Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Marko Kočevar

Primerjava implementacij verige blokov na primeru registra vpogledov

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: doc. dr. Jurij Mihelič
Somentor: Jernej Srebrnič

Ljubljana, 2019

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Primerjava implementacij verige blokov na primeru registra vpogledov

Tematika naloge:

Tehnologija verige blokov v zadnjem času močno pridobiva na pomembnosti. Pametne pogodbe omogočajo izjemno prilagodljivost in posledično novi primeri uporabe tehnologije nastajajo skorajda vsakodnevno. Še vedno pa obstaja ogromno odprtih vprašanj, še posebej kadar se mora uporabnik odločiti, katero izmed izvedb tehnologije vzeti kot osnovo za razvoj aplikacij.

V okviru diplomske naloge ponudite praktično primerjavo in primer uporabe tehnologije verige blokov. Primer uporabe vzemite iz realnosti, pri čemer uporabite različne izvedbe verige blokov za razvoj iste aplikacije. Izberite aplikacijo, za katero je uporaba verige blokov smiselna in utemeljena. Izbrane izvedbe verige blokov primerjajte, predstavite razlike, prednosti in slabosti. Postavitev tudi manjše testno okolje, v katerem izbrane izvedbe eksperimentalno ovrednotite in komentirajte rezultate.

Zahvaljujem se doc. dr. Juriju Miheliču za mentorstvo in pomoč pri oblikovanju dimplomskega dela.
Zahvala gre tudi družini in prijateljem za podporo in vzpodbudo tekom študija.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
GDPR	General Data Protection Regulation	splošna uredba o varstvu osebnih podatkov
DPO	Data Protection Officer	pooblaščena oseba za varstvo osebnih podatkov
JSON	JavaScript Object Notation	JavaScript zapis objektov
HTTP	Hypertext Transfer Protocol	protokol za prenos hiperbesedila
IP	Internet Protocol	internetni protokol
POW	Proof of Work	dokazilo dela
POS	Proof of Stake	dokazilo deleža
DPOS	Delegated Proof of Stake	delegirani POS
dBFT	Delegated Byzantine Fault Tolerance	delegirana Bizantinska toleranca odpovedi
EVM	Ethereum Virtual Machine	Ethereum navidezni stroj
MSP	Membership Service Provider	Fabric komponenta, ki skrbi za izdajanje in preverjanje certifikatov
ACID	Atomicity, Consistency, Isolation, and Durability	atomarnost, konsistentnost, izolacija, trajnost
Java EE	Java Enterprise Edition	poslovna različica Jave

SQL	Standard Query Language	strukturiran povpraševalni jezik za delo s podatkovnimi bazami
IPC	Inter-process Communication	medprocesna komunikacija
EJB	Enterprise Java Bean	poslovna zrna Java
JDBC	Java Database Connectivity	Java standard za baze podatkov
REST	Representational State Transfer	arhitektura za izmenjavo podatkov med spletnimi storitvami
JMS	Java Messaging System	sporočilna storitev Java
SDK	Software Development Kit	paket za razvoj programske opreme
MDB	Message Driven Bean	Javino poslovno zrno, ki omogoča asinhrono procesiranje sporočil
RPC	Remote Procedure Call	klici oddaljenih procedur

Povzetek

Naslov: Primerjava implementacij verige blokov na primeru registra vpogledov

Avtor: Marko Kočevar

V diplomski nalogi skozi primer implementacije registra vpogledov v osebne podatke predstavimo in primerjamo dve implementaciji tehnologije verige blokov. Register vpogledov omogoča obdelovalcem osebnih podatkov večji nadzor nad zbranimi podatki in hkrati pomaga k izpoljnevanju nekaterih pogojev uredbe GDPR. Tehnologiji verige blokov se v zadnjem času posveča vse več pozornosti, glavni razlog pa je vse večje zanimanje za kriptovalute, katere so bile tudi prve, ki so uspešno implementirale to novo tehnologijo. V nalogi si pogledamo, kako je tehnologija zasnovana, kaj nam ponuja, osnovne lastnosti, prednosti in slabosti. Podrobneje predstavimo dve implementaciji, Ethereum in Hyperledger Fabric, ter prikažemo, kako smo jih umestili v našo aplikacijo, in ponudimo primerjavo med njima.

Ključne besede: Veriga blokov, GDPR, Ethereum, Hyperledger.

Abstract

Title: Comparison of blockchain implementations using a register of inquiries

Author: Marko Kočevar

The thesis presents and compares two implementations of blockchain technology. The comparison is based on a register of insights into personal data, as an example use case. The register enables various institutions and companies to have better control, as well as a cleaner overview of insight into personal data of their customers from within their computer systems. Additionally, it helps users towards compliance with the new GDPR regulation regarding personal data. Blockchain technology is becoming increasingly popular, mainly influenced by the rise of cryptocurrencies, which were the first to implement this technology. In the thesis we take a look into the technology, its design, properties, capabilities as well as pros and cons. There is a more detailed focus on two main implementations, the Ethereum and Hyperledger Fabric platforms, with comparison and description of our application.

Keywords: Blockchain, GDPR, Ethereum, Hyperledger.

Poglavje 1

Uvod

V današnjem informacijskem času in s pojavom raznih družbenih omrežij se je močno povečal obseg osebnih podatkov posameznika, ki jih podjetja zbirajo in obdelujejo za dosego svojih ciljev. Zato je s 25. marcem 2018 stopila v veljavo nova uredba EU za varstvo osebnih podatkov [10] oziroma GDPR (*angl. General Data Protection Regulation*). Zakon predpisuje pravila za podjetja in ustanove (upravljavci oziroma obdelovalci osebnih podatkov), ki hranijo in/ali obdelujejo osebne podatke posameznikov, kako ravnati z le-temi. Posameznikom zagotavlja večji nadzor in pregled nad tem, kje se njihovi podatki uporabljajo. V podrobnosti same uredbe se v diplomskem delu ne bomo poglabljali, spodaj je strnjenih nekaj njenih poglavitnejših točk.

Privolitev Posameznik mora izrecno privoliti v zbiranje in obdelavo svojih osebnih podatkov.

Dostop do informacij Upravljavci morajo posamezniku zagotoviti pregledne informacije o obdelavah njegovih podatkov.

Pooblaščena oseba Podjetja, ki redno zajemajo in obdelujejo osebne podatke posameznika, morajo imenovati odgovorno osebo za varstvo osebnih podatkov ali DPO (*angl. data protection officer*).

Evidenca dejavnosti obdelave Vodenje evidence obdelav in vpogledov v osebne podatke.

Možnost pozabe Posameznik ima pravico zahtevati izbris oziroma pozabo njegovih zbranih osebnih podatkov.

Uredba v 30. členu navaja, da mora vsak upravljavec in obdelovalec voditi evidenco dejavnosti obdelav, ki vsebuje informacije, kot so namen obdelav, vrste osebnih podatkov, kdo je videl te podatke, prenosi v tretje države in predvideni roki za izbris podatkov.

Namen projekta, zasnovanega na delovnem mestu, je bil razviti aplikacijo, register vpogleda osebnih podatkov, ki bi hranil prej omenjene informacije, podjetjem in ustanovam omogočal enostavno integracijo v njihove obstoječe sisteme in ponujal pregleden ter enostaven uporabniški vmesnik, preko katerega bi dostopali do evidenc obdelav. Dodana vrednost je zagotavljanje integritete shranjenih podatkov in zabeleženih vpogledov, kar dosežemo z uporabo tehnologije verige blokov (*angl. blockchain*). Veriga blokov je novejša tehnologija, najbolj znana po uporabi v kriptovalutah. V grobem lahko rečemo, da nam ponuja decentralizirano shrambo podatkov, hkrati pa zagotavlja nespremenljivost in varnost.

Cilj diplomskega dela je predstaviti idejo in izdelavo registra ter podrobno pogledati implementacijo dveh različnih tehnologij verige blokov in njihovih zmožnosti. Prva od teh tehnologij je Ethereum [23], ki jo poznamo predvsem po kriptovaluti Ether. Poleg tega, da gostuje omenjeno kriptovaluto, je bila prva, ki nam je omogočila ustvarjanje t. i. pametnih pogodb (*angl. smart contracts*) in decentraliziranih aplikacij. Druga tehnologija, katero si bomo pogledali, je Hyperledger Fabric. Je eden izmed produktov iniciative Hyperledger ali Hyperledger project [14], ki je bila ustanovljena leta 2015 s strani Linux Foundation-a [13] skupaj s mnogimi podporniki (IBM, Cisco, Intel, Red Hat, ...). Zavzemajo se za razvijanje naprednih odprtokodnih rešitev tehnologije verige blokov s poudarkom na uporabnosti v industriji.

Najprej si bomo v naslednjem poglavju pogledali, kako smo zasnovali projekt registra vpogleda osebnih podatkov. Sledi opis uporabljenih tehnologij. Podrobneje bomo opisali tehnologijo verige blokov in obe implementaciji. Nato bomo na kratko predstavili izdelavo registra, bolj se bomo osredotočili

na to, kako smo vključili posamezno tehnologijo verige blokov v naš projekt. Omenili bomo težave in omejitve, s katerimi smo se srečali, ter predstavili načine za njihovo reševanje. V predzadnjem poglavju si bomo pogledali primerjavo obeh implementacij verige blokov. Za konec bomo strnili naše ugotovitve in se ozrli na nadaljnji razvoj in izboljšave.

Poglavje 2

Zasnova projekta

Poglejmo si, kako je bil register zasnovan, kakšne so njegove funkcionalnosti ter zmožnosti. Glavna funkcionalnost registra je beleženje vpogledov v osebne podatke posameznika znotraj različnih aplikacij in sistemov podjetij ali ustanov.

Pri zasnovi projekta smo imeli tri glavne cilje. Prvi in glavni cilj je bil izdelati register, ki je sposoben beležiti vpoglede v osebne podatke posameznika znotraj različnih aplikacij in sistemov podjetij ali ustanov ter hraniti revizijsko sled teh, hkrati pa omogočiti čim lažjo integracijo v obstoječe sisteme. Drugi cilj je zajemal zagotavljanje varnosti shranjevanja podatkov o vpogledih, tako da jih ni mogoče enostavno prirediti, zakriti ali izbrisati. Zadnji cilj je bil ponuditi vmesnik, preko katerega omogočimo dostop do teh podatkov pooblaščenim osebam.

Za uresničitev prvega cilja smo se odločili, da bo register beležil vpoglede preko vhodnega spletnega vmesnika, na katerega aplikacije pošiljajo opise dogodkov (vpogledov) v JSON obliki preko HTTP POST zahtevka. Tako smo zasnovali splošen vhodni vmesnik, ki lahko beleži vpoglede iz različnih sistemov, saj morajo te poskrbeti le za pošiljanje zahtevkov s pravilno vsebino v naš register.

Posamezen opis dogodka vsebuje razne podatke o vpogledu, med njimi nekaj obveznih:

- tip in časovni žig vpogleda;
- podatki uporabnika in izvornega sistema ali aplikacije;
- podatki o osebah, katerih podatki so bili obdelani oziroma vpogledani;
- podatki o dokumentu, v katerem nastopajo.

Tip vpogleda nam določi, kakšna obdelava podatkov se je zgodila. Poznamo 5 osnovnih tipov vpogleda:

- **C** - ustvarjanje (*Create*) novega dokumenta, določenega tipa, ki je vezan na eno ali več oseb;
- **R** - bralni (*Read*) dostop v določen dokument;
- **U** - posodobitev (*Update*) podatkov dokumenta;
- **D** - brisanje (*Delete*) brisanje dokumenta;
- **E** - izvoz (*Export*) podatkov.

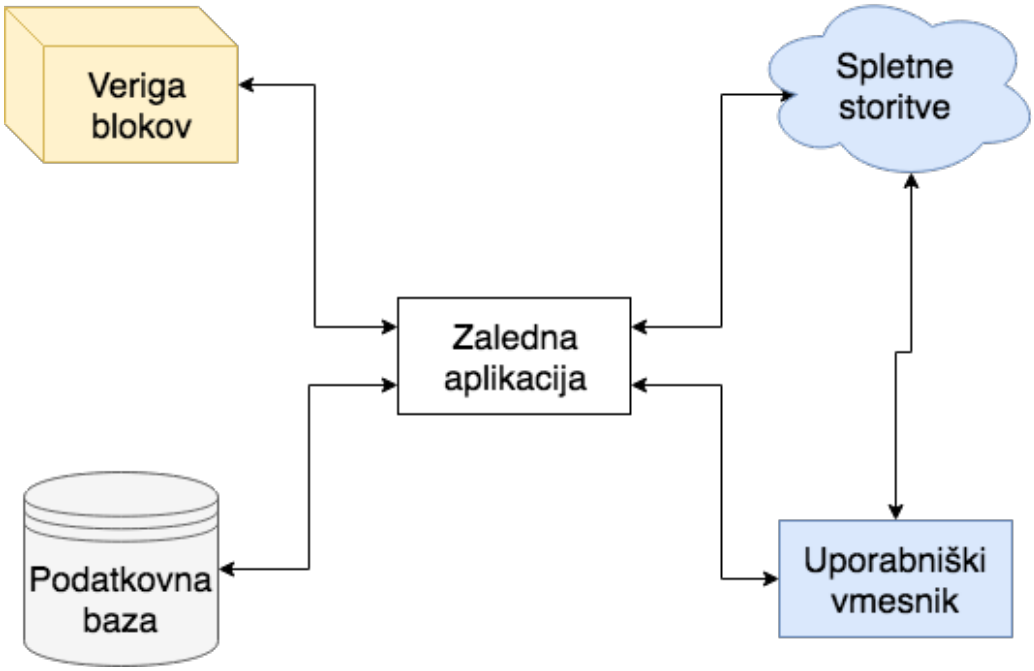
Predpostavljamo, da so osebni podatki posameznika vedno navedeni v nekem dokumentu določenega tipa, kateri točno definira, katere osebne podatke vsebuje. Dokument si lahko prestavljamo kot papirnati dokument (npr. pogodba o zaposlitvi, aneksi, zdravstveni izvidi, ...) ali virtualni zapis podatkov, pomembno je le, da ga izvorna aplikacija identificira z enoličnim identifikatorjem. Tip dokumenta nam pove, kateri osebni podatki lahko v njem nastopajo, kot tudi čas veljavnosti in hrambe ter datum izbrisa dokumenta. Zato moramo pred integracijo v obstoječe sisteme pridobiti podatke o tipih dokumentov s katerimi upravljajo.

Podatki o posamezniku in dokumentih se hranijo v podatkovni bazi. Pred shranjevanjem se osebni podatki in drugi identifikatorji kriptirajo, tako da niso prosto berljivi. Okleščeni opisi vpogledov se nato shranijo na verigo blokov. Ker so podatki na verigi trajni in neizbrisljivi, imamo na njej zapisane samo osnovne podatke vpogleda. Vsebujejo pa interni identifikator dokumenta, preko katerega lahko zapise povežemo z drugimi podatki v bazi.

Register s pomočjo teh podatkov vodi revizijsko sled za vsak dokument. Sled mora slediti logičnemu zaporedju, kar pomeni, da mora biti za posamezen dokument najprej poslan dogodek za ustvaritev dokumenta. Temu lahko sledi poljubno zaporedje vpogledov, ki označujejo branje, posodobitev ali izvoz podatkov. Sled se zaključi z dogodkom brisanja. Register zazna neregularne vpoglede, ki ne sledijo logičnemu sosledju, in na te opozori v uporabniškem vmesniku, kjer se jih lahko razreši. Neregularni dogodki so dogodki, ki ne sledijo logičnemu sosledju, kot vpogled v še ne ustvarjen dokument ali večkratni dogodek ustvarjanja dokumenta z istim identifikatorjem.

Uporabniški vmesnik omogoča pooblaščenim osebam vpogled v evidenco obdelav osebnih podatkov. Omogoča filtriranje po identifikatorju osebe ali dokumenta, pregled dokumentov, dodajanje tipa dokumenta in prikaz neregularnih dogodkov, ki jih zazna. Posamezne funkcionalnosti se omogočijo ali onemogočijo glede na pravice uporabnika.

Naš register je tako sestavljen iz več delov, ki so med seboj povezani, kot vidimo na sliki 2.1. Osrednji del registra je zaledna aplikacija, ta vsebuje vso logiko za shranjevanje in poizvedovanje evidenc obdelav. Prav tako skrbi za komunikacijo in pretok informacij med ostalimi deli aplikacije, ki so na kratko opisani v nadaljevanju. Relacijska podatkovna baza, v kateri so shranjeni podatki o posamezniku, dokumentih in tipih. Veriga blokov na kateri so shranjeni opisi vpogledov. Spletni vhodni vmesnik, preko katerega sprejema nove dogodke. Uporabniški vmesnik, ki je namenjen pooblaščenim osebam in služi poizvedovanju evidenc.



Slika 2.1: Osnovna shema aplikacije

Poglavje 3

Uporabljene tehnologije

3.1 Veriga blokov

Veriga blokov je novejša tehnologija z začetki v letu 1991, ki predstavlja kriptografsko povezan seznam blokov, kateri vsebujejo shranjene informacije. Prva prava implementacija se je pojavila šele leta 2008, ki jo danes poznamo kot eno največjih in najbolj razširjenih kriptovalut Bitcoin [19]. Kriptovalute uporabljajo verigo blokov kot decentralizirano javno knjigo vseh transakcij oziroma prenosov virtualnih kovancev med uporabniki. Knjiga transakcij je dostopna vsakemu udeležencu omrežja, tako lahko vsak preveri, kakšno je stanje omrežja. Ni več potrebe po centralni entiteti (v tem primeru banke), ki bi nadzirala, koliko kovancev ima kdo in komu je kdo koliko nakazal, saj omrežje samo skrbi za veljavna stanja in transparentne prenose kriptovalut.

Glavni prednosti verige blokov sta decentralizacija baze podatkov, kar pomeni, da podatkov nima samo ena glavna entiteta, ampak jih imajo vsi udeleženci v omrežju. Druga pomembna lastnost je nespremenljivost podatkov. Veriga blokov je v zasnovi odporna na poseganje v shranjenje podatke in spreminjanje teh.

3.1.1 Osnovni pojmi

Implementacij tehnologije verige blokov je danes že precej, vsaka z določenimi izboljšavami ali popravki. Veliko si jih je med seboj podobnih, na primer, nekatere popularnejše kriptovalute so kloni Bitcoina (Litecoin, Dogecoin, Monero, Dash). Nekatere ponujajo dodatno funkcionalnost pametnih pogodb, te označujemo kot verige blokov druge generacije (Ethereum). Večina teh pa deluje po istem principu oziroma imajo podobno logiko delovanja.

Sledi opis nekaterih osnovnih pojmov, ki so skupni večini verig in jih moramo poznati za razumevanje samega delovanja tehnologije verige blokov. Pri opisu se bomo osredotočili na implementacije, kakršne imajo nekatere glavne verige (Ethereum, Bitcoin).

Omrežje je sestavljeno iz poljubnega števila vožlišč (*angl. nodes*), ki med seboj komunicirajo po omrežni arhitekturi vsak z vsakim (*angl. peer to peer*).

Poznamo več vrst omrežij. Z napredkom tehnologije se pojavljajo vedno nova, v grobem pa jih lahko razdelimo na sledeče tri tipe; odprti, zaprti in mešani tip omrežja.

Odprti tip pomeni, da se lahko omrežju pridruži vsak, ki ima dostop do interneta in postavi svoje vozlišče. Vsako vozlišče lahko generira nove transakcije in bloke, komunicira z drugimi udeleženci omrežja in pomaga pri delovanju (validaciji novih blokov) omrežja. Hkrati pa so vozlišča delno anonimna, saj jih določa le internetni naslov (*angl. IP address*). Primer takega tipa so kriptovalute (npr. Bitcoin, Ethereum, Litecoin), kjer se lahko vsakdo pridruži omrežju, pridobi svoj naslov (denarnico) in začne upravljati s svojimi virtualnimi sredstvi.

Zaprti tip uporabljajo predvsem podjetja in konzorciji za učinkovit, zaupen in transparenten prenos informacij. Vozlišča omrežja so vnaprej definirana, določeno je, katera skrbijo za delovanje omrežja. Dostop do podatkov je možen samo znotraj podjetja ali konzorcija. Take tipe razvijajo pri Hyperledger projektu.

Mešani tip je podoben zaprtemu, le da so podatki dostopni tudi javnosti.

Za verifikacijo in delovanje omrežja še vedno skrbijo določena vozlišča, drugi pa imajo omejen dostop do podatkov ali nekaterih funkcij.

Vozlišča so programska oprema, implementirana po določenem protokolu verige blokov (npr. Bitcoin ali Ethereum). Omogočajo nam priključitev v omrežje, kreiranje denarnice, pošiljanje transakcij in pregled zgodovine blokov. Vozlišča, ki hranijo celotno zgodovino verige blokov, se imenujejo polna vozlišča (*angl. full nodes*). Ta lahko sodelujejo tudi pri generiranju novih blokov ter podpori omrežja. Ker je celotna zgodovina nekaterih verig že v stotih gigabajtih (celotna Bitcoin zgodovina zasede več kot 200 gigabajtov prostora, zgodovina Ethereuma pa kar 700), so se pojavila vozlišča, katera hranijo le zadnjih nekaj blokov (*angl. light nodes*). Taka vozlišča v ozadju komunicirajo s polnim vozliščem, primerna pa so predvsem za upravljanje denarnice.

Denarnica je ločena funkcionalnost vozlišča ali samostojna programska oprema, ki komunicira z oddaljenim vozliščem. Določena je s parom javnega in privatnega kriptografskega ključa, katerima pripada en ali več naslovov na verigi blokov. Naslovi so navadno izpeljani iz javnega ključa denarnice. Generiranje naslova je odvisno od določenega protokola, ponavadi se pridobi s pomočjo različnih zgoščevalnih funkcij (Bitcoin uporablja HASH160, Ethereum pa Keccak-256). Naslovi se uporabljajo za prenos kriptovalut med uporabniki in za beleženje stanja denarnice. Sama denarnica ne hrani vrednosti, njeno stanje je implicitno določeno z vsemi prejetimi in poslanimi transakcijami na oziroma z njenih naslovov. Privatni ključ se uporabi za podpisovanje poslanih transakcij, tako je zagotovljena verodostojnost pošiljatelja. Denarnica nam omogoča pošiljanje transakcij (kovancev) in pregled našega stanja. Pri Ethereumu je še potrebna za kreiranje pametnih pogodb ter klicanje njihovih funkcij.

Zgoščevalne funkcije (*angl. hash function*) so algoritmi, ki iz poljubnega vhodnega sporočila na izhod vrnejo binarno vrednost fiksne dolžine (zgoščena

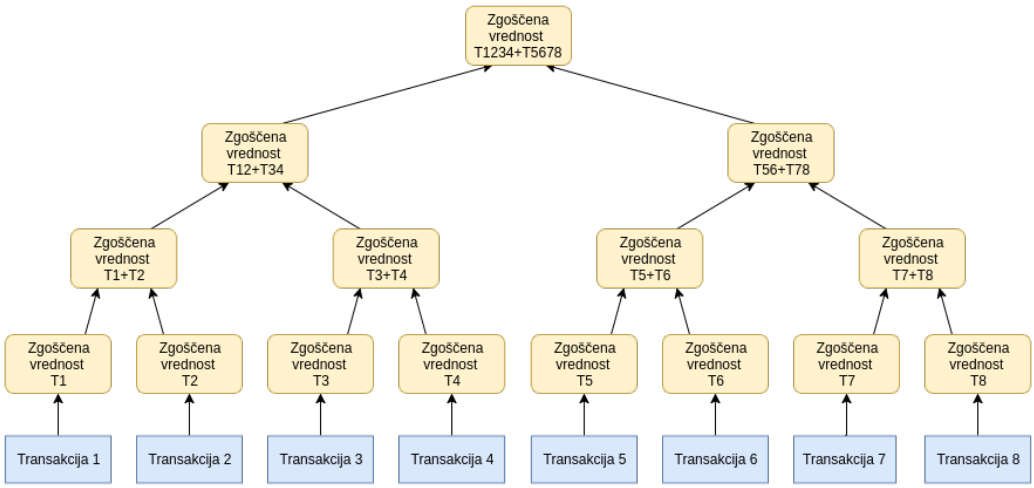
vrednost). Imajo dve pomembni lastnosti, in sicer, da so nepovratne, kar pomeni, da je skoraj nemogoče najti vhodno sporočilo, četudi poznamo izhodno vrednost, in da že majhna sprememba vhodnega sporočila povzroči generiranje povsem drugačnega izhoda.

Zaradi teh lastnosti so zelo uporabne v tehnologiji verige blokov, saj lahko s pomočjo zgoščenih vrednosti enostavno preverimo, če so transakcije in bloki ostali nedotaknjeni. Različne implementacije verige blokov uporabljajo različne funkcije, med širše uporabljenima sta SHA-256 (pri Bitcoinu) ali SHA-3 (pri Ethereumu).

Transakcije se uporabljajo za prenos kriptovalut med denarnicami ali za interakcijo s pametnimi pogodbami, kjer tehnologija to omogoča. Ko ustvarimo novo transakcijo in se ta podpiše s privatnim ključem denarnice, jo razpršeno oddamo vsem drugim vozliščem. Vozlišča take, še nepotrjene, transakcije hranijo v svojem bazenu transakcij (*angl. mempool*), dokler jih katero od vozlišč ne spravi v blok. Ko je enkrat transakcija shranjena v bloku, kateri je priključen verigi, se smatra kot potrjena. Transakciji, poleg vrednosti, katero hočemo poslati, dodamo še plačilo (*angl. fee*), ki služi kot nagrada vozlišču, ki je generiralo blok. Transakcijam, ki komunicirajo s pamentimi pogodbami, lahko dodamo še podatke, kateri služijo kot vhodni parametri funkcijam.

Zgoščeno dvojiško drevo transakcij dobimo tako, da za liste drevesa uporabimo vse transakcije, ki so vsebovane v bloku. Nato za vsako transakcijo izračunamo njeno zgoščeno vrednost iz njenih podatkov. Potem se premikamo proti korenu drevesa in na vsakem nivoju za posamezno vozlišče izračunamo zgoščeno vrednost združenih zgoščenih vrednosti otrok. To ponavljamo, dokler ne pridemo do korena drevesa in dobimo ene same vrednosti.

Primer postopka izračuna je predstavljen tudi na sliki 3.1. Končna vrednost se potem shrani v glavo bloka in nam omogoča enostaven način preverjanja pristnosti transakcij v bloku.

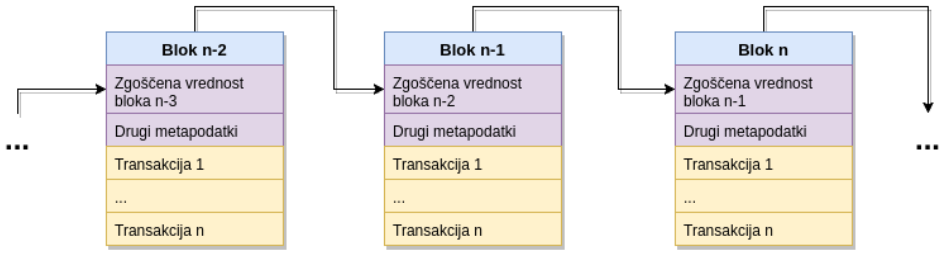


Slika 3.1: Diagram postopka za izračun končne vrednosti zgoščenega binarnega drevesa

Bloki so glavni členi verige. V osnovi so sestavljeni iz glave (*angl. block header*) in množice transakcij. Glava vsebuje razne metapodatke, ki nosijo informacijo o dotičnem bloku. Kateri metapodatki so zapisani v glavi, je odvisno od posamezne implementacije, med njimi je ponavadi podatek o verziji, časovnem žigu, številu vsebovanih transakcij, zgoščeni vrednosti predhodnega bloka in vrednosti zgoščenega dvojiškega drevesa transakcij [12] (*angl. Merkle tree oziroma hash tree*). Tako drevo zakodira vse transakcije bloka v eno samo vrednost. Prednost drevesa je, da moramo za dokaz, da je neka transakcija vsebovana, izračunati le logaritemsko število zgoščenih vrednosti glede na število listov drevesa.

Posamezen blok se enolično identificira z zgoščeno vrednostjo njegovih metapodatkov. Lahko ga referenciramo tudi z njegovo zaporedno številko v verigi, vendar je možno, da ima več blokov enako vrednost, več o tem nekoliko pozneje.

Veriga blokov je torej enosmerno povezan seznam blokov, kjer se za kazalec uporablja zgoščena vrednost predhodnega bloka, kot je prikazano na sliki



Slika 3.2: Poenostavljena shema verige blokov

3.2. Veriga se začne z izhodiščnim blokom (*angl. genesis block*), kateri je ponavadi zapečen v samo programsko kodo in je edini, ki nima predhodnika. Verigi se nato postopoma priključujejo novi bloki, katere generirajo vozlišča z novimi transakcijami.

Generiranje novih blokov je proces, pri katerem se ustvari nov blok transakcij, ki se nato lahko priključi verigi. Generiranje poteka tako, da vozlišče izbere podmnožico nepotrjenih transakcij iz svojega bazena in po specifičnem protokolu soglasja (*angl. consensus protocol*) ustvari nov blok transakcij. Vozlišča prednostno izbirajo transakcije z višjim plačilom, saj ta znesek dobijo kot nagrado za uspešno ustvarjen blok. Blok se nato razpršeno pošlje po celotnem omrežju. Omrežje preveri verodostojnost bloka in ga potrdi ter priključi verigi ali zavrne. Večina verig ima definiran časovni interval za generiranje novega bloka (*angl. block time*). Pri Bitcoinu je ta čas približno 10 minut, pri Ethereumu pa samo 14 sekund.

Vejitev verige (*angl. forking*) je pojav, ki nastane, ko se veriga na nekem mestu razcepi na dva ali več delov. Ločimo med namernim in naključnim vejanjem.

Naključno vejanje nastane, kadar se skoraj istočasno generira več ustreznih blokov transakcij. Tako lahko različni deli omrežja dodajo v verigo drug blok transakcij, zaradi česar je lahko stanje omrežja nekaj časa neusklajeno.

Vejanje se samodejno razreši, ko ena od verig preseže dolžino drugih. Krajše verige se zavržejo, omrežje pa prevzame stanje, kakršno ima najdaljša veriga. Tako vejanje je lahko pogosto, zato moramo počakati nekaj blokov, da lahko zanesljivo trdimo, da je bila neka transakcija res potrjena.

Namerno vejanje predstavlja spremembo samega protokola, po katerem komunicira omrežje. V tem primeru je potrebno posodobiti vozlišča, da generirajo in pravilno validirajo generirane bloke po novem protokolu. Nekaj vejanj je bilo izvedenih tudi zato, da se je izničilo posledice vdora v omrežje.

Protokol soglasja določa, kako je poskrbljeno za varnost v samem omrežju, validacijo novega bloka in kako se vozlišča ukladijo glede stanja verige. Obstaja več različnih, vsak ima svoje prednosti in slabosti.

Trenuntno je najbolj razširjen algoritem POW (*angl. proof of work*), katerega uporabljata tudi omrežji Bitcoin in Ethereum. Pri tem algoritmu se je pojavil izraz rudarjenje (*angl. mining*), katerega pomen sledi v nadaljevanju.

Da vozlišče ustvari legitimen nov blok, kateri se lahko priključi verigi, mora generirati zgoščeno vrednost bloka, tako da ta ustreza težavnosti. Težavnost določa število ničel, s katerimi se mora začeti zgoščena vrednost bloka, da se smatra kot veljaven. Kot smo omenili, se zgoščeno vrednost bloka izračuna iz metapodatkov glave. Eden izmed teh podatkov je `nonce`. To je vrednost, katero vozlišče neprestano spreminja, s čimer poskuša dobiti zahtevano zgoščeno vrednost glave. Ko vozlišču uspe dobiti zahtevano zgoščeno vrednost, blok razpošlje po omrežju. Druga vozlišča lahko enostavno preverijo, če ustreza zahtevam, tako da sama izračunajo zgoščeno vrednost podatkov, in če ustreza, se ta blok priključi verigi.

Ta operacija je zelo zahtevna oziroma je potrebno ogromno poskusov, da nam uspe najti pravo zgoščeno vrednost. Prvotno se je ta operacija izvajala samo na procesorjih, nato so se začele uporabljati grafične kartice, saj so zaradi svoje arhitekture sposobne opraviti veliko več izračunov. Z omenjeno težavnostjo se regulira tudi čas med generiranimi bloki. Več kot je rudarjev, vozlišč katera ustvarjajo nove bloke, hitrejše bodo našli zahtevano zgoščeno

vrednost, zato je potrebno prilagajati težavnost, da ostane čas med bloki približno enak.

Razvijajo se vedno novi algoritmi, kateri bi nadomestili POW. Njegov problem je velika poraba električne energije, saj rudarjenje danes poteka predvsem na grafičnih karticah, katere porabijo precej energije. V podrobno delovanje ostalih se ne bomo spuščali, saj niso pomembni za temo diplomske naloge, ampak jih bomo samo omenili. V zadnjem času razvijajo predvsem algoritem POS (*angl. proof of stake*), kjer vozlišča zagotovijo veljaven blok tako, da zastavijo nekaj svojega denarja. Obstajata še dva protokola, ki sta trenutno pospešeno v razvoju, DPOS (*angl. delegated proof of stake*) in dBFT (*angl. delegated byzantine fault tolerance*).

3.2 Ethereum

Ethereum je odprtokodna platforma odprtega tipa, ki temelji na tehnologiji verige blokov in ponuja funkcionalnost poganjanja pametnih pogodb. Implementirana je v večih jezikih, Go, C++ in Rust. Omrežje je bilo prvič zagnano konec julija 2015, od takrat pa je postala najbolj priljubljena platforma za razvijanje aplikacij, ki temeljijo na tehnologiji verige blokov. Na njej živi tudi zelo razširjena kriptovaluta Ether, kakor tudi mnogi drugi žetoni, ki so bili ustvarjeni s pomočjo pametnih pogodb. Osnovna enota Ethra se imenuje Wei in predstavlja 10^{-18} delež Ethra.

Pametne pogodbe se izvajajo znotraj navideznega stroja Ethereum Virtual Machine (EVM). EVM je del vsakega vozlišča, kar pomeni, da se pametne pogodbe izvajajo pri vsakem udeležencu omrežja. Trenutno uporablja še POW protokol soglasja, vendar bo z naslednjo verzijo prešel na POS.

Spodaj bomo pogledali nekaj posebnosti samega Ethereum protokola, katere so zanimive z vidika implementacije našega registra.

Pametne pogodbe so programska koda, ki živi na dodeljenem naslovu na verigi. Vsaka pogodba ima svojo kodo, notranje stanje in stanje v Ethru. Iz-

vajanje se sproži s transakcijo ali sporočilom druge pogodbe, ob tem se izvede poljubno kompleksno zaporedje ukazov, ki lahko manipulira notranje stanje ali pokliče druge pametne pogodbe. Ethereum transakcije imajo dodatno polje `data`, ki služi za pošiljanje vhodnih parametrov funkcijam pametnih pogodb in za sprejem vrednosti, ki jih te lahko vrnejo.

Pametne pogodbe so navadno napisane v domensko specifičnem jeziku, ki se potem prevede v strojno kodo, katera se zapiše na verigo blokov pod določen naslov. Najbolj razširjen jezik za pisanje pogodb je Solidity [20]. Še vedno je v razvoju, vendar omogoča večino programskih konstruktov, ki jih poznamo v sodobnih programskih jezikih (zanke, strukture, funkcije, pogojni stavki, ...).

Spodaj imamo primer enostavne pogodbe napisane v Solidity jeziku. Prvi ukaz je zahteva po minimalni verziji prevajalnika, v našem primeru 0.4.0. Sestavljena je samo iz enega nepredznačenega celega števila `storedData` in dveh funkcij `set`, ki sprejme vhodni parameter `x` in nastavi vrednost spremenljivke in `get`, ki vrne nastavljeno vrednost.

```
pragma solidity ^0.4.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

Gorivo (*angl. Gas*) je posebna enota na platformi, s katero merimo strošek izvajanja pametne pogodbe oziroma transakcij. Vsak strojni ukaz ima vnaprej določeno ceno goriva [7]. Seštevek cen ukazov določene funkcije znotraj pogodbe nam pove, koliko goriva potrebuje, da se lahko izvede. Ceno posamezne enote goriva določa omrežje samo, transakcijam podamo samo zgornji meji; koliko enot smo pripravljeni plačati (*gas limit*) in ceno posamezne enote (*gas price*). Končna cena izvedbe funkcije se nato izračuna tako, da se dejansko število porabljenih enot goriva pomnoži z ceno ene enote. Cena goriva je merjena v Ethru, tako da je tudi končna cena, ki jo moramo plačati, v Ethru.

Ker je izvajanje pogodb in njihovih funkcij drago, saj se te izvajajo na vseh vozliščih omrežja, uvedba goriva spodbuja pisanje dobre kode, preprečuje pojav neskončnih zank in zmanjšuje smetenje po verigi.

Z gorivom je omejena tudi velikost posameznega bloka. Seštevek porabe goriva vseh transakcij mora biti manjša od vrednosti, ki je določena glede na trenutno stanje omrežja. Ta vrednost se stalno spreminja. Če je transakcij manj in so bloki bolj prazni, se zmanjša, drugače se povečuje.

3.3 Hyperledger Fabric

Hyperledger Fabric je ena od implementacij verige blokov znotraj Hyperledger projekta, ki je namenjena poslovnim aplikacijam. Omogoča postavitev modularnega omrežja zaprtega tipa oziroma privatne verige blokov. Pri Fabric projektu se osredotočajo na zmogljivost, skalabilnost omrežja in modularnosti posameznih delov.

Kakor Ethereum, omogoča pisanje in poganjanje pametnih pogodb (*angl. chaincode*), v sami implementaciji pa se tehnologiji močno razlikujeta. Fabric trenutno podpira pisanje pogodb v programskih jezikih Go in Node.js, načrtujejo pa tudi podporo Jave. Te razlike so nastale zaradi različnega tipa omrežja in višjenivojskega pogleda na implementacijo pri Fabric projektu. Nekaj glavnih razlik Fabric omrežja glede na Ethereum:

- vsa vozlišča so znana in vnaprej registrirana;
- ločevanje funkcionalnosti vozlišč glede na tip;
- bolj zapletena arhitektura omrežja;
- nima implementirane svoje valute;
- za transakcije in pogodbe ni potrebno plačati;
- hrani bazo trenutnega stanja omrežja v podatkovni bazi (*angl. state database*);
- preprosto poizvedovanje podatkov;
- implementira zaupne kanale.

Fabric loči med tremi tipi vozlišč, vsak tip opravlja svojo funkcijo v omrežju:

Vrstniška vozlišča izvajajo pametne pogodbe, dostopajo do podatkov na verigi in komunicirajo z aplikacijami. Lahko imajo vlogo odobravanja transakcij.

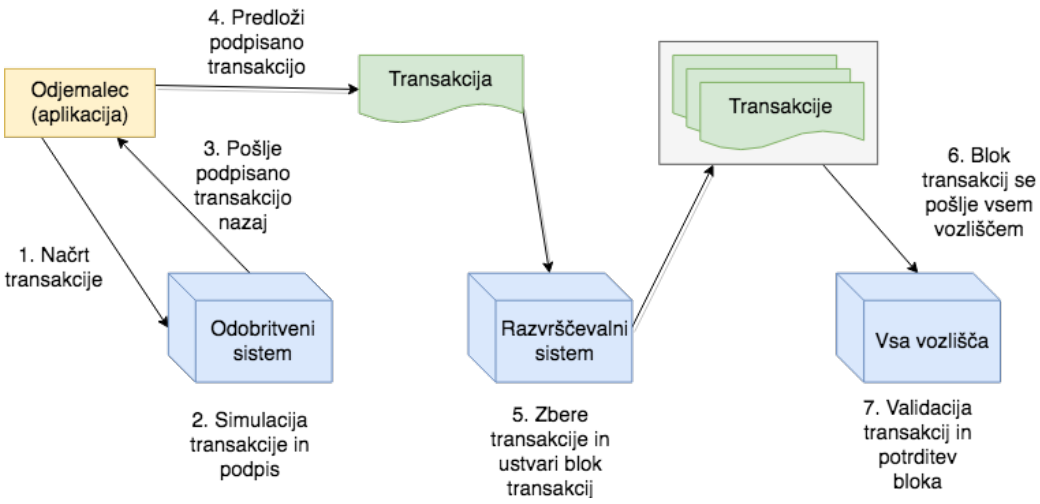
Uredniška vozlišča zagotavljajo konsistenco omrežja in skrbijo za zanesljivo dostavljanje transakcij (blokov) drugim vozliščem v omrežju.

Odjemalci prožijo nove transakcije in dostavljajo podpisane transakcije razvrščevalnemu sistemu. Morajo biti povezani z vrstniškim vozliščem, da lahko komunicirajo z verigo blokov.

Poleg tega imajo tudi ločeno komponento MSP (*angl. membership service provider*), katera skrbi za izdajanje in preverjanje certifikatov vozlišč in overjanje uporabnikov.

Zaradi različnih tipov vozlišč je posledično bolj zapletena arhitektura omrežja in potrjevanje transakcij. Medtem ko se te pri Ethereumu samo razpršeno oddajo med vsa vozlišča, je tukaj postopek bolj definiran.

Potrjevanje transakcij je prikazano z diagramom na sliki 3.3. Ko določena aplikacija ustvari transakcijo, se ta najprej pošlje določeni množici vozlišč, katera sestavljajo odobritveni sistem. Odobritveni sistem sestavljajo vozlišča, katera so določena s pametno pogodbo, simulirajo izvajanje pametne pogodbe, ustvarijo množici bralnih in pisalnih dostopov pogodbe in podpišejo transakcijo. Podpisana transakcija se pošlje nazaj aplikaciji, katera jo posreduje naprej razvrščevalnemu sistemu. Ta ustvari blok transakcij in jih dostavi ostalim vozliščem. Ko ta prejmejo blok, za vsako transakcijo še preverijo, če je odobrena od vseh zahtevanih vozlišč in če se množici pisalnih in bralnih dostopov ne prekrivata. Če ni težav, se blok doda verigi in trenutno stanje omrežja v bazi se posodobi. Omenimo še, da Fabric ne generira novih blokov, če ni novih transakcij.



Slika 3.3: Diagram toka potrjevanja transakcij

Baza trenutnega stanja hrani trenutno stanje verige, kar omogoča enostavno in hitro poizvedovanje podatkov v verigi. Pogodbe vršijo svoje operacije nad podatki v verigi, za hitro izvajanje teh operacij so v bazi shranjeni vsi podatki, ki so bili kadarkoli spremenjeni. Vsak podatek je shranjen pod svojim ključem, tako se ni potrebno sprehajati po celi verigi, da bi dobili

določeno iskano vrednost. Za privzeto bazo uporablja LevelDB [18], druga možnost je CouchDB [2]. CouchDB je uporaben predvsem, kadar shranjujemo podatke v JSON obliki, saj ponuja obogatene poizvedbe po ključih ali vrednostih.

Kanali so zasebna podomrežja, sestavljena iz dveh ali več vozlišč, ki omogočajo privatno in zaupno komunikacijo med njimi. Vsak kanal hrani ločeno verigo in trenutno stanje, katero je vidno samo vozliščem, ki so pridruženi kanalu. S kanali lahko ustvarimo varen komunikacijski kanal med določenimi udeleženci.

Da povzamemo, Hyperledger Fabric projekt je namenjen poslovnim aplikacijam, katere združujejo ali beležijo različne delovne procese. Njegov povdarek je na zmogljivosti, skalabilnosti in privatnosti omrežja. Pisanje pametnih pogodb je mogoče v že poznanih programskih jezikih, tako se nam ni potrebno učiti še enega. Če uporabimo CouchDB bazo stanja, nam izbira omogoča shranjevanje celotnih JSON struktur in bogate poizvedbe kar po JSON objektih.

3.4 Splošno

V tem razdelku naštejemo in na kratko predstavimo ostale tehnologije, ki smo jih uporabili za izvedbo registra vpogledov.

Java EE (Java Enterprise Edition) je izpeljana iz osnovne različice Java SE in ponuja dodatne knjižnice, ki ponujajo funkcionalnosti za implementacijo programskih rešitev, ki temeljijo na odjemalec-strežnik arhitekturi. Omogoča razvoj varnih, večnivojskih in modularnih spletnih aplikacij.

Wildfly je odprtokodni aplikacijski strežnik, na katerem je mogoče poganjati aplikacije, napisane v Java EE. Pohvali se lahko z majhno porabo sistemskih virov in močno optimizacijo zagona strežnika.

PostgreSQL je odprtokodni objektno-relacijski sistem za upravljanje podatkovnih baz, ki pripisuje veliko veljavo skladnosti standardom. Primeren je tako za manjše, kot velike projekte z večimi sočasnimi uporabniki. Skladen je z ACID lastnostmi, ki zagotavljajo pravilno delovanje baze, tudi v primeru napak, izpadov ali drugih težav.

Hibernate je odprtokodno orodje, ki skrbi za objektno-relacijsko mapiranje Java objektov na tabele objektno-relacijske baze in Java razredov na SQL podatkovne tipe. Za mapiranje objektov na tabele se poslužuje Java anotacij. Razvijalcem omogoča lažje delo z podatkovnimi bazami iz Jave, saj avtomatično generira SQL klice in omogoča poizvedovanje ter pridobivanje podatkov iz baze.

ReactJS je JavaScript knjižnica za izdelavo interaktivnih uporabniških vmesnikov. Omogoča grajenje enostavnih komponent, ki skrbijo same za svoje stanje, združujemo jih lahko v bolj kompleksne vmesnike.

Web3j [22] je knjižnica za Javo in Android, katero smo uporabili za komuniciranje naše zaledne aplikacije z verigo blokov. Je lahka in modularno zasnovana, omogoča nam enostavno integracijo in delo z pametnimi pogodbami. Z vozlišči komunicira preko klicev za oddaljen postopek preko HTTP protokola ali vtičnice za medprocesno komunikacijo (*angl. IPC*).

Docker [1] je orodje za virtualizacijo, ki uporablja vsebniško tehnologijo (*angl. containerisation*). Z virtualizacijo lahko na istem strežniku poganjamo več aplikacij v ločenih izvajalnih okoljih. Vsebniška tehnologija skrbi za izolacijo procesov in virtualizacijo virov, do katerih procesi dostopajo.

Fauxton [9] je spletni vmesnik za CouchDB bazo. Ponuja nam osnoven vmesnik do večine funkcionalnosti baze, kot so ustvarjanje, pregled, posodabljanje in brisanje shranjenih dokumentov. Prav tako omogoča dostop do konfiguracijskih parametrov in vmesnik za replikacijo baze.

Poglavje 4

Implementacija rešitve

V tem poglavju bomo predstavili implementacijo registra. Najprej si bomo pogledali, kako smo implementirali zaledno aplikacijo, saj ta skrbi za pravilno delovanje celotnega registra in je pri obeh verzijah verig blokov enaka. Nato bomo podrobneje predstavili, kako smo vključili obe verigi blokov v aplikacijo.

4.1 Zaledna aplikacija in baza podatkov

Arhitektura zaledne aplikacije je pri obeh implementacijah ostala v veliki meri nespremenjena. Shemo vidimo na sliki 4.1, kjer so prikazani glavni deli aplikacije. Aplikacija je sestavljena iz več med seboj povezanih komponent, katere komunicirajo med sabo preko poslovnih zrn Java (*angl. Java Enterprise beans - EJB*). Posamezni deli komunicirajo tudi direktno z bazo preko Javine JDBC (*angl. Java database connectivity*) komponente.

Vhodni spletni servis za beleženje vpogledov je implementiran po RESTful arhitekturi z uporabo Javine JAX-RS tehnologije [16]. Preko HTTP POST zahtevka sprejme podatke v JSON podatkovni obliki. Servis je povezan s podatkovno bazo, da zabeleži nove osebe ali dokumente iz podatkov vpogleda. Prav tako komunicira z JMS vrsto, kamor odlaga sporočila, v katerih je Javin objekt, ki vsebuje podatke vpogleda. Pobiralec je zadolžen za

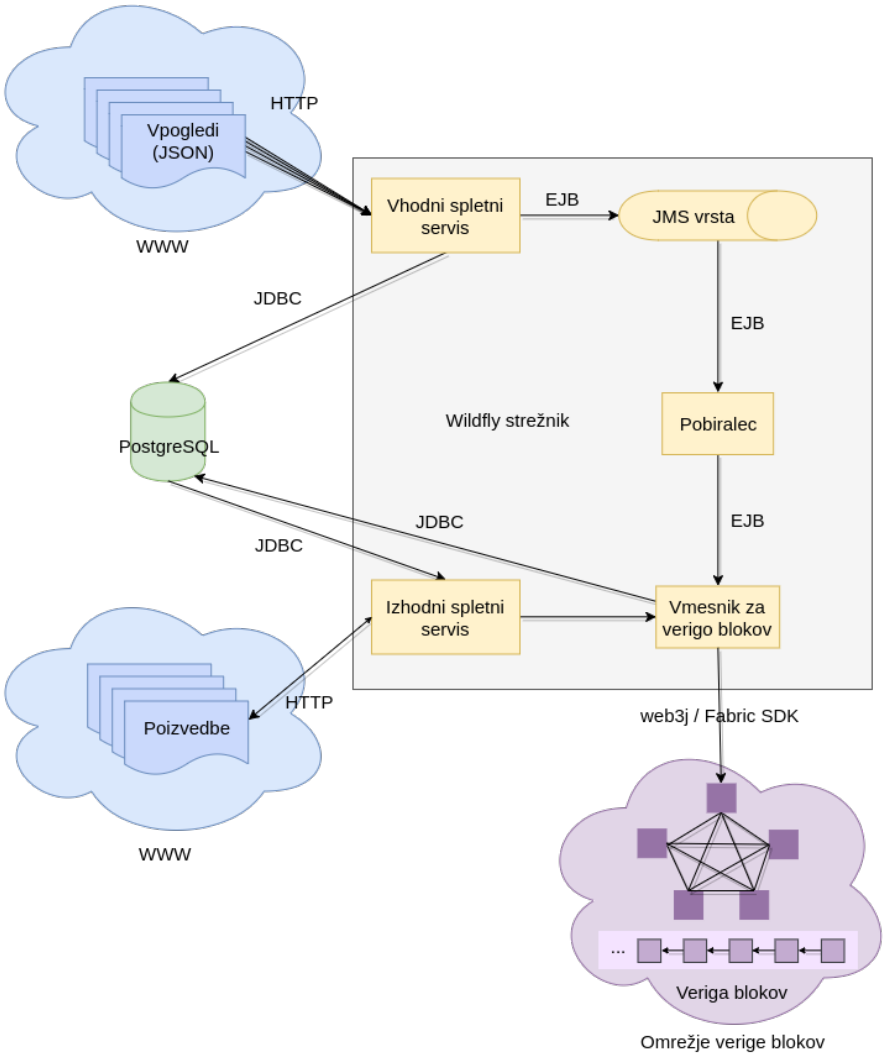
pobiranje sporočil z vrste, katera potem posreduje vmesniku za komunikacijo z verigo blokov. Vmesnik je pri posamezni implementaciji drugačen, saj mora upravljati z drugo verigo. Za Ethereum smo uporabili web3j knjižnico, za Fabric pa programski paket za razvoj programske opreme (*angl. SDK*) Java SDK for Hyperledger Fabric. Da lahko knjižnici komunicirata z verigo blokov in prožita transakcije, morata biti povezani z vozliščem. Vmesnik pri obeh implementacijah komunicira tudi z bazo, iz katere prebere potrebne podatke. Iz sporočila izlušči podatke in sestavi transakcijo, katero preko vozlišča odda v omrežje. Izhodni spletni servis ima različne končne točke, katere sprejmejo HTTP GET in POST zahtevke. Preko njega lahko poizvedujemo po bazi in verigi blokov za željene informacije.

Podatkovna baza vsebuje podatke o osebah, dokumentih in razne šifrante vrednosti. Prav tako povezuje vpoglede posamezne osebe z transakcijami, v katerih je zabeležen vpogled. Občutljivi podatki se pred shranjevanjem v bazo kriptirajo. Kriptiranje uporabimo zato, da podatki niso berljivi s prostim očesom v bazi. Tako lahko v primeru vdora v podatkovno bazo zagotovimo neuporabnost podatkov, pri čemer je predpogoj primerno zaščiten kriptirni ključ. Za kriptiranje smo uporabili simetrično kritpografijo, in sicer algoritem AES [4].

4.2 Tok informacij

Diagram 4.2 prikazuje natančen tok podatkov ob prejemu novega zapisa vpogleda.

Vhodni servis prejme opis vpogleda v JSON podatkovni strukturi. Ob prejemu se ta prevede v Javin objekt, iz katerega lahko enostavno preberemo posamezne podatke. Nato se poizve v bazi, če imamo nastopajoče podake že zavedene, drugače se vnesejo. Ko so podatki v bazi usklajeni, se celoten objekt pošlje na JMS vrsto. JMS je Javina storitev za zanesljivo asinhrono izmenjavo sporočil znotraj sistema. Z uporabo JMS-ja zagotovimo, da se določeno sporočilo res sprocesira do konca (v našem primeru shrani v blok),



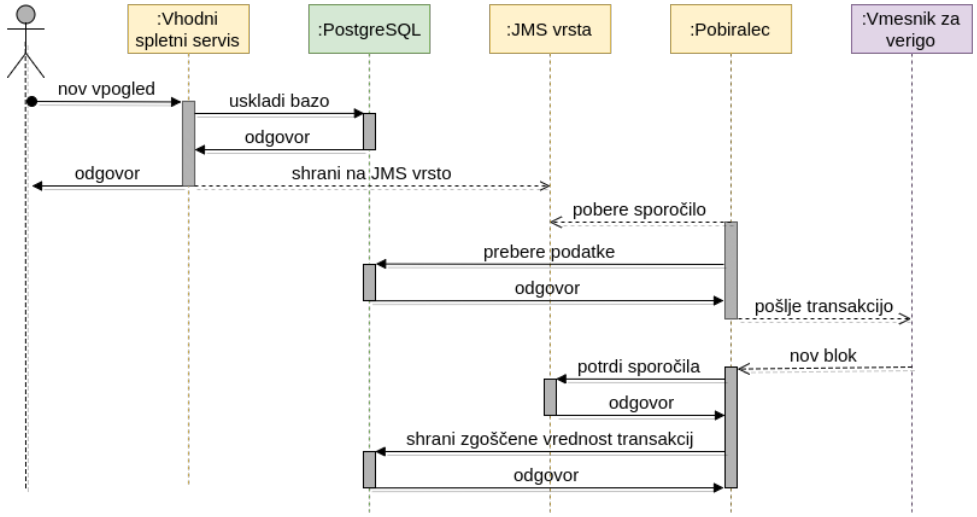
Slika 4.1: Shema zaledne aplikacije

šele nato ga potrdimo in odstranimo iz vrste. Ker je klic asinhron, se v tem trenutku zahteva za beleženje novega vpogleda konča.

Na JMS vrsto je poleg pobiralca povezan tudi MDB (*angl. Message Driven Bean*). Ta samo posluša na vrsti in ko se pojavi novo sporočilo,

sproži pobiralca. Pobiralec nato iz vrste pobere sporočilo in iz njega pridobi podatke. Del podatkov mora pridobiti še iz baze, nato vse potrebno pošlje vmesniku za verigo.

Vmesnik je povezan na vozlišče in ima vse informacije o pametni pogodbi. Iz dobljenih podatkov sestavi transakcijo, katero odda preko vozlišča v omrežje verige blokov. Ko odda transakcijo, dobi za povratno informacijo njeno zgoščeno vrednost, katero sporoči pobiralcu. Ta hrani zgoščeno tabelo transakcij in JMS sporočil. Pobiralec ima preko vmesnika registriranega poslušalca novih transakcij, tako za vsak blok izve, katere transakcije vsebuje. Ko se pojavi transakcija, katero ima shranjeno v tabeli, potrdi povezano sporočilo in to se odstrani z vrste. Če transakcija ne nastopi v naslednjih nekaj blokih, se JMS sporočilo zavrne in ponovno vrne v vrsto.



Slika 4.2: Prikaz toka informacij ob sprejemu novega vpogleda

Zgoraj opisana arhitektura zaledne aplikacije in tok informacij novega vpogleda sta pri obeh implementacijah zelo podobna.

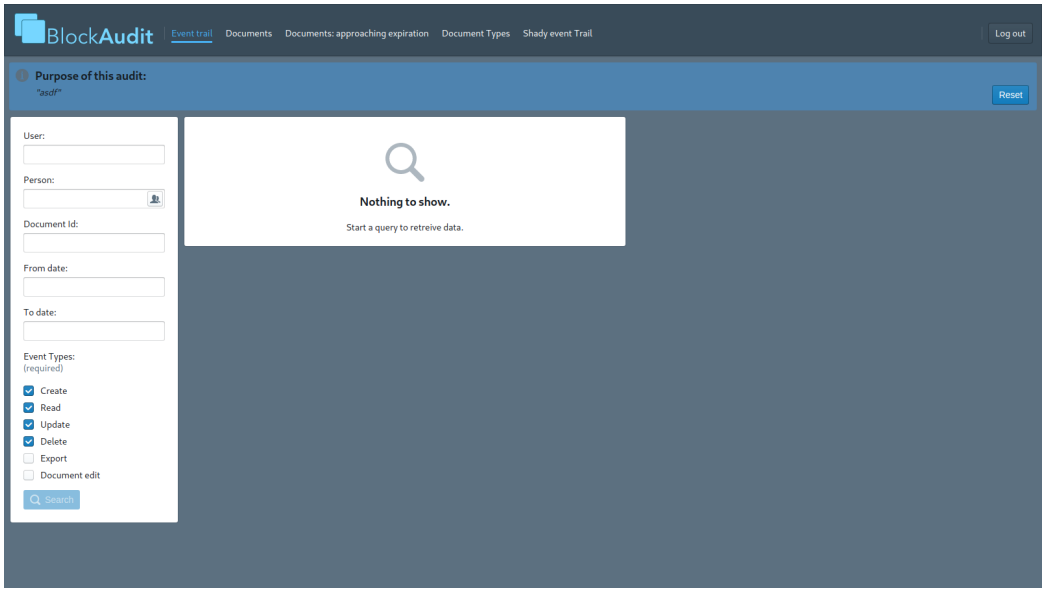
4.3 Uporabniški vmesnik

Uporabniški vmesnik je bil narejen s pomočjo knjižnice ReactJS. Podatke pridobiva s pomočjo izhodnega spletnega servisa zaledne aplikacije.

Ima pet glavnih funkcionalnosti:

- pregled revizijske sledi vpogledov;
- pregled dokumentov;
- pregled tipov dokumentov;
- pregled dokumentov, kateri bodo potekli;
- pregled neregularnih dogodkov.

Vsaka funkcionalnost je vezana na določeno pravico uporabnika, tako lahko uporabnikom omejimo dostop do podatkov.



Slika 4.3: Uporabniški vmesnik

4.4 Uporaba tehnologije Ethereum

Da smo lahko začeli s pisanjem pametnih pogodb, smo si najprej morali podrobneje pogledati programski jezik Solidity. Ker je bil razvit po vzoru C++, Pythona in JavaScripta, so nam bile strukture in programski konstrukti hitro domači.

Odločili smo se, da postavimo privatno razvojno omrežje za potrebe testiranja pogodb. Z uporabo lastnega omrežja je proces iterativnega razvijanja in testiranja hitrejši in lažji. Obstaja tudi nekaj javnih testnih omrežij (Ropsten, Kovan in Rinkeby), vendar so ta bolj primerna za testiranje v poznejših fazah razvoja, da vidimo, kako naša koda deluje v večjem omrežju.

4.4.1 Postavitev privatnega omrežja

Naše omrežje je bilo sestavljeno iz treh vozlišč, dva sta rudarila oziroma skrbela za generiranje novih blokov transakcij, na tretjem smo testirali pametne pogodbe in prožili transakcije. Za vozlišča smo uporabili Go Ethereum [11] programsko opremo, imenovano tudi Geth. Geth je ena od uradnih implementacij Ethereum protokola, napisana v programskem jeziku Go, katero uporablja približno polovica vseh vozlišč v glavnem Ethereum omrežju [6]. Odlikuje ga aktivna skupnost, hiter razvoj in številne funkcionalnosti. Omogoča rudarjenje, funkcionalnosti denarnice, delo z pametnimi pogodbami in pregled zgodovine blokov. Za upravljanje ponuja tri vmesnike; vmesnik z ukazno vrstico, vmesnik za klice za oddaljen postopek (*angl. RPC calls*) in Javascript konzolo.

Za postavitev privatnega omrežja smo morali naprej konfigurirati začetni blok (angl. Genesis block) naše verige. Konfiguracijska datoteka se ponavadi imenuje genesis.json in vsebuje nekaj osnovnih parametrov, s katerimi se nato inicializira omrežje. Primer take datoteke vidimo spodaj.

```
{
  "config": {
        "chainId": 31,
```

```
        "homesteadBlock": 0,
        "eip155Block": 0,
        "eip158Block": 0
  },
  "alloc": {
        "3ee9779e286a2f12ebe31881ae26c86816e0b942": {
              "balance": "613329000000000000000000000000"
        },
  },
  "coinbase"   : "0x0000000000000000000000000000000000000000",
  "difficulty" : "0x20000",
  "extraData"  : "",
  "gasLimit"   : "0x2fefd8",
  "nonce"      : "0x0000000000000042",
  "mixhash"    : "0x0000000000000000000...00000000000000000",
  "parentHash" : "0x0000000000000000000...00000000000000000",
  "timestamp"  : "0x00"
}
```

Pomembnejši parametri so:

- `chainId` - identifikator omrežja, lahko je poljubna vrednost, razen enice, saj ta označuje glavno omrežje;
- `alloc` - s tem parametrom lahko vnaprej generiranim naslovom podamo začetno vrednost Ethra (v Wei);
- `difficulty` - začetna težavnost za rudarje;
- `gasLimit` - začetna omejitev porabe goriva posameznega bloka.

Predhodno smo kreirali tudi nekaj naslovov, katerim smo nato pri konfiguraciji dodelili večjo količino Ethra. Tako smo imeli pripravljene račune, ki so imeli dovolj virtualnega denarja, da smo lahko brez omejitev ustvarjali in

testirali pametne pogodbe. Posamezno vozlišče je bilo potem potrebno inicializirati s prej ustvarjeno konfiguracijsko datoteko, nakar smo lahko zagnali naše privatno omrežje.

Inicializacija Geth vozlišča se izvede z naslednjim ukazom:

```
geth --datadir pot/do/izbrane/mape/za/podatke init genesis.json
```

S parametrom `datadir` določimo pot do krovne mape, znotraj katere bodo vsi podatki te verige. Vse kar je še potrebno je ukaz `init`, kateremu podamo ime konfiguracijske datoteke.

Vozlišče se nato zažene z ukazom:

```
geth --datadir pot/do/izbrane/mape/za/podatke --networkid 31
```

Temu ukazu spet podamo pot do krovne mape in identifikator omrežja, ki mora biti enak nastavljenemu v konfiguracijski datoteki. Vozlišče se nato poveže z ostalimi v omrežju in preveri stanje verige. Nato potrebuje nekaj časa, da od drugih vozlišč prejme pretekle bloke in se tako sinhronizira. Čas sinhronizacije je odvisen predvsem od dolžine verige.

4.4.2 Pametna pogodba

Za shranjevanje podatkov na verigo blokov smo izkoristili EVM-jevo funkcionalnost dogodkov (*angl. events*). Ko se pokliče dogodek, se argumenti klica shranijo v transakcijski dnevnik, posebno podatkovno sturkturo na verigi blokov. Ti zapisi se referencirajo z naslovom pametne pogodbe in vključijo v bloke, kjer ostanejo shranjeni, dokler imamo dostop do bloka. Do tri parametre lahko označimo kot indeksirane, kar nam potem omogoča iskanje in filtriranje po njih. Po indeksiranih parametrih lahko samo filtriramo, ne moremo pa dobiti njihove vrednosti. Ostali argumenti se shranijo v podatkovni del dnevniškega zapisa. Uporaba transakcijskega dnevnika je najcenejša oblika shranjevanja podatkov na verigo blokov, poleg tega pa nam omogoča, da na dnevniške dogodke obesimo asinhrone povratne klice (*angl. callbacks*).

Spodaj je koda naše najbolj enostavne pametne pogodbe. Funkcijo `newLog` pokličemo z petimi parametri:

- `appid` - identifikator aplikacije, v kateri je bil opravljen vpogled;
- `operation` - identifikator tipa vpogleda;
- `user` - uporabniško ime uporabnika;
- `person` - identifikator osebe, ki je bila vpogledana;
- `timestamp` - časovni žig dogodka.

```
pragma solidity ^0.4.4;

contract AuditRecord {

  event LogRecord (uint appid,
                   uint operation,
                   bytes32 indexed user,
                   bytes32 indexed person,
                   uint timestamp);

  function newLog (uint appid,
                   uint operation,
                   bytes32 user,
                   bytes32 person,
                   uint timestamp) {

      LogRecord(appid, operation, user, person, timestamp);
  }
}
```

To je bila prva pametna pogodba, katero smo testirali. Vse, kar naredi, je, da prejete parametre zapiše v transakcijski dnevnik. `Uint` predstavlja nepredznačeno celo število, medtem ko `bytes32` dvaintrideset bajtov, kar je ravno velikost besede EVM-ja.

4.4.3 Omejitve

Tekom meritev zmogljivost in hitrosti se je izkazalo, da s tako enostavno pametno pogodbo ne uspemo shraniti dovolj velikega števila vpogledov večjih sistemov. Problem je v tem, da se generira preveliko število transakcij, katerih omrežje ne uspe pravočasno sprocesirati in shraniti v bloke. Ker je generiranje transakcij zahtevnejša operacija, je bil tudi register precej obremenjen z ustvarjanjem le-teh.

Znano je, da je glavno Ethereum omrežje sposobno obdelati le do 15 transakcij na sekundo, kar je posledica trenutne implementacije samega omrežja. Pri naših testih zaradi majhnega omrežja do te omejitve ni prišlo, z razširitvijo omrežja pa bi se nedvomno začeli približevati tej meji. Ta številka je precej nizka in trenutno predstavlja največji problem omrežja. S prehodom na POS algoritem soglasja in nekaterimi drugimi prijemi, naj bi se ta številka v prihodnje močno povečala. Čeprav je bilo naše testno omrežje in pametna pogodba sposobna obdelati do 100 transakcij na sekundo, smo se odločili, da poskusimo zmanjšati število potrebnih transakcij in tako povečati prepustnost omrežja.

4.4.4 Prilagoditve

Da bi rešili zgoraj opisani problem, smo morali najti način, kako spraviti več zapisov o vpogledih v eno transakcijo. Ker Solidity še ne podpira pošiljanja strukture kot vhodnega parametra funkcij, smo morali vpoglede beležiti drugače. Prišli smo do sledeče rešitve. Pametno pogodbo smo predelali, da je sprejela tabele kot parametre, iz njih izluščila podatke o posameznem vpogledu in ga zabeležila v translacijski dnevnik.

Posodobljeno pogodbo vidimo spodaj.

```
pragma solidity ^0.4.18;

contract LogExtended {
    event logRecord (bytes32 fixedParams,
```

```
                    bytes32 indexed personId,
                    bytes32 indexed userId,
                    bytes32[] data);

    function submitData  (bytes32[] fixedParams,
                          bytes32[] personId,
                          bytes32[] userId,
                          uint[] dataIdx,
                          bytes32[] data) public {

        for (uint i = 0; i < fixedParams.length; i++) {
            uint start = dataIdx[i];
            uint end = dataIdx[i+1];
            bytes32[] memory currData =
              new bytes32[](end-start);

            for(uint j = start; j< end; j++) {
                currData[j-start] = data[j];
            }

            logRecord(fixedParams[i], personId[i],
                      userId[i], currData);
        }
    }
}
```

Pogodba zdaj preko funkcije `submitData` sprejme pet tabel. Pri prvih treh je vsak vpogled pod svojim indeksom. Podatke o tipu vpogleda, časovnem žigu in izidu vpogleda smo združili v en niz, saj imajo fiksno dolžino. Te nize potem pošljemo v tabeli `fixedParams`. V tabelah `personId` in `userId` pošljemo primarna ključa vpogledane osebe in tistega, ki je opravil vpogled iz baze. Tabela `data` vsebuje celotne JSON opise vpogledov. Ker so

posamezni JSON opisi večji kakor 32 bajtov, so raztegnjeni čez več elementov tabele. Zato `dataIdx` tabela hrani začetni in končni indeks vpogleda v `data` tabeli. Pogodba se nato sprehodi skozi vse podatke in vsak vpogled zabeleži, kakor tudi prej, v transakcijski dnevnik verige.

Tekom testiranja zmogljivosti smo ugotovili, da lahko s tako pametno pogodbo shranimo nekaj deset vpogledov naenkrat.

4.5 Uporaba tehnologije Hyperledger Fabric

Zaradi bolj kompleksne arhitekture omrežja smo porabili kar nekaj časa, da smo se spoznali s samim projektom in njegovim delovanjem. Fabric ponuja odlično dokumentacijo [8] z razlago, primeri in navodili za delo z Hyperledger Fabric verigo blokov.

4.5.1 Postavitev testnega omrežja

Za zagon testnega omrežja je naprej potrebno namestiti nekaj programov, ki so predpogoj, da bo omrežje delovalo. To so cURL, Docker, Go, Node.js in Python. Fabric ponuja skripto, ki samodejno prenese potrebne datoteke, skripte, zabojnike vozlišč in primere pametnih pogodb ter testnih omrežij. Skripto sprožimo s sledečim ukazom:

```
curl -sSL http://bit.ly/2ysbOFE | bash -s 1.2.0
```

Ta nam ustvari mapo `fabric-samples`, v kateri je več primerov pametnih pogodb in postavitev omrežij.

Ker Fabric vozlišča tečejo znotraj navideznega sistema Docker, smo lahko testno omrežje postavili samo na enem strežniku. Omrežje je sestavljeno iz le enega vrstniškega vozlišča, enega uredniškega vozlišča in enega odjemalca. Za bazo trenutnega stanja uporablja CouchDB.

4.5.2 Pametna pogodba

Pametno pogodbo za Fabric smo napisali v programskem jeziku Go. V splošnem je precej enostavna, ima dve funkciji, ena sprejema nove opise vpogledov, druga je namenjena poizvedovanju po verigi.

Go pametne pogodbe imajo določeno strukturo. Pomembno je, da uvozimo knjižnico, katera ponuja API-je za delo z verigo blokov. To naredimo s sledečim ukazom:

`import "github.com/hyperledger/fabric/core/chaincode/shim"`.

Poleg tega mora pametna pogodba vsebovati `main()` funkcijo, ki je izhodišče vsakega Go programa. Znotraj nje poženemo našo pametno pogodbo. Pametna pogodba je predstavljena kot Go struktura, implementirati pa mora dve metodi. Prva je `Init()`, ki se pokliče, ko se pogodba prvič umesti v omrežje. Funkcijo pokliče vsako vozlišče, ki bo poganjalo svojo instanco pogodbe. Uporabi se za začetno nalaganje kode in nastavljanje parametrov. Druga metoda je `Invoke()`. Znotraj nje morajo biti klici vseh metod, ki spreminjajo stanje verige, torej razne operacije tipa ustvarjanja, posodabljanja ali brisanja podatkov. Ker te operacije spreminjajo stanje verige, bo Fabric samodejno ustvaril transakcijo, v kateri bodo zapisani ukazi, katera se bo nato izvedla in zapisala v bloke.

Spodaj je navedena funkcija za beleženje novih vpogledov naše pametne pogodbe. Zaradi preglednosti smo odstranili preverjanje uspešnosti posameznih korakov v kodi.

```
func (s *gdprChaincode) saveJSON(
        APIStub shim.ChaincodeStubInterface,
        args []string) sc.Response {

    rawIn := json.RawMessage(args[0])
    bytes, err := rawIn.MarshalJSON()

    var p GdprStruct
```

```
    err = json.Unmarshal(bytes, &p)

    docID := p.Request.DocumentUniqueID
    txid := APIStub.GetTxID()
    compositeIndexName := "docID~txID"

    compositeKey, compositeErr :=
    APIStub.CreateCompositeKey(compositeIndexName,
        []string{docID, txid})

    compositePutErr := APIStub.PutState(compositeKey, bytes)

    return shim.Success([]byte(compositeKey))
}
```

Funkcija sprejme niz, ki predstavlja JSON objekt, in ga pretvori v strukturo, ki predstavlja posamezen vpogled. Iz strukture prebere identifikator dokumenta tega vpogleda in skupaj z identifikatorjem transakcije ustvari sestavljen identifikator (ključ). Pod tem ključem nato shrani dokument na verigo.

Poizvedovanje dokumentov na verigi je tako mogoče s pomočjo ključa. Tako lahko poiščemo vse zapise, ki se navezujejo na določen identifikator dokumenta. Ker smo uporabili CouchDB bazo trenutnega stanja, nam ta omogoča poizvedovanje kar po elementih JSON objekta, ki je shranjen na verigi.

4.5.3 Poizvedovanje

Poizvedovanje se opravi z deklarativno JSON sintakso. Spodaj vidimo primer take poizvedbe:

```
{
"selector": {
```

```
"year": {"$gt": 2010}
},
"fields": ["_id", "_rev", "year", "title"],
"sort": [{"year": "asc"}],
"limit": 2,
"skip": 0
}
```

V `selector` delu določimo, po katerih elementih JSON dokumenta iščemo in nastavimo določene omejitve. V zgornjem primeru iščemo po dokumentih, v katerih ima element `year` vrednost večjo od 2010. S `fields` lahko določimo, izključno kateri elementi nas zanimajo. Omogoča nam tudi sortiranje po določenem elementu in omejitev rezultatov.

Taka poizvedba se pošlje kot POST HTTP zahtevek na določeno dostopno točko CouchDB-ja. Ostale možnosti so navedene v spletnem viru [3]. Kot vidimo, lahko sestavimo zapletene selektorje, ki nam omogočajo podobne poizvedbe kot pri SQL bazah.

Poglavje 5

Primerjava implementacij

V tem poglavju si bomo pogledali primerjavo obeh implementacij. Naprej nas bo zanimala zahtevnost posamezne rešitve, nato še rezultati meritev hitrosti in prostorske zahtevnosti. Meritve smo izvedli ob predpostavki, da Ethereum omrežje trenutno ni sposobno obdelati več kot 15 transakcij na sekundo in da generira nove bloke na 14 sekund. Tako smo dobili realne rezulte, kakršne bi lahko pričakovali v produkcijskem okolju.

5.1 Zahtevnost implementacij

Za uspešno implementacijo projekta smo se morali poglobiti v delovanje obeh tehnologij verige blokov.

Pri delu z Ethereum platformo smo ugotovili, da ima veliko enostavnejšo arhitekturo omrežja v primerjavi z Hyperledger Fabric. Postavitev testnega oziroma zasebnega omrežja je enostavna in zahteva le osnovno znanje o delovanju verige blokov. Tudi pri konfiguraciji omrežja ne moremo veliko vplivati, ne da bi direktno posegali v samo implementacijo vozlišč. Zaradi tega smo lahko več časa posvetili samemu razvoju pametnih pogodb in njihovem testiranju. Tukaj se pokaže drugačna slika. Za razvoj pametnih pogodb smo se morali spoznati z novim programskim jezikom Solidity. Čeprav je podoben nekaterim, ki jih že poznamo, smo porabili precej časa za raziskovanje in

razhroščevanje napak. Solidity je dobro dokumentiran, prav tako najdemo veliko primerov kode, vendar smo opazili, da nekatere podrobnosti niso omenjene, zato je potrebno odgovore iskati po klepetalnicah in forumih. Poleg tega imata Solidity in vozlišča Geth hiter razvojni cikel, zato moramo redno spremljati izide novih verzij, saj navadno odpravijo veliko napak ali vsebujejo nove funkcionalnosti.

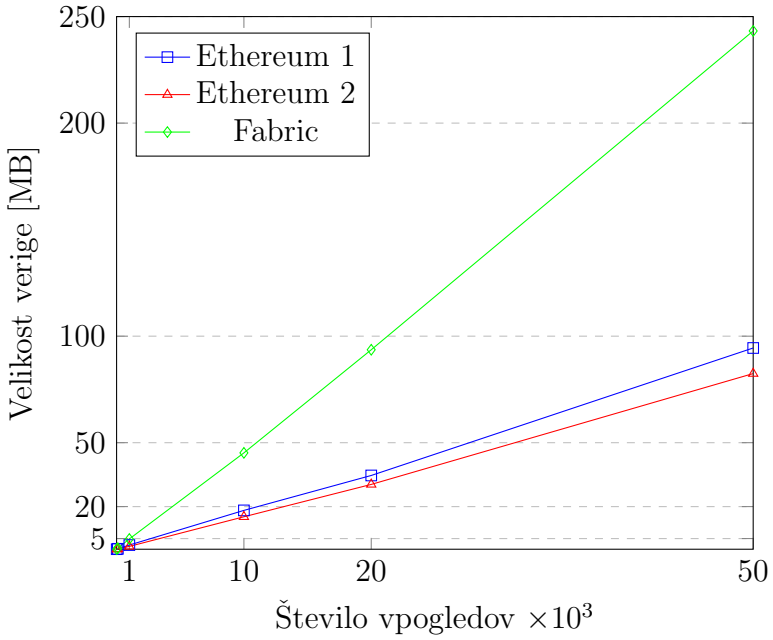
Pri delu z Hyperledger Fabric platformo so stvari ravno obratne. Sama arhitektura omrežja je močno deljena, obstajajo različna vozlišča, katera opravljajo različne funkcije. Postavitev testnega omrežja je s pomočjo ponujenih skript in primerov hitra, za pravilno razumevanje delovanja pa je potrebnega veliko časa. Samo omrežje in vozlišča so tudi veliko bolj nastavljiva v primerjavi z Ethereum omrežjem, vendar je potrebno tudi veliko širše znanje. Pri pisanju pametnih pogodb smo tukaj naleteli na manj težav. Čeprav nismo bili vešči Go-ja, smo porabili manj časa, da smo spisali delujočo pametno pogodbo. Velika prednost je bila tudi, da smo lahko podatke na verigo shranili v JSON obliki in jih v povezavi s couchDB-jem enostavno poizvedovali.

5.2 Prostorske zahteve

Tukaj si bomo pogledali, koliko prostora zasede samo omrežje in vozlišča ter kako hitro raste sama veriga.

Da postavimo zasebno Ethereum omrežje, potrebujemo vsaj dve vozlišči Geth. Sama izvedljiva datoteka Geth programa je velika le 40 megabajtov, tako da potrebujemo za postavitev zasebnega omrežja s tremi vozlišči le okoli 120 megabajtov, kar je izjemno malo. Kako hitro raste veriga, si bomo pogledali pozneje.

Ker Fabric vozlišča tečejo znotraj Docker sistema kot ločeni zabojniki, so ta precej večja. Za začetno postavitev testnega omrežja z uporabo ponujenih skript potrebujemo vsaj 4 zabojnike. To so zabojniki za Orderer in Peer vozlišči, MSP in couchDB bazo stanja. Zabojnika za Orderer in Peer vozlišči sta velika okoli 200 megabajtov, za MSP 300 megabajtov, couchDB zabojnik



Slika 5.1: Graf velikosti podatkov verige glede na število vpogledov

pa je velik kar 1,5 gigabajta. Zasebno omrežje tako že v začetku zasede približno 2,2 gigabajta prostora.

Tukaj lahko vidimo, da je že v izhodišču Fabric omrežje veliko večje. Razlog za to sta dejstvi, da omrežje poganja vozlišča znotraj Docker sistema in da je kompleksnost arhitekture omrežja večja. Čeprav Fabric porabi veliko več prostora, je razlika v porabi za današnje računalniške sisteme skoraj zanemarljiva. Bolj zanimivo bo videti, kako hitro rastejo podatki verige, ko začnemo beležiti vpoglede, kar si bomo pogledali v nadaljevanju.

Da smo dobili pravilne rezultate, smo za merjenje naraščanja verige posodobili prvotno pametno pogodbo na Ethereum omrežju, da je tudi ta sprejela in zapisala celoten JSON opis vpogleda na verigo.

Meritve smo opravili tako, da smo zabeleži začetno velikost verige na disku, zabeležili n vpogledov in ponovno pogledali velikost. Razlika med odčitanima vrednostima nam pove, koliko prostora zasede določeno število zabeleženim vpogledov.

Rezultate merjenj vidimo v tabeli 5.1 in na sliki 5.1. Prva pametna po-

	Število vpogledov					
	10	100	1000	10000	20000	50000
`Ethereum 1`	0.02	0.22	1.96	18.73	34.24	94.46
`Ethereum 2`	0.02	0.14	1.33	15.24	30.12	82.43
`Fabric`	0.05	0.55	4.82	45.64	93.32	243.52

Tabela 5.1: Tabela velikosti verige (v MB) glede na število vpogledov in pametne pogodbe

godba je označena kot `Ethereum 1`, njena izboljšana verzija kot `Ethereum 2`. Pametna pogodba na Hyperledger Fabric platformi je enostavno `Fabric`.

V tabeli lahko hitro opazimo, da izboljšana verzija pogodbe porabi približno 20 % manj prostora. To lahko pripišemo temu, da omrežje potrebuje manj transakcij in blokov, da shrani isto število vpogledov. Fabric je pri prostoru zelo potraten, porabi okoli 370 % več prostora kot izboljšana Ethereum pogodba. Razlog za tako veliko porabo prostora je število metapodatkov, ki jih Fabric shrani za vsako transakcijo. Iz tehnične specifikacije Ethereum protokola [23] (*angl. yellow paper*) lahko ugotovimo, da posamezna transakcija vsebuje 8 metapodatkov. Pri Fabricu je teh metapodatkov precej več, iz slike 2 v članku [21] lako razberemo, da je teh vsaj 25, ne da bi šteli nabor pisalnih in bralnih dostopov. To pomeni, da so stroški režije 4-krat dražji, kar se pri večjem številu transakcij hitro pozna.

Na sliki 5.1 vidimo, da zasedenost prostora raste povsem linearno v odvisnosti od števila vpogledov, posledično števila transakcij.

5.3 Analiza hitrosti

Poglejmo še, kako je s hitrostjo obdelave in shranjevanjem vpogledov obeh implementacij verige blokov.

Iz naših podatkov smo dobili grobo oceno, da mora biti naša rešitev sposobna obdelati in shraniti vsaj 30 vpogledov na sekundo. Pri tem je pomembno, da se vpogledi kar se da hitro shranijo na verigo blokov, tako da je

stanje v aplikaciji čim bolj usklajeno z dogodki vpogledov. Vedeti moramo tudi, da v realnem svetu število vpogledov na sekundo ni enakomerno porazdeljeno, to pomeni, da lahko nastopijo obdobja, kjer je vpogledov precej več od povprečja.

5.3.1 Postopek testiranja

Testiranje smo izvedli s pomočjo programa Apache JMeter [17], odprtokodnega programa, ki je namenjen merjenju zmogljivosti spletnih aplikacij. Z njegovo pomočjo smo v naš register poslali večje število testnih vpogledov. Merili smo čas, potreben, da je register vse vpoglede obdelal, ustvaril in poslal transakcije v omrežje, ter da so se te shranile na verigo. Meritve smo ponovili 5-krat in rezultate povprečili. Tako smo dobili zanesljive rezultate o zmožnosti našega registra vpogledov in različnih pametnih pogodb ter omrežij.

Kot že prej omenjeno, lahko glavno Ethereum omrežje obdela do nekje 15 transakcij na sekundo. Omenimo, da to trenutno šteje okoli 10000 vozlišč [5]. Do te omejitve pri naši testni postavitvi omrežja ni prišlo, saj je bilo to sestavljeno iz le treh vozlišč, eno vozlišče je sprejemalo nove transakcije, drugo je skrbelo za generiranje novih blokov, tretje pa je bilo samo povezano na prva dva. V literaturi nismo zasledili, pri kakšnem številu vozlišč v omrežju pride do upada hitrosti zaradi same komunikacije in usklajevanja le-teh.

	Število vpogledov		
	10000	20000	50000
`Ethereum 1`	102	183	459
`Ethereum 2`	49	115	325
`Fabric`	31	65	189

Tabela 5.2: Tabela povprečne hitrosti (v sekundah) zapisa množice vpogledov na verigo glede na število le-teh

Rezultate meritev vidimo v tabeli 5.2. Opazimo lahko, da je tudi pr-

votna pametna pogodba presegla naše zahteve po hitrosti. Zmožna je bila obdelati okoli 100 transakcij na sekundo. Razlog je v tem, da je bilo naše testno omrežje precej majhno, zato lahko trenutno samo ugibamo, pri kakšnem številu vozlišč v testnem omrežju bi ta hitrost začela upadati v smeri omejitve glavnega (15 transakcij na sekundo). V prvih poskusih meritev je bila omejitev poraba goriva v blokih, zato smo to vrednost nastavili na praktično neomejeno (10^{12}), da smo lahko res prišli do zgornje meje našega registra in samega omrežja. Poraba goriva je dosegala tudi do 100 milijonov. Za primerjavo; glavno Ethereum omrežje ima trenutno porabo okoli 8 milijonov.

S posodobljeno pogodbo smo uspeli hitrost povečati za več kot 50 %. Glavni razlog za pospešitev je manjše število transakcij, saj sedaj shranimo do približno 30 vpogledov v eno transakcijo. Posledično mora naš register generirati manj transakcij, kar je zahtevna operacija, prav tako pa jih mora omrežje obdelati manj.

Da smo lahko izkoristili posodobljeno pogodbo, je bilo potrebno prilagoditi generiranje transakcij z vpogledi. Za prvo pametno pogodbo smo generirali transakcijo takoj, ko smo zabeležili nov vpogled. Za posodobljeno pogodbo smo sedaj morali spraviti čim več vpogledov v eno transakcijo, hkrati pa paziti, da se te še vedno generirajo hitro. Sedaj ob vsakem novem vpogledu po oddani transakciji počakamo, da se nabere več vpogledov, katere interno hranimo v programu, preden kreiramo transakcijo. Po pretečenem času (čakamo eno sekundo) ali če se je vpogledov nabralo toliko, da zapolnimo transakcijo z njimi, ustvarimo transakcijo z zbranimi vpogledi in jo oddamo v omrežje.

Hyperledger Fabric omrežje je sposobno obdelati veliko več transakcij na sekundo. Iz analize v članku [21] vidimo, da lahko s pravilnimi prijemi dosežemo hitrost tudi do 2000 transakcij na sekundo, kar je dva reda velikosti več kot Ethereum omrežje. Tukaj nam samo omrežje ne predstavlja ovire in smo lahko z našo pametno pogodbo brez težav shranili vse vpoglede. V tabeli 5.2 vidimo, da se sicer nismo preveč približali maksimalni hitrosti Fabric omrežja. Razlog je v tem, da nam omejitev predstavlja naša imple-

mentacija registra, saj obdelava vpogledov vzame več časa kakor generiranje in pošiljanje transakcij.

Poglavje 6

Sklepne ugotovitve

Skozi implementacijo registra vpogledov smo spoznali delovanje tehnologije verige blokov, podrobneje tudi dve implementaciji, ki bosta igrali pomembno vlogo v nadaljnjem razvoju tehnologije. Tehnologija je še v razvoju, hitro se pojavljajo nove rešitve in izboljšave trenutnim izvedbam, vendar že zdajšnje implementacije ponujajo veliko funkcionalnosti. Glavne prednosti verige blokov so transparetnost transakcij, porazdeljena baza podatkov in nespremenljivost oziroma integriteta podatkov. Zaradi teh lastnosti se lahko tehnologija aplicira na različna področja, kot na primer:

- tokenizacija (digitalizacija) dobrin;
- sledenje preskrbovalnih mrež;
- digitalna identita;
- zdravstvo;
- kriptovalute.

Na vseh navedenih področjih, kot tudi drugih, poteka že ogromno projektov, kateri poskušajo ponuditi rešitve, ki uporabljajo tehnologije verige blokov. Zares izdelanih rešitev na trgu sicer še ni, razen na področju kriptovalut, kjer je bil največji poudarek v zadnjih letih. Nedvomno pa se bodo zanimive rešitve pojavile tudi v drugih panogah, predvsem ko tehnologija še dozori.

Trenutno implementacijo Ethereum omrežja omejuje predvsem količina prometa, ki ga je sposobno sprocesirati. Težava je vidna predvsem na glavnem omrežju, kjer je že večkrat prišlo do preobremenjenosti in upočasnitve delovanja omrežja. Testna oziroma privatna omrežja so še vedno precej hitra, a lahko trenutno le ugibamo, če hitrost omrežja začne upadati že pri nekaj 10 ali šele nekaj 1000 vozliščih. V času pisanja diplomske naloge še vedno ni prišlo do načrtovane nadgradnje na POS protokol soglasja, kateri bi med drugim povečal tudi pretočnost omrežja. Samo delo s tehnologijo je enostavnejše in hitro osvojimo potrebno znanje za implementacijo rešitve, ki uporablja Ethereum verigo blokov.

Hyperledger Fabric nima težav z pretočnostjo, saj je bil že v osnovi zasnovan za večje sisteme, ki potrebujejo večjo skalabilnost. Implementacija deluje bolj celovita implementacije tehnologije verige blokov, vendar s sabo prinese tudi večjo kompleksnost samega omrežja. Več časa porabimo za spoznavanje tehnologije in delovanja, vendar nam ta ponuja veliko več kot Ethereum implementacija.

Ethereum in Hyperledger Fabric se posvečata različnim ciljnim skupinam. Medtem ko je prvi bolj primeren za manjše projekte in aplikacije, namenjene širši javnosti, je Fabric boljši za večje industrijske rešitve, ki potrebujejo boljšo zanesljivost, večjo skalabilnost in konfigurabilnost.

Naš register vpogledov smo uspešno implementirali z uporabo obeh tehnologij. Zaradi omenjene slabe pretočnosti Ethereum omrežja, bo nadaljnji razvoj potekal na Hyperledger Fabric tehnologiji. Poudarek pri razvoju bo na varni hrambi osebnih podatkov znotraj našega registra, kot tudi na izboljšanju delovnega procesa shranjevanja. Da bomo lahko izkoristili hitrost, ki nam jo ponuja samo Hyperledger Fabric omrežje, bomo morali pohitriti proces med sprejemom novega vpogleda in generiranja transakcije.

Razmisliti bomo morali tudi, kako bi boljše poskrbeli za možnost izbrisa oziroma pozabe osebnih podatkov, ki so shranjeni na verigi blokov, kar se trenutno rešuje z izbrisom dekriptirnega ključa posamezne osebe. Za dosego tega cilja nam nova verzija Fabric-a [15] ponuja novo funkcionalnost zaupnih

zbirk podatkov (*angl. private data collections*), ki nam omogočajo brisanje podatkov, saj so hranjeni v ločeni bazi, na verigo blokov pa se shrani samo zgoščena vrednost.

Literatura

[1] Dejan Benedik. Analiza in uporaba vsebnikov docker v arhitekturi mikrostoritev. Diplomska naloga, Fakulteta za elektrotehniko in računalništvo, Univerza v Ljubljani, 2016.

[2] CouchDB. `http://couchdb.apache.org/`. [Dostopano: 20.08.2018].

[3] CouchDB Documentantion. `http://docs.couchdb.org/en/stable/api/database/find.html`. [Dostopano: 22.09.2018].

[4] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[5] Seznam vozlišč ethereum. `https://www.ethernodes.org/network/1`. [Dostopano: 15.12.2019].

[6] Ethereum nodes. `https://www.ethernodes.org/network/1`. [Dostopano: 21.08.2018].

[7] Cene strojnih ukazov EVM. `https://docs.google.com/spreadsheets/d/1m89CVujrQe5LAFJ8YAUCcNK950dUzMQPMJBxRtGCqs/edit#gid=0`. [Dostopano: 19.08.2018].

[8] Fabric dokumentacija. https://hyperledger-fabric.readthedocs.io/en/release-1.2/index.html. [Dostopano: 22.08.2018].

[9] Fauxton. `https://github.com/apache/couchdb-fauxton`. [Dostopano: 22.09.2018].

[10] Uredna EU o varstvu podatkov. Doseglijvo: https://eur-lex.europa.eu/legal-content/SL/TXT/?uri=uriserv:OJ.L_.2016.119.01.0001.01.SLV. [Dostopano: 12. 7. 2018].

[11] Go ethereum. https://github.com/ethereum/go-ethereum. [Dostopano: 13.08.2018].

[12] Hash tree. https://en.wikipedia.org/wiki/Merkle_tree. [Dostopano: 16.08.2018].

[13] Hyperledger. Doseglijvo: https://www.linuxfoundation.org/press-release/linux-foundation-unites-industry-leaders-to-advance-blockchain-technology/. [Dostopano: 17.07.2018].

[14] Hyperledger project. Dosegljivo: https://www.hyperledger.org/. [Dostopano: 17.07.2018].

[15] Hyperledger project. Dosegljivo: https://hyperledger-fabric.readthedocs.io/en/release-1.2/whatsnew.html. [Dostopano: 20.10.2018].

[16] JAX-RS. https://en.wikipedia.org/wiki/Java_API_for_RESTful_Web_Services. [Dostopano: 14.08.2018].

[17] Apache JMeter. https://jmeter.apache.org/. [Dostopano: 15.12.2019].

[18] LevelDB. https://github.com/google/leveldb. [Dostopano: 20.08.2018].

[19] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[20] Solididy. http://solidity.readthedocs.io/en/v0.4.24/. [Dostopano: 25.07.2018].

[21] Parth Thakkar, Senthil Nathan, and Balaji Vishwanathan. Performance benchmarking and optimizing hyperledger fabric blockchain platform. *CoRR*, abs/1805.11390, 2018.

[22] Web3j. `https://github.com/web3j/web3j`. [Dostopano: 13.08.2018].

[23] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. Dosegljivo: https://ethereum.github.io/yellowpaper/paper.pdf, 2018. [Dostopano: 23.08.2018].