Primjena distribuiranih baza podataka i pametnih ugovora u sustavima elektroničkog poslovanja

Miškulin, Ivan

SVEUČILIŠTE U ZAGREBU

FAKULTET PROMETNIH ZNANOSTI

Ivan Miškulin

PRIMJENA DISTRIBUIRANIH BAZA PODATAKA I PAMETNIH UGOVORA U SUSTAVIMA ELEKTRONIČKOG POSLOVANJA

DIPLOMSKI RAD

ZAGREB 2018.

Sveučilište u Zagrebu

Fakultet prometnih znanosti

DIPLOMSKI RAD

PRIMJENA DISTRIBUIRANIH BAZA PODATAKA I PAMETNIH UGOVORA U SUSTAVIMA ELEKTRONIČKOG POSLOVANJA

IMPLEMENTATION OF DISTRIBUTED DATABASES AND SMART CONTRACTS IN ELECTRONIC BUSINESS SYSTEMS

Mentor: prof. dr. sc. Tonči Carić

Student: Ivan Miškulin

JMBAG: 0135228486

Zagreb, rujan 2018.

PRIMJENA DISTRIBUIRANIH BAZA PODATAKA I PAMETNIH UGOVORA U SUSTAVIMA ELEKTRONIČKOG POSLOVANJA

SAŽETAK

Konstantnim povećanjem postotka ljudske populacije u gradovima povećava se i broj automobila u gradovima. S povećanjem broja automobila dolazi do većeg opterećenja prometa te zahtjeva za novim parkirnim mjestima. Kako bi smanjilo opterećenje prometnica i potražnja za parkirnim mjestima potrebno je osmišljavati nove metode. Jedna od tih metoda je i dijeljenje automobila kojoj je cilj smanjiti vrijeme koje vozilo provode na parkirnom mjestu. Za implementaciju takve metode potrebno je kreirati sustav koji će omogućivati iznajmljivanje i dijeljenje automobila te plaćanje usluge. U ovome radu izraditi će se dio aplikacije za dijeljenje automobila koja će funkcionirati na pametnom ugovoru unutar blockchain-a odnosno distribuiranog zapisnika. Postupak izrade te funkcionalnosti aplikacije objašnjene su unutar rada. Aplikacija se fokusira na plaćanje koristeći pametne ugovore te su opisani ostali dijelovi sustava potrebni za implementaciju aplikacije koje nisu unutar okvira ovoga rada.

KLJUČNE RIJEČI: distribuirane baze podataka, pametni ugovor, elektroničko poslovanje

SUMMARY

Number of vehicles in cities rises with the constant increase of human population in the cities. Bigger number of vehicles creates bigger traffic congestion and demand for parking spaces. To reduce traffic congestion and demand for parking spaces it is necessary to devise new methods. One of those methods is car sharing which goal is to reduce time vehicles spends in the parking spot. To implement that kind of method it is necessary to create a system which will enable car sharing and service payment. Part of application for car sharing will be created in this paper. Application will be focused on smart contract and blockchain. The process of making the application is explained within the paper. Application is focused on payments using smart contract and the other parts of the system needed for implementation are described but are not within the scope of this paper.

KEY WORDS: distributed databases, smart contract, electronic business

SADRŽAJ

1. Uvod

Promet cestovnih vozila u gradovima je problem koji se javlja u svim gradovima te se uz njega povezuje problem parkirnih mjesta unutar gradova te količina površine koja se izdvaja za parkirališta. Jedan od načina smanjenja potražnje za parkirnim mjestima je smanjenje vremena koje vozilo provode na parkirnom mjestu. Da se vozilo vrati u promet potrebno je vraćanje vozača u vozilo ili dolazak novog vozača koji preuzima vozilo. Iz toga proizlazi ideja o dijeljenju vozila od koje određene prednosti imaju vozači i gradovi. Sustav dijeljenja vozila bi izgledao tako da je vlasnik vozila treća strana koja daje automobil na iznajmljivanje kojeg onda iznajmljuju drugi vozači na željeno vrijeme te ostavljaju vozilo na njihovom odredištu.

Svrha ovoga rada je prikazati mogućnosti upotrebe distribuiranih baza podataka i pametnih ugovora za rješavanje jednog od prometnih problema. Cilj rada je izrada aplikacije bazirane na navedenim tehnologijama koja bi omogućivala dijeljenje vozila te plaćanje usluge iznajmljivanja vozila.

Rad je podijeljen u sedam cjelina:

1. Uvod
2. Distribuirane baze podataka
3. Programski jezik Solidity
4. Pametni ugovori
5. Sučelje za distribuiranu bazu podataka
6. Web aplikacija za izradu pametnih ugovora na distribuiranoj bazi
7. Zaključak

U drugom poglavlju objašnjavaju se distribuirane baze podataka i različite izvedenice distribuiranih baza podataka poput distribuiranog zapisnika i *blockchain-a*. Kroz treće poglavlja objašnjava se programski jezik Solidity koji se koristi za izradu pametnih ugovora. Pametni ugovor i njegove funkcionalnosti prikazane su i objašnjene u četvrtom poglavlju čime počinje

izrada aplikacije. U petome poglavlju prikazuje se sučelje između pametnog ugovora u *blockchain-u* i internetskog preglednika. Šesto poglavlje prikazuje upotrebu pametnog ugovora putem internetske stranice te primjere korištenja aplikacije.

2. Distribuirane baze podataka

U ovome poglavlju prvo će se proći kroz same definicije baze podataka, sustava za upravljanje bazama podataka te će se nakon toga prijeći na tematiku distribuiranih baza podataka gdje će se objasniti i *blockchain* tehnologija.

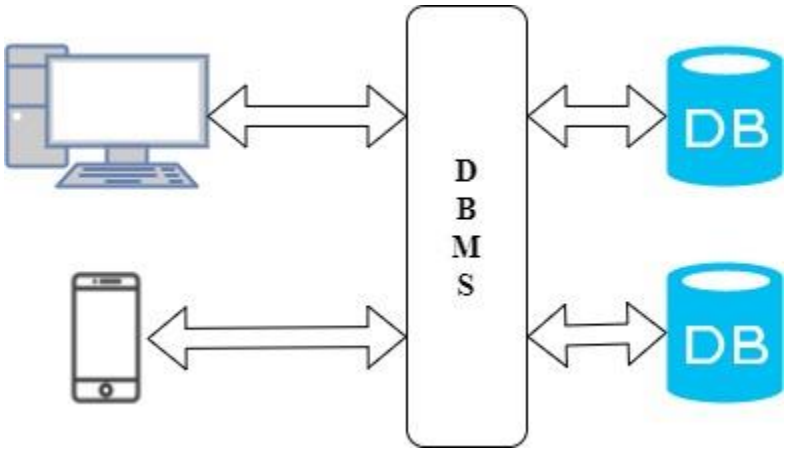
2.1 Osnove baza podataka

Baza podataka je skup međusobno povezanih i strukturiranih podataka dok se upravljanje podacima u bazi podataka vrši sa Sustavom za upravljanje bazom podataka DBMS (engl. *Database Managment System*). DBMS je program koji se koristi za pohranjivanje, dohvaćanje, brisanje i mijenjanje podataka u bazi podataka. Glavni cilj DBMS-a je osiguravanje konzistentnosti i redundantnosti podataka te pružiti siguran pristup podacima.[1]

Korisnici šalju strukturirane upite kako bi dohvatili podatke iz baze podataka. Upiti nad bazom podataka pišu se po pravilima upitnog jezika (engl. *query language*) koji se koristi. Primjer takvog jezika je Structured Query Language koji je postao ISO standard u 1986. godini.[2]

[1] Chhanda, R.: *Distributed Database Systems*, Pearson Education Inc., Dorling Kindersley, 2009., p. 1.

[2] URL: http://www.informit.com/articles/article.aspx?p=29583 (pristupljeno: 2. kolovoz 2018.)



Slika 1. Shematski prikaz strukture sustava baze podataka

Slikom 1. prikazan je shematski prikaz razmjene podataka između baza podataka označenih s DB, sustava za upravljanje bazama podataka DBMS i korisničkih uređaja. Iz slike je vidljivo da korisnici komuniciraju s DBMS-om koji korisničke naredbe izvršava nad bazom podataka te onda dobiven rezultat vraća korisniku.[3]

2.2 Osnove distribuiranih baza podataka

Distribuirana baza podataka iz korisničkog pogleda izgleda kao jedinstvena baza podataka dok je u stvarnosti ona pohranjena na više lokacija. Baze podataka mogu biti na više računala na jednoj geografskoj lokaciji ili na više računala na više geografskih lokacija. Baze podataka koje čine distribuiranu bazu podataka povezane su mrežnim tehnologijama. Time distribuirana baza podataka obuhvaća baze podataka i računalne mreže. Svakom bazom podataka u distribuiranoj bazi podataka upravlja lokalni DBMS koji surađuje s ostalim DBMS-ovima u distribuiranoj bazi kako bi se održala konzistentnost podataka. DBMS-ovi putem mreže razmjenjuju informacije o promjenama podataka kako bi onda lokalni DBMS-ovi ažurirali bazu podataka kojom upravljaju.[4]

[3] URL: http://www.sqlrelease.com/sql-server-tutorial/dbms-rdbms-and-sql-server (pristupljeno: 2. kolovoz 2018.)
[4] URL: https://docs.oracle.com/cd/A57673_01/DOC/server/doc/SCN73/ch21.htm (pristupljeno: 2. kolovoz 2018.)

Dalje u poglavlju obraditi će se prednosti i nedostaci distribuiranih baza podataka. Određeni načini funkcioniranja sustava distribuiranih baza podataka su ujedno i prednost i nedostatak ovisno o pogledu tako da će se u sljedećim potpoglavljima prvotno obraditi tema prednosti i svih mogućih pozitivnih učinaka distribuiranih baza podataka te nakon toga nedostaci i negativni učinci.

2.2.1 Prednosti distribuiranih DBMS

Jedna od glavnih prednosti sustava distribuiranih baza podataka je dijeljenje podataka u sustavu. Tako korisnik koji se spaja s jedne lokacije u sustavu može pristupiti podacima koji se nalaze na drugoj lokaciji u drugoj bazi podataka u sustavu te se time postiže lakše dijeljenje podataka.[5]

Sustav distribuiranih baza podataka moguće je izvesti tako da se u bazama koje čine sustav pohranjuje samo podskup svih podataka. U tom slučaju ukoliko korisnik traži podatke koji su pohranjeni u bazi podataka na njemu najbližoj geografskoj lokaciji on ih dobije brže nego u slučaju da pristupa udaljenoj centralnoj bazi podataka. U ovome slučaju treba se obratiti pažnja pri odabiru podatka koji će se pohranjivati u određenoj bazi podataka.

Distribuirane baze podataka omogućuju simultano obrađivanje podataka na više lokacija tako da je moguće jedan upit rastaviti u više manjih upita koji se onda paralelno izvršavaju te time ubrzava proces dohvaćanja podataka.

Ukoliko se u obzir uzme da tvrtka ima više zgrada na više lokacija, uvođenje distribuirane baze podataka povećava lokalnu autonomiju. Svaka lokacija koja ima bazu podataka odgovorna je za lokalnu kontrolu podataka dok u centraliziranom sustavu samo je jedna baza podataka gdje je jedna osoba ili odjel odgovorna za kontrolu cijelog skupa podataka.

Ukoliko poslužitelj jedne od baza podataka u sustavu prestane raditi moguće je pristupiti setovima podataka drugih baza podataka u sustavu čime se povećava dostupnost i pouzdanost sustava. U distribuiranim sustavima prestanak rada jednog čvora ne uzrokuje pad cijelog sustava.

[5] Chhanda, R.: *Distributed Database Systems,* Pearson Education Inc., Dorling Kindersley, 2009., p. 32.

Ovakav sustav zahtijeva dodatan mehanizam za otkrivanje grešaka te ponovo pokretanje baze podataka i ažuriranje podataka.

Bolje performanse cijelog sustava mogu se postići prilagođavanjem zahtijevima krajnjih korisnika tako da se podaci pohranjuju tamo gdje je najveća potražnja za tim podacima.[6]

2.2.2 Nedostaci distribuiranih DBMS

Upravljanje distribuiranim podacima kompleksnije je od upravljanja podacima centraliziranog sustava. Distirbuirani sustav baza podataka koji s korisničke strane ne izgleda kao distribuirani sustav znatno je kompliciranije napraviti takav sustav i njime upravljati. Kopiranje i distribuiranje podataka u sustavu stvara dodatnu kompleksnost uz koje se vežu zadaće, koje također u distribuiranom sustavu postaju kompleksnije. Primjeri tih zadaća su: upravljanje transakcijama, optimizacija upita te upravljanje oporavka baze podataka.

Proširenje sustava s jedne na više lokacija donosi troškove nabavke potrebne opreme te zahtijeva komunikacijsku mrežu za funkcioniranje sustava. Također veći su troškovi održavanja sustava te i potreba za samim kadrom koji radi na održavanju sustava zbog većeg broja poslužitelja, baza podataka i DBMS-ova te komunikacijske mreže.

U distribuiranim sustavima zbog većeg broja lokacija na kojima se pohranjuju podaci povećava se vjerojatnost da se dogodi sigurnosni propust. Uz to podaci se u ovome sustavu prenose putem mreže pa je potrebno da taj prijenos podataka između baza podataka bude siguran.

Kopiranje cijelog skupa ili podskupa podataka zahtijeva veću memoriju nego centralizirani sustav te dodijeljivanje pristupa čvrstom disku i upravljanje pohranjivanjem podataka postaje sve kompleksnije s rastom distribuiranosti sustava.

Održavanje integriteta baze podataka koji se gleda kroz validnost i konzistentnost podataka te postavljenih ograničenja vrlo je teško za izvest. Ograničenje se odnosi na određene uvjete i prijedloge koje baza podataka ne smije prekršiti. Provođenje integriteta kroz ograničenja zahtijeva pristup velikom broju podataka koja definiraju ograničenje.[7]

---

[6] Chhanda, R.: *Distributed Database Systems*, Pearson Education Inc., Dorling Kindersley, 2009., p. 33. – 34.

[7] Chhanda, R.: *Distributed Database Systems*, Pearson Education Inc., Dorling Kindersley, 2009., p. 34. – 35.

2.3 Homogeni i heterogeni sustavi distribuiranih baza podataka

Homogeni sustav distribuiranih baza podataka je mreža dvaju ili više baza podataka koje koriste isti DBMS za upravljanje bazama podataka. Upotrebom istog softvera u sustavu osigurava se njihova kompatibilnost te se pojednostavljuje dohvaćanje podataka iz drugih baza podataka u sustavu.

U heterogenom sustavu distribuiranih baza podataka postoje najmanje dvije različite vrste DBMS-a. Takvom izvedbom sustava nije osigurana kompatibilnost koja onda može uzrokovati da određeni korisnici nisu u mogućnosti dohvatiti podatke sa svih baza podataka u sustavu. Za ostvarivanje kompatibilnosti potrebno je uložiti određene resurse.[8]

2.4. Fragmentacija podataka

U distribuiranim bazama podataka bitno je prije repliciranja podataka odabrati koji skup podataka će se distribuirati, u koju bazu podataka te na koji način će se fragmentirati podaci koji se distribuiraju. Fragmentacija podataka je postupak dijeljenja jednog objekta u više dijelova. U slučaju baze podataka objekt može biti korisnik, tablica ili cijela baza podataka. Fragmente je moguće pohraniti na različite baze podataka na više geografskih lokacija u sustavu.

Fragmentacija podataka može se izvesti na tri načina: horizontalna, vertikalna i hibridna fragmentacija podataka. Za objašnjenje primjera različitih vrsta fragmentacije koristiti će se tablica 1.[9]

[8] URL: https://docs.oracle.com/cd/A87860_01/doc/server.817/a76960/ds_conce.htm (pristupljeno: 3. kolovoz 2018.)
[9] Singh, S.:Database Systems: Concepts, Design and Applications, Pearson Education Inc., Dorling Kindersley, 2009., p. 556.

Tablica 1. Primjer tablice podataka zaposlenika koja se pohranjuje u bazu podataka

ID	Ime	Prezime	Odjel	Plaća
1	Ana	Anić	Marketing	5500,00
2	Ivo	Ivić	Računovodstvo	6000,00
3	Petar	Perić	Održavanje	5700,00
4	Marija	Marić	Razvoj sustava	6500,00

Prikazana tablica 1. prikaz je mogućeg jednostavnog pohranjivanja podataka o zaposlenicima. Tablicom odnosno relacijom na slici prikazuju se zaposlenici. Svaki zaposlenik ima pet atributa: ID, ime, prezime, odjel i plaća. Kroz sljedeća potpoglavlja objasniti će se i prikazati različite vrste fragmentacije.

2.4.1 Horizontalna fragmentacija podataka

Horizontalna fragmentacija tablice odnosno relacije je dijeljenje cijelog skupa na određeni broj redaka tako da svaki redak ima sve atribute relacije. Horizontalna fragmentacija horizontalno dijeli relaciju tako da retci zadovoljavaju određeni kriterij poput: ime počinje slovom A ili odabir zaposlenika gdje je odjel Marketing.[10]

Fragmente relacije onda je moguće pohraniti na određenu lokaciju u distribuiranom sustavu.

[10] Singh, S.:Database Systems: Concepts, Design and Applications, Pearson Education Inc., Dorling Kindersley, 2009., p. 557.

Lokacija 1 - Zagreb

ID	Ime	Prezime	Odjel	Plaća
1	Ana	Anić	Marketing	5500,00
2	Ivo	Ivić	Računovodstvo	6000,00

Lokacija 2 - Rijeka

ID	Ime	Prezime	Odjel	Plaća
3	Petar	Perić	Održavanje	5700,00
4	Marija	Marić	Razvoj sustava	6500,00

Slika 2. Horizontalno fragmentirana Tablica 1

Na slici 2. prikazane su tablice koje se dobiju horizontalnom fragmentacijom tablice 1. Dobivene tablice je onda moguće pohraniti na dvije lokacije kao što je prikazano na slici 2, Zagreb i Rijeka.

2.4.2 Vertikalna fragmentacija podataka

Vertikalna fragmentacija podataka dijeli relaciju odnosno tablicu vertikalno po stupcima odnosno atributima. Vertikalnom fragmentacijom fragmentirana relacija zadržava samo određene atribute. Na određenim lokacijama i bazama podataka možda nije potrebno pohraniti sve atribute pa sve vrši vertikalna fragmentacija te se pohranjuju samo potrebni atributi. Prilikom fragmentacije potrebno je u svim fragmentiranim relacijama dodati primarni ključ kako bi se one mogle ponovno spojiti na ispravan način.[11]

[11] Singh, S.:Database Systems: Concepts, Design and Applications, Pearson Education Inc., Dorling Kindersley, 2009., p. 559.

Lokacija 1 - Zagreb

ID	Ime	Prezime
1	Ana	Anić
2	Ivo	Ivić
3	Petar	Perić
4	Marija	Marić

Lokacija 2 - Rijeka

ID	Odjel	Plaća
1	Marketing	5500,00
2	Računovodstvo	6000,00
3	Održavanje	5700,00
4	Razvoj sustava	6500,00

Slika 3. Vertikalno fragmentirana Tablica 1

Vertikalnom fragmentacijom dobiju se dvije tablice prikazane na slici 3. Prva tablica dobivena fragmentiranjem sadrži ID, Ime i Prezime zaposlenika i pohranjena je u lokaciji odnosno Zagreb. Druga tablica, koja se pohranjuje u Rijeci, sadrži atribute ID, Odjel i Plaća. Vertikalna fragmentacija u ovome slučaju omogućuje odvajanje moguće osjetljivih podataka poput plaće te pohranu te tablice na posebnu lokaciju te ograničiti pristup.

2.4.3 Hibridna fragmentacija podataka

Postupak hibridne fragmentacije podataka obuhvaća i vertikalnu i horizontalnu fragmentaciju podataka jedne relacije. Relaciju je moguće podijeliti ili prvo vertikalno i nakon toga dobivene fragmente dodatno još horizontalno fragmentirati. Moguće je napraviti i obrnuti postupak ovisno o postavljenim kriterijima i ciljevima fragmentacije.

Potreba za hibridnom fragmentacijom podataka polazi od korisničkih upita koji često zahtijevaju samo određeni podskup podataka koji je kombinacija vertikalnih i horizontalnih fragmenata.[12]

[12] Navathe S, Karlapalem K, Ra M. A mixed fragmentation methodology for initial distributed database design. Journal of Computer and Software Engineering. 1995 Jun;3(4):395-426.

Lokacija 1.1 - Zagreb

ID	Ime	Prezime
1	Ana	Anić
2	Ivo	Ivić

Lokacija 1.2 - Zagreb

ID	Odjel	Plaća
1	Marketing	5500,00
2	Računovodstvo	6000,00

Lokacija 2.1 - Rijeka

ID	Ime	Prezime
3	Petar	Perić
4	Marija	Marić

Lokacija 2.2 - Rijeka

ID	Odjel	Plaća
3	Održavanje	5700,00
4	Razvoj sustava	6500,00

Slika 4. Vertikalno i horizontalno fragmentirana Tablica 1

Slikom 4. prikazan je jednostavan primjer hibridne fragmentacije Tablice 1. Kombinirale su se prethodno korištene fragmentacije. Prvo je korištena horizontalna fragmentacija te se odvoje zaposlenici ovisno o njihovom identifikatoru te se dobiju tablice prikazane na slici 2.

Sljedeći korak je vertikalno fragmentiranje dobivenih tablica gdje se odvajaju argumenti Ime i Prezime u jednu tablicu te argument Odjel i Plaća u drugu tablicu. Vertikalno fragmentiranje se vrši na obje tablice prikazane na slici 2. te se dobiju četiri tablice prikazane na slici 3.

2.5 Distributed ledger tehnologija

Tehnologija distribuiranog zapisnika (engl. *Distributed ledger technology*, DLT) u najstrožem smislu je vrsta baze podataka koja se dijeli svim čvorovima u mreži. Upotreba DLT-a u raznim aplikacijama je kombinacija komponenti koje uključuju *peer-to-peer*[13] (dalje u tekstu P2P) mrežu, distribuiranu bazu podataka i kriptografiju koja ovisno o upotrebi može promijeniti način na koji se podaci pohranjuju i imovina trasferira.

U DLT-u čvorovi u mreži su uređaji na kojima je pokrenut DLT softver koji održava zapise baze podataka. Čvorovi su međusobno povezani kako bi dijelili i validirali informacije. Ovakvom strukturom omogućuje se da entitet (npr. krajnji korisnik ili financijska institucija) koji posjeduje čvor u mreži dijeli odgovornost upravljanja bazom podataka s drugim entitetima u P2P mreži.

DLT tehnologija omogućuje i jednom entitetu da održava zapise baze podataka na više čvorova kako bi se povećala dostupnost baze podataka.

Izvedbu P2P mreže moguće je izvesti na dva načina. Prvi u kojem jedan entitet dijeli zapise na određene čvorove u vlastitoj P2P mreži. Drugi način je da više entiteta dijele distribuirani zapisnik. Svaki entitet putem svog čvora u dijeljenoj mreži ima pristup zapisniku.[14]

Upotreba kriptografije u DLT-u poput: privatnih i javnih ključeva te *hash*-a objasniti će se u sljedećem potpoglavlju.

2.6 Blockchain

Kako bi se objasnila tehnologija *blockchain* potrebno je prvo razumjeti tematiku kriptografije poput: privatnog i javnog ključa, *hash* funkcija i digitalnog potpisa. Prvo će se objasniti navedene tehnologije u potpoglavljima kako bi se olakšalo objašnjavanje načina funkcioniranja *blockchain-a*.

[13] Računalna mreža u kojoj sva računala u ravnopravnom odnosu te imaju jednake ovlasti, mogućnosti i zadatke

[14] Mills, D., Wang, K., Malone, Bernard, Ravi, A., Marquardt, J., Chen, C., Badev, A., Brezinski, T., Fahy, L., Liao, K., Kargenian, V., Ellithorpe, W., Baird, M.: *Distributed ledger technology in payments, clearing and settlement*, Finance and Economics Discussion Series 2016-095, Washington, D.C., 2016.

2.6.1 Kriptografska hash funkcija

Kriptografska *hash* funkcija je matematička funkcija koja za ulazne podatke varijabilne duljine daje binarnu sekvencu određene duljine. *Hash* funkcija je napravljena tako da je vrlo teško napraviti obrnuti proces odnosno iz izlazne sekvence dobiti ulazne podatke.

Hash funkcija za iste ulazne podatke daje istu izlaznu sekvencu. Ukoliko se nad ulaznim podacima napravi i najmanja promjena ona uzrokuje značajnu promjenu na izlaznoj sekvenci. Tako promjena jednog bita u ulaznim podacima uzrokuje promjenu više od polovice bitova u izlaznoj sekvenci.

Ulazni podaci mogu biti vrlo različiti. Mogu biti kratki te sadržavati jedan znak ili sadržavati cijeli dokument. U oba slučaja se dobije fiksna dužina određena algoritmom. Primjer *hash* algoritma je Secure Hashing Algorithm 256 odnosno SHA256 koji daje sekvencu od 256 bitova.[15]

Svojstva koja *hash* funkcija treba ispunjavati su: *preimage resistance*, *second preimage resistance* i *collision resistance*.

Za dobivenu *hash* vrijednost *h* trebalo bi biti teško izračunati ulazne podatke *m*. Broj pokušaja potrebnih za dobivanje ulaznih podataka treba biti ekstremno velik te je potrebna velika količina vremena i resursa. Ovime se definira *preimage resistance*.

Za slučaj kada se za zadane ulazne podatke m_1 dobije pripadajuća *hash* vrijednost *h*, treba biti teško pronaći druge ulazne podatke m_2 koji daju istu *hash* vrijednost, što čini drugo svojstvo *hash* funkcije odnosno *second preimage resistance*.

Treće svojstvo je *collision resistance* koje je vrlo slično drugom svojstvu gdje se traži par ulaznih podataka m_1 i m_2 koji daju istu hash vrijednost *h*. U ovome slučaju nije zadana vrijednost m_1 već se samo traži par ulaznih podataka koji daju istu hash vrijednost.[16]

2.6.2 Kriptografski simetrični i asimetrični ključevi

Enkripcija ključevima označava kriptiranje ulaznih podataka odnosno poruka koje se nazivaju *plaintext*[17]. Za enkripciju se koriste privatni ili javni ključ. Kod simetrične kriptografije

[15] URL: http://www.aspencrypt.com/crypto101_hash.html (pristupljeno: 17. kolovoz 2018.)
[16] URL: https://www.denimgroup.com/resources/blog/2007/11/properties-of-1/ (pristupljeno: 17. kolovoz 2018.)

isti ključ se koristi za enkripciju i dekripciju poruka. Ukoliko se u komunikaciji koriste simetrični ključevi dolazi do problema distribucije ključa koji mora ostati tajan jer svaki korisnik koji posjeduje ključ može dekriptirati poruke.

Asimetrična kriptografija bazirana je na privatnom i javnom ključu. Korisnik posjeduje oba ključa. Privatni ostaje tajan dok je javni ključ dostupan svima. Za komunikaciju s korisnikom se koristi njegov javni ključ za enkripciju poruka. Korisnik onda koristi pripadajući privatni ključ za dekripciju.

Prednost asimetričnih ključeva što nema problema distribucije ključeva, ali zato zahtijeva više vremena i procesne snage za enkripciju i dekripciju poruka.[18]

2.6.3 Digitalni potpis

Digitalni potpis koristi *hash* funkciju i javni ključ kako bi se potvrdila autentičnost te osigurao integritet poruke. Korisnik prvo koristi *hash* algoritam da dobije *hash* vrijednost poruke. Enkripcija *hash* vrijednosti postiže se potenciranjem sekvence bitova vrijednošću privatnog ključa. Na odredištu se zaprima poruka s digitalnim potpisom. Izdvaja se digitalni potpis te se potenciranjem sekvence bitova digitalnog potpisa vrijednošću javnog ključa dobiva *hash* vrijednost.

Na odredištu se uspoređuje *hash* vrijednost dobivena sažimanjem poruke i *hash* vrijednosti dobivene iz digitalnog potpisa čime se osigurava da sadržaj poruke nije promijenjen. Dok upotreba javnog ključa, ovog puta za dekripciju, osigurava da je upotrebljen privatni ključ te korisnik ne može negirati da je poslao poruku pošto je korišten njegov privatni ključ.[19]

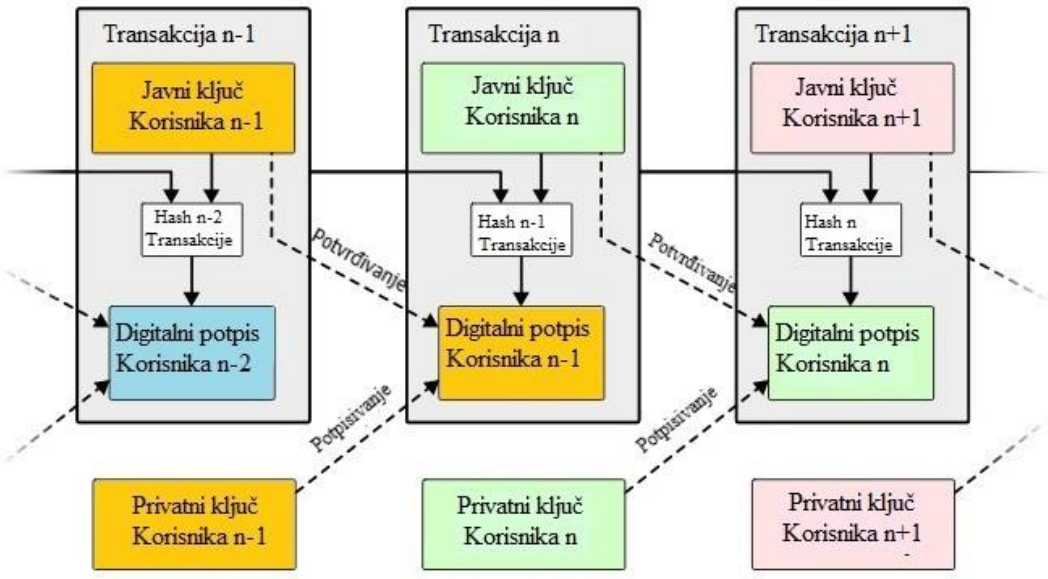
Blockchain je lista kriptografski potpisanih i neporecivih zapisa koju dijele svi čvorovi u mreži. Svaki zapis sadrži vremensku oznaku, transakcije te referencu na prethodni zapis.

---

[17] Plaintext prestavlja korisničke podatke koji su čitljivi odnosno podatke koji nisu kriptirani
[18] Peterson, L., Davie, B.: Computer Networks: System Approach – Fifth Edition, Elsevier, SAD, 2012., p. 638 - 643
[19] Kurose, J., Ross, K.: Computer Networking: A Top-Down Approach – Sixth Edition, Pearson, SAD, 2013., p. 683 - 689

Izračunava se *hash* vrijednost transakcije te se dobivena vrijednost predaje sljedećoj transakciji. Svi zapisi su međusobno povezani u lanac referencama, odnosno *hash* vrijednostima[20]

Transakcije koje se zapisuju u *blockchain* sadrže elektronički novac koji se može definirati kao lanac digitalnih potpisa. Svaki korisnik transferira elektronički novac korištenjem digitalnog potpisa i javnog ključa korisnika kojemu se šalju novci.



Slika 5. Prikaz povezivanja transakcija u lanac [13]

Slikom 5 prikazuje se upotreba *hash* vrijednosti, digitalnog potpisa te javnog i privatnog ključa u transakcijama. Šalje se javni ključ pošiljatelja transkacije koji služi identifikaciju te potvrđivanje integriteta digitalnog potpisa i podataka transakcije. *Hash* vrijednost služi za povezivanje transakcija u lanac tako da se hash vrijednost prethodne tranakcije doda u polje unutar trenutne transkacije.

Svaka transakcija sadrži *hash* vrijednost prethodne transakcije. Transakcije se skupljaju u blokove te se dodaje vremenska oznaka (engl. *timestamp*) kako bi se osiguralo da se isti novci ne pošalju dva puta.

[20] URL: https://www.gartner.com/it-glossary/blockchain (pristupljeno: 17. kolovoz 2018.)

Proof-of-work je postupak izračunavanje *hash* vrijednosti za skup transakcija u bloku. *Hash* vrijednost mora ispunjavati određeni uvjet npr. da *hash* vrijednost počinje s određenim brojem nula. Kada se *hash* vrijednost pretvori u dekadski broj može se reći da se traži *hash* vrijednost koja je manja od nekog određenog dekadskog broja.

Za potrebe pronalaska određene *hash* vrijednosti dodaje se polje *nonce* kao što je prikazano na slici 6. Vrijednost polja *nonce* se povećava dok se ne pronađe tražena *hash* vrijednost.

Proces je moguće opisati kroz šest koraka:

1. Nove transakcije se razašilju svim čvorovima u mreži
2. Svaki čvor prikuplja nove transakcije u blok
3. Svaki čvor radi *proof-of-work* za svoj blok
4. Kada čvor pronađe potrebnu *hash* vrijednost, razašilje popunjeni blok u mrežu
5. Čvorovi prihvaćaju blok ako su sve transakcije validne i nisu prethodno upisane
6. Čvor prihvaća blok tako da radi na stvaranju novog bloka koristeći *hash* vrijednost dobivenog bloka

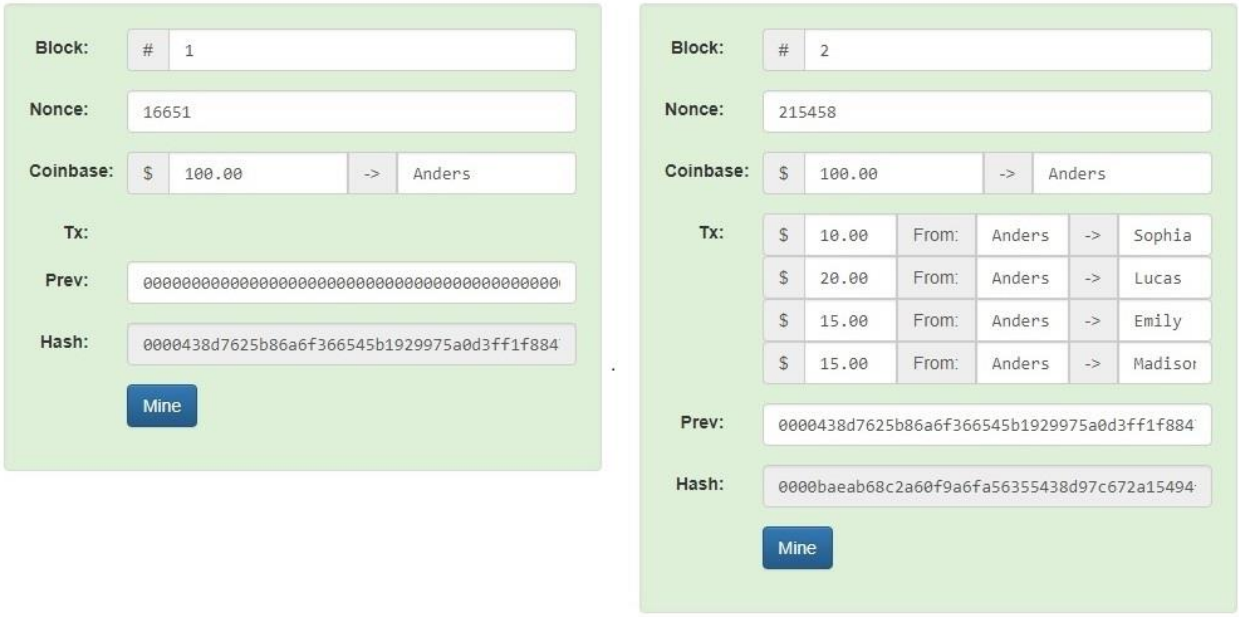
U situacijama kada dva čvora simultano razašilju dva različita bloka, neki čvorovi će primiti jedan blok prije od drugog dok će drugi čvorovi primiti blokove obrnutim redoslijedom. U tom slučaju čvor radi na prvom zaprimljenom i spremaju drugi nastali lanac. Time nastaju dva lanca jednake duljine s različitim zadnjim blokom. Razlika između njih dolazi nakon *proof-of-work* postupka i dodavanja novog bloka. Svi čvorovi koji su radili na lancu koji je ostao kraći prebacuju se na duži lanac.

Nove transakcije ne moraju uvijek doći do svih čvorova u mreži. Potrebno je da dođu do većine čvorova te završe u bloku transakcija. Ukoliko čvor ne primi blok, zatražiti će ga kada prilikom primitka sljedećeg bloka shvati da nedostaje blok. Blok koji nedostaje može zatražiti od ostalih čvorova putem njegove *hash* vrijednosti koja je zapisana u novom bloku.[21]

Čvorovi u mreži se dijele na čvorove koji imaju cijeli *blockchain* lanac (engl. *full network node*) i one koje ima samo dio lanca. Time se proširuje broj uređaja koji mogu poslati i provjeriti

[21] URL: https://nakamotoinstitute.org/bitcoin/ (pristupljeno: 18. kolovoz 2018.)

transakcije u mreži zbog manje količine memorije koja je potrebna. Time se dobiva mogućnost jednostavne provjere transakcije (engl. *Simplified Payment Verification*).[22]



Slika 6. Shematski prikaz lanca blokova[15]

Slikom 6 prikazan je prvi blok u lancu gdje je prva transakcija dodjeljivanja novca kreatoru lanca. Izračunava se *hash* vrijednost pomoću polja *nonce*. Dobivena *hash* vrijednost prvog bloka vidljiva je u drugom bloku u polju *Prev* što označava prethodni blok. Lanac se nastavlja po prethodno objašnjenim pravilima.

[22] URL: https://nakamotoinstitute.org/bitcoin/ (pristupljeno: 18. Kolovoz 2018.)

3. Programski jezik Solidity

Solidity je jezik za implementaciju pametnih ugovora. Na njega su utjecali programski jezici C++, Python i JavaScript te je dizajniran za rad s Ethereum virtualnom mašinom (engl. *Ethereum Virtual Machine*). Solidity je statički pisan, podržavanja nasljeđivanje, knjižnice i korisnički definirane tipove podataka.

Remix je integrirano razvojno okruženje (engl. *Integrated Development Environment*) za pisanje pametnih ugovora koristeći programski jezik Solidity. Remix se pokreće u internetskom pregledniku te je u njemu moguće razviti, pokrenuti i testirati pametne ugovore.[23]

Svaki pametni ugovor napisan u Solidity-u počinje s definiranjem verzije čija se oznaka koristi za odabir prevoditelja (engl.*Compiler*).[24]

```
1  pragma solidity ^0.4.20;
2
```

Slika 7. Početna linija pametnog ugovora

Slikom 7 prikazana je verzija koja će se koristiti dalje u radu za izradu pametnog ugovora.

3.1 Tipovi varijabli

Solidity je statički pisan jezik što znači da tip svake varijable mora biti specificiran ili barem poznat prilikom prevođenja. Solidity sadrži nekoliko osnovnih tipova koji se onda mogu kombinirati za stvaranje kompleksnih tipova.

[23] URL: https://solidity.readthedocs.io/en/v0.4.24/ (pristupljeno: 18. kolovoz 2018.)
[24] URL: https://solidity.readthedocs.io/en/v0.4.24/layout-of-source-files.html (pristupljeno: 18. kolovoz 2018.)

Vrijednosni tipovi (engl. *value types*) pohranjuju određenu vrstu vrijednosti i to su:

- Bool
- Cijeli brojevi (engl. *integers*)
- Brojevi fiksnih točaka (engl. *Fixed point numbers*)
- Adresa
- Red bajtova fiksne dužine (engl. *fixed-size byte arrays*)
- Red bajtova dinamične dužine (engl. *dynamically-sized byte array*)
- Adresni literali
- Racionalni i cjelobrojni literali
- Litrali niza znakova
- Heksadecimalni literali
- Enums
- Funkcije[25]

Prva tri tipa poznate su vrste vrijednosnih tipova. Adresa je prvi tip koji je specifičan za jezik Solidity. Adresa je duljina 20 bajtova što je i duljina Ethereum adrese. Adresa ima svoja posebna svojstva potrebna za lakše funkcioniranje pametnog ugovora.

```
<address>.balance(uint256)
<address>.transfer(uint 256 amount)
<address>.send(uint256 amount) return (bool)
```

Slika 8. Dodatne opcije vezane uz adresu[18]

Prva opcija *.balance* prikazana na slici 8 vraća stanje računa pripadajuće adrese u Wei[26]. Druga opcija *.transfer* koja šalje elektroničke novce u Wei prema danoj adresi. U slučaju greške

[25] URL: https://solidity.readthedocs.io/en/v0.4.24/types.html#value-types (pristupljeno: 18. kolovoz 2018.)
[26] 1 Wei je 10^{-18} Ether

vraća dobivenu grešku te šalje 2300 *gas*. *Gas* će biti kasnije u tekstu obrađen. Zadnja opcija adrese je *.send* je niži oblik *.transfer*. Također šalje zadani iznos novca u Wei određenoj adresi. U slučaju greške vraća bool vrijednost *false* te prosljeđuje 2300 *gas*. Dodatne opcije adrese su *.call*, *.callcode* i *.delegatecall*[27]

Prva vrsta referencirajućih tipova su redovi (engl. *array*). Redovi mogu biti dinamični ili fiksne veličine. Moguće je redove označiti kao javne (engl. *public*) čime se stvara funkcija za dohvaćanje (engl. *getter*). Opcije koje dolaze uz redove su: dohvaćanje duljine s *.length* i dodavanje na kraj reda s opcijom *.push*.[28]

Solidity pruža mogućnost definiranja novih tipova kroz formu *Struct*. *Struct* može sadržavati ostale tipove poput adrese, *uint* i *string* dok ne može sadržavati drugi *struct*.[29]

```
struct Driver {
    address driverAddress;
    string firstName;
    string lastName;
    uint accountBalance;
}
```

Slika 9 Primjer tipa struct

Na slici 9 prikazan je *struct* Vozač koji se sastoji od vozačeve adrese, imena i prezime te stanja računa. Ovakvim načinom se jednostavno povezuju svi podaci vezani uz vozača koji su potrebni za njegovo opisivanje.

Tip pohranjivanja podataka *mapping* povezuje određene vrijednosti s pripadajućim ključem. Ključevi u *maping-u* se ne pohranjuju već njihove *hash* vrijednosti. Zbog toga nije moguće na ključevima raditi funkciju *.length* ili iterirati kroz mapping. *Mapping* dohvaća vrijednost pridruženu *hash* vrijednosti predanog ključa.

[27] URL: https://solidity.readthedocs.io/en/v0.4.24/units-and-global-variables.html#address-related (pristupljeno: 18. kolovoz 2018.)
[28] URL: https://solidity.readthedocs.io/en/v0.4.24/types.html#arrays (pristupljeno: 18. kolovoz 2018.)
[29] URL: https://solidity.readthedocs.io/en/v0.4.24/types.html#structs (pristupljeno: 19. kolovoz 2018.)

Mapping se može označiti kao javan te onda Solidity automatski stvara funkciju dohvaćanja koja ima ključ za obavezan ulazni parametar.[30]

```
// list of drivers adresses
address[] public drivers;

// mapping drivers address with struct Driver
mapping(address => Driver) public listOfDrivers;
```

Slika 10. Primjer mapping-a adrese i vozača

Na slici 10 prikazan je primjer upotrebe *mapping-a*. Prvo je deklariran dinamičan red adresa *drivers* koji je javno dostupan. Nakon toga je deklariran *mapping listOfDrivers* koji povezuje adresu i *struct-om* Driver koji je prikazan na slici 9. Ovime se dobije red koji pohranjuje adrese svih vozača te *mapping* koji služi za pristupanje detaljima jednog vozača putem njegove adrese.

3.2 Funkcije

Funkcije su izvršive linije koda unutar pametnog ugovora koje mogu primiti ulazne parametre te vratiti vrijednosti kao izlazne parametre.

```
// adding a driver
function addDriver(string _firstName, string _lastName) public {

    // adding address to list of drivers
    drivers.push(msg.sender);

    // creating a Driver with inputs
    Driver storage driver = listOfDrivers[msg.sender];
    driver.driverAddress = msg.sender;
    driver.firstName = _firstName;
    driver.lastName = _lastName;
    driver.accountBalance = msg.sender.balance;
}
```

Slika 11. Primjer funkcije za dodavanje vozača

[30] URL: https://solidity.readthedocs.io/en/v0.4.24/types.html#mappings (pristupljeno: 19. kolovoz 2018.)

Na slici 11 prikazana je funkcija koja prima ulazne parametre imena i prezimena kao nizove znakova *_firstName* i *_lastName*. Funkcija je označena kao javna odnosno da je dostupna svima za korištenje. Prva linija u funkciji dodaje adresu pošiljatelja u dinamični red naziva *drivers* koji pohranjuje sve adrese vozača. Adresa se dohvaća putem objekta *msg* kod kojega *.sender* opcija označava adresu pošiljatelja.

Ulazni parametri ime i prezime te stanje računa koje se također dohvaća putem objekta *msg* pohranjuju se u *struct* driver odnosno vozač koji je prikazan na slici 9.[31]

Izlazni parametri funkcije deklariraju se nakon ključne riječi *returns*. Nazive ulaznih i izlaznih parametara moguće je izostaviti.

```
function retrieveDrivers() public view returns(address[]){
    return drivers;
}

// retrieve a driver with an address parameter
function retrieveADriver(address _driverAddress) public view returns(
    string,
    string,
    uint
    )
    {
    return (
        listOfDrivers[_driverAddress].firstName,
        listOfDrivers[_driverAddress].lastName,
        listOfDrivers[_driverAddress].accountBalance
    );
}
```

Slika 12. Primjeri funkcija s izalznim parametrima

Na slici 12 prikazane su dvije funkcije koje vraćaju određene izlazne parametre. Funkcija *retrieveDrivers* vraća dinamični red adresa koji sadrži adrese svih vozača.[32]

Nakon toga može se odabrati adresa iz dobivenog reda te predati funkciji *retrieveADriver* kao ulazni parametar. Funkcija *retrieveADriver* zatim dohvaća podatke vozača vezane za tu adresu. Funkcija vraća tri parametra: dva *string* i jedan *uint* koji odgovaraju tipovima imena, prezimena i stanja računa.

[31] URL: https://solidity.readthedocs.io/en/v0.4.24/control-structures.html#function-calls (pristupljeno: 20. kolovoz 2018.)
[32] URL: https://solidity.readthedocs.io/en/v0.4.24/control-structures.html#function-calls (šristupljeno: 20. Kolovoz 2018.)

Modifikatori funkcija služe kako bi se jednostavno promijenilo ponašanje funkcije. Modifikatori funkcije mogu se koristiti za provjeravanje uvjeta prije nego što se tijelo funkcije počne izvršavati.

```
    function pickWinner() public payable restricted {
    uint index = random() % players.length;
    players[index].transfer(this.balance);
    players = new address[](0);
}

modifier restricted() {
    require(msg.sender == manager);
    _;
}
```

Slika 13. Primjer modifikatora funkcije

Modifikator funkcije prikazan je na donjem dijelu slike 13 te je primjenjen na funkciji iznad. Modifikator je nazvan *restricted* te zahtijeva da adresa pošiljatelja bude jednaka *manager-u* odnosno upravitelju. Naziv modifikatora dodaje se u deklaraciju svih funkcija na kojima se želi primjeniti taj modifikator. Funkcija *pickWinner* označena je kao *public payable* i *restricted* te odabire pobjednika pseudo-slučajnim odabirom indeksa u redu adresa. Oznaka *restricted* osigurava da je funkciju pokrenuo upravitelj pametnog ugovora, te da ju ne može pokrenuti niti jedan drugi korisnik. Nakon odabira pobjednika, pobjedniku se transferiraju svi novci koji pametni ugovor posjeduje. Kod prikazan na slici 13 isječak je iz pametnog ugovora za jednostavan loto. Oznakom *payable* se označavaju funkcije putem kojih se šalje određena količina novca, koji ne uključuje *gas*.[33]

3.3 Gas – naknada za rudarenje

U *blockhain* mreži za svaki blok transakcija treba se pronaći određena *hash* vrijednost za koju treba izdvojiti računalne resurse. Naknada za te čvorove dolazi u obliku *gas*. Termin *gas* ima više asocirajućih termina s različitim značenjima: *gas cost*, *gas price*, *gas limit* i *gas fee*. Cijena transakcije se želi održati istom, no zbog volatilnosti čvorova koji rudare i vremena koje je potrebno da se izračuna *hash* vrijednost za blok, ona se mijenja.

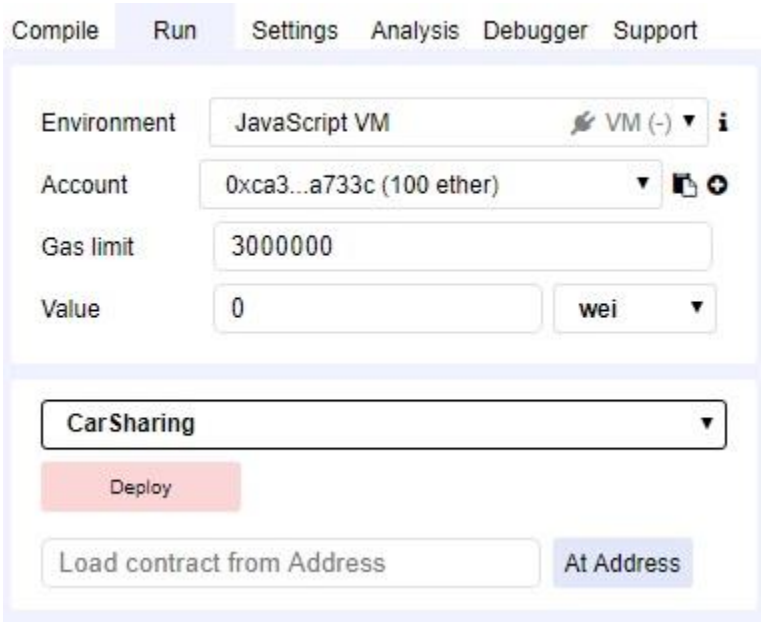
[33] URL: https://solidity.readthedocs.io/en/v0.4.24/contracts.html#modifiers (pristupljeno: 20. kolovoz 2018.)

Gas cost je statična vrijednost koliko vrijede troškovi računanja (engl. *Computational costs*) i cilj je da stvarna vrijednost *gas-a* se nikad ne promijeni tako da bi ova vrijednost trebala biti ista kroz vrijeme.

Gas price označava koliko *gas* vrijedi u drugim elektroničkim novcima ili tokenima. Kako bi vrijednost *gas-a* bila stabilna *gas price* je promijenjiva vrijednost koja se mijenja tako da poništi promjene vrijednosti valute. Vrijednost *gas price* se postavlja ovisno o tome koliko su korisnici spremni platiti za rudarenje transakcija te koliko čvorovi za rudarenje zahtijevaju za taj rad.

Gas limit je maksimalni iznos *gas-a* koji može biti iskorišten po bloku. Smatra se maksimalnim računalnim opterećenjem (engl. *Computational load*), maksimalnom veličinom transakcije ili bloka te rudari mogu postepeno mijenjati ovu vrijednost kroz vrijeme.

Gas fee je količina *gas-a* koju je potrebno platiti kako bi se određena transakcija ili pametni ugovor izvršio. *Gas fee* u bloku može biti korišten kako bi se implicirala količina računalnog opterećenja, veličine transakcije ili bloka te se na kraju *gas fee* se šalje rudarima.[34]



Slika 14. Gas limit u razvojnom okruženju Remix

[34] URL: http://ethdocs.org/en/latest/ether.html#gas-and-ether (pristupljeno: 20. kolovoz 2018.)

Prilikom slanja transakcije ili pametnog ugovora u mrežu jedna od stavki koju je potrebno ispuniti je *gas limit*. U većini slučajeva *gas limit* je automatski popunjen. Na slici 14 u razvojnom okruženju Remix specificira se adresa s koje se šalje trasnakcija u polju *Account*. Polje ispod toga je *Gas Limit* kojim korisnik označava koliko je spreman platiti za se transakcija zapiše u *blockchain*. Zadnje polje je *Value* u koje se unosi elektronički novac ukoliko je potreban prilikom slanja transakcije. Pokraj polja za unos je polja za odabir jedinicu plaćanja koja je u ovom slučaju Wei. Wei je najmanja jedinica Ether valute. Jedan Wei je 10^{-18} Ether.

Tablica 2. Skala Ether valute po veličini [29]

Unit	Wei Value	Wei
wei	1 wei	1
Kwei (babbage)	1e3 wei	1,000
Mwei (lovelace)	1e6 wei	1,000,000
Gwei (shannon)	1e9 wei	1,000,000,000
microether (szabo)	1e12 wei	1,000,000,000,000
milliether (finney)	1e15 wei	1,000,000,000,000,000
ether	1e18 wei	1,000,000,000,000,000,000

Tablicom 2 prikazane su jedinice Ether valute po veličine od Wei do Ether. Počevši od Wei prema Etheru svaka jedinica je veća za 10^3 od prethodne.

4. Pametni ugovori

Pametni ugovor je skup funkcija i podataka koji su pohranjeni na određenoj adresi u Ethereum *blockchain-u*. Moguće ga je smatrati jednim mjestom u bazi podataka iz kojega se mogu dohvaćati podaci i mijenjati podatke pozivajući pripadajuće funkcije.

Sve promjene koje se rade u *blockchain-u* nazivaju se transakcije te ih ostali čvorovi u mreži moraju prihvatiti odnosno potvrditi. Transakcije podrazumijevaju da se radi promjena na podacima, ne samo dohvaćanje podataka, te se transakcije izvršavaju u cijelosti ili se uopće ne izvrše. Ukoliko se žele promijeniti dvije vrijednosti, promijeniti će se obje vrijednosti ili se neće promijeniti ni jedna vrijednost.

Ukoliko se želi transferirati novac s jednog računa na drugi, transakcijsko obilježje *blockchain-a* osigurava da se iznos preuzet s jednog računa uvijek doda na drugi račun. Ukoliko iz nekog razlog primatelj nije u stanju primiti elektronički novac, iznos pošiljateljovog računa ostaje nepromijenjen. Transakcije uvijek potpisuje digitalnim potpisom pošiljatelj te se time sprječavaju modifikacije u bazi podataka.[35]

Za potrebe ovoga rada razvijen je pametni ugovor za dijeljenje automobila te će on biti prikazan i objašnjen dalje u ovom poglavlju. Detaljno će se objasniti njegove varijable i funkcije koje su potrebne za funkcioniranje samog pametnog ugovora.

```
pragma solidity ^0.4.20;

contract CarSharing {

    address public admin;

    //Seting admin address to the account that creates smart contract
    function CarSharing() public {
        admin = msg.sender;
    }

    modifier isAdmin() {
        require(msg.sender == admin);
        _;
    }
```

Slika 15.Deklaracija pametnog ugovora i konstrukcijska funkcija

[35] URL: https://solidity.readthedocs.io/en/v0.4.24/introduction-to-smart-contracts.html (pristupljeno: 1. rujan 2018.)

Slikom 15 prikazan je početni dio pametnog ugovora za dijeljenje automobila nazvan *CarSharing*. Već je objašnjeno da prva linija koda definira verziju za potrebe prevoditelja. Druga linija koda je deklariranje pametnog ugovora te postavljanje njegovog imena. Sa samim stvaranjem pametnog ugovora, uvijek se jednom na početku pokreće kontrukcijska funkcija. Kontrukcijsku funkciju se deklarira tako da mora imati isto ime kao i pametni ugovor, u ovome slučaju *CarSharing*. Konstrukcijska funkcija uzima adresu pošiljatelja odnosno stvaratelja pametnog ugovora te je postavlja za administratora. Iz tog razlog je deklarirana varijabla *admin* tipa adresa. Ispod kontruktorske funkcije definiran je *modifier isAdmin* koji se postavlja na određene funkcije koje će samo administrator biti u mogućnosti izvršiti.

```
// Declaring Driver data type
struct Driver {
    address driverAddress;
    string firstName;
    string lastName;
    uint accountBalance;
}

// list of drivers adresses
address[] public drivers;

// mapping drivers address with struct Driver
mapping(address => Driver) public listOfDrivers;
```

Slika 16. Varijable vezane za vozače

Varijable potrebne za upravljanje vozačima prikazane su na slici 16. Prvo se koristi *struct* da se definira nova vrsta varijable nazvana *Driver* odnosno vozač koja sadrži podatke o vozaču. Varijabla vozač sastoji se od četiri elementa: adrese vozača, njegovog imena i prezimena te stanja računa. Nakon toga deklariran je dinamičan red adresa *drivers* u kojega će se zapisivati sve adrese vozača koji se registriraju. Za kraj je napravljen *mapping listOfDrivers* koji povezuje Ethereum adresu vozača sa *struct-om driver*. On će služiti za jednostavno dohvaćanje podataka o jednome vozaču putem njegove adrese.

Ovakav način dohvaćanja podataka uvijek koristi jednake računalne resurse neovisno o tome koliki je broj vozača registriran. To dolazi od samog načina funkcioniranja *mapping-a* koji će uzeti adresu te provjeriti podatke povezane s *hash* vrijednosti unesene adrese. Iz toga razloga

nije moguće iterirati kroz mapping nego je moguće samo dohvatiti podatke vezane za samo jednu adresu.

```
// Declaring Vehicle data type
struct Vehicle {
    address vehicleAddress;
    string brand;
    string model;
    string registerationPlates;
    uint hourlyPrice;
    bool availability;
    address owner;
    address currentDriver;
    uint timeOfNextAvailability;
    uint startOfLastDrive;
}

// list of vehicle addresses
address[] public vehicles;

// mapping vehicle address with struct Vehicle
mapping(address => Vehicle) public listOfVehicles;

address[] public availableVehicles;
```

Slika 17. arijable vezane za vozila

Deklariranje i dohvaćanje podataka o vozilima postavljeno je na isti način kao i za vozače. Razlike su u samim podacima vezanim za *struct* vozilo kao što je prikazano na slici 17. Definiran je *struct vehicle* odnosno vozilo koje se sastoji od većeg broja podataka nego vozač. Dio varijabli se koristi da opišu vozilo dok će se ostale koristiti kasnije unutar funkcija.

Opisne varijable unose se prilikom registriranja vozila:

- vehicleAddress – Ethereum adresa vozila
- brand – marka automobila
- model – model automobila
- registrationPlates – registracijske oznake automobila
- hourlyPrice – cijena sata iznajmljivanja automobila izražena u Wei
- owner – Ethereum adresa vlasnika automobila koji može mijenjati podatke automobila

Ostale varijable koje će se koristiti u ostalim funkcijama su:

- availability – dostupnost vozila

- currentDriver – Ethereum adresa trenutnog vozača
- timeOfNextAvailability – vrijeme kada će vozilo biti ponovno dostupno
- startOfLastDrive – vrijeme početka posljednje vožnje

Ispod definiranja vozila na slici 17. deklariran je dinamičan red adresa vozila *vehicles* gdje se zapisuju adrese svih registriranih vozila te nakon njega *mapping listOfVehicles* koji služi za dohvaćanje podataka o vozilo pomoću njegova Ethereum adrese. Na kraju je deklariran dinamičan red adresa *availableVehicles* koji će sadržavati adrese trenutno slobodnih vozila.

```
// adding a driver
function addDriver(string _firstName, string _lastName) public {
    //Check to see if address is already used
    if(listOfDrivers[msg.sender].driverAddress == 0x0
    && listOfVehicles[msg.sender].vehicleAddress == 0x0) {
    // adding address to list of drivers
        drivers.push(msg.sender);

        // creating a driver with input arguments
        Driver storage driver = listOfDrivers[msg.sender];
        driver.driverAddress = msg.sender;
        driver.firstName = _firstName;
        driver.lastName = _lastName;
        driver.accountBalance = msg.sender.balance;
    } else {
        revert();
    }
}
```

Slika 18. Funkcija za dodavanje vozača

Dodavnanje vozača vrši se pokretanjem funkcije *addDriver* koja je prikazana na slici 18. Funkcija *addDriver* uzima dva ulazna parametra *_firstName* i *_lastName* koji se koriste za postavljanje vrijednosti imena i prezimena vozača. Funkcija prvo provjerava je li adresa pošiljatelja koji želi dodati vozača već korištena. Za provjeru se koriste *mapping listOfDrivers* i *listOfVehicles* gdje se provjeravaju podaci koji pripadaju adresi pošiljatelja odnosno *msg.sender*.

Mapping funkcionira tako da će za svaku unesenu vrijednost vratiti podatke ako postoje te ako ne postoje vratiti će zadane vrijednost (engl. *default values*). Za shvaćanje provjeravanja uvjeta potrebno je prvo pogledati liniju koda *Driver storage driver = listOfDriver[msg.sender];*. Tom linijom koda stvara se nova instanca varijable *Driver* naziva *driver*. Deklarirana varijabla *driver* nakon toga pridružuje se u *mapping listOfDrivers* gdje je ključ adrese pošiljatelja

msg.sender. Nakon toga se postavlja prva varijabla unutar varijable *driver*. Upisuje se adresa pošiljatelja kao adresa vozača kroz liniju *driver.driversAddress = msg.sender;*. Iz toga je vidljivo da korisnik ne kontrolira postavljanje adrese i da ne može postaviti svoju adresu.

Provjeru korištenja adrese izvršava se tako da se pristupi *mapping-u* vrijednosti pridruženoj ključu *msg.sender* odnosno pošiljatelju. Unutar dobivenih podataka provjerava se vrijednost zapisane adrese. Ukoliko je ona 0x0 može se zaključiti da adresa pošiljatelja nije korištena jer je vrijednost 0x0 moguća samo kao zadana vrijednost. Na isti način provjerava se je li adresa korištena za registraciju vozila i vozača.

Nakon provjere prvo se adresa pošiljatelja dodaje u dinamičan red adresa *drivers*. Podaci o vozaču zapisuju u varijablu *driver* koja je adresom povezana u *mapping listOfDrivers*. Ukoliko provjera adrese ne prođe, radi se *revert();* kako se ne bi trošio dodatan *gas*.

```
//adding a vehicle
function addVehicle(string _brand, string _model,
    string _registrationPlates, uint _hourlyPrice, address _owner) public {
    if(listOfDrivers[msg.sender].driverAddress == 0x0
    && listOfVehicles[msg.sender].vehicleAddress == 0x0) {
        // adding address to a list of vehicles
        vehicles.push(msg.sender);

        // creating a Vehicle with inputs
        Vehicle storage vehicle = listOfVehicles[msg.sender];
        vehicle.vehicleAddress = msg.sender;
        vehicle.brand = _brand;
        vehicle.model = _model;
        vehicle.registerationPlates = _registrationPlates;
        vehicle.hourlyPrice = _hourlyPrice; //in wei
        vehicle.availability = true;
        vehicle.owner = _owner;
        vehicle.currentDriver = 0x0;
        vehicle.timeOfNextAvailability = 0;
        vehicle.startOfLastDrive = 0;
        availableVehicles.push(msg.sender);
    } else {
        revert();
    }
}
```

Slika 19. Funkcija dodavanja vozila

Funkcija dodavanja vozila *addVehicle* prikazana na slici 19 i funkcija dodavanja vozača *addDriver* imaju istu provjeru adrese te isti princip zapisivanja podataka. Razlika je samo u *struct-u Vehicle* koji uzima više ulaznih podataka te upisuje još druge dodatne podatke koji su prethodno opisani. Princip dodavanja vozila je isti kao i vozača te se koriste pripadajući

dinamični red adresa *vehicles* i mapping *listOfVehicles*. Na kraju funkcije se adresa vozila dodaje na kraj dinamičnog reda adresa dostupnih vozila *availableVehicles*.

```
    // retirieve an array with addresses of all vehicles
function retrieveVehicles() public view returns(address[]) {
    return vehicles;
}

// retrieve an array of address that contains all addresses
function retrieveDrivers() public view returns(address[]){
    return drivers;
}
```

Slika 20. Funkcije dohvaćanja adresa svih vozila i svih vozača

Funkcije za dohvaćanje svih adresa vozila i vozača prikazane su na slici 20. Obje funkcije ne zahtijevaju ulazne parametre te vraćaju dinamičan red adresa vozila ili vozača. Dinamičnim redovima adresa *vehicles* i *drivers* prethodno je prikazan deklaracija na slikama 16. i 17. te je prikazano dodavanje adresa u te redove slikama 18. i 19.

```
// retrieve a driver with an address parameter
function retrieveADriver(address _driverAddress) public view returns(
    string,
    string,
    uint
    )
    {
    return (
        listOfDrivers[_driverAddress].firstName,
        listOfDrivers[_driverAddress].lastName,
        listOfDrivers[_driverAddress].accountBalance
    );
}
```

Slika 21. Funkcija dohvaćanja podataka jednog vozača

Nakon što se dohvate svi vozači moguće je dohvatiti dodatne podatke o svakom vozaču pomoću funkcije *retrieveADriver* prikazane na slici 21. Funkcija uzima adresu *_driverAddress* kao jedini ulazni parametar. Adresa se onda koristi da se pristupi podacima u varijabli tipa *Driver* koja je u *mapping-u listOfDrivers* povezana s pripadajućom adresom. Dohvaćaju i vraćaju se podaci: ime i prezime te stanje računa vozača.

```
// retrieve a vehicle with an address
function retrieveAVehicle(address _vehicleAddress) public view returns(
    string,
    string,
    string,
    uint,
    bool
) {
    return (
        listOfVehicles[_vehicleAddress].brand,
        listOfVehicles[_vehicleAddress].model,
        listOfVehicles[_vehicleAddress].registerationPlates,
        listOfVehicles[_vehicleAddress].hourlyPrice,
        listOfVehicles[_vehicleAddress].availability
    );
}

// retrieve additional vehicle information
function retrieveVehicleInfo(address _vehicleAddress) public view returns (
    address,
    address,
    uint,
    uint
) {
    return (
        listOfVehicles[_vehicleAddress].owner,
        listOfVehicles[_vehicleAddress].currentDriver,
        listOfVehicles[_vehicleAddress].timeOfNextAvailability,
        listOfVehicles[_vehicleAddress].startOfLastDrive
    );
}
```

Slika 22. Funkcije za dohvaćanje podataka o vozilu

Solidity postavlja ograničenje na maksimalan broj varijabli koje funkcija može vratiti. Jedna funkcija može vratiti sedam varijabli dok je vozilo opisano s devet varijabli. Zbog toga vraćanje podataka o vozilu je razdvojeno u dvije funkcije kao što je prikazano na slici 22.

Obje funkcije dohvaćaju podatke po istom principu kao i prethodno objašnjena funkcija za dohvaćanje podataka o vozaču. Također imaju jedan ulazni parametar, adresu vozila.

Funkcija *retrieveAVehicle* prikazana na slici 22. služi za dohvaćanje osnovnih informacija o vozilu koje bi mogućem vozaču bile korisne: marka i model automobila, njegova registracijska oznaka, cijena po satu te dostupnost vozila. Funkcija *retrieveAVehicleInfo* služi za dohvaćanje preostalih podataka o vozilu: adresa vlasnika, trenutni vozač, vrijeme sljedeće dostupnosti i vrijeme početka vožnje.

```
//address and number of hours wanted to rent a vehicle are needed
//to rent a vehicle
function rentAVehicle(address _vehicleAddress, uint _numberOfHours)
    public payable{
    //Check to see if address is registered as driver
    if(listOfDrivers[msg.sender].driverAddress != 0x0){
        //It is required that the vehicle is availble
        require(listOfVehicles[_vehicleAddress].availability == true);

        //Amount of wei that needs to send to rent a vehicle
        //for certain amount of hours
        uint totalPrice = listOfVehicles[_vehicleAddress].hourlyPrice * _numberOfHours;

        //Checking if amoount sends is right
        if(msg.value == totalPrice){
            Vehicle storage vehicle = listOfVehicles[_vehicleAddress];
            vehicle.availability = false;
            vehicle.timeOfNextAvailability = block.timestamp + (3600 * _numberOfHours);
            vehicle.currentDriver = msg.sender;
            vehicle.startOfLastDrive = now;
            _vehicleAddress.transfer(totalPrice);
            removeFromAvailableVehicles(_vehicleAddress);
        } else {
            revert();
        }
    } else {
        revert();
    }

}
```

Slika 23. Funkcija iznajmljivanja vozila

Iznajmljivanje vozila izvršava se funkcijom *rentAVehicle* prikazanoj na slici 23. Funkcija uzima dva ulazna parametra: adresu i broj sati na koji se želi iznajmiti automobil. Prvo se provjerava da je pošiljatelj registriran kao vozač. Nakon toga se provjerava dostupnost vozila. Ukoliko dostupnost vozila *availability* vraća vrijednost *false*, funkcija će se prekinuti. Ukoliko se zadovolji taj uvjet kreće se izračunavanje ukupnog iznosa koju pošiljatelj mora poslati unutar transakciji za plaćanje iznajmljivanja vozila. Ukupni iznos je umnožak broju sati za koji se želi iznajmiti vozilo i iznosu cijene sata iznajmljivanja vozila. Ukupni iznos deklariran je kao *totalPrice* i formula za njegov izračun prikazana je na slici 23.

Slijedi provjera da je u transakciji poslan iznos koji odgovara prethodno izračunatom ukupnom iznosu. Slijede promjene na podacima vozila koje se iznajmljuje. Dostupnost se prebacuje u vrijednost *false*, izračunava se vrijeme sljedeće dostupnosti, postavlja se adresa pošiljatelja za trenutnog vozača i početak vožnje. Linija koda *_vehicleAddress.transfer(totalPrice);* preusmjerava primljeni iznos na Ethereum adresu vozila

koje je iznamljeno. Moguće je promijeniti kod tako da se iznos preusmjerava vlasniku vozila. Za kraj se poziva funkcija *removeFromAvailableVehicles();* koja uklanja adresu vozila, koje se iznamljuje, s popisa adresa dostupnih vozila.

Za izračunavanje vremena kada će vozilo biti ponovno dostupno koristi se *block.timestamp*. Svaki blok ima vremensku oznaku. Vremenska oznaka trenutnog bloka je *block.timestamp* te izražava broj proteklih sekundi od *unix epoch*. *Unix epoch* predstavlja datum 1. siječanj 1970. u 00:00 h.[36] Broj sati za koji će se vozilo iznajmiti se pretvara u sekunde te dodaje vrijednosti *block.timestamp*. U varijablu *startOfLastDrive* sprema se vrijednost *now* koja je iste vrijednosti kao i *block.timestamp* te je samo kraći naziv iste vrijednosti.[37]

```
function removeFromAvailableVehicles(address _vehicleAddress) public {
    for (uint i = 0; i < availableVehicles.length; i++){
        if(_vehicleAddress == availableVehicles[i]){
            address keyToMove = availableVehicles[availableVehicles.length-1];
            availableVehicles[i] = keyToMove;
            availableVehicles.length--;
        }
    }
}

function getAvailableVehicles() public view returns(address[]){
    return availableVehicles;
}
```

Slika 24. Funkcije postavljanja dostupnih vozila i dohvaćanja dostupnih vozila

Funkcija iznajmljivanja vozila *rentAVehicle* završava pozivanjem funkcije *removeFromAvailableVehicles* koja je prikazana na slici 24. Funkcija *removeFromAvailableVehicles* upravlja dinamičnim redom adresa *availableVehicles* čija je deklaracija prikazana na slici 17. Prethodno je objašnjeno dodavanja adrese vozila na listu dostupnih vozila prilikom registracije vozila. Funkcija za uklanjanje vozila s tog popisa pomoću *for* petlje iterira kroz dinamičan red te uspoređuje adresu dohvaćenu iz dinamičnog reda s adresom koja je predana kao ulazni parametar funkcije. Usporedba se vrši unutar *if* petlje gdje se uspoređuje adresa predana u funkciju i adresa iz dinamičnog reda *availableVehicles* na indeksu i koji se povećava sa svakom iteracijom *for* petlje.

[36] URL: http://unixepoch.com/ (pristupljeno: 1. rujan 2018.)
[37] URL : https://solidity.readthedocs.io/en/v0.4.24/units-and-global-variables.html (pristupljeno: 1. rujan 2018.)

Kada dogodi ispunjenje *if* uvjeta, odnosno pronađe tražena adresa, deklarira se pomoćna varijabla *keyToMove* koja se postavlja za zadnje mjesto u dinamičnom redu. Nakon toga slijedi premještanje tražene adrese na kraj reda te skraćivanje reda za jednu adresu čime se uklanja tražena adresa iz reda te se podešava duljina reda.

Kada bi se koristila funkcija *.remove()* na traženoj adresi u redu, adresa koja se briše bi u redu bila zamijenjena zadanom vrijednosti 0x0 te bi duljina reda ostala nepromijenjena.

```
function timeTillEndOfRent(address _vehicleAddress)
public view returns(uint) {
    uint end =
    (listOfVehicles[_vehicleAddress].timeOfNextAvailability
    - listOfVehicles[_vehicleAddress].startOfLastDrive) / 3600;
    return end;
}
```

Slika 25. Funkcija za izračunavanje vremena do kraja iznajmljivanja automobila

Nakon što je vozilo iznajmljeno, moguće je da bi vlasnik ili neki drugi korisnik želio znati koliko je vremena preostalo do kraja iznajmljivanja. Iz tog razloga napravljena je funkcija *timeTillEndOfRent* koja izračunava vrijeme do kraja iznamljivanja kao što je prikazano na slici 25. Funkcija zahtijeva adresu vozila kao ulazni parametar te izračunava koliko je preostalo sati do kraja iznamljivanja pomoću varijabli *timeOfNextAvailability* i *startOfLastDrive*. Vrijednosti varijabli se postavljaju prilikom iznajmljivanja vozila kao što je prikazano na slici 24. Vrijednosti *timeOfNextAvailability* se oduzme *startOfLastDrive* te se podijeli s 3600 kako bi se pretvorilo iz sekunde u sate.

```
//Ending drive drive - function that is called from remote server
//Checks if admin is caling it and that the time of rent has expired
function endDrive(address _vehicleAddress) public {
    require(now >= listOfVehicles[_vehicleAddress].timeOfNextAvailability &&
    (msg.sender == admin || msg.sender == listOfVehicles[_vehicleAddress].currentDriver));

    Vehicle storage vehicle = listOfVehicles[_vehicleAddress];

    //Setting vehicle to be available
    vehicle.availability = true;
    vehicle.timeOfNextAvailability = 0;
    vehicle.currentDriver = 0x0;
    vehicle.startOfLastDrive = 0;
}
```

Slika 26. Funkcija završetka iznajmljivanja vozila

Nakon završetka vožnje potrebno je postaviti vrijednosti određenih varijabli vezanih za vozilo kako bi to vozilo bilo u mogućnosti da se ponovno iznajmi. Za to služi funkcija *endDrive* prikazana na slici 26. kojom se postavlja dostupnost vozila, vrijeme sljedeće dostupnosti, trenutni vozač i početak vožnje. Dostupnost vozila *availability* se postavlja u bool vrijednost *true* čime se omogućuje da se vozilo ponovno iznajmi. Vrijeme sljedeće dostupnosti *timeOfNextAvailability*, trenutni vozač *currentDriver* i početak vožnje *startOfLastDrive* postavljaju se na vrijednost nule, odnosno time se brišu podaci vožnje prije.

Da bi se navedene varijable ažurirale prvo se radi provjera više uvjeta. Vrijeme trenutnog bloka mora biti veće od vremena sljedeće dostupnosti vozila. Uz taj uvjet potrebno je zadovoljiti provjeru pošiljatelja. Adresa koja poziva funkciju mora biti administrator ili trenutni vozač vozila.

```
function numberOfDrivers() public view returns(uint) {
    return drivers.length;
}

function numberOfVehicles() public view returns(uint) {
    return vehicles.length;
}
```

Slika 27. Funkcije dohvaćanja broja registriranih vozača i vozila

Dvije jednostavne funkcije za dohvaćanje broja registriranih vozača i vozila prikazane su na slici 27. Ovakve funkcije moguće je koristiti na početnoj internetskoj stranici web aplikacije kako bi se korisniku prikazalo koliko je trenutno registriranih vozila i vozača.

```
function editDriverFirstName(
    address _driverAddress,
    string _newFirstName
    )
    public {
    require(msg.sender == listOfDrivers[_driverAddress].driverAddress);

    Driver storage driver = listOfDrivers[_driverAddress];

    driver.firstName = _newFirstName;
    }

function editDriverLastName(
    address _driverAddress,
    string _newLastName)
    public {
    require(msg.sender == listOfDrivers[_driverAddress].driverAddress);

    Driver storage driver = listOfDrivers[_driverAddress];

    driver.lastName = _newLastName;
    }
```

Slika 28. Funkcije za ažuriranje podataka vozila

Za potrebe mijenjanja podataka vozača napravljene su funkcije prikazane na slici 28. Obje funkcije su identične te prva služi za promjenu imena vozača dok druga funkcija služi za promjenu prezimena vozača. Obje funkcije za ulazne parametre primaju dva parametra: adresu i niz znakova. Adresa označava adresu vozača kojemu se želi promijeniti ime ili prezime. Provjerava se je li adresa pošiljatelja poruke odgovara adresi registriranog vozača. Nakon što je zadovoljen uvjet funkcija mijenja ime ili prezime u primljeni niz znakova *_newFirstName* ili *_newLastName*.

```
function editVehicleBrand(address _vehicleAddress, string _newBrand) public {
    require(msg.sender == listOfVehicles[_vehicleAddress].owner);

    listOfVehicles[_vehicleAddress].brand = _newBrand;
}

function editVehicleModel(address _vehicleAddress, string _newModel) public {
    require(msg.sender == listOfVehicles[_vehicleAddress].owner);

    listOfVehicles[_vehicleAddress].model = _newModel;
}

function editVehicleRegistrationPlates(address _vehicleAddress, string _newRegistrationPlates) public {
    require(msg.sender == listOfVehicles[_vehicleAddress].owner);

    listOfVehicles[_vehicleAddress].registerationPlates = _newRegistrationPlates;
}

function editVehicleHourlyPrice(address _vehicleAddress, uint _newHourlyPrice) public {
   require(msg.sender == listOfVehicles[_vehicleAddress].owner);

    listOfVehicles[_vehicleAddress].hourlyPrice = _newHourlyPrice;
}

function editVehicleOwner(address _vehicleAddress, address _newOwner) public {
    require(msg.sender == listOfVehicles[_vehicleAddress].owner);

    listOfVehicles[_vehicleAddress].owner = _newOwner;
}
```

Slika 29. Funkcije za ažuriranje podataka vozila

Vozilo je opisano s više varijabli nego vozač i zato je potrebno više funkcija za promjenu pripadajućih podataka. Promjena podataka se mogla vršiti unutar jedne funkcije, ali onda to zahtijeva promjenu svih podataka odjednom. Funkcijama prikazanim na slici 29. omogućuje se promjena samo jednog podatka za slučaj npr. krivo unesenog modela ili bilo kakvog krivo unesenog podatka. Isto tako omogućuje se samo promjena cijene sata iznamljivanja *hourlyPrice* ukoliko vlasnik želi povećati ili smanjiti cijenu sata iznamljivanja. Sve funkcije za promjenu podataka vozila prihvaćaju dva ulazna parametra: adresu vozila te dodatnu varijablu koja se odnosi na podatak o vozilu koji se mijenja. Tako ukoliko se mijenja marka, model i registracijske oznake vozila, funkcije uzimaju uz adresu vozila varijablu tipa *string* jer su tako i pohranjene varijable marka, model i registracijske oznake vozila.

Svaka funkcija na početku zahtijeva da adresa pošiljatelja je adresa vlasnika vozila. Time se omogućuje samo vlasniku vozila da mijenja podatke o samom vozilu. Na kraju funkcija pristupa određenoj varijabli vozila te u nju sprema novu vrijednost koja joj je predana kao drugi ulazni parametar.

Zadnje funkcije u pametnom ugovoru *CarSharing* odnose se na brisanje vozača i vozila iz pripadajućih redova adresa.

```
function deleteDriver(address _driverAddress) public {
    require(msg.sender == listOfDrivers[_driverAddress].driverAddress);

    Driver storage driver = listOfDrivers[_driverAddress];

    driver.driverAddress = 0x0;
    driver.firstName = '';
    driver.lastName = '';
    driver.accountBalance = 0;

    for(uint i = 0; i < drivers.length; i++){
        if(_driverAddress == drivers[i]){
            address keyToMove = drivers[drivers.length-1];
            drivers[i] = keyToMove;
            drivers.length--;
        }
    }
}
```

Slika 30. Funkcija brisanja vozača

Funkcija brisanja vozila *deleteDriver* prikazana na slici 30. koristi dijelove logike prethodno prikazanih funkcija *endDrive* i *removeFromAvailableVehicles*. Za izvršavanje funkcije prvo se vrši provjera adrese pošiljatelja koja mora odgovarati vrijednosti adrese vozača pohranjenoj u *mapping-u listOfDrivers* na mjestu adrese koja predana kao ulazni parametar *_driverAddress*. Nakon toga se sve vrijednosti u navedenom *mapping-u* na mjestu adrese postavljaju na nule zbog nemogućnosti brisanja vrijednosti iz *mapping-a*. Nakon toga slijedi pronalazak adrese u dinamičnom redu adresa *drivers*. Nakon pronalaska indeksa adrese ona se pomiče na kraj te se duljina reda smanjuje za jedan te se time briše adresa iz reda.

```
function deleteVehicle(address _vehicleAddress) public {
    require(msg.sender == listOfVehicles[_vehicleAddress].owner);

    Vehicle storage vehicle = listOfVehicles[_vehicleAddress];

    vehicle.vehicleAddress = 0x0;
    vehicle.brand = '';
    vehicle.model = '';
    vehicle.registerationPlates = '';
    vehicle.hourlyPrice = 0;
    vehicle.availability = false;
    vehicle.owner = 0x0;
    vehicle.currentDriver = 0x0;
    vehicle.timeOfNextAvailability = 0;
    vehicle.startOfLastDrive = 0;

    for(uint i = 0; i < vehicles.length; i++){
        if(_vehicleAddress == vehicles[i]){
            address keyToMove = vehicles[vehicles.length-1];
            vehicles[i] = keyToMove;
            vehicles.length--;
        }
    }
}
```

Slika 31. Funkcija brisanja vozila

Zadnja funkcija u pametnom ugovora je brisanje vozila, prikazana na slici 31, koja izvršava brisanje vozila na isti način kao i funkcija za brisanje vozača. Razlike počivaju u pristupanju drugom *mapping-u listOfVehicles* i dinamičnom redu adresa *vehicles.* Na isti način kao i prethodno potrebno je postaviti vrijednosti vezane za vozilo na vrijednosti nule te ukloniti adresu vozila iz reda registriranih vozila *vehicles*.

Ovime završava poglavlje te se prelazi na sučelje između pametnog ugovora u distribuiranoj bazi i web aplikacije.

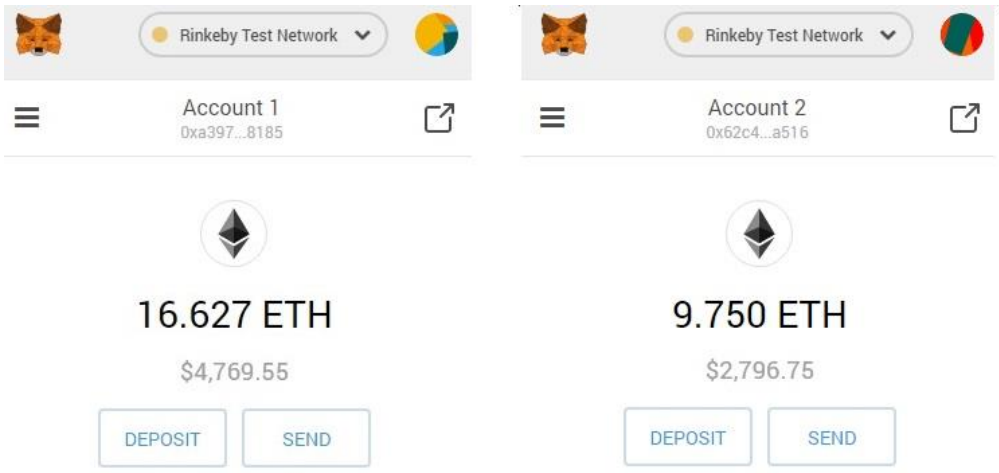
5. Sučelje za distribuiranu bazu podataka

Za korištenje aplikacija koje se baziraju na *blockchain-u* potrebno je posjedovati račun. Jedan od načina je instaliranje dodatka MetaMask na internetski preglednik. MetaMask predstavlja most između internetskog preglednika i distribuirane mreže na kojoj je pokrenuta aplikacija te će se MetaMask koristiti u ovome radu. MetaMask omogućuje pokretanje distribuiranih aplikacija (engl. *Distributed applications*) na Ethereum mreži u internetskom pregledniku bez zahtjeva da se pokreće cijeli Ethereum čvor.

5.1 Kreiranje računa potrebnih za rad s aplikacijom

MetaMask uključuje sigurnu pohranu računa, sučelje za upravljanje računima te potpisivanje transakcija u *blockchain-u*. MetaMask uz pristup glavnoj Ethereum mreži omogućuje pristup više testnih mreža poput: Ropsten, Kovan i Rinkeby.[38]

Svaki račun je zapravo niz određenih Ethereum adresa i njima pripadajućih privatnih i javnih ključeva. Stvaranje novog računa putem MetaMask-a zahtijeva unošenje lozinke.



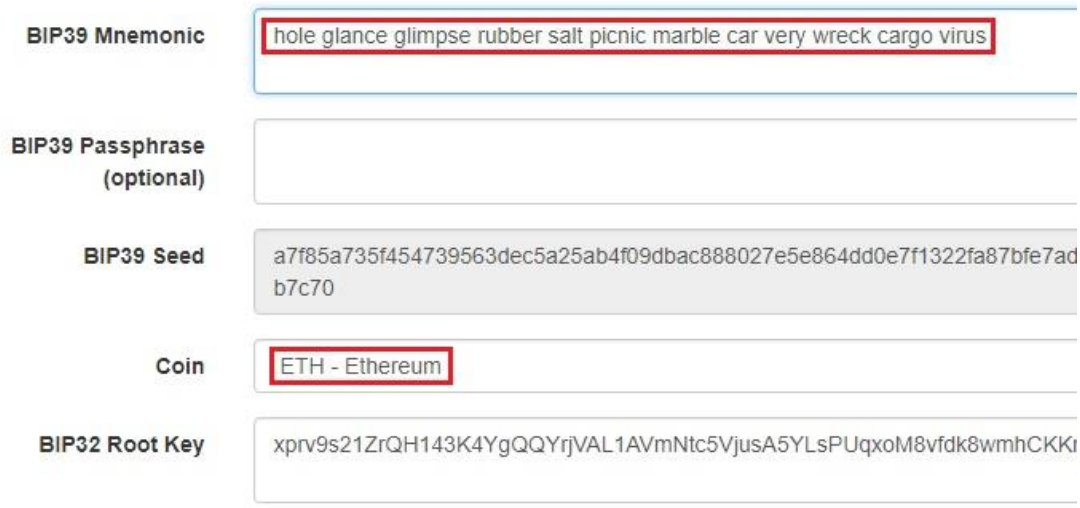
Slika 32. Prva dva računa iz skupa računa u MetaMask-u

[38] URL: https://metamask.io/ (pristupljeno: 3. rujan 2018.)

Slikom 32. prikazana su prva dva računa u nizu. Prvi račun s Ethereum adresom koja počinje s *0xA397*... te drugi račun s Ethereum adresom koja počinje *0x62C4*...

Nakon kreiranja računa prikaže se 12 riječi koje su važne za kreirani niz računa. Primjer takvih 12 dobivenih riječi je: "*hole glance glimpse rubber salt picnic marble car very wreck cargo virus*". Riječi se čine nasumične, ali su generirane po određenom postupku te s određenim uvjetima kako bi bile lakše za zapamtiti.[39]

Kada se dobivenih 12 riječi unese u *Mnemonic Code Converter* te odabere pripadajuća valuta kako je prikazano na slici 33. dobije se pristup nizu računa.



Slika 33. Unošenje podataka u Mnemonic Code Converter

Uz niz računa putem tih 12 riječi dobiju se i pripadajući javni i privatni ključeva kao što je prikazano na slici 34.

[39] URL: https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki (pristupljeno: 4. rujan 2018.)

Putanja	Adresa	Javni ključ	Privatni ključ
m/44'/60'/0'/0/0	0xA397cb6d96fe656314409B827b52DEbE6dfa8185	0x02dedf8431d46a1c40b215c579c1c6c2d99a7fcfbde5a5529325d17f155392fcbb	0x543fdef82222302e8bb3edb
m/44'/60'/0'/0/1	0x62c4c882Cc9062692dfb00A367275b81757cA516	0x032e0e08867719f426e5a58089e05f6f0f68e19ef850936bdae1bfa80d85e8760d	0xe96a1f7d35a4f267493af3e
m/44'/60'/0'/0/2	0x37C81EaBEaC638CaC656EB6142E5BB0d29829861	0x03d53ce9f1ebc6a77768d1e7bcc26e0918671270f628cb6a70951e5afb6be45921	0x7472d078d181a37398502c8
m/44'/60'/0'/0/3	0x06835468E7761Fa793641C6d95BAc9dc02d33939	0x03901bc4752de3131781767dccb482866dc4fd24773e9fe734013e17d0bddf1f8a	0x7ebd300c87c50aa11cd3d9c

Slika 34. Dobiveni niz adresa i pripadajućih ključeva

Dobiveni niz adresa jednak je adresama računa koji su stvoreni putem MetaMask-a. Generiranih 12 riječi omogućuje pristup istom nizu adresa i pripadajućih ključeva bez upotrebe lozinke. Ovaj način koristit će se kasnije u radu za stvaranje pametnog ugovora u Rinkeby testnoj mreži.[40]

5.2 Prevoditelj pametnog ugovora

Standardan način za interakciju s pametnim ugovorom unutar Ethereum ekosustava i izvan *blockchain-a* je putem aplikacijskog binarnog sučelja (engl. *Application Binary Interace*, ABI).[41] ABI i *bytecode* pametnog ugovora dobit će se korištenjem posebnog *npm* (engl. Node Package Manager) paketa Solidity prevoditelja.[42]

[40] URL: https://iancoleman.io/bip39/ (pristupljeno: 4. rujan 2018.)
[41] URL: https://solidity.readthedocs.io/en/develop/abi-spec.html (pristupljeno: 4. rujan 2018.)
[42] URL: https://www.npmjs.com/package/solc (pristupljeno: 4. rujan 2018.)

```
const path = require('path');
const solc = require('solc');
const fs = require('fs-extra');

const buildPath = path.resolve(__dirname, 'build');
fs.removeSync(buildPath);

const carsharingPath = path.resolve(__dirname, 'contracts', 'CarSharing.sol');
const source = fs.readFileSync(carsharingPath, 'utf8');

const output = solc.compile(source, 1).contracts;

fs.ensureDirSync(buildPath);

for (let contract in output) {
  fs.outputJsonSync(
    path.resolve(buildPath, contract.replace(':', '') + '.json'),
    output[contract]
  );
}
```

Slika 35. Skripta za prevođenje pametnog ugovora

Za početak se dohvaćaju svi potrebni paketi i moduli potrebni za prevođenje uključujući i sam Solidity prevoditelj *solc* na način koji je prikazan na slici 35. Nakon toga se osigurava da je mapa *build* obrisana, što se radi u slučaju višestrukog korištenja prevoditelja te brisanja starih datoteka. Pametni ugovor je spremljen kao *CarSharing.sol* te se dohvaća put do njega u odnosu na poziciju skritpe za prevođenje. Od rezultata prevođenja pohranjuje se samo *contracts* obilježje u kojemu se nalazi ABI i *bytecode*.

```
λ node compile.js
{ contracts:
   { ':CarSharing':
      { assembly: [Object],
        bytecode: '608060405234801561001057600080fd5b506000
1000000000000000000000000000000000000000000000000000000000000060
32a5047e914610408578063342 43a5d1461042f57806345c43280146104
8097d146105ca5780638d560159146106315780639063920e1461064857
214610a5c578063b6f0b19a14610a7d578063b7edb4e214610b83578063
10c9f578063f6ffc6d314610cb4578063f79e545014610cd8578063f851
        interface: '[{"constant":true,"inputs":[{"name":"","
me":"firstName","type":"string"},{"name":"lastName","type":"
e":"function"},{"constant":false,"inputs":[{"name":"_driverA
onpayable","type":"function"},{"constant":false,"inputs":[{"
eBrand","outputs":[],"payable":false,"stateMutability":"nonp
_model","type":"string"},{"name":"_registrationPlates","type
ddVehicle","outputs":[],"payable":false,"stateMutability":"n
s"},{"name":"_newOwner","type":"address"}],"name":"editVehic
":false,"inputs":[{"name":"_firstName","type":"string"},{"na
":"nonpayable","type":"function"},{"constant":false,"inputs"
```

Slika 36. Zapis ABI i bytecode-a u obilježju contrcts

Rezultat prevođenja prikazan je na slici 36. gdje je vidljivo obilježje *contracts* koji sadrži *bytecode* i *interface* odnosno ABI koji će se koristiti dalje u radu. Zbog duljine oba obilježja prikazani su samo dijelovi. Skripta za prevođenje na kraju stvara datoteku *CarSharing.json* unutar mape *build*. Naziv *CarSharing* dolazi iz samog pametnog ugovora kao što je vidljivo na slici 35. te se uklanja dvotočje ispred naziva.

5.3 Stvaranje instance pametnog ugovora

Prevođenjem pametnog ugovora stvorena je datoteka *CarSharing.json* koja omogućuje pristup svim dijelovima potrebnim za stvaranje instance pametnog ugovora te slanje instance u testnu mrežu Rinkeby. Koristi se testna mreža kako se ne bi stvarnim novcem plaćale transkacije i stvaranja instance pametnog ugovora te transakcije dodavanja vozača i vozila i slično. Princip stvaranje instance i slanja u glavnu Ethereum mrežu je isti kao i za testnu mrežu koji je prikazan na slici 37.

```
const HDWalletProvider = require('truffle-hdwallet-provider');
const Web3 = require('web3');
const CarShare = require('./build/CarSharing');

const provider = new HDWalletProvider(
  'hole glance glimpse rubber salt picnic marble car very wreck cargo virus',
  'https://rinkeby.infura.io/z9TH3Y9RbANwEIWXdTLk'
);
const web3 = new Web3(provider);

const deploy = async () => {
  const accounts = await web3.eth.getAccounts();

  console.log('Attempting to deploy from account', accounts[0]);

  const result = await new web3.eth.Contract(JSON.parse(CarShare.interface))
    .deploy({ data: '0x' + CarShare.bytecode })
    .send({ from: accounts[0], gas: '3000000' });

  console.log('Contract deployed to ', result.options.address);
};
deploy();
```

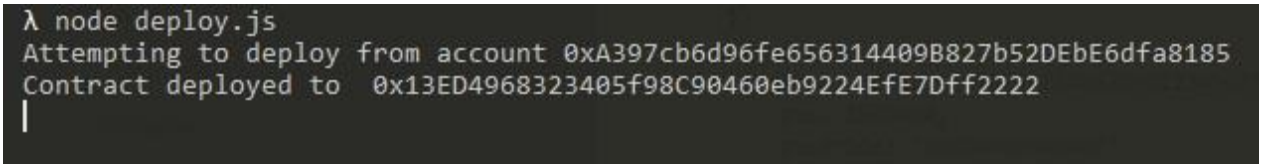
Slika 37. Stvaranje instance pametnog ugovora i slanje instance u testnu mrežu Rinkeby

U skriptu prikazanu na slici 37. prvo se uvoze svi paketi potrebni za stvaranje instance te povezivanje s Rinkeby mrežom. *HDWalletProvider* se koristi za digitalno potpisivanje

transakcija adresa dobivenih iz 12 riječi koje su prethodno objašnjene.[43] Za interakciju s lokalnim i udaljenim Ethereum čvorovima putem HTTP (engl. *Hyper Text Transfer Protocol*) ili IPC (engl. *Inter-procces communication*) konekcije koristi se kolekcija *web3.js*.[44] Kako bi se pojednostavilo povezivanje s Rinkeby mrežom putem Ethereum čvora koristi se usluga Infura. Infura putem ključa omogućuje jednostavno povezivanje sa Infura čvorom koji je ujedno i dio Ethereum mreže, u ovome slučaju Rinkeby mreže.[45]

Nakon uvoženja potrebnih paketa i datoteka deklarira se i postavlja *provider* koji omogućuje spajanje s određenim čvorom. *Provider-u* se omogućuje pristup adresama i pripadajućim setovima ključeva te se onda taj *provider* predaje u novo stvorenu instancu *web3-a*.

Deklarira se funkcija *deploy* unutar koje se dohvaćaju računi koji su dobiveni iz *web3* koje je on dobio iz *provider-a*. Nakon toga se stvara instanca pametnog ugovora pozivanjem *.Contract* opcije u koju se predaje sučelje dobiveno prevođenjem. Dalje u funkciji se poziva metoda *.deploy* u koju se unosi *bytecode* koji će se slati u Rinkeby mrežu. Na kraju se specificira račun s kojega se šalje transakcija te koliko je *gas-a* korisnik spreman platiti za provođenje transkacije. Na slici 38. vidljivo je da je specificiran račun s indeksom 0 *accounts[0]* te da je za provođenje transakcije spreman platiti 3 000 000 *gas*. Rezultat funkcije pohranjuje se u varijabli *result* koja se koristi kako bi se prikazala Ethereum adresu novog stvorene instance pametnog ugovora.[46][47]



Slika 38. Rezultat pokretanja deploy skripte

Prva adresa prikazana na slici 39. odgovara prethodno prikazanoj adresi iz niza adresa na slici 35. To je adresa pošiljatelja ili stvaratelja pametnog ugovora dok je druga adresa prikazana adresa instance kreiranog pametnog ugovora.

[43] URL: https://github.com/trufflesuite/truffle-hdwallet-provider (pristupljeno: 5. rujan 2018.)
[44] URL: https://web3js.readthedocs.io/en/1.0/ (pristupljeno: 5. rujan 2018.)
[45] URL: https://infura.io/docs (pristupljeno: 5. rujan 2018.)
[46] URL: https://web3js.readthedocs.io/en/1.0/web3-eth-contract.html#deploy (pristupljeno: 5. rujan 2018.)
[47] URL: https://web3js.readthedocs.io/en/1.0/web3.html#parameters (pristupljeno: 5. rujan 2018.)

5.4 Postavljanje web3 provider-a za web aplikaciju

U prethodnom potpoglavlju prikazano je postavljanje vrijednosti *web3 provider-a* za potrebe stvaranja instance pametnog ugovora putem *node.js-a*. Za potrebe dohvaćanja i mijenjanja podataka na pametnom ugovoru potrebno je postaviti novu vrijednosti *web3 provider-a*.

Prethodno prikazani *provider* koristio je podatke od administratora za pristup adresi i pripadajućim ključevima za slanje transakcije kojom se stvara instanca pametnog ugovora. Za potrebe web aplikacije potrebno je pristupiti podacima korisnika u internetskom pregledniku. Za stvaranje web aplikacije s više internetskih stranica u ovome radu koristi se *next.js* što ima utjecaj na uvjete postavljanja web3 *provider-a*.

web3.js

```
import Web3 from 'web3';

let web3;

if (typeof window !== 'undefined' && typeof window.web3 !== 'undefined') {
  // We are in the browser and metamask is running
  web3 = new Web3(window.web3.currentProvider);
} else {
  // We are on the server or the user is not running metamask
  const provider = new Web3.providers.HttpProvider(
    'https://rinkeby.infura.io/z9TH3Y9RbANwEIWXdTLk'
  );
  web3 = new Web3(provider);
}

export default web3;
```

Slika 39. Postavljanje web3 provider-a za web aplikaciju

Postavljanje vrijednosti ovisi o mjestu gdje se pokušava podići stranica te je li MetaMask definiran. Slikom 39. prikazana su dva uvjeta za različito postavljanje *web3 provider-a*. Prvi uvjet provjerava da li se stranica pokreće u internetskom pregledniku i je li MetaMask ubacio (engl. *Injected*) *web3* u preglednik. U tome slučaju koristi se ubačeni *provider* od MetaMask-a.

Ukoliko prvi uvjet nije zadovoljen smatra se da se stranicu podiže *next.js* poslužitelj ili internetski preglednik nema ubačen *provider* od MetaMask-a što znači da nema instaliran

dodatak MetaMask. Tada se kreira varijabla *provider* u koju se stavlja *HttpProvider* s pristupom Infura čvoru, putem Infura ključa, te se dodaje u *web3* instancu. Na kraju se izvozi instanca *web3* da bude dostupna svim stranicama koje će ju koristiti.[48]

[48] URL: https://web3js.readthedocs.io/en/1.0/web3.html#value (pristupljeno: 5. rujan 2018.)

6. Web aplikacija za izradu pametnih ugovora na distribuiranoj bazi

U ovome poglavlju prikazati će se izrada web aplikacije koja se bazira na prethodno objašnjenom i prikazanom pametnom ugovoru *CarSharing* te tehnologijama prikazanim u prošlom poglavlju poput *web3* i Metamask. Cilj aplikacije je omogućiti korisnicima da se registriraju kao vozači, registriraju vozilo, pregledavaju listu svih vozila i dostupnih vozila te plate iznajmljivanje vozila. Unutar ovoga rada neće se obraditi cijeli sustav gdje bi web aplikacija i prethodno prikazani pametni ugovor bio samo jedan od dijelova sustava.

Za funkcioniranje cijelog sustava potrebno bi bilo osmisliti generiranje vremenski ovisnih virtualnih ključeva i njihovo dodjeljivanje fizičkom ključu. Primjer toga je NFC kartica (engl. *Near Field Communication*) koji bi se pričvrstila na mobilni uređaj, te bi u sebe pohranjivala Ethereum adresu vozača i virtualni ključ. S druge strane vozilo bi moralo imati takvu ključanicu da može komunicirati s NFC karticom te otključati vozilo nakon pozitivne usporedbe virtualnih ključeva. Potrebno je također da vozilo ima internetsku vezu kako bi moglo sigurno primiti virtualni ključ. Na vozilu bi se mogla nalaziti pasivna NFC oznaka koja bi emitirala Ethereum adresu vozila koju bi mogući korisnici mogli unjeti u aplikaciju te provjeriti dostupnost i cijenu po satu.

Web aplikaciju koja će se prikazati mogao bi nadopunjavati klasičan poslužitelj u smislu prikazivanja podataka koji se vrlo rijetko mijenjaju. Također bi se taj poslužitelj mogao koristiti za pohranjivanje slika vozila te za prikazivanje lokacije vozila ukoliko je vozilo dostupno.

U sljedećim potpoglavljima prikazati će se i objasniti napravljena web aplikacija i njene mogućnosti.

6.1 Početna stranica i stranice s podacima o jednom vozilu i vozaču

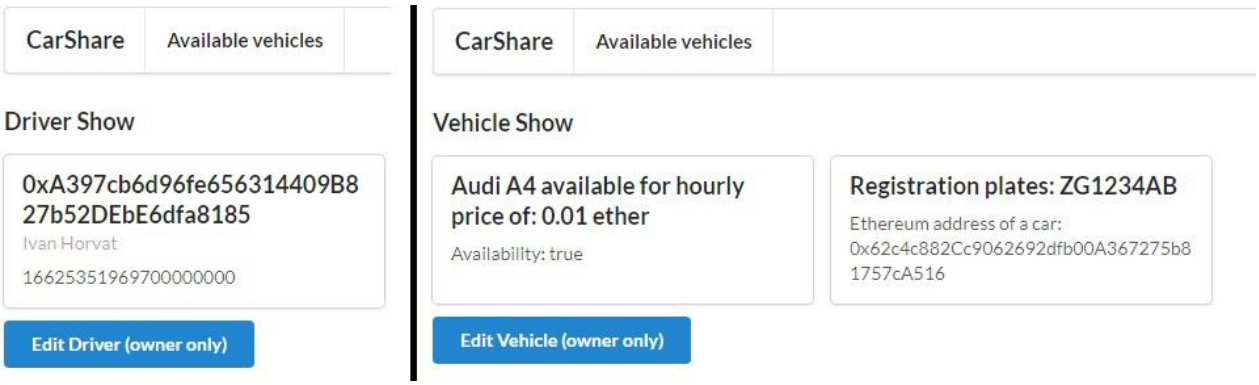
Početna stranica aplikacije prikazuje popis registriranih vozača i vozila te uz njih gumbove za registraciju vozača i registraciju vozila. Na vrhu stranice nalazi se zaglavlje koje se prikazuje na svim stranicama te omogućuje povratak na početnu stranicu te prikazivanje stranica

s popisom svih registriranih i svih dostupnih vozila te popis registriranih vozača kao što je prikazano na slici 40.

Slika 40. Početna stranica aplikacije s prvim podacima

Pritiskom na jedan od gumbova na desnoj strani otvara se stranica za registraciju vozača ili vozila. Također pritiskom na određeni dio zaglavlja otvara se pripadajuća stranica, npr. pristiskom na *Available vehicles* otvara se stranica s popisom dostupnih vozila za iznajmljivanje. Ispod adresa vozila i vozača nalazi se veza za preusmjeravanje koja otvara stranicu i prikazuje sve podatke vezane za pripadajuće vozilo ili vozača. S povećanjem broja registriranih vozila i vozača povećavati će se i liste koje se prikazuju na početnoj stranici.

Pritiskom na *View driver* ispod prve adrese u listi registriranih vozača otvaraju se detalji o vozaču.



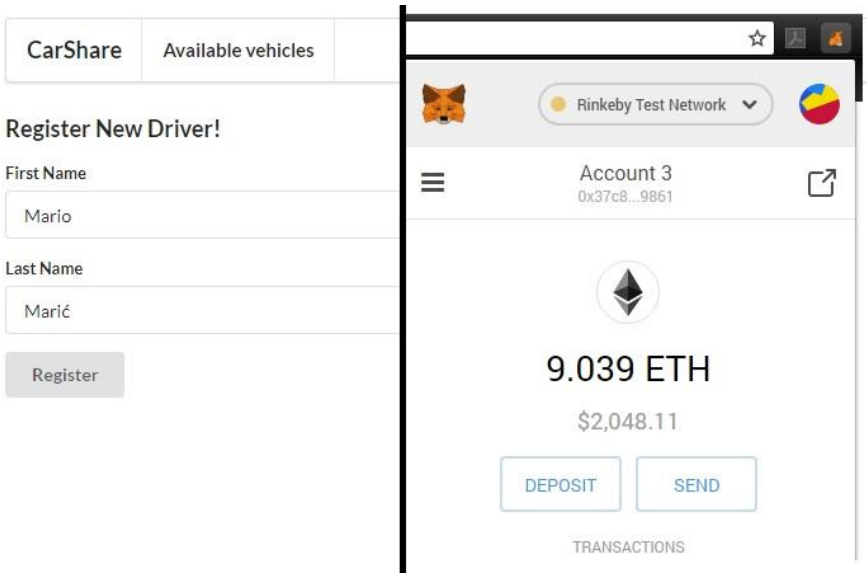
Slika 41. Podaci o registriranom vozilu i vozaču

Slikom 41. prikazani su podaci prvog registriranog vozača i vozila. Na lijevoj strani slike prikazana je Ethereum adresa vozača, njegovo ime i prezime te stanje računa u Wei valuti. Uz to je prikazan i gumb putem kojega se može otvoriti stranica za promjenu podataka o vozaču *Edit Driver*. Funkciju promjene podataka o vozaču može izvršiti samo vlasnik računa. Promjenu podataka o vozilu može izvršiti samo vlasnik, odnosno adresa pošiljatelja mora odgovarati adresi vlasnika koja je upisana prilikom registracije.

Uz podatke o vozilu sa strane se prikazuje forma za iznajmljivanje vozila koja će se prikazati i objasniti nakon registriranja novog vozača i vozila.

6.2 Registriranje novog vozača i vozila

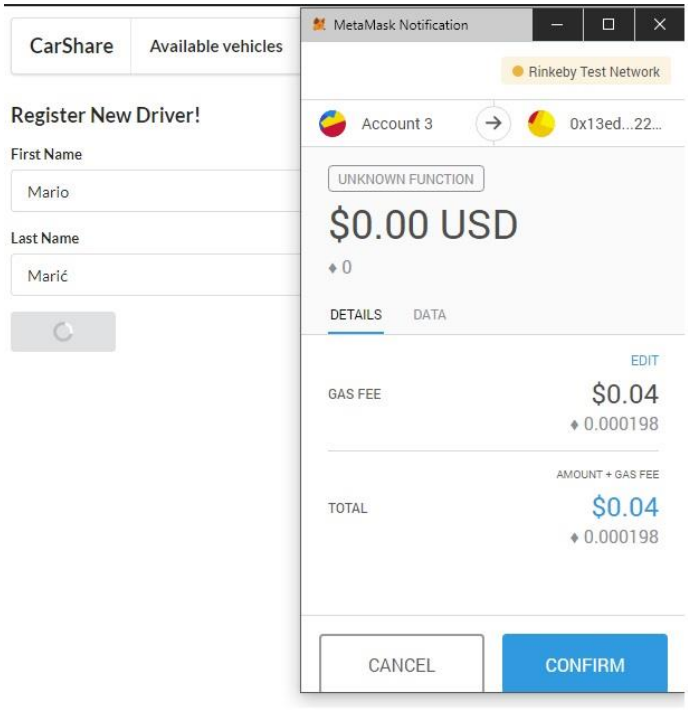
Za registriranje novog vozača i vozila potrebno je koristiti MetaMask dodatak te korisnik mora biti prijavljen u svoj račun koji želi iskoristiti za registraciju. Adresa se ne može koristit za registriranje dva ili više vozača te za vozilo i vozača. Jedna adresa može biti dio samo jednog vozača ili vozila.



Slika 42. Registriranje novog vozača

Na lijevoj strani slike 42. prikazan je ispunjena forma za registraciju vozača dok je na desnoj strani prikazan račun koji se koristi za regitriranje vozača. Pritiskom na gumb *Register* pokreće se funkcija *addDriver*. Za izvršavanje funkcije potrebno je potrošiti malu količinu valute

Ethereum za rudarenje. Tako pritiskom na gumb *Register* iskače prozor MetaMask-a koji traži potvrdu transakcije.

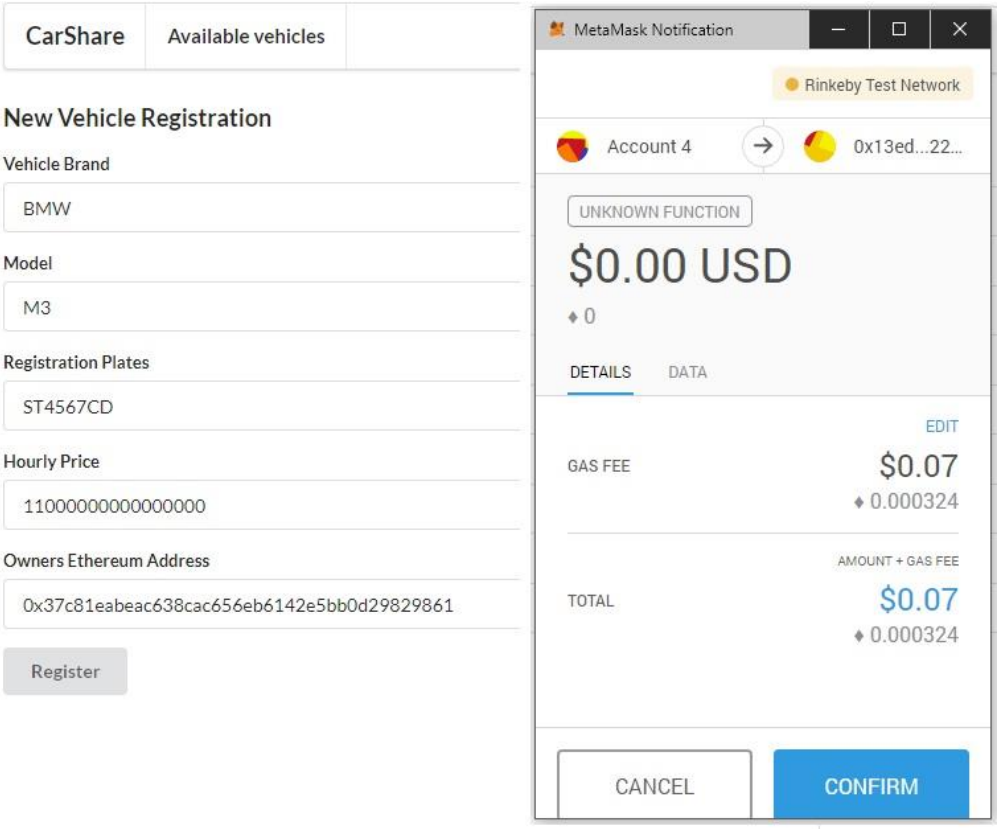


Slika 43. Plaćanje registriranja vozača

U skočnom prozoru prikazanom na slici 43. vidljivo je da se za potvrdu transakcije registracije vozača plaća samo *Gas fee* odnosno naknada za rudarenje. Naknadu za rudarenje MetaMask automatski pretvara iz Ethereum-a u američke dolare te je iznos za ovu transakciju četiri centa. Za potvrdu transakcije čeka se oko 15 sekundi[49] te nakon toga aplikacija preusmjerava korisnika na početnu stranicu s ažuriranim popisima vozača i vozila.

Za registriranje novog vozila potrebno je unjeti više podataka kao što je već prikazano i objašnjeno u poglavlju pametni ugovori. Funkcija registriranja novog vozila također vrši provjeru adrese tako da je potrebno koristiti adresu odnosno račun koji nije prethodno korišten.

[49] URL : https://www.rinkeby.io/#stats (pristupljeno: 6. rujan 2018.)



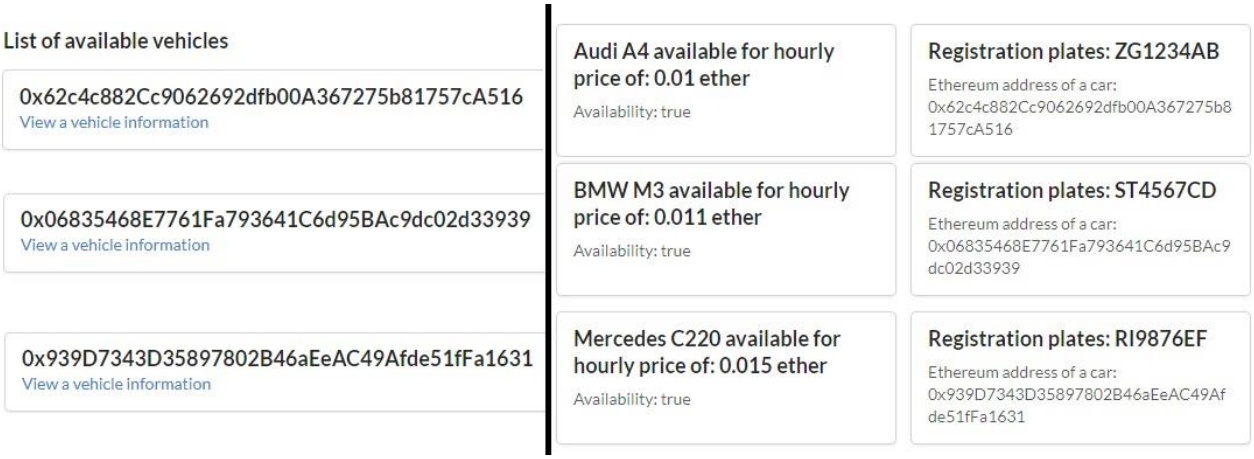
Slika 44. Plaćanje registriranja vozila

Funkcije registriranja novog vozača i vozila funkcioniraju na isti način uz razliku u tome što se za vozilo unosi više podataka. Cijena sata iznajmljivanja vozila ovdje je izražena u Wei. Zbog veće količine podataka veća je i cijena transakcije koja je u ovome slučaju sedam centi američkog dolara. Nakon potvrde transakcije korisnika se vraća na početnu stranicu s ažuriranim popisima. Unutar funkcije *addVehicle* koja je prethodno opisana pokreće se i funkcija koja dodaje vozilo na popis dostupnih vozila *availableVehilces* te se nakon ove funkcije i ažurira popis dostupnih vozila.

6.3 Iznajmljivanje vozila

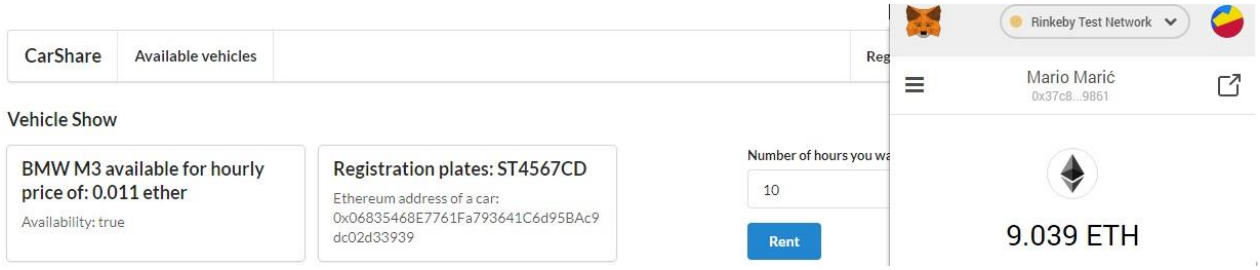
Funkcija iznajmlivanja vozila *rentAVehicle* zahtijeva da je adresa pošiljatelja korištena za registraciju vozača odnosno da je korisnik registriran kao vozač te da ima dovoljna sredstva na računu za platiti iznajmljivanje vozila na određen broj sati.

U ovome potpoglavlju će se iznamljivanje vozila prikazati tako da će se prikazati lista dostupnih vozila prije i poslije iznamljivanja te iznosi po računima vozača i vozila koja će sudjelovati u procesu.



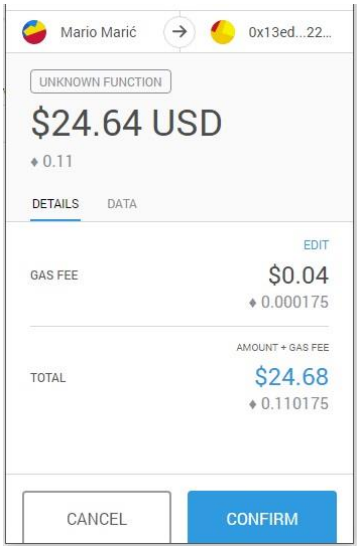
Slika 45. Lista dostupnih vozila

Za potrebe primjera registrirana su tri vozila. Na lijevoj strani slike 45. prikazan je popis dostupnih vozila dok su na desnoj strani pripadajuće stranice s detaljima vozila u kojima je vidljiva marka, model i registracijske oznake vozila, njegova dostupnost i cijena po satu. Otvorivši jedno od vozila klikom na *View a vehicle information* ispod adrese vozila u listi dostupnih vozila otvara se stranica prikazana na slici 46.



Slika 46. Iznajmljivanje vozila

Iz podataka prikazanih na slici 46. vidljivo je da je vozilo slobodno te ga se može iznajmiti. Potrebno je unesti broj sati za koji se želi iznajmiti vozilo te kliknuti na gumb *Rent*. Slikom 46. Prikazan je korisnik Mario Marić koji želi iznajmiti vozilo BMW M3 na 10 sati.



Slika 47. Iznos za iznajmljivanje vozila

Pritiskom na gumb automatski se izračunava iznos koji je potrebno platiti za iznajmljivanje vozila te se na taj iznos dodaje i prethodno objašnjeni *gas fee*. Cijena iznajmljivanja vozila po satu je 0.011 Ether-a i kad se pomnoži s brojem sati dobiva se iznos 0.11 Ether-a prikazan na slici 47. Kada se taj iznos Ether-a prebaci u američke dolare dobiva se iznos od 24.64 američka dolara. Cijena za rudarenje transkacije je 4 centa te onda ukupna cijena transakcije iznosi 24.68 američka dolara. Potrebno kliknuti gumb *Confirm* te pričekati da se transakcija potvrdi.

Nakon potvrde se dostupnost vozila prebacuje u vrijednost *false* te se uklanja s popisa dostupnih vozila. Stanja računa vozača i vozila prije i poslije iznajmljivanja prikazana su na slici 48.

Slika 48. Stanja računa vozača i vozila prije i poslije iznajmljivanja

Na lijevoj stani slike 48 prikazana su stanja računa vozača Maria Marića koji je iznajmio vozilo BMW M3 čija su stanja računa prikazana na desnoj strani slike. Sa slike je vidljivo da se stanje računa vozača umanjilo za 0.11 Ether te da se stanje računa vozilo uvećalo za taj iznos.

Ovim poglavljem prikazani su osnovni primjeri korištenja aplikacije s korisničke perspektive poput registiranja vozača i iznajmljivanja vozila. Za dovršetak aplikacije potrebno bi bilo napraviti poslužitelj u kojeg bi se pohranjivale slike i određeni podaci o vozilima i vozačima koje bi se onda moglo ukloniti s pametnog ugovora i pojednostaviti pametni ugovor.

7. Zaključak

Pametni ugovori i distribuirane baze podataka imaju potencijal za promjenu mnogih područja i ovim radom htjela se prikazati jedna od mogućih primjena. Tehnologije samostalno nisu u mogućnosti promijeniti sustav već je potrebno uskladiti ih s postojećim i dobro razvijenim tehnologijama kako bi se stvorila nova usluga koja se onda može ponuditi korisnicima.

Blockchain omogućava nove načine plaćanja, povjerenja u sustav baziran na kriptografiji i decentralizaciji. S druge strane potrebno bi bilo koristiti klasične poslužitelje za pohranu korisničkih podataka kako bi se primarno osigurala privatnost. Poslužitelji bi se koristili za pohranjivanje svih podataka koji nisu potrebni za provođenje transakcije u blockchain-u.

Time bi se pametni ugovor koristio za identifikaciju korisnika u aplikaciji i izvršavanje naplate. Smanjenjem podataka koij se zapisuju u blockchain smanjuje se i iznos koju je korisnik potreban platiti za potvrđivanje transakcije. Manjim troškovima bi se lakše privuklo više korinsika.

Aplikacija za dijeljenje automobila koristila bi se na nekom ograničenom geografskom području poput grada te bi omogućivala vlasnicima vozila da iznajmljuju i zarađuju iznajmljivanjem vozila. Iz perspektive grada vozila bi manje vremena provela na parkirnom mjestu. Korisnicima odnosno vozačima bi omogućila jednostavno iznajmljivanje vozila na određeni kratki vremenski period te ih možda oslobodila potrebe za kupnjom automobila.

Upotreba ovakve aplikacije mogla bi biti od većeg značaja u budućnosti s dolaskom autonomnih vozila gdje bi se moglo platiti vozilo da dođe na određenu lokaciju te odveze korisnika na odredište. Naplata bi se vršila kao što je prikazano u radu, ali bi autonomna vozila potpuno promijenila uslugu i povećala broj korisnika te riješila određene probleme poput ostavljanja vozila izvan zona korisnika.

Literatura

[1.] Chhanda, R.: *Distributed Database Systems*, Pearson Education Inc., Dorling Kindersley, 2009.

[2.] Singh, S.:Database Systems: Concepts, Design and Applications, Pearson Education Inc., Dorling Kindersley, 2009.

[3.] Peterson, L., Davie, B.: Computer Networks: System Approach – Fifth Edition, Elsevier, SAD, 2012.

[4.] Kurose, J., Ross, K.: Computer Networking: A Top-Down Approach – Sixth Edition, Pearson, SAD, 2013.

[5.] Navathe S, Karlapalem K, Ra M. A mixed fragmentation methodology for initial distributed database design. Journal of Computer and Software Engineering. 1995 Jun;3(4): 395-426.

[6.] Mills, D., Wang, K., Malone, Bernard, Ravi, A., Marquardt, J., Chen, C., Badev, A., Brezinski, T., Fahy, L., Liao, K., Kargenian, V., Ellithorpe, W., Baird, M.: *Distributed ledger technology in payments, clearing and settlement*, Finance and Economics Discussion Series 2016-095, Washington, D.C., 2016.

[7.] URL: http://www.informit.com/articles/article.aspx?p=29583 (pristupljeno: kolovoz 2018.)

[8.] URL: http://www.sqlrelease.com/sql-server-tutorial/dbms-rdbms-and-sql-server (pristupljeno: kolovoz 2018.)

[9.] URL: https://docs.oracle.com/cd/A57673_01/DOC/server/doc/SCN73/ch21.htm (pristupljeno: kolovoz 2018.)

[10.] URL: https://docs.oracle.com/cd/A87860_01/doc/server.817/a76960/ds_conce.htm (pristupljeno: kolovoz 2018.)

[11.] http://www.aspencrypt.com/crypto101_hash.html (pristupljeno: kolovoz 2018.)

[12.] URL: https://www.denimgroup.com/resources/blog/2007/11/properties-of-1/ (pristupljeno: kolovoz 2018.)

[13.] URL: https://nakamotoinstitute.org/bitcoin/ (pristupljeno kolovoz 2018.)

[14.] URL: https://www.gartner.com/it-glossary/blockchain (pristupljeno: 17. kolovoz 2018.)

[15.] URL: https://anders.com/blockchain/coinbase.html (pristupljeno: kolovoz 2018.)

[16.] URL: https://solidity.readthedocs.io/en/v0.4.24/ (pristupljeno: kolovoz 2018.)

[17.] URL: https://solidity.readthedocs.io/en/v0.4.24/layout-of-source-files.html (pristupljeno: kolovoz 2018.)

[18.] URL: https://solidity.readthedocs.io/en/v0.4.24/units-and-global-variables.html#address-related (pristupljeno: kolovoz 2018.)

[19.] URL: https://solidity.readthedocs.io/en/v0.4.24/types.html#value-types (pristupljeno: 18. kolovoz 2018.)

[20.] URL: https://solidity.readthedocs.io/en/v0.4.24/types.html#enums (pristupljeno: kolovoz 2018.)

[21.] URL: https://solidity.readthedocs.io/en/v0.4.24/types.html#reference-types (pristupljeno: kolovoz 2018.)

[22.] URL: https://solidity.readthedocs.io/en/v0.4.24/types.html#arrays (pristupljeno: kolovoz 2018.)

[23.] URL: https://solidity.readthedocs.io/en/v0.4.24/types.html#structs (pristupljeno: kolovoz 2018.)

[24.] URL: https://solidity.readthedocs.io/en/v0.4.24/types.html#mappings (pristupljeno: kolovoz 2018.)

[25.] URL: https://solidity.readthedocs.io/en/v0.4.24/control-structures.html#function-calls (pristupljeno: kolovoz 2018.)

[26.] URL: https://solidity.readthedocs.io/en/v0.4.24/control-structures.html#function-calls (pristupljeno: kolovoz 2018.)

[27.] URL: https://solidity.readthedocs.io/en/v0.4.24/contracts.html#modifiers (pristupljeno: kolovoz 2018.)

[28.] URL: http://ethdocs.org/en/latest/ether.html#gas-and-ether (pristupljeno: kolovoz 2018.)

[29.] URL: http://ethdocs.org/en/latest/ether.html (pristupljeno: kolovoz 2018.)

[30.] URL: https://solidity.readthedocs.io/en/v0.4.24/introduction-to-smart-contracts.html (pristupljeno: rujan 2018.)

[31.] URL: http://unixepoch.com/ (pristupljeno: rujan 2018.)

[32.] URL:https://solidity.readthedocs.io/en/v0.4.24/units-and-global-variables.html (pristupljeno: rujan 2018.)

[33.] URL: https://metamask.io/ (pristupljeno: rujan 2018.)

[34.] URL: https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki (pristupljeno: rujan 2018.)

[35.] URL: https://iancoleman.io/bip39/ (pristupljeno: rujan 2018.)

[36.] URL: https://solidity.readthedocs.io/en/develop/abi-spec.html (pristupljeno: rujan 2018.)

[37.] URL: https://www.npmjs.com/package/solc (pristupljeno: rujan 2018.)

[38.] URL: https://github.com/trufflesuite/truffle-hdwallet-provider (pristupljeno: rujan 2018.)

[39.] URL: https://web3js.readthedocs.io/en/1.0/ (pristupljeno: rujan 2018.)

[40.] URL: https://infura.io/docs (pristupljeno: rujan 2018.)

[41.] URL: https://web3js.readthedocs.io/en/1.0/web3-eth-contract.html#deploy (pristupljeno: rujan 2018.)

[42.] URL: https://web3js.readthedocs.io/en/1.0/web3.html#parameters (pristupljeno: rujan 2018.)

[43.] URL: https://web3js.readthedocs.io/en/1.0/web3.html#value (pristupljeno: rujan 2018.)

[44.] URL : https://www.rinkeby.io/#stats (pristupljeno: rujan 2018.)

Popis kratica

DBMS – Database Managment System

SQL – Structured Query Language

ISO – International Organization for Standardization

DLT – Distibuted Ledger Technology

P2P – Peer – to – Peer Network

ABI – Application Binary Interface

HTTP – Hypertext Transfer Protocol

IPC – Interprocess Communication

NFC – Near-field Communication

NPM – Node Package Manager

SOLC – Solidity Compiler

Popis slika

Popis tablica

Sveučilište u Zagrebu
Fakultet prometnih znanosti
10000 Zagreb
Vukelićeva 4

IZJAVA O AKADEMSKOJ ČESTITOSTI I SUGLASNOST

Izjavljujem i svojim potpisom potvrđujem kako je ovaj diplomski rad isključivo rezultat mog vlastitog rada koji se temelji na mojim istraživanjima i oslanja se na objavljenu literaturu što pokazuju korištene bilješke i bibliografija.

Izjavljujem kako nijedan dio rada nije napisan na nedozvoljen način, niti je prepisan iz necitiranog rada, te nijedan dio rada ne krši bilo čija autorska prava.

Izjavljujem također, kako nijedan dio rada nije iskorišten za bilo koji drugi rad u bilo kojoj drugoj visokoškolskoj, znanstvenoj ili obrazovnoj ustanovi.

Svojim potpisom potvrđujem i dajem suglasnost za javnu objavu diplomskog rada pod naslovom **Primjena distribuiranih baza podataka i pametnih ugovora u sustavima elektroničkog poslovanja** na internetskim stranicama i repozitoriju Fakulteta prometnih znanosti, Digitalnom akademskom repozitoriju (DAR) pri Nacionalnoj i sveučilišnoj knjižnici u Zagrebu.

Student/ica:

U Zagrebu, 17.9.2018

(potpis)