen izračun nagrajen z vnaprej določeno količino

kriptokovancev. Težavnostni prag dela se vsakih 2016 blokov prilagodi glede na računsko moč omrežja. Povprečni čas potrditve novega bloka transakcij se na ta način ohranja pri desetih minutah [1].

Vsak blok prek zgoščene vrednosti, kodirane s SHA-256 zgoščevalnim algoritmom, vsebuje povezavo na prejšnji blok [1]. Tako se s povezavami med bloki tvori veriga blokov. S časom potrjeni bloki v omrežju prejmejo nove potrditve o skladnosti zapisanih podatkov, zato so starejši bloki z večjim številom potrditev med uporabniki deležni višje stopnje zaupanja o celovitosti in veljavnosti shranjenih podatkov [6].

3.1.4 Podpora za pametne pogodbe

Platforma Bitcoin ne ponuja neposredne podpore za pametne pogodbe. Te so podprte z uporabo zunanjih odportokodnih rešitev, kot je npr. platforma Rootstock [37], ki pametnih pogodb ne namesti neposredno v verigo blokov Bitcoin, ampak poskrbi za vzdrževanje lastne verige blokov s pametnimi pogodbami. Uporabnik s sredstvi Bitcoin upravlja s pomočjo posebnega dvosmernega priključka.

3.2 Ethereum

Ethereum je kriptovaluta (oznaka ETH) in porazdeljena računalniška platforma, ki jo je leta 2013 kot odportokodni projekt oblikoval raziskovalec kriptovalut in programer Vitalik Buterin [44]. Temelji na tehnologiji verige blokov in vključuje funkcionalnost pametnih pogodb. Uporabnikom prek virtualnega stroja Ethereum omogoča izvajanje programov in skript na vozliščih porazdeljenega omrežja. Platforma v zameno za ponujeno računsko moč vozliščem nudi kompenzacijo v obliki digitalne kriptovalute Ether, s katero lahko uporabniki trgujejo in jo prenašajo med računi [15]. Podrobnejše specifikacije platforme Ethereum so navedene v tabeli 3.2.

Kriptovaluta:	ETH
Leto objave:	2014
Način potrjevanja blokov:	dokazilo o delu
Čas potrditve bloka:	15 sekund
Število transakcij:	povprečno 16.487 na uro [7]
Cena transakcije:	povprečno 0,50 USD [7]
Pametne pogodbe:	da
Programski jezik pametnih pogodb:	Solidity

Tabela 3.2: Specifikacije platforme Ethereum.

3.2.1 Omrežje

Glavno omrežje platforme Ethereum je tako kot omrežje verige blokov Bitcoin javno in v vsakem trenutku dostopno vsem uporabnikom.

Platforma uporabnikom poleg javnega omrežja ponuja tudi vzpostavitev zasebnega omrežja, katerega vozlišča niso povezana na javno omrežje. Za njegovo vzpostavitev mora upravitelj omrežja poskrbeti sam, pri čemer mora v omrežju definirati tudi vozlišče za začetni zagon, prek katerega se med seboj povezujejo preostala vozlišča in si izmenjujejo transakcije. Ker Ethereum za potrjevanje novih blokov uporablja sistem rudarjenja, mora znotraj omrežja delovati tudi manjše število rudarjev. Za potrjevanje ni potrebna tako velika računska moč kot za rudarjenje v javnem omrežju, saj upravitelj omrežja ob konfiguraciji omrežja poskrbi za ustrezno nastavitev težavnostnega praga, ki ne narašča tako hitro kot v javnem omrežju [13].

3.2.2 Vrste transakcij

Transakcije znotraj javnega omrežja platforme Ethereum so, enako kot pri platformi Bitcoin, v vsakem trenutku vidne vsem uporabnikom v omrežju. Naključne osebe, ki bi rade dostopale do podatkov v javnem omrežju, lahko do njih dostopajo prek različnih spletnih raziskovalcev verige blokov - najbolj

poznano odprtokodno spletno orodje je Etherscan [16]. Uporabniške identitete so, tako kot pri Bitcoinu, predstavljene s pomočjo naslovov, zato so vsi uporabniki deležni enake mere anonimnosti.

Zasebno omrežje uporabnikom zagotavlja dodatno stopnjo varnosti zapisanih podatkov, saj lahko do transakcij dostopajo zgolj uporabniki, ki so pridruženi omrežju. Ti so del omrežja postali ob njegovi vzpostavitvi [13].

3.2.3 Način potrjevanja transakcij

Platforma za potrjevanje blokov, podobno kot platforma Bitcoin, uporablja sistem rudarjenja in t. i. dokazilo o delu.

3.2.3.1 Dokazilo o delu

Ethereum za generiranje blokov in dokazovanje njihove integritete uporablja algoritem Ethash, ki je bil zasnovan posebej za to platformo. Algoritem v splošnem deluje v skladu s spodnjimi smernicami:

1. Za vsak nov blok na podlagi pregleda vseh zapisanih podatkov v glavah blokov do trenutnega bloka algoritem izračuna seme (ang. seed).

2. Iz semena algoritem izračuna psevdonaključen 16 MB pomnilnik, ki ga hranijo odjemalci z minimalno različico platforme Ethereuma.

3. Na podlagi pomnilnika algoritem poskrbi za generiranje podatkovne zbirke velikosti 1 GB, kjer je vsak element odvisen od majhnega števila podatkov v pomnilniku. Podatkovno zbirko hranijo odjemalci s polno različico platforme Ethereum. Ta sčasoma narašča linearno [12].

3.2.3.2 Rudarjenje

Rudarjenje na platformi Ethereum deluje tako kot na platformi Bitcoin in vključuje združevanje naključnih delov podatkovne zbirke, hranjene na odjemalcu, v enolično zgoščeno vrednost. Podatkovna zbirka se posodablja na

vsakih 30.000 blokov, zato velika večina rudarjev ves čas zgolj bere podatke iz zbirke in je nikoli ne spreminja.

Preverjanje podatkov, zapisanih v blok, je preprosto, saj algoritem na podlagi zgoščene vrednosti podatkov in hranjenega pomnilnika izračuna dele podatkovne zbirke [12].

Ker rudarji za izračun števila nonce porabljajo velike količine električne energije, npr. za rudarjanje kriptovalut Bitcoin in Ethereum se dnevno porabi za miljon dolarjev električne energije [14], razvijalci platforme Ethereum pripravljajo nadgradnjo platforme s prehodom iz dokazila o delu (proof-of-work) na dokazilo o deležu (proof-of-stake). V sistemu, ki uporablja dokazilo o deležu, je namreč rudar, ki potrdi in ustvari nov blok, izbran naključno glede na količino kovancev Ether v posamezni digitalni denarnici, ki jo ima v lasti, pri čemer je nagrajen zgolj s provizijo transakcij, za razliko od sistema z dokazilom o delu, kjer rudar prejme vnaprej določeno količino kriptokovancev [35].

3.2.4 Podpora za pametne pogodbe

Pametne pogodbe so v platformi Ethereum podprte s pomočjo programskega jezika Solidity. Predstavljajo programe, ki so prevedeni v bitno kodo. Razbrati in izvajati jo zna virtualni stroj Etherem - Ethereum Virtual Machine (EVM). Vsako vozlišče nato samo poskrbi za izvajanje pametne pogodbe znotraj lastnega EVM-ja v skladu s pravili, ki jih je programer predhodno vključil v pametno pogodbo.

Pametne pogodbe v platformi Ethereum lahko:

- Služijo kot večpodpisne funkcije, ki uporabnikom zagotavljajo, da so sredstva porabljena le, če se s tem strinja določen delež ljudi.

- Upravljajo dogovore med uporabniki, npr. dogovore ob sklenitvi zavarovanja med podjetjem in zavarovancem.

- Delujejo kot deljene programske knjižnice, ki drugim pametnim pogodbam zagotavljajo uporabo implementiranih lastnih funkcij.

- Hranijo podatke o aplikaciji, npr. podatke o registraciji domene ali članske evidence uporabnikov [21, 22].

3.3 Hyperledger Fabric

Hyperledger Fabric je platforma verige blokov, ki ga je razvila neprofitna organizacija Linux Foundation. Platforma omogoča razvoj modularnih porazdeljenih rešitev, z visoko stopnjo varnosti, prožnosti, prilagodljivosti in razširljivosti. Zasnovana je tako, da podpira implementacije različnih komponent, kot je npr. lastna implementacija soglasja o potrjevanju blokov. Hyperledger Fabric uporablja tehnologijo vsebnikov Docker za gostovanje pametnih pogodb, imenovanih verižne kode, ki vsebujejo aplikacijsko logiko sistema. Podrobnejše specifikacije platforme Hyperledger Fabric so navedene v tabeli 3.3.

Kriptovaluta:	brez
Leto objave:	2016
Način potrjevanja zapisov:	dokazilo o delu
Število transakcij:	približno 10.000 na uro (točno število ni navedeno)
Cena transakcije:	brezplačno
Pametne pogodbe:	da
Programski jezik pametnih pogodb:	Go

Tabela 3.3: Specifikacije platforme Hyperledger Fabric.

3.3.1 Omrežje

Platforma Hyperledger Fabric uporabnikom omogoča vzpostavitev lastnega zasebnega omrežja verige blokov, v katerem je upravitelj omrežja odgovoren za njegovo začetno konfiguracijo in nadaljnje vzdrževanje. Končnim uporabnikom to omogoča prilagajanje zmogljivosti omrežja glede na lastne potrebe

in gostovanje omrežja na lastni infrastrukturi ter jim obenem zagotavlja dodatno stopnjo varnosti.

Vsi postopki povezani s konfiguracijo in vzpostavitvijo omrežja so opisani v poglavjih 5.2.3 in 5.2.3.

3.3.2 Vrste transakcij

Vsak uporabnik platforme Hyperledger Fabric mora za dokazovanje lastne identitete in dostop do verige blokov imeti v lasti digitalni certifikat, s katerim izkazuje svojo identiteto. Uporabniška identiteta se uporablja pri proženju transakcije, zato je uporabnik v omrežju izpostavljen kot njen avtor.

Hyperledger Fabric podpira dve vrsti transakcij - javne in zasebne transakcije. Ker platforma v glavno knjigo shrani celotno izvorno transakcijo, je treba razlikovati med obema vrstama transakcij. Vsebina javnih transakcij je dostopna vsem uporabnkom omrežja, medtem ko je vsebina zaupnih transakcij šifrirana z zasebnimi ključi, ki so poznani zgolj lastniku transakcije, vozliščem, ki potrjujejo transakcijo, in pooblaščenim revizorjem [25]. Uporabnik lahko tako sam, glede na zaupnost podatkov, ki jih želi zapisati v verigo blokov, izbira med vrsto transakcije, ki jo želi izvesti.

3.3.3 Način potrjevanja transakcij

Platforma Hyperledger Fabric ne uporablja principa rudarjenja blokov in dokazila o delu. Za razliko od platform Bitcoin in Ethereum namreč nima podprte kriptovalute, s katero bi nagrajevala rudarje za njihovo opravljeno delo. Hyperledger Fabric za doseganje soglasja uporablja praktično bizantinsko odpornost na napake (PBFT), medtem ko za generiranje novih blokov uporablja poseben, spodaj opisan, postopek [25].

Ker mora imeti vsak uporabnik v lasti digitalni certifikat za izkazovanje identitete, lahko platforma namesto računsko zahtevnejših algoritmov za potrjevanje blokov, kot sta dokazilo o delu (proof-of-work) in dokazilo o deležu (proof-of-stake), uporablja praktično bizantinsko odpornost na na-

pake [23, 24, 27].

3.3.3.1 Praktična bizantinska odpornost na napake

Validacija transakcij se zgodi z večkratnim izvajanjem transakcij znotraj verižne kode ob predpostavki soglasja PBFT, torej da med n vozlišči vrstnika obstaja največ $f < n/3$ vozlišč, ki se obnašajo poljubno - torej lahko sporočajo resnične podatke ali lažne podatke o rezultatih transakcije, medtem ko preostala vozlišča poskrbijo za pravilno izvajanje transakcij verižne kode.

Pri uporabi praktične bizantinske odpornosti na napake je pomembno, da so transakcije verižne kode deterministične, saj v nasprotnem primeru enak vrstni red izvajanja transakcij na posameznih vozliščih ni zagotovljen [2].

3.3.3.2 Postopek generiranja blokov

Namesto rudarjenja platforma za generiranje blokov uporablja sledeč postopek:

1. Transakcija je posredovana enemu izmed zaupanja vrednih vozlišč vrstnika.

2. Vozlišče vrstnika poskrbi za razpošiljanje prejete transakcije vsem ostalim vozliščem vrstnika.

3. Vozlišča vrstnika soglasje dosežejo z uporabo algoritma, ki vključuje postopek praktične bizantinske odpornosti na napake.

4. Vsako vozlišče vrstnika je samo zadolženo za izvajanje transakcij in za grajenje blokov z izvedenimi transakcijami.

Zaradi determinističnega izvajanja transakcij in vnaprej določenega števila transakcij znotraj bloka bo število generiranih blokov na vseh vozliščih enako [25].

3.3.4 Podpora za pametne pogodbe

Veriga blokov Hyperleger Fabric podporo za pametne pogodbe vpeljuje prek t. i. verižne kode, ki uporabniku omogoča manipulacijo z zapisanimi podatki. Podrobnejši opis verižne kode se nahaja v poglavju 4.5.

3.4 Primerjava platform verige blokov

Primerjava platform verige blokov vključuje primerjavo opisanih platform verige blokov glede na različne kriterije, pomembne za končnega uporabnika. Ključni kriteriji so primerjave specifikacij, vrste omrežja, vrste transakcij, načinov potrjevanja transakcij in podpore za pametne pogodbe.

3.4.1 Primerjava specifikacij

Glede na performančne sposobnosti in želje končnega uporabnika je primerjava specifikacij posamenih platform zelo subjektivne narave. Vse platforme namreč omogočajo hitro procesiranje velikega števila transakcij.

Platforma Hyperledger Fabric se s platformama Bitcoin in Ethereum izjemoma razlikuje v ceni transakcije. Proženje transakcij je prek platforme Hyperledger Fabric namreč brezplačno, medtem ko mora uporabnik ob izvedbi transakcije na platformu Bitcoin ali Ethereum plačati določeno ceno, ki predstavlja del nagrade za rudarje blokov.

3.4.2 Primerjava omrežja

Platforma Hyperledger Fabric za razliko od platform Bitcoin in Ethereum uporabniku omogoča zgolj pripravo zasebnega omrežja, medtem ko preostali dve platformi uporabnikom ponujata popolnoma javno omrežje. To ima svoje prednosti in slabosti. Glavna prednost je v zaupnosti shranjenih podatkov, saj lahko uporabniki platforme Hyperledger Fabric z dostopom do omrežja upravljajo vidnost informacije, medtem ko so v javnem omrežju vse informacije v vsakem trenutku vidne vsem uporabnikom, kot tudi naključnim

osebam. Glavna slabost zasebnega omrežja pa je v hranjenju podatkov, saj je takšna vrsta omrežja omejena, zato so podatki podvojeni zgolj na omejenem številu naprav, medtem ko so v javnem omrežju podatki podvojeni na vseh vključenih napravah.

3.4.3 Primerjava vrste transakcij

Vse opisane platforme podpirajo izvajanje javnih transakcij, katerih rezultati so vidni vsem uporabnikom omrežja. Platformi Ethereum in Hyperledger Fabric dodatno omogočata izvajanje zasebnih transakcij, katerih razultati so vidni zgolj izbranim deležnikom. Glavna razlika med platformama Bitcoin in Ethereum na eni strani in platformo Hyperledger Fabric na drugi strani je v poznavanju klicatelja transakcije. Uporabniki platforme Hyperledger Fabric morajo namreč za proženje transakcij imeti digitalni certifikat, s katerim izkazujejo lastno identiteto, medtem ko uporabniki v platformah Bitcoin in Ethereum niso neposredno izpostavljeni ob proženju transakcij, saj se v omrežju identificirajo z javnimi ključi, ki niso povezani z njihovimi identitetami.

3.4.4 Primerjava načinov potrjevanja transakcij

Platformi Bitcoin in Ethereum za potrjevanje transakcij uporabljata sistem dokazila o delu, ki od rudarjev zahteva velike količine električne energije, saj morajo ti preveriti veliko število kombinacij za ustrezno generiranje števila nonce. Platforma Hyperledger Fabric za potrjevanje transakcij uporablja praktično bizantinsko odpornost na napake. Takšen način potrjevanja transakcije ne terja velikih količin električne energije, saj v sistemu ne obstajajo rudarji, ki bi potrjevali bloke, ampak je privzeto, da določen delež vključenih vozlišč zagotovo sporoča resnične podatke o rezultatih transakcije.

3.4.5 Primerjava podpore za pametne pogodbe

Platforma Bitcoin za razliko od platform Ethereum in Hyperledger Fabric privzeto ne podpira pametnih pogodb. Tako Ethereum kot Hyperledger Fabric podporo za pametne pogodbe vpeljujeta prek dveh različnih sistemov. Ethereum uporabnikom omogoča zapis poslovne logike s pomočjo programskega jezika Solidity, ki je bil razvit in prilagojen za izvajanje na Ethereum virtualnem stroju. Hyperledger Fabric na drugi strani uporabnikom omogoča pripravo pametnih podgodb, imenovanih verižna koda, zapisanih v programskem jeziku Go. Obe platformi s podprtimi jeziki za pametne pogodbe omogočata veliko podobnih funkcionalnosti, zato je končna izbira glede izbire platforme za gostovanje pametnih pogodb odvisna od uporabniških želja.

3.4.6 Tabelarična primerjava platform verige blokov

Platforma	Kriptovaluta	Omrežje	Transakcije	Soglasje	Pametne pogodbe
Bitcoin	bitcoin	javno	anonimne	POW	ne
Ethereum	ether	javno ali zasebno	anonimne ali zasebne	POW	da, prek Solidity
Hyperledger Fabric	brez	zasebno	javne ali zasebne	PBFT	da, prek verižne kode

Tabela 3.4: Primerjava platform verige blokov.

Kot je razvidno iz tabele 3.4, se posamezne platforme verige blokov med seboj razlikujejo v večini kriterijev. Izbira platforme je odvisna od končnega uporabnika, ki bi želel s pomočjo verige blokov implementirati aplikacijo. Pri izbiri mora biti uporabnik pozoren na želene performančne sposobnosti aplikacije, razsežnost infrastrukture, ki mu je na voljo, in na način implementacije poslovne logike aplikacije s pomočjo pametnih pogodb.

V procesu iskanja in primerjave je bilo pregledanih veliko različnih platform, ki se zelo malo razlikujejo v primerjavi z zgoraj opisanimi. Izmed treh

najbolj priljubljenih platform, ki smo jih opisali, smo za nadaljnji razvoj praktičnega primera izbrali platformo Hyperledger Fabric, saj ta končnemu uporabniku zagotavlja popolno avtonomijo nad konfiguracijo in pripravo zasebnega omrežja glede na dostopno infrastrukturo, enostavno preverjanje identitete udeležencev v omrežju, komunikacijo z verigo blokov prek povezave zaščitene s TLS in enostavno implementacijo pametnih pogodb s pomočjo programskega jezika Go.

Poglavje 4

Hyperledger Fabric

4.1 Arhitektura

Veriga blokov Hyperledger Fabric je porazdeljen sistem, sestavljen iz številnih vozlišč, ki komunicirajo eden z drugim. Skrbi za shranjevanje podatkov o stanju in porazdeljeni glavni knjigi ter izvajanje transakcij in programov, imenovanih verižna koda (ang. chaincode). Verižna koda je osrednji element verige blokov, saj skrbi za proženje in izvajanje transakcij. Transakcije morajo biti potrjene in samo potrjene transakcije so lahko izvedene in vplivajo na stanje podatkov v verigi blokov. V verigi blokov lahko obstajajo posebne vrste verižnih kod, imenovane sistemske verižne kode, ki vključujejo parametre in funkcije za upravljanje stanja [24].

4.1.1 Transakcije

Hyperledger Fabric omogoča proženje dveh vrst transakcij:

- **Namestitvene transakcije** kot parameter sprejmejo program za izvajanje in ustvarijo novo verižno kodo. Ob uspešni namestitvi je ustvarjena verižna koda, nameščena v verigo blokov in pripravljena na uporabo.

- **Klicne transakcije**, ki izvajajo operacije na predhodno nameščeni

verižni kodi. Klicna transakcija se sklicuje na obstoječo verižno kodo in proži eno izmed njenih funkcij. Te funkcije lahko vključujejo tudi spreminjanje trenutnega stanja in vračanje ustreznega odgovora.

Namestitvene transakcije predstavljajo poseben primer klicnih transakcij, kjer namestitvena transakcija ustreza klicni transakciji, ki proži ustrezno funkcijo za pripravo in namestitev nove verižne kode [24].

4.1.2 Podatkovni strukturi verige blokov

4.1.2.1 Stanje

Zadnje stanje verige blokov je predstavljeno kot verzioniran zapis oblike ključ/vrednost, kjer ključ predstavlja ime stanja in vrednost poljuben zapis blob. Ti podatki so upravljani prek verižne kode, ki je nameščena v verigi blokov in vključuje operaciji za pridobivanje (get) in posodabljanje (put) nad KVS. Izvajanje operacij nad KVS je dovoljeno samo vozliščem vrstnika. Stanje je v verigi blokov hranjeno trajno. Vse prožene transakcije, povezane s posodabljanjem in spreminjanjem stanja, so dodatno zabeležene v posebno datoteko.

Formalno je stanje s element preslikave opisane z enačbo 4.1:

$$K \rightarrow (V \times N), \quad (4.1)$$

kjer je:

- K množica ključev,
- V množica vrednosti,
- N neskončna urejena množica števil različic. Injektivna funkcija $next : N \rightarrow N$ sprejme element N in vrne naslednjo številko različice.

Množici V in N vsebujeta tudi poseben element, imenovan $\backslash bot$, ki v množici N predstavlja najmanjši element. Začetne vrednosti vseh ključev so privzeto nastavljene na $(\backslash bot, \backslash bot)$.

Naj bo stanje s v odvisnosti od ključa k enako $s(k) = (v, ver)$, kjer v predstavlja vrednost $s(k).value$ in ver vrednost $s(k).version$. Operacije nad KVS so definirane kot:

- $put(k, v)$ kot argumenta sprejme ključ in novo vrednost za spremembo stanja. Ključ k je element množice K in v poljubna vrednost iz množice V. Funkcija posodobi stanje s v novo stanje s' tako, da je $s'(k) = (v, next(s(k).version))$, pri čemer velja $s'(k') = s(k')$ za vsak $k' \neq k$.

- $get(k)$ vrne vrednost $s(k)$, ki predstavlja par $(vrednosti, različica)$.

Razvijalci lahko namesto privzete podatkovne baze za hranjenje stanja verige blokov, goleveldb, uporabijo podatkovno bazo CouchDB. CouchDB podatke hrani v obliki JSON in poleg privzetih funkcionalnosti, ki jih podpira tudi podatkovna baza goleveldb, vključuje tudi možnost izvajanja naprednejših poizvedb o stanju verige blokov.

4.1.2.2 Glavna knjiga

Glavna knjiga hrani zgodovino vseh poskusov sprememb stanja med delovanjem sistema. Uspešne spremembe stanja so posledice izvedb potrjenih transakcij, medtem ko so neuspešni poskusi spreminjanja stanja posledica neveljavnih transakcij. Glavno knjigo ustvarijo deljene storitve za naročanje, v lasti naročnika, kot kronološko urejeno zaporedje blokov, kodirano v verigo enoličnih zgoščenih vrednosti, kjer vsak blok vsebuje zbirko kronološko urejenih veljavnih in neveljavnih transakcij. Takšen način beleženja podatkov v sistemu vzpostavi popolno kronološko urejenost vseh transakcij.

Porazdeljena je med vsa vozlišča vrstnikov in po potrebi tudi med podskupine vozlišč naročnika. Glavna knjiga, deljena med vrstniki, se razlikuje od glavne knjige, deljene med naročniki v lokalno vzdrževani bitni maski, ki razlikuje veljavne transakcije od neveljavnih.

Vrstniki lahko ponovno izvedejo vse transakcije iz zgodovine transakcij in na ta način rekonstruirajo zadnje shranjeno stanje. Podatkovna struk-

tura stanja, opisana v poglavju 4.1.2, zato spada med opcijske podatkovne strukture [24].

4.1.3 Vozlišča verige blokov

Vozlišča so komunikacijske entitete verige blokov. Ker lahko na enem strežniku hkrati teče več vrst vozlišč, je vsako izmed njih predstavljeno kot logična enota. Povezana so s subjekti, ki jih upravljajo, in med seboj združena v zaupanja vredne skupine.

Veriga blokov Hyperledger Fabric pozna tri vrste vozlišč: odjemalec, vrstnik in naročnik. Opisali jih bomo v nadaljevanju.

4.1.3.1 Odjemalec

Odjemalec je vozlišče, ki vozlišču vrstnika z dodatnimi pravicami za odobravanje transakcij predloži poziv za izvedbo transakcije in poskrbi za prenos predloga transakcije do deljenih storitev za naročanje.

Predstavlja entiteto, s katero upravlja končni uporabnik. Da ima odjemalec omogočeno komunikacijo z verigo blokov, mora sam poskrbeti za povezavo s poljubnim vozliščem vrstnika. Po vzpostavitvi povezave lahko odjemalec ustvari in proži transakcije. Vozlišče odjemalca komunicira z vozlišči vrstnika, kot tudi z vozlišči naročnika.

Odjemalec se mora pred pričetkom oddajanja sporočila povezati na kanal in šele nato lahko prične z njegovim oddajanjem. Oddano sporočilo je tako posredovano vsem vozliščem vrstnika. Kanal obenem podpira atomsko dostavo vseh sporočil, t. j. dostavo sporočil vsem povezanim vrstnikom v enakem vrstnem redu, kot so bila ta posredovana. Sporočila, oddana s strani odjemalca, vsebujejo predloge transakcij za upravljanje stanja verige blokov.

4.1.3.2 Vrstnik

Vrstnik je vozlišče, ki potrdi transakcijo, vzdržuje stanje verige blokov in poskrbi za izdelavo kopije glavne knjige.

S strani storitev naročnika v obliki blokov prejema zahteve po posodobitvi stanja verige blokov in skrbi za vzdrževanje njegove zadnje različice.

Vozlišča vrstnika lahko zavzamejo tudi posebno vlogo za odobravanje transakcij glede na potrebe verižne kode po potrjevanju transakcij pred njihovo izvedbo. Vsaka verižna koda lahko namreč skupini vrstnikov z dodatnimi pravicami za odobravanje določi politiko odobravanja transakcij, ki določa potrebne in zadostne pogoje za potrditev oz. zavrnitev transakcije. V primeru, da namestitev verižne kode poteka prek namestitvene transakcije, je namestitvena politika odobravanja določena v sistemu verige blokov.

Vrstniki se na kanal povezujejo prek vmesnika, ki ga ponuja storitev naročanja in vključuje 2 osnovni asinhroni operaciji:

- Operacijo *broadcast* (*blob*), ki jo odjemalec pokliče ob oddajanju blob sporočila prek komunikacijskega kanala.

- Operacijo *deliver* ($seqno, prevhash, blob$), ki jo proži vrstnik preko storitve za naročanje. Ta odjemalcu dostavi sporočilo blob, ki vsebuje tudi nenegativno zaporedno številko sporočila in enolično zgoščeno vrednost zadnjega dostavljenega sporočila blob. Operacija predstavlja odgovor ob zaključku dogodka storitve za naročanje.

4.1.3.3 Naročnik

Naročnik je vozlišče, na katerem tečejo komunikacijske storitve, ki zagotavljajo zanesljivo dostavo sporočil.

Naročniki in njihove storitve za naročanje tvorijo komunikacijsko strukturo, ki uporabnikom zagotavlja zanesljivo dostavo sporočil. Storitve za naročanje so lahko implementirane kot centralizirane storitve (za razvoj in testiranje) ali kot decentralizirani protokoli, namenjeni različnim vrstam omrežja in različnim modelom vozlišč. Storitve odjemalcem in vrstnikom ponujajo deljen komunikacijski kanal, ki vključuje tudi storitve za oddajanje sporočil vsem vrstnikom.

Storitve naročanja lahko podpirajo več kanalov, podobno kot tema - sistem za objavljanje in naročanje na sporočila. Kanale si lahko predstavljamo tudi kot particije, kjer povezani odjemalci oddajajo in prejemajo sporočila. Vsak kanal oz. particija se ne zaveda obstoja drugih kanalov, kar odjemalcem omogoča povezavo na več različnih kanalov.

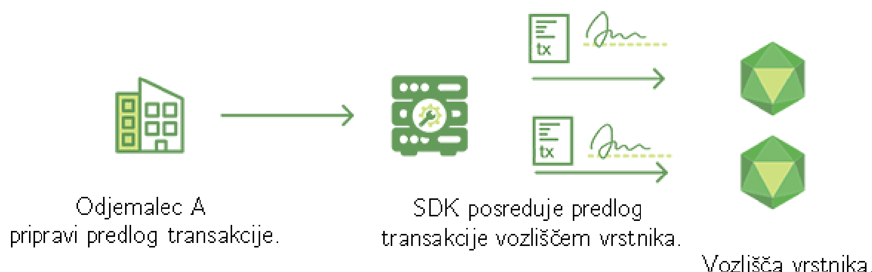
Glavna knjiga hrani vse odgovore storitev za naročanje, zato si jo lahko predstavljamo kot zaporedje izhodov operacij $deliver(seqno, prevhash, blob)$, ki tvori verigo zgoščenih vrednosti. Zaradi učinkovitosti storitve naročanja poskrbijo za združitev posameznih odgovorov operacij $deliver(seqno, prevhash, blob)$ v blok, ki je nato poslan kot odgovor ene operacije deliver. Število odgovorov v posameznem bloku je določeno dinamično glede na implementacijo storitev za naročanja [24].

4.2 Potek transakcije

Scenarij mehanizma transakcij vključuje dva odjemalca, A in B, ki kupujeta in prodajata redkvice. Vsakemu izmed odjemalcev pripada eno vozlišče vrstnika, prek katerega pošiljata transakcije in komunicirata z glavno knjigo. Odjemalcu A pripada vozlišče vrstnikA in odjemalcu B vozlišče vrstnikB.

1. Odjemalec A pripravi predlog transakcije.
   Odjemalec A pošlje zahtevo za nakup redkvic. Zahteva se nanaša na vozlišči vrstnikA in vrstnikB, ki predstavlja odjemalca A in B. Politika odobravanja navaja, da morata transakcijo odobriti oba odjemalca, zato je zahteva posredovana vozliščema vrstnikA in vrstnikB.
   Aplikacija prek podprtega SDK-ja poskrbi za sestavo predloga transakcije, ki predstavlja zahtevo po izvedbi funkcije, implementirane v verižni kodi. Funkcija klicatelju omogoča branje in/ali zapisovanje podatkov v glavno knjigo, t. j. dodajanje novega para ključ/vrednost ali posodabljanje obstoječega. SDK poskrbi za pretvorbo predloga transakcije v ustrezno arhitekturno obliko in digitalni podpis predloga z odjemalčevim certifikatom. Slika 4.1 prikazuje pripravo predloga tran-

sakcije.



Slika 4.1: Odjemalec A pripravi predlog transakcije [24].

2. Vozlišča vrstnikov odobrijo in izvedejo transakcijo. Vozlišče vrstnika s pravicami za odobravanje transakcij preverijo, da:

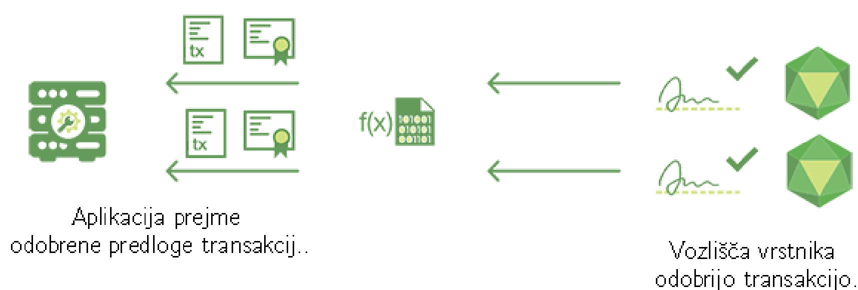
   (a) je predlog transakcije ustrezno strukturiran,

   (b) transakcija ne predstavlja že obravnane transakcije,

   (c) je digitalni podpis predloga transakcije ustrezen in

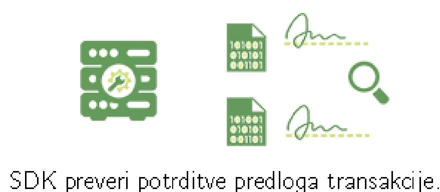
   (d) ima odjemalec A pravice za izvajanje zahtevane operacije.

   Vrstnik izvede zahtevano funkcijo verižne kode z argumenti, ki predstavljajo vhode predloga transakcije. Izvedba predstavlja zgolj simulacijo, zato glavna knjiga na tej točki še ni posodobljena. Po zaključku izvajanja vrstnik odjemalcu posreduje rezultate transakcije, obvestilo o odobritvi oz. zavrnitvi in digitalni podpis sporočila. SDK poskrbi za ustrezno pretvorbo prejetega sporočila. Slika 4.2 prikazuje odobritev in izvedbo transakcije.



Slika 4.2: Vozlišča vrstnikov odobrijo in izvedejo transakcijo [24].

3. Pregledani so odzivi na predlog transakcije.
   Aplikacija preveri digitalni podpis vsakega prejetega odziva s strani vozlišč vrstnika in med seboj primerja vse prejete rezultate transakcije. Tako ugotovi, ali je izvedba transakcije povsod proizvedla enake rezultate in ali so bili izpolnjeni vsi pogoji politike odobravanja. Po zaključku pregleda aplikacija lahko ugotovi, ali je bil predlog transakcije odobren s strani obeh vozlišč - vrstnikaA in vrstnikaB. Arhitektura obenem zagotavlja dodatne varnostne mehanizme preverjanja odzivov o odobritvi transakcije, saj politika odobravanja zahteva preverjanje ustreznosti transakcije tudi pred njeno dejansko izvedbo. Slika 4.3 prikazuje pregled potrditev predloga transakcije.



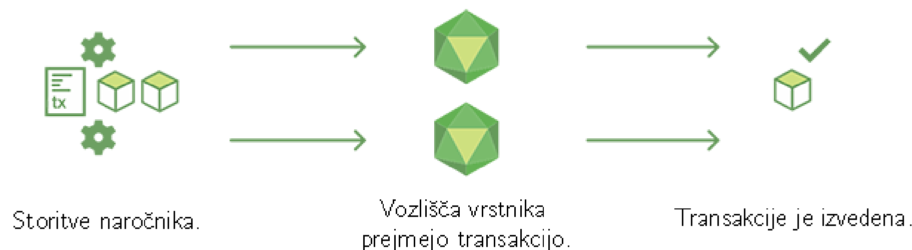
Slika 4.3: Pregledani so odzivi na predlog transakcije [24].

4. Odjemalec v transakcijo vključi odobritve transakcije.

Aplikacija storitvam za naročanje v obliki transakcije posreduje predlog transakcije in prejete odzive o odobritvi s strani vozlišč vrstnika. Storitve naročanja poskrbijo za kronološko ureditev prejetih transakcij na posameznem kanalu in njihovo združitev v blok transakcij. Blok transakcij nato pripravljen čaka na posredovanje vozliščem vrstnika in izvedbo. Slika 4.4 prikazuje pripravo predloga transakcije.

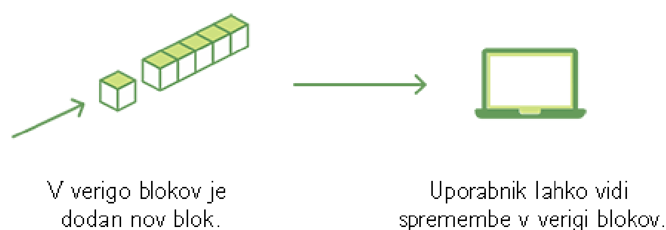
Slika 4.4: Odjemalec v transakcijo vključi odobritve transakcije [24].

5. Transakcija je potrjena in izvršena.
   Blok transakcij je posredovan vsem vozliščem vrstnika na danem kanalu. Transakcije znotraj bloka so pred izvedbo ponovno preverjene, da ustrezajo zahtevam politike odobravanja. Znotraj bloka so transakcije označene kot veljavne ali neveljavne. Slika 4.5 prikazuje potrditev in izvedbo transakcije.



Slika 4.5: Transakcija je potrjena in izvršena [24].

6. Posodobitev glavne knjige. Vsako vozlišče vrstnika blok z novimi transakcijami doda verigi izbranega kanala. Za vsako veljavno transakcijo je posodobljeno trenutno stanje verige blokov. V primeru uspešne izvedbe transakcije aplikacija prejme obvestilo, da je bilo posodobljeno stanje verige blokov, ki sedaj vključuje tudi spremembe povezane z izvedbo transakcije. V primeru neuspešne izvedbe je aplikacija obveščena o veljavnosti oz. neveljavnosti transakcije [24]. Slika 4.6 prikazuje posodobitev glavne knjige.



Slika 4.6: Posodobitev glavne knjige [24].

4.3 Odobritev in izvedba transakcije

Pred vključitvijo predlaganih sprememb v verigo blokov, ki jih vključuje transakcija, mora biti transakcija s strani ostalih udeležencev omrežja ustrezno pregledana in odobrena za izvedbo. V nadaljnjih podpoglavjih so opisane podrobnosti povezane z izoblikovanjem transakcije, njeno odobritvijo, njenim posredovanjem preostalim vozliščem in samo izvedbo transakcije.

4.3.1 Izoblikovanje transakcije

Za izoblikovanje transakcije odjemalec poljubno izbrani množici vrstnikov s pravicami za odobravanje transakcij posreduje $PROPOSE$ sporočilo, ki je oblike $< PROPOSE, tx, [anchor] >$:

- $PROPOSE$ določa vrsto sporočila.

- $tx =< clientID, chaincodeID, txPayload, timestamp, clientSig >$, kjer:
  - $clientID$ predstavlja odjemalčev identifikator,
  - $chaincodeID$ se nanaša na verižno kodo, kjer naj bo izvedena transakcija,
  - $txPayload$ vsebuje predloženo transakcijo:
    * Pri namestitveni transakciji je vrednost $txPayload =< source, metadata, policies >$, kjer *source* označuje izvorno kodo verižne kode, *metadata* vključuje atribute, povezane z verižno kodo in aplikacijo, *policies* vsebuje politiko dostopov do operacij.
    * Pri transakcijah, ki izvajajo operacije, je vrednost $txPayload =< operation, metadata >$, kjer *operation* vsebuje vrsto operacije in njene argumente, *metadata* vsebuje atribute, povezane z izvajanjem operacije.
  - *timestamp* je monotono naraščajoče število, ki ga za vsako transakcijo izračuna vsak odjemalec sam,
  - *clientSig* vključuje digitalni podpis odjemalca,
- *anchor* vsebuje odvisnosti, povezane z zahtevo.

Množica vozlišč vrstnikov, ki imajo pravice za odobravanje transakcij na verižni kodi, določeni s chaincodeID, je odjemalcu dostopna prek navadnega vozlišča vrstnika, ki množico vozlišč razbere iz politike odobravanja transakcij [24].

4.3.2 Odobritev transakcije

Ko vozlišče s pravicami za odobravanje transakcij prejme sporočilo $< PROPOSE, tx, [anchor] >$, najprej preveri podpis stranke - *clientSig* in nato simulira transakcijo. Simuliranje transakcije vključuje oblikovanje lokalne

kopije stanja in izvajanje transakcije, shranjene v polju $txPayload$, v okolju izbrane verižne kode, določene v polju $chaincodeID$. Spremembe stanja, povezane z izvajanjem simulacije transakcije, ne vplivajo na zadnje shranjeno stanje verižne kode.

Če se vozlišče vrstnika na podlagi logike odobravanje transakcije odloči za odobritev, odjemalcu, shranjenemu v polju $tx.clientID$, posreduje sporočilo oblike $< TRANSACTION - ENDORSED, tid, tran - proposal, epSig >$, kjer je:

- $TRANSACTION - ENDORSED$ vrsta poslanega sporočila,
- $tid = HASH(tx)$ enolična zgoščena vrednost predlagane transakcije,
- $tran - proposal := (epID, tid, chaincodeID, txContentBlob, readset, writeset)$, kjer:
  - $epID$ predstavlja vrstnikov identifikator,
  - tid določa enolično zgoščeno vrednost predlagane transakcije,
  - $chaincodeID$ se nanaša na verižno kodo, kjer je bil simuliran predlog transakcije,
  - $txContentBLob$ vsebuje specifične informacije o verižni kodi ali transakciji,
  - $readset$ vsebuje pare $(k, s(k).version)$, ki jih je vrstnik shranil pred simuliranjem transakcije,
  - $writeset$ vsebuje pare $(k, s(k).version)$ ključev, ki so bili med simuliranjem transakcije spremenjeni.
- $epSig$ digitalni podpis vozlišča vrstnika.

V primeru zavrnitve transakcije vrstnik odjemalcu posreduje sporočilo oblike $< TRANSACTION - INVALID, tid, REJECTED >$ [24].

4.3.3 Posredovanje transakcije

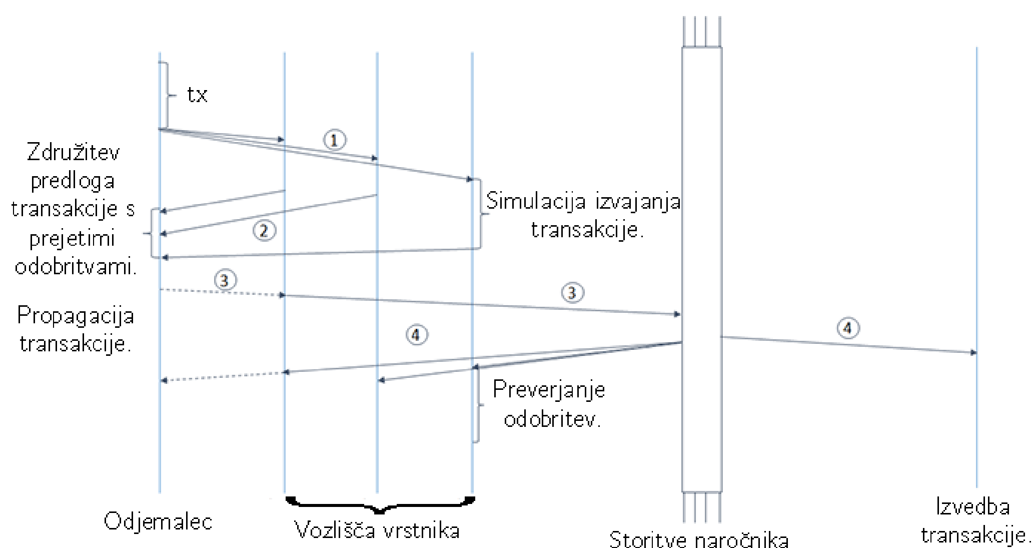
Odjemalec prek storitev naročnika čaka na prejetje sporočil $< TRANSAC TION - ENDORSED, tid, *, * >$, ki vključujejo digitalno podpisano odobritev transakcije s strani različnih vrstnikov. Z zagotovostjo lahko trdi, da je bil predlog transakcije odobren šele, ko prejme zadostno število prejetih sporočil z odobritvijo transakcije. Natančno število zadostnih odobritev predloga transakcije je navedeno v politiki odobravanja transakcij. Če odjemalec ne uspe zbrati zadostnega števila odobritev, zavrže predlog transakcije z možnostjo ponovnega poskusa predložitve predloga transakcije za odobritev.

Po uspešni odobritvi transakcije odjemalec pokliče operacijo *broadcast* (*blob*) iz nabora operacij storitev za naročanje, kjer sporočilo blob vsebuje odobritve transakcije. Če odjemalec nima pravic za proženje operacije *broad cast*, lahko sporočilo posreduje vozlišču vrstnika, ki poskrbi za njegovo nadaljnjo propagacijo. Odjemalec si sam izbere pooblaščeno vozlišče vrstnika, saj bi se lahko zgodilo, da bi nepooblaščen vrstnik s spreminjanjem sporočila lahko preprečil izvajanje transakcije [24].

4.3.4 Izvedba transakcije

Za izvedbo transakcije mora vozlišče vrstnika vsebovati zadnje aktualno stanje verige blokov. Ko zazna, da je bila prožena operacija $deliver(seqno, prev hash, blob)$, najprej preveri, ali so odobritve transakcije veljavne in na katero verižno kodo se transakcija nanaša.

Vozlišče vrstnika poskrbi za izvedbo transakcije, če je zadoščeno vsem pogojem za njeno izvedbo. Ko je transakcije izvedena, vrstnik v lokalno shranjeni bitni maski transakcijo označi z zastavico 1 in posodobi stanje verige blokov. Če preverjanje veljavnosti odobritev transakcije ni uspešno, vrstnik transakcijo v bitni mapi označi z zastavico 0 in ne izvede transakcije [24]. Slika 4.7 prikazuje potek odobritve transakcije.



Slika 4.7: Potek odobritve transakcije [24].

4.4 Politika odobravanja transakcije

Vsaka transakcija mora pred samo izvedbo na vozliščih omrežja verige blokov pridobiti ustrezno število odobritev, ki označujejo njeno veljavnost in pravilnost. Vsi postopki, povezani z odobritvijo transakcije in zahtevanim številom prejetih odobritev posamezne transakcije, so navedeni v politiki odobravanja transakcij.

4.4.1 Specifikacije politike odobravanja

Politika odobravanja transakcij je sestavljena iz pravil, ki morajo biti izpolnjena, da je transakcija odobrena. Ta lahko vključuje tudi parametre, ki so določeni med izvedbo namestitvene transakcije. Politika je vključena v verigo blokov ob namestitvi verižne kode. Da je zgotovljena zadostna stopnja varnosti, ima nabor pravil navedene samo nujno potrebne funkcije, ki imajo omejen čas izvajanja. Dinamično dodajanje pravil politike odobravanja transakcij zaradi varnostnih razlogov ni mogoče [24].

4.4.2 Ovrednotenje transakcije glede na politiko odobravanja

Transakcija je veljavna, če je bila odobrena v skladu s politiko odobravanja transakcij. Formalna politika odobravanja predstavlja predikat, kateregа rezultata sta lahko zgolj vrednosti TRUE ali FALSE. Predikat lahko vključuje tudi različne vrste spremenljivk verige blokov, kot so ključi ali identitete, povezane z verižno kodo, metapodatki o verižni kodi, elementi odobritvene transakcije in druge. Predlog transakcije je na vsakem vozlišču vrstnika s pravicami za odobravanje lahko obravnavan lokalno, saj vsa vozlišča sledijo enakemu vrstnem redu pravil politike odobravanja [24].

4.4.3 Primer politike odobravanja

Množica vozlišč vrstnikov, ki imajo pravice za odobravanje transakcije naj bo določena v verižni kodi in enaka množici $E = \{Anja, Bine, Cene, Domen, Eva, France, Grega\}$. Primeri pravil politike odobravanja so lahko sledeči:

- Predlog transakcije je odobren, če odobritev digitalno podpišejo vsi člani množice E.

- Predlog transakcije je odobren, če odobritev digitalno podpiše kateri koli član množice E.

- Predlog transakcije je odobren, če odobritev digitalno podpiše 5 izmed 7 članov množice.

- Naj imajo digitalni podpisi posameznih oseb sledečo težo $\{Anja = 49, Bine = 15, Cene = 15, Domen = 10, Eva = 7, France = 3, Grega = 1\}$, katere seštevek je enak 100. Predlog transakcije je odobren, če seštevek odobritev znaša več kot 50.

4.5 Verižna koda

Verižna koda je program, napisan v programskem jeziku Go, ki implementira predpisani vmesnik. Teče kot samostojen proces Docker, izoliran od procesov vrstnikov, ki odobravajo transakcije. Verižna koda inicializira in upravlja stanje verige blokov prek transakcij, ki jih prožijo različne aplikacije. Vsebuje poslovno logiko, ki ji sledijo vsi člani omrežja, zato jo lahko imenujemo tudi pametna pogodba. Upravljanje inicializiranega stanja je možno samo prek verižne kode, ki je ustvarila in inicializirala začetno stanje verige blokov. Neposreden dostop ene verižne kode do druge znotraj istega omrežja je mogoč na podlagi ustreznih pravic med sklicujočima verižnima kodama [24].

Vsaka verižna koda mora implementirati vmesnik, katerega funkcije so prožene kot odziv na prejete transakcije. Izsek programske kode 4.1 opisuje vmesnik verižne kode.

```
type Chaincode interface {
    Init(stub ChaincodeStubInterface) pb.Response
    Invoke(stub ChaincodeStubInterface) pb.Response
}
```

Izsek programske kode 4.1: Vmesnik verižne kode.

Funkcija Init je poklicana, ko verižna koda prejme transakcijo, ki vključuje zahtevo po inicializaciji ali nadgradnji verižne kode. Funkcija takrat poskrbi za vso potrebno inicializacijo, vključno z inicializacijo stanja aplikacije.

Funkcija Invoke je poklicana kot odziv na prejetje klicne transakcije za obdelavo predloga transakcije [24].

4.6 Ključne lastnostni platforme Hyperledger Fabric

Platforma Hyperledger Fabric uporabnikom in razvijalcem ponuja vzpostavitev zasebnega omrežja verige blokov, znotraj katerega je proženje in izvajanje transakcij brezplačno.

Konfiguracija omrežja je poljubna, saj lahko upravitelji omrežja sami poskrbijo za njegovo nastavitev glede na želje naročnika in širitev glede na performančne potrebe omrežja. Platforma razlikuje med tremi vrstami vozlišč: vozlišča odjemalca skrbijo za predložitev predloga transakcij, vozlišča vrstnika skrbijo za potrjevanje in izvajanje transakcij, vozlišča naročnika omrežju zagotavljajo komunikacijske storitve za medsebojno izmenjavo sporočil in transakcij.

Prek politike odobravanja transakcij vozlišča vrstnika odločajo o potrditvi transakcij. Da je transakcija potrjena in odobrena za izvajanje v omrežju, mora prejeti ustrezno število potrditev. Ker Hyperledger Fabric poleg javnih transakcij omogoča tudi proženje zasebnih transakcij, lahko uporabniki v omrežju, glede na zaupnost podatkov, omejijo dostop do pregleda in urejanja zapisanih informacij.

Vsak uporabnik omrežja mora za proženje transakcij preostalim udeležencem v omrežju izkazati pristnost svoje identite, kar stori s pripadajočim digitalnim certifikatom. Za vse spremembe v omrežju je tako enostavno odkriti njihovega avtorja.

Izbira soglasja o potrjevanju blokov je prepuščena razvijalcem oz. potrebam naročnika, saj platforma omogoča implementacijo lastne različice soglasja. Privzet način doseganja soglasja je podprt s praktično bizantinsko odpornostjo na napake, ki, za razliko od sistemov z dokazilom o delu ali dokazilom o deležu za potrjevanje blokov, ne potrebuje velike računske moči in posledično velike porabe električne energije.

Sistem pametnih pogodb je v platformi podprt z uporabo t. i. verižne kode, ki vključuje vso poslovno logiko za upravljanje zapisanih podatkov. Na-

mestitev poslovne logike v omrežje, shranjene v verižni kodi, je omogočeno prek posebne vrste transakcij, imenovanih namestitvene transakcije, druga vrsta transakcij, imenovanih klicne transakcije, pa končnemu uporabniku omogoča proženje v verižni kodi implementiranih funkcij.

Poglavje 5

Zapisovanje dnevnika dostopov v verigo blokov z uporabo mikrostoritev

5.1 Vsebinski opis aplikacije

Današnji centralizirani sistemi beleženja dnevnika dostopov do aplikacij snovalcem programske opreme ne zagotavljajo popolne integritete v primeru vdora v računalniški sistem, na katerem gostuje aplikacija. Ob vdoru lahko napadalci namreč poskrbijo za prikritje dogajanja in dostopov do podatkov znotraj aplikacije med njihovim napadom.

Zapisovanje dnevnika dostopov v verigo blokov z uporabo mikrostoritev snovalcem programske opreme zagotavlja vzpostavitev decentraliziranega sistema beleženja podatkov, v katerem je podatke mogoče enostavno zapisovati in praktično nemogoče spreminjati brez njihove vednosti. To snovalcem v primeru vdora v računalniški sistem omogoča enostavnejšo identifikacijo razsežnosti ter posledic vdora v aplikacijo in hitrejše ter zanesljivejše ukrepanje po njem.

Aplikacija tako končnim uporabnikom ponuja enostavno beleženje podatkov o dostopih v verigo blokov prek klicev REST in snovalcem programske

opreme samodejno beleženje podatkov o dostopih z uporabo mikrostoritve KumuluzEE Logs.

11. julija 2017 je bila izdana različica 1.0. platforme Hyperledger Fabric, pri razvoju katere je sodelovalo 159 inženirjev iz 27 različnih organizacij. Izdaja prve različice je za skupnost platforme Hyperledger Fabric predstavljala ogromen mejnik. Ker Hyperledger Fabric predstavlja novo platformo, katere dokumentacija zaenkrat še ni popolna, glede na navedene arhitekturne zmogljivosti platforme opisan praktični primer zaobjema večino trenutnih zmožnosti platforme Hyperledger Fabric.

5.2 Tehnični opis aplikacije

5.2.1 Zasnova aplikacije

Aplikacija za zapisovanje dnevnika dostopov v verigo blokov z uporabo mikrostoritev je sestavljena iz več medsebojno povezanih komponent, ki končnemu uporabniku omogočajo beleženje dnevnika dostopov v verigo blokov prek končne točke REST ali samodejno beleženje dostopov do funkcij znotraj programa z uporabo modula KumuluzEE Logs.

Verižna koda oz. pametna pogodba vsebuje vso potrebno poslovno logiko za manipulacijo s shranjenimi zapisi v verigi blokov. Uporabniku ponuja možnost shranjevanja podatkov o novem dostopu, branje že obstoječih shranjenih zapisov o dostopih in poizvedovanje po zapisih v skladu z iskalnimi kriteriji.

Node.js na računalniku z nameščenimi komponentami verige blokov vzpostavi vmesnik, ki uporabniku olajša komunikacijo z verigo blokov s pomočjo ustrezno naslovljenih zahtevkov REST.

Modul KumuluzEE Logs razvijalcem programske opreme ponuja uporabo anotacije, ki s pomočjo Log4J2 in priprave zahtevkov REST poskrbi za avtomatsko shranjevanje dnevnika dostopov v verižno kodo.

5.2.2 Arhitektura aplikacije

5.2.2.1 Implementacija verižne kode

Verižna koda je napisana v programskem jeziku Go in vključuje implementacijo predpisanega vmesnika s funkcijami za manipulacijo podatkov o dnevniku dostopov v verigi blokov.

Zapis dnevnika dostopa je predstavljen kot podatkovna struktura *struct*, ki je sestavljena iz spremenljivk podatkovnega tipa string. Vsaka spremenljivka ima določeno preslikavo pripadajočega elementa JSON, kar nam olajša pretvorbo podatkov iz prejetega JSON-a v spremenljivke, s katerimi lahko upravljamo znotraj programskega jezika Go. Izsek programske kode 5.1 opisuje podatkovno strukturo zapisa dnevnika dostopa.

```
type Log struct {
    Timetamp string ‘json:”timestamp”‘
    Level string ‘json:”level”‘
    Marker string ‘json:”marker”‘
    LogMessage string ‘json:”log_message”‘
    Class string ‘json:”class”‘
    Method string ‘json:”method”‘
    Parameters string ‘json:”parameters”‘
    ResponseTime string ‘json:”response_time”‘
    Result string ‘json:”result”‘
}
```

Izsek programske kode 5.1: Podatkovna struktura zapisa dnevnika dostopa.

Funkcija `initLedger` poskrbi za inicializacijo globalne spremenljivke zaporedne številke zapisa. Spremenljivka obenem predstavlja tudi identifikator novega zapisa dnevnika dostopov. Izsek programske kode 5.2 opisuje funkcijo `initLedger`.

```
func (s *SmartContract) initLedger(APIstub shim.ChaincodeStubInterface)
    ↪ sc.Response {
        // === Initialize global variable value ===
        insertId = 1

        return shim.Success(nil)
}
```

Izsek programske kode 5.2: Funkcija initLedger.

Funkcija Invoke poskrbi za pravilno proženje klica funkcije, ki ga uporabnik navede kot parameter klicne transakcije. Končni uporabnik lahko z zapisanimi podatki upravlja prek funkcij:

- initLedger,
- createLog,
- readLog,
- getAllLogs in
- queryLogs.

Funkcija createLog na podlagi prejetega zapisa v strukturi JSON ustvari nov zapis dnevnika dostopa v verigo blokov. Sprva funkcija preveri, ali je uporabnik podal pravilno število argumentov, nato poskrbi za pretvorbo prejetega JSON-a in nazadnje podatke shrani v verigo blokov. Če je bilo shranjevanje uspešno, funkcija poveča vrednost globalne spremenljivke številke zapisa in zaključi z izvajanjem. Ob napaki pri shranjevanju uporabniku vrne sporočilo napake in prejetega zapisa ne shrani v blokovno bazo. Izsek programske kode 5.3 opisuje funkcijo createLog.

```
func (t *SmartContract) createLog(stub shim.ChaincodeStubInterface, args
    ↪ []string) sc.Response {
        // === Check if there is correct number of arguments ===
        if len(args) != 1 {
                return shim.Error("Incorrect number of arguments.
                    ↪ Expecting 1.")
        }

        // === Parse JSON ===
        s := args[0]
        data := &Log{}
        err := json.Unmarshal([]byte(s), data)
        logJSONasBytes, err := json.Marshal(data)

        // === Save log to state ===
        err = stub.PutState("LOG" + strconv.Itoa(insertId),
            ↪ logJSONasBytes)
        if err != nil {
                return shim.Error(err.Error())
        }

        // === Increment global variable ===
        insertId += 1

        return shim.Success(nil)
}
```

Izsek programske kode 5.3: Funkcija createLog.

Funkcija `readLog` kot argument prejme identifikator zapisa dnevnika dostopov, na podlagi katerega uporabniku vrne informacije o želenem zapisu dnev-

nika dostopov. Po preverjanju pravilnega števila argumentov funkcija poskrbi za deklaracijo lokalnih spremenljivk in pridobitev identifikatorja. Nato iz verige blokov poskuša pridobiti informacije o želenem dnevniku dostopa. Če je pridobivanje informacij uspešno, so te posredovane uporabniku, ob napaki funkcija uporabniku vrne sporočilo napake. Izsek programske kode 5.4 opisuje funkcijo readLog.

```
func (t *SmartContract) readLog(stub shim.ChaincodeStubInterface, args
    ↪ []string) sc.Response {
        // === Check if there is correct number of arguments ===
        if len(args) != 1 {
                return shim.Error("Incorrect number of arguments.
                    ↪ Expecting id of the log to read.")
        }

        // === Declare local variables ===
        var id, jsonResp string

        // === Get log id ===
        id = args[0]

        // === Get log or return error ===
        valAsbytes, err := stub.GetState("LOG" + id)
        if err != nil {
                jsonResp = "{\"Error\":\"Failed to get state for id=" +
                    ↪ id + "\"}"

                return shim.Error(jsonResp)
        } else if valAsbytes == nil {
                jsonResp = "{\"Error\":\"Log does not exist: id=" + id
                    ↪ + "\"}"
```

```
                return shim.Error(jsonResp)
        }

        // === Return log ===
        return shim.Success(valAsbytes)
}
```

Izsek programske kode 5.4: Funkcija readLog.

Funkcija getAllLogs glede na podana identifikatorja zapisa dnevnika dostopov uporabniku vrne vse zapise med navedenima mejama. Sprva iz argumentov pridobi iD-ja in nato v verigi blokov poskuša poiskati zapise dnevnika dostopov z identifikatorjem znotraj iskane meje. Če je bilo iskanje uspešno, funkcija prejete podatke pretvori v strukturo JSON in jo vrne uporabniku. Ob napaki pri iskanju uporabniku vrne sporočilo napake. Če uporabnik želi pridobiti informacije o vseh zapisih dnevnika dostopov, shranjenih v verigi blokov, mora namesto identifikatorjev začetnega in končnega zapisa podati prazna niza. Hyperledger Fabric funkcija GetStateByRange nato samodejno razbere, da uporabnik želi pridobiti vse zapise dnevnika dostopov. Izsek programske kode 5.5 opisuje funkcijo getAllLogs.

```
func (s *SmartContract) getAllLogs(APIstub shim.ChaincodeStubInterface
    ↪ , args []string) sc.Response {
        // === Check if there is correct number of arguments ===
        if len(args) != 2 {
                return shim.Error("Incorrect number of arguments.
                    ↪ Expecting 2.")
        }

        // === Get start and end key ===
```

```
    startKey := args[0]
    endKey := args[1]

    // === Get all logs between start key inclusively and end key
      ↪ exclusively ===
    resultsIterator, err := APIstub.GetStateByRange(startKey,
      ↪ endKey)
    if err != nil {
        return shim.Error(err.Error())
    }
    defer resultsIterator.Close()

    // ===
    // Block of code which packages all returning logs into JSON
      ↪ buffer.
    // ===

    // === Return logs ===
    return shim.Success(buffer.Bytes())
}
```

Izsek programske kode 5.5: Funkcija getAllLogs.

Funkcija `queryLogs` kot argumente sprejme iskalne parametre posameznih spremenljivk. Iz podanih parametrov ustvari JSON spodnje oblike, na podlagi katerega verižna koda izvede poizvedbo po shranjenih zapisih dnevnika dostopa. Zapisi, ki ustrezajo iskalnim parametrom, so po končani poizvedbi posredovani uporabniku v obliki JSON. Izsek programske kode 5.6 opisuje stukturo JSON za iskanje po podatkih in izsek programske kode 5.7 funkcijo `queryLogs`.

```
{
  "selector":
    {
      "<ime_spremenljivke>": "<iskana_vrednost>"
    }
}
```

Izsek programske kode 5.6: Struktura JSON z iskalnimi parametri.

```
func (t *SmartContract) queryLogs(stub shim.ChaincodeStubInterface,
    ↪ args []string) sc.Response {
        // === Check if there is correct number of arguments ===
        if len(args) < 9 {
                return shim.Error("Incorrect number of arguments.
                    ↪ Expecting 9.")
        }

        // === Initialize local variables with argument values ===
        timestamp := args[0]
        level := args[1]
        marker := args[2]
        logMessage := args[3]
        class := args[4]
        method := args[5]
        parameters := args[6]
        responseTime := args[7]
        result := args[8]

        // ===
        // Block of code which creates query JSON.
```

```
        // ===

        // === Query all logs ===
        queryResults, err := getQueryResultForQueryString(stub,
            ↪ queryString)
        if err != nil {
                return shim.Error(err.Error())
        }

        // === Return logs matching query ===
        return shim.Success(queryResults)
}
```

Izsek programske kode 5.7: Funkcija queryLogs.

5.2.2.2 Priprava končne točke REST s pomočjo Node.js

Node.js uporabniku omogoča implementacijo končne točke za komunikacijo z verigo blokov prek klicev REST. Pred uporabo mora uporabnik poskrbeti za zagon skripte `server.js`, ki na portu 3000 vzpostavi končno točko za poslušanje na uporabniške zahteve. Izsek programske kode 5.8 opisuje skripto `server.js`.

```
var express = require('express'),
  app = express(),
  port = process.env.PORT || 3000,
  bodyParser = require('body-parser');

app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());

var routes = require('./api/routes/loggerListRoutes');
```

```
routes(app);

app.listen(port);
```

Izsek programske kode 5.8: Skripta server.js.

Skripta glede na vrsto prejetega klica v skladu s pravili, določenimi v datoteki, `loggerListRoutes.js` poskrbi za nadaljnje proženje klica za interakcijo z verigo blokov. Izsek programske kode 5.9 opisuje skripto `loggerListRoutes.js`.

```
'use strict';

module.exports = function(app) {
    var logger = require('../controllers/loggerController');

    app.route('/logs')
      .get(logger.list_all_logs)
      .put(logger.create_a_log);

    app.route('/logs/query')
      .get(logger.query_logs);

    app.route('/logs/:logId')
      .get(logger.read_a_log);
};
```

Izsek programske kode 5.9: Skripta loggerListRoutes.js s pravili za nadaljnje proženje funkcij.

Končna točka mora za nadaljnji klic funkcije znotraj verižne kode pripraviti gRPC klic, saj Hyperledger Fabric to določa kot privzet način komunikacije.

Izsek programske kode 5.10 prikazuje pripravo asinhronega klica, ki poskrbi za pridobitev podatkov o zapisu dnevnika dostopov z identifikatorjem, podanim znotraj klica REST.

```
'use strict';

var hfc = require('fabric-client');
var path = require('path');
var util = require('util');

var options = {
    wallet_path: path.join(__dirname, '../../network/creds'),
    user_id: 'PeerAdmin',
    channel_id: 'mychannel',
    chaincode_id: 'logging',
    network_url: 'grpc://localhost:7051',
    peer_url: 'grpc://localhost:7051',
    event_url: 'grpc://localhost:7053',
    orderer_url: 'grpc://localhost:7050'
};

var channel = {};
var client = null;
var targets = [];
var tx_id = null;

exports.read_a_log = function(req, res) {

  Promise.resolve().then(() => {
      console.log("Create a client and set the wallet location");
      client = new hfc();
```

```
    return hfc.newDefaultKeyValueStore({ path: options.wallet_path });
}).then((wallet) => {
    console.log("Set wallet path, and associate user ", options.user_id, "
        ↪ with application");
    client.setStateStore(wallet);
    return client.getUserContext(options.user_id, true);
}).then((user) => {
    console.log("Check user is enrolled, and set invoke function URL in
        ↪ the network");
    if (user === undefined || user.isEnrolled() === false) {
        console.error("User not defined, or not enrolled - error");
    }
    channel = client.newChannel(options.channel_id);
    channel.addPeer(client.newPeer(options.network_url));
    return;
}).then(() => {
    console.log("Query function");
    var transaction_id = client.newTransactionID();
    console.log("Assigning transaction_id: ", transaction_id.
        ↪ _transaction_id);

    const request = {
        chaincodeId: options.chaincode_id,
        txId: transaction_id,
        fcn: 'readLog',
        args: [req.params.logId]
    };

    return channel.queryByChaincode(request);
}).then((query_response) => {
    console.log("Returned from function invoke");
```

```
        if (!query_response.length) {
            console.log("No payloads were returned from function invoke");
        } else {
            console.log("Invoke result count = ", query_response.length)
        }
        if (query_response[0] instanceof Error) {
            console.error("error from function invoke = ", query_response[0]);
        }
        console.log("Response is ", query_response[0].toString());

        res.send(query_response[0].toString());
    }).catch((err) => {
        console.error("Caught Error", err);
    });

};
```

Izsek programske kode 5.10: Primer funkcije za branje zapisa dnevnika dostopov iz verige blokov.

5.2.2.3 Integracija beleženja dnevnikov dostopov v verigo blokov znotraj modula KumuluzEE Logs

Modul KumuluzEE Logs je odprtokodno ogrodje, posebej oblikovano za beleženje zapisov mikrostoritev. Uporabniku omogoča enostavno in učinkovito beleženje zapisov, povezanih s klici metod, z dostopi do zunanjih virov in drugimi dogodki. Ponuja tudi možnost samodejnega beleženja parametrov in performančnih metrik posameznih funkcij. Beleženje zapisov je omogočeno z uporabo anotacije @Log. KumuluzEE Logs je zasnovan tako, da podpira različne vrste ogrodij za beleženje zapisov, trenutno podprti ogrodji sta Log4J2 in java.util.logging [30].

Ker je Java SDK platforme Hyperledger Fabric v tem trenutku še v fazi razvoja, je bil modul KumuluzEE Logs dopolnjen s funkcijo logToBlockchain, ki podatke v verigo blokov beleži prek pripravljene končne točke REST. Funkcija logToBlockchain pridobi podatke o časovnem žigu - `timestamp`, stopnji beleženja zapisov - `level`, značkah - `marker`, sporočilu zapisa - `log_message`, razredu funkcije - **`class`**, klicani funkciji - method, podanih parametrih - `parameters`, času izvedbe klicane funkcije - `response_time` in vrnjenem rezultatu - `result`. Pridobljene podatke funkcija zapakira v ustrezno strukturo JSON (opisano v izseku programske kode 5.24) in jih prek zahtevka PUT posreduje končni točki REST za zapis v verigo blokov. Izsek programske kode 5.11 opisuje funkcijo `logToBlockchain`.

```
private void logToBlockchain(LogLevel level, Marker marker, Marker
    ↪ parentMarker, String message) {
        try {

            URL url = new URL("http://localhost:3000/logs");
            HttpURLConnection httpCon = (HttpURLConnection) url.
                ↪ openConnection();
            httpCon.setDoOutput(true);
            httpCon.setRequestMethod("PUT");
            httpCon.setRequestProperty("Content-Type", "application/
                ↪ json");
            httpCon.setRequestProperty("Accept", "application/json");

            OutputStreamWriter out = new OutputStreamWriter(
                ↪ httpCon.getOutputStream());
            out.write(String.format("{" +
                            "\"timestamp\":\"%s\", " +
                            "\"level\":\"%s\", " +
                            "\"marker\":\"%s\", " +
```

```
                    "\"log_message\":\"%s\", " +
                    "\"class\":\"%s\", " +
                    "\"method\":\"%s\", " +
                    "\"parameters\":\"%s\", " +
                    "\"response_time\":\"%s\", " +
                    "\"result\":\"%s\"" +
                    "}",
                Long.toString(Instant.now().getEpochSecond()),
                Log4j2LogUtil.convertToLog4j2Level(level),
                MarkerManager.getMarker(marker.toString()).
                    ↪ setParents(MarkerManager.getMarker(
                    ↪ parentMarker.toString())),
                message,
                ThreadContext.get("class"),
                ThreadContext.get("method"),
                ThreadContext.get("parameters"),
                ThreadContext.get("response-time"),
                ThreadContext.get("result")
        ));
        out.flush();
        out.close();

        httpCon.getInputStream();

    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Izsek programske kode 5.11: Funkcija logToBlockchain, ki podatke o dostopu zabeleži v verigo blokov.

5.2.3 Zagon in delovanje aplikacije

5.2.3.1 Priprava okolja za vzpostavitev lokalne verige blokov

Za uspešno vzpostavitev lokalnega okolja platforma Hyperledger Fabric privzema, da so na lokalnem računalniku prednameščene komponente:

- platforma Docker različice 17.03.1-ce,
- programski jezik Go različice 1.7.x,
- ogrodje Node.js različice 6.9.x.

Da lahko upravitelj omrežja v verigo blokov uspešno namesti verižno kodo, napisano v programskem jeziku Go, mora pred tem poskrbeti za nastavitev okoljske spremenljivke $GOPATH s pomočjo ukazov, opisanih v izseku programske kode 5.12.

```
export GOPATH=$HOME/go
export PATH=$PATH:$GOPATH/bin
```

Izsek programske kode 5.12: Nastavitev okoljske spremenljivke $GOPATH.

Po uspešni namestitvi komponent in nastavitvi okoljske spremenljivke za programski jezik Go mora upravitelj omrežja poskrbeti za ustrezno definicijo omrežja in lastnosti posameznih vozlišč, ki jo platforma Docker razume in na podlagi katere lahko vzpostavi želeno omrežje. To upravitelj stori z določitvijo nastavitev posameznega vozlišča znotraj datoteke `docker-compose.yaml`. Datoteka `docker-compose.yaml` je sestavljena iz definicije:

- enega vozlišča naročnika,
- več vozlišč vrstnika,
- komponent za interakcijo z vozliščem vrstnika,

- podatkovne baze CouchDB posameznega vozlišča vrstnika.

Definicija vozlišča naročnika in vozlišč vrstnika vključuje nastavitev imena vsebnika - container_name, namestitvene slike s programom za delovanje vozlišča naročnika - image, okoljskih spremenljivk - environment, delovne mape Docker vsebnika - working_dir, ukaza, ki se zažene ob namestitvi programa - command, preslikavo zunanjih vrat za povezovanje na notranja vrata - ports in povezave vseh datotečnih odvisnosti - volumes. Izsek programske kode 5.13 opisuje primer definicije vozlišča naročnika.

```
orderer.example.com:
  container_name: orderer.example.com
  image: hyperledger/fabric-orderer:x86_64-1.0.0-rc1
  environment:
    - ORDERER_GENERAL_LOGLEVEL=debug
    - ORDERER_GENERAL_LISTENADDRESS=0.0.0.0
    - ORDERER_GENERAL_GENESISMETHOD=file
    - ORDERER_GENERAL_GENESISFILE=/etc/hyperledger/configtx/
        ↪ genesis.block
    - ORDERER_GENERAL_LOCALMSPID=OrdererMSP
    - ORDERER_GENERAL_LOCALMSPDIR=/etc/hyperledger/msp/
        ↪ orderer
  working_dir: /opt/gopath/src/github.com/hyperledger/fabric/orderer
  command: orderer
  ports:
    - 7050:7050
  volumes:
      - ./:/etc/hyperledger/configtx
      - ./crypto-config/ordererOrganizations/example.com/orderers/
          ↪ orderer.example.com/msp:/etc/hyperledger/msp/orderer
```

Izsek programske kode 5.13: Primer definicije vozlišča naročnika.

Komponenta za interakcijo z vozliščem vrstnika potrebuje nastavitev imena vsebnika - container_name, namestitvene slike s programom za delovanje komponente - image, dostopa preko konzole - tty, okoljskih spremenljivk - environment, delovne mape Docker vsebnika - working_dir, ukaza, ki se zažene ob namestitvi programa - command, povezave vseh datotečnih odvisnosti - volumes in povezave z vsemi vsebniki, s katerimi želi upravljati - depends_on. Izsek programske kode 5.14 opisuje definicijo komponente za interakcijo z vozliščem vrstnika.

```
cli01:
  container_name: cli01
  image: hyperledger/fabric-tools:x86_64-1.0.0-rc1
  tty: true
  environment:
    - GOPATH=/opt/gopath
    - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
    - CORE_LOGGING_LEVEL=DEBUG
    - CORE_PEER_ID=cli
    - CORE_PEER_ADDRESS=peer0.org1.example.com:7051
    - CORE_PEER_LOCALMSPID=Org1MSP
    - CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/
        ↪ hyperledger/fabric/peer/crypto/peerOrganizations/org1.
        ↪ example.com/users/Admin@org1.example.com/msp
    - CORE_CHAINCODE_KEEPALIVE=10
  working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
  command: /bin/bash
  volumes:
      - /var/run/:/host/var/run/
```

```
    - ./../chaincode:/opt/gopath/src/github.com/fabcar/
    - ./crypto-config:/opt/gopath/src/github.com/hyperledger/fabric/
      ↪ peer/crypto/
  depends_on:
    - orderer.example.com
    - peer0.org1.example.com
    - couchdb01
```

Izsek programske kode 5.14: Primer definicije komponente za interakcijo z vozliščem vrstnika.

Podatkovna baza CouchDB za uspešno namestitev potrebuje nastavitev imena vsebnika - `container_name`, namestitvene slike s programom za delovanje podatkovne baze - `image`, preslikave zunanjih vrat za povezovanje na notranja vrata - `ports` in okoljske spremenljivke - `environment`. Izsek programske kode 5.15 opisuje definicijo podatkovne baze CouchDB.

```
couchdb01:
  container_name: couchdb01
  image: hyperledger/fabric-couchdb:x86_64-1.0.0-rc1
  ports:
    - 5984:5984
  environment:
    DB_URL: http://localhost:5984/member_db
```

Izsek programske kode 5.15: Primer definicije podatkovne baze CouchDB.

Po uspešni namestitvi potrebnih komponent, nastavitvi okoljske spremenljivke za programski jezik Go in pripravi datoteke `docker-compose.yaml` je upravitelj omrežja poskrbel za ustrezno pripravo okolja, v katerem bo delovalo omrežje verige blokov. Upravitelj mora poskrbeti še za zagon vsebnikov Docker, v katerem bodo tekle definirane storitve.

5.2.3.2 Zagon omrežja verige blokov

Zagon omrežja verige blokov je mogoč s pomočjo ukaza, opisanega v izseku programske kode 5.16. Ukaz sprva preveri, ali so v lokalno okolje že prenešene namestitvene slike posameznih komponent in jih po potrebi posodobi oz. prenese iz repozitorijev na Docker Hubu. Nato sledi oblikovanje in zagon vsebnikov Docker, znotraj katerih tečejo posamezne komponente omrežja.

```
docker-compose -f docker-compose.yaml up -d
```

Izsek programske kode 5.16: Ukaz za zagon omrežja verige blokov.

Zaustavitev omrežja verige blokov je možna z ukazom, opisanim v izseku programske kode 5.17, ki obenem poskrbi za ustrezno uničenje zaustavljenih vsebnikov Docker.

```
docker-compose -f docker-compose.yaml down
```

Izsek programske kode 5.17: Ukaz za zaustavitev omrežja verige blokov.

5.2.3.3 Ustvarjanje in pridružitev kanalu

Upravitelj omrežja lahko nov kanal ustvari s pomočjo ukaza, opisanega v izseku programske kode 5.18.

```
peer channel create -o <ime_containerja_Ordererja>:<port> -C <ime-
    ↪ kanala> -f <pot_do_datoteke_ime-kanala.tx>
```

Izsek programske kode 5.18: Ukaz za ustvarjanje novega kanala.

Vozlišče vrstnika, ki je ustvarilo nov kanal, se mu lahko pridruži zgolj z izvedbo ukaza, opisanega v izseku programske kode 5.19, medtem ko morajo preostala vozlišča vrstnika predhodno pridobiti genesis datoteko kanala.

```
peer channel join −b <ime_genesis_datoteke>
```

Izsek programske kode 5.19: Ukaz za pridružitev kanalu.

Genesis datoteko kanala vozlišče vrstnika pridobi z izvršitvijo ukaza, opisanega v izseku programske kode 5.20.

```
peer channel fetch oldest <ime_genesis_datoteke> −o <
    ↪ ime_containerja_Ordererja>:<port> −C <ime−kanala>
```

Izsek programske kode 5.20: Ukaz za pridobitev genesis datoteke Channela.

5.2.3.4 Namestitev in incializacija verižne kode

Namestitev verižne kode poteka v dveh korakih:

1. V prvem koraku je treba verižno kodo s poslovno logiko za manipulacijo s podatki v verigi blokov namestiti na vozlišče vrstnika. Namestitev je mogoča prek komponente za interakcijo z vozliščem vrstnika z ukazom, opisanem v izseku programske kode 5.21.

```
peer chaincode install −n <ime_instance_verizne_kode> −v <
    ↪ razlicica_Hyperledger_Fabric> −p <pot_do_go_datoteke>
```

Izsek programske kode 5.21: Ukaz za namestitev verižne kode.

2. V drugem koraku je treba poskrbeti za ustvarjanje instance in inicializacijo verižne kode. Zadošča, da za oblikovanje instance in inicializacijo verižne kode na enem izmed vozlišč vrstnika sprožimo ukaz, opisan v izseku programske kode 5.22.

```
peer chaincode instantiate −o <ime_containerja_Ordererja>:<port>
    ↪ −C <ime_kanala> −n <ime_instance_Chaincodea> −v <
    ↪ razlicica_Hyperledger_Fabric> −c <argumenti, npr. '{"Args":[
    ↪ ""]}'> −P <politika_odobravanja, npr. "OR␣('Org1MSP.
    ↪ member','Org2MSP.member')">
```

Izsek programske kode 5.22: Ukaz za ustvarjanje instance in inicializacijo verižne kode.

Da lahko vozlišče vrstnika proži operacije za manipulacijo s podatki, shranjenimi v verigi blokov, mora biti na njem predhodno nameščena verižna koda z ustrezno poslovno logiko.

5.2.3.5 Zagon končne točke Node.js REST

Za zagon končne točke Node.js REST upravitelj omrežja poskrbi s proženjem spodnjega ukaza, ki vzpostavi vse potrebne komponente za poslušanje na uporabniške zahteve. Izsek programske kode 5.23 opisuje ukaz za zagon končne točke.

```
node server.js
```

Izsek programske kode 5.23: Ukaz za zagon končne točke Node.js REST.

5.2.3.6 Zapisovanje dnevnika dostopov v verigo blokov

Uporabniku je zapisovanje dnevnika dostopov v verigo blokov omogočeno prek končne točke Node.js REST, ki jo je predhodno vzpostavil upravitelj omrežja (poglavje 5.2.2). Da je zapisovanje podatkov uspešno, mora uporabnik podatke o dnevniku dostopa shraniti v dokument JSON, pripravljen v skladu s strukturo, opisano v izseku programske kode 5.24.

```
{
```

```
    "timestamp": "<vrednost>",
    "level": "<vrednost>",
    "marker": "<vrednost>",
    "log_message": "<vrednost>",
    "class": "<vrednost>",
    "method": "<vrednost>",
    "parameters": "<vrednost>",
    "response_time": "<vrednost>",
    "result": "<vrednost>"
}
```

Izsek programske kode 5.24: Dokument JSON za shranjevanje podatkov o dnevniku dostopa.

Pripravljen dokument JSON uporabnik vstavi v telo zahtevka PUT, naslovljene na naslov `http://<ip_naslov_naprave>:3000/logs`. Ko končna točka Node.js prejme zahtevo REST, v skladu s pravili, določenimi v datoteki `loggerListRoutes.js`, poskrbi za preusmeritev klica na ustrezno funkcijo, ki poskrbi za nadaljnje shranjevanje podatke o dnevniku dostopa v verigo blokov.

5.2.3.7 Branje informacij o dnevniku dostopov iz verige blokov

Branje informacij o dnevniku dostpov je iz verige blokov možno na 3 načine:

1. Branje informacij o vseh zapisih dnevnika dostopov:
   Uporabnik lahko informacije o vseh zapisih dnevnika dostopov, shranjenih v verigi blokov, pridobi z naslovitvijo zahtevka GET na naslov `http://<ip_naslov_naprave>:3000/logs`, ki uporabniku podatke vrne v obliki JSON, opisani v izseku programske kode 5.25.

```
[
  {
    "Key":"LOG1",
```

```
      "Log":{
        <Struktura JSON, opisana v izseku programske kode številka
            ↪ 5.24>
      }
    },
    {
      "Key":"LOG2",
      "Log":{
        <Struktura JSON, opisana v izseku programske kode številka
            ↪ 5.24>
      }
    },
    .
    .
    .
]
```

Izsek programske kode 5.25: Struktura vrnjenega dokumenta JSON s podatki o vseh zapisih dnevnika dostopov.

2. Branje informacij o določenem zapisu dnevnika dostopov Informacije o določenem zapisu dnevnika dostopov lahko uporabnik pridobi z naslovitvijo zahtevka GET na naslov `http://<ip_naslov_naprave ↪ >:3000/logs/<id_loga>`, v katerem kot pot navede identifikator iskanega dnevnika dostopa. Končna točka uporabniku informacije o dnevniku dostopa vrne v strukturi JSON, opisani v izseku programske kode številka 5.24.

3. Poizvedovanje po zapisih, shranjenih v verigi blokov Poizvedovanje je uporabniku omogočeno z naslovitvijo zahtevka GET na naslov `http ↪ ://<ip_naslov_naprave>:3000/logs/query?<iskalni_ ↪ parametri>`, v katerem kot iskalne parametre navede pare `ime_ ↪ polja=vrednost`, npr. poizvedba `http://<ip_naslov_na`

↪ prave>:3000/logs/query?level=TRACE uporabniku vrne vse zapise dnevnika dostopov, ki imajo v polju level shranjeno vrednost TRACE. Vrnjene informacije o logih so oblikovane v skladu s strukturo JSON, opisano v izseku programske kode številka 5.25.

5.2.3.8 Samodejno beleženje dnevnikov dostopov v verigo blokov znotraj modula KumuluzEE Logs

Pred uporabo modula KumuluzEE Logs mora razvijalec sam poskrbeti za vključitev odvisnosti na ustrezno ogrodje za beleženje zapisov. V opisan praktični primer je bila vključena odvisnost na Log4J2. Izsek programske kode 5.26 opisuje vstavitev odvisnosti na Log4J2.

```
<dependency>
    <artifactId>kumuluzee-logs-log4j2</artifactId>
    <groupId>com.kumuluz.ee.logs</groupId>
    <version>${kumuluzee-logs.version}</version>
</dependency>
```

Izsek programske kode 5.26: Vstavljena odvisnost na Log4J2.

Razvijalci programske kode lahko nato samodejno beleženje dnevnika dostopov v verigo blokov omogočijo z ustrezno označitvijo funkcije ali razreda funkcije z anotacijo @Log. Po ustreznem zagonu pripravljene kode je nato ob vsakem klicu funkcije ta ustrezno zabeležen in shranjen v verigo blokov [30].

5.3 Priprava aplikacije za produkcijsko okolje

Migracija aplikacije in verige blokov iz lokalnega razvojnega okolja v produkcijsko okolje od upravitelja omrežja zahteva, da sprva poskrbi za ustrezno pripravo arhitekture omrežja, znotraj katerega bodo medsebojno komunicirala posamezna vozlišča.

Skupno `docker-compose.yaml` datoteko, opisano v poglavju 5.2.2, znotraj katere se nahajajo definicije posameznih vozlišč, mora upravitelj razbiti na dve datoteki. Ena datoteka je namenjena definiciji vozlišča naročnika, druga pa definiciji vozlišča vrstnika, komponenti za interakcijo z vozliščem vrstnika in podatkovne baze CouchDB. Datoteke mora upravitelj prenesti na posamezne naprave v omrežju, šele nato lahko prične z namestitvijo aplikacije v produkcijsko okolje, opisano v poglavju 5.4.

5.4 Namestitev aplikacije v produkcijsko okolje

Upravitelj omrežja s pomočjo namestitvenih datotek Docker na posamezni napravi znotraj omrežja poskrbi za namestitev pravilne vrste vozlišča verige blokov. Znotraj omrežja mora obstajati eno vozlišče naročnika, ki skrbi za medsebojno komunikacijo s preostalimi vozlišči vrstnika, in eno ali več vozlišč vrstnika.

Da lahko vozlišča vrstnika med seboj komunicirajo, morajo biti pridružena skupnemu kanalu. Upravitelj mora zato ustvariti kanal in poskrbeti za pridružitev posameznega vozlišča na ustvarjen kanal. Vsa povezana vozlišča znotraj kanala bodo samodejno poskrbela za sinhronizacijo verige blokov v omrežju.

Ker končni uporabnik z verigo blokov lahko komunicira zgolj prek vozlišča vrstnika, mora upravitelj omrežja na vsakem izmed vozlišč vrstnika namestiti verižno kodo s poslovno logiko za manipulacijo s podatki v verigi blokov in na enem izmed njih poskrbeti za ustvarjanje instance in inicializacijo verižne kode. Če uporabnik želi z zapisi v verigi blokov upravljati tudi prek končne točke REST, mora upravitelj omrežja poskrbeti za njeno vzpostavitev in zagon s pomočjo Node.js.

Vsi postopki za zagon omrežja, ustvarjanje in pridružitev kanalu, namestitev in inicializacijo verižne kode in zagon končne točke Node.js REST so opisani

v poglavju 5.2.3.

Poglavje 6

Zaključek

V diplomskem delu smo spoznali osnovne koncepte verige blokov, njeno dosedanjo zgodovino in nadgrajeno prvo generacijo verige blokov. Primerjali smo tri najpogostejše platforme verige blokov: Bitcoin, Ethereum in Hyperledger Fabric. Za nadaljnje raziskovanje smo izbrali platformo Hyperledger Fabric, saj je najprimernejša za razvoj praktičnega primera zapisovanja dnevnika dostopov v verigo blokov. V opisu praktičnega primera smo spoznali, kako s pomočjo verižne kode pripraviti poslovno logiko za upravljanje podatkov, shranjenih v verigi blokov, kako pripraviti končno točko REST za zapisovanje podatkov in kako samodejno zapisovati podatke o dnevniku dostopov v verigo blokov s pomočjo modula KumuluzEE. Raziskali smo potrebne postopke za konfiguracijo in vzpostavitev lastnega omrežja verige blokov, proženje funkcij, implementiranih v verižni kodi, in vključitev samodejnega zapisovanja podatkov.

Praktični primer je zaenkrat mogoče varnostno izboljšati z uporabo certifikatov v okviru protokola TLS, kjer z lastnim certifikatom vsak uporabnik izkazuje svojo identiteto. Posledično je v omrežju potrebno vzpostaviti tudi certifikatno agencijo s pomočjo vsebnika Docker. Vsak uporabnik, ki bi ob nadgraditvi omrežja želel sodelovati v njem, bi lahko vse potrebne certifikate prejel s strani certifikatne agencije.

Razvijalci platforme Hyperledger Fabric v prihodnosti nameravajo povezovanje z verigo blokov in proženje transakcij omogočiti tudi prek Java SDK-ja, ki je trenutno še v razvoju. Po izdaji Java SDK-ja je samodejno beleženje podatkov znotraj modula KumuluzEE Logs iz proženja zahtevkov REST na pripravljeno končno točko REST možno posodobiti z uporabo primernejšega Java SDK-ja.

Tehnologija verige blokov predstavlja tehnologijo 21. stoletja, katere razvoj se je v preteklih letih šele pričel. Primeri aplikacij večinoma zaobjemajo finančni sektor, saj sama tehnologija odpravlja omejitve, povezane s tradicionalnim finančnim poslovanjem. Menim, da je večina aplikacije trenutno šele zgolj ideja, nekatere izmed njih si svojo pot razvoja poskušajo zagotoviti z javnim financiranjem - ICO, ki se je v preteklem letu močno razširilo med širšo javnost.

Literatura

[1] Andreas M. Antonopoulos. *Mastering Bitcoin. Unlocking Digital Crypto-Currencies.* O'Reilly Media, 2014.

[2] Architecture of the hyperledger blockchain fabric [online]. Dostopano: 29. 8. 2017. URL: https://www.zurich.ibm.com/dccl/papers/cachin_dccl.pdf.

[3] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. *A survey of attacks on Ethereum smart contracts; 6th International Conference on Principles of Security and Trust (POST).* European Joint Conferences on Theory and Practice of Software, 2017.

[4] Dave Bayer, Stuart Haber, and W. Scott Stornetta. Improving the efficiency and reliability of digital time-stamping. *Sequences*, 2:329–334, 1992.

[5] Nirupama Devi Bhaskar and David LEE Kuo Chuen. *Bitcoin Mining Technology; Handbook of Digital Currency: Bitcoin, Innovation, Financial Instruments, and Big Data.* Academic Press, 2015.

[6] Bitcoin proof of work [online]. Dostopano: 28. 8. 2017. URL: https://en.bitcoin.it/wiki/Proof_of_work.

[7] Bitcoincharts [online]. Dostopano: 30. 8. 2017. URL: https://bitinfocharts.com.

[8] Blockchain size [online]. Dostopano: 8. 8. 2017. URL: https://blockchain.info/charts/blocks-size?timespan=3years.

[9] Blockexplorer [online]. Dostopano: 12. 9. 2017. URL: https://blockexplorer.com.

[10] Jerry Brito and Andrea Castillo. *Bitcoin: A Primer for Policymakers*. Mercatus Center, George Mason University, Fairfax, VA, 2013.

[11] Joshua Davis. The crypto-currency: Bitcoin and its mysterious inventor. *The New Yorker*, 2011.

[12] Ethash [online]. Dostopano: 29. 8. 2017. URL: https://github.com/ethereum/wiki/wiki/Ethash.

[13] Ethereum private network [online]. Dostopano: 29. 8. 2017. URL: https://github.com/ethereum/go-ethereum/wiki/Private-network.

[14] Ethereum proof of stake [online]. Dostopano: 7. 8. 2017. URL: https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ.

[15] Ethereum white paper [online]. Dostopano: 7. 8. 2017. URL: https://github.com/ethereum/wiki/wiki/White-Paper.

[16] Etherscan [online]. Dostopano: 7. 9. 2017. URL: https://etherscan.io.

[17] Stuart Haber and W. Scott Stornetta. How to time-stamp a digital document. *Journal of Cryptology*, 3(2):99–111, 1991.

[18] Nikolai Hampton. Understanding the blockchain hype: Why much of it is nothing more than snake oil and spin. *Computerworld*, 2016.

[19] Hard fork [online]. Dostopano: 30. 8. 2017. URL: http://www.investopedia.com/terms/h/hard-fork.asp.

[20] Hashcash [online]. Dostopano: 28. 8. 2017. URL: https://en.bitcoin.it/wiki/Hashcash.

[21] How do ethereum smart contracts work? [online]. Dostopano: 29. 8. 2017. URL: https://www.coindesk.com/information/ethereum-smart-contracts-work/.

[22] How ethereum works [online]. Dostopano: 29. 8. 2017. URL: https://www.coindesk.com/information/how-ethereum-works/.

[23] Hyperledger fabric [online]. Dostopano: 12. 8. 2017. URL: http://hyperledger.org/projects/fabric.

[24] Hyperledger fabric documentation [online]. Dostopano: 8. 8. 2017. URL: https://hyperledger-fabric.readthedocs.io.

[25] Hyperledger vs corda [online]. Dostopano: 29. 8. 2017. URL: https://medium.com/chain-cloud-company-blog/hyperledger-vs-corda-pt-1-3723c4fa5028.

[26] Marco Iansiti and Karim R. Lakhani. The truth about blockchain. *Harvard Business Review*, 2017.

[27] Ibm hyperledger fabric [online]. Dostopano: 12. 8. 2017. URL: https://www.ibm.com/blockchain/hyperledger.html.

[28] If you understand hash functions, you'll understand blockchains [online]. Dostopano: 5. 9. 2017. URL: https://decentralize.today/if-you-understand-hash-functions-youll-understand-blockchains-9088307b745d.

[29] Hossein Kakavand, Bart Chilton, and Nicolette Kost De Sevres. *The Blockchain Revolution: An Analysis of Regulation and Technology Related to Distributed Ledger Technologies.* Luther Systems & DLA Piper, 2016.

[30] Kumuluzee logs [online]. Dostopano: 7. 9. 2017. URL: https://github.com/kumuluz/kumuluzee-logs.

[31] Robert McMillan. How bitcoin lets you spy on careless companies. *wired.co.uk*, 2013.

[32] Merkle tree [online]. Dostopano: 7. 9. 2017. URL: https://en.wikipedia.org/wiki/Merkle_tree.

[33] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System.* Gmane, 2008.

[34] Lam Pak Nian and David LEE Kuo Chuen. *A Light Touch of Regulation for Virtual Currencies; Handbook of Digital Currency: Bitcoin, Innovation, Financial Instruments, and Big Data.* Academic Press, 2015.

[35] Serguei Popov. A probabilistic analysis of the nxt forging algorithm. *Ledger*, 1:69–83, 2016. doi:10.5195/LEDGER.2016.46.

[36] Siraj Raval. *What Is a Decentralized Application?; Decentralized Applications: Harnessing Bitcoin's Blockchain Technology.* O'Reilly Media, Inc., 2016.

[37] Rootstock [online]. Dostopano: 7. 9. 2017. URL: http://www.rsk.co.

[38] Katherine Sagona-Stophel. *Bitcoin 101 white paper.* Thomson Reuters, 2016.

[39] Tom Simonite. Mapping the bitcoin economy could reveal users' identities. *MIT Technology Review*, 2013.

[40] Soft fork [online]. Dostopano: 30. 8. 2017. URL: http://www.investopedia.com/terms/s/soft-fork.asp.

[41] Economist Staff. Blockchains: The great chain of being sure about things. *The Economist*, 2015.

[42] Nick Szabo. Smart contracts: Building blocks for digital markets. *Extropy*, 2016.

[43] The great chain of being sure about things [online]. 2015. Dostopano: 2. 8. 2017. URL: http://www.economist.com/news/briefing/21677228-technology-behind-bitcoin-lets-people-who-do-not-know-or-trust-each-other-build-dependable.

[44] Understanding ethereum [online]. 2016. URL: https://www.coindesk.com/research/understanding-ethereum-report/.

[45] What is bitcoin? [online]. 2013. Dostopano: 2. 8. 2017. URL: http://money.cnn.com/infographic/technology/what-is-bitcoin.

[46] What is double spending & how does bitcoin handle it? [online]. Dostopano: 5. 9. 2017. URL: https://coinsutra.com/bitcoin-double-spending/.

[47] What is the bitcoin block size debate and why does it matter? [online]. Dostopano: 5. 9. 2017. URL: https://www.coindesk.com/what-is-the-bitcoin-block-size-debate-and-why-does-it-matter/.

[48] Who is satoshi nakamoto? [online]. 2015. Dostopano: 2. 8. 2017. URL: https://www.economist.com/blogs/economist-explains/2015/11/economist-explains-1.

[49] David Yanofsky, Ritchie S. King, and Sam Williams. By reading this article, you're mining bitcoins. *qz.com*, 2013.