

**SVEUČILIŠTE U ZAGREBU**  
**PRIRODOSLOVNO-MATEMATIČKI FAKULTET**  
**MATEMATIČKI ODSJEK**

Filip Maček

**PAMETNI UGOVORI POMOĆU BLOCKCHAIN TEHNOLOGIJE**

Diplomski rad

Voditelj rada:

prof.dr.sc. Luka Grubišić

Zagreb, kolovoz 2020.

# Sadržaj

Sadržaj

Uvod	1
<b>1. Uvod u blockchain</b>	<b>3</b>
1.1 Uvod .....	
1.2 Povijest .....	
1.2 Osnovne karakteristike .....	
1.3 Ethereum .....	
<b>2. Pametni ugovori</b>	
2.1 Uvod .....	
2.2 Oracle problem .....	
2.3 Opis projekta i pametnog ugovora .....	
2.4 Implementaciju pametnog ugovora .... ..	
<b>3. Mobilna i web aplikacija</b>	
3.1 Uvod .....	
3.2 Tehnički opis aplikacija i korištenih tehnologija. ....	
3.3 Pregled Dapp web aplikacije .....	
3.4 Pregled mobilne android aplikacije .....	
<b>4. Chainlink čvor</b>	
4.1 Uvod .....	
4.2 Opis tehnologija i proces .....	
4.3 Implementacija .....	

# Uvod

Pametni ugovori naziv su za bilo kakav kompjutorski program ili transakcijski protokol koji automatski izvršava i kontrolira izvršavanje nekog ugovora i svih njegovih dijelova. Ideja je da su uvjeti ugovora kao i njegova etape izvršavanja potpuno prepušteni kompjutorskom kodu.

Jedan važna odluka kod pametnih ugovora jest gdje će se taj program tj. ugovor izvršavati . Većina današnjih pametnih ugovora se izvršavaju na centralnim računalima onog jednog od sudionika tog ugovora. Ali idealni scenarij bi bio da se ugovor izvršava na serverima na kojima nitko od sudionika nema kontrola, a svi sudionici mogu vjerovati da se ugovor izvršiti po unaprijed određenim pravilima.

U prošlom poglavlju „Uvod u blockchain” uspjeli smo identificirati blockchain kao sigurni i decentraliziranu mrežu za procesiranje i spremanje podataka te se on nameće kao jedno od rješenja gdje se pametni ugovori mogu izvršavati.

Ethereum je jedan od najvećih blockchain platformi za izvršavanje pametnih ugovora, i naša primjer pametnog ugovora će se tamo izvršavati.

Pametni ugovori nude mnoge prednosti nad današnjim digitalnim ugovorima

Glavna korisnost pametnih ugovora je u tome što se oni ne ovise o nikakvim sigurnim posrednicima ili arbitražama i svim troškovima koje takvo posredstvo nosi. Svi ostali problemi i prevare koji mogu nastati u izvršavanju ugovora također su izostavljeni jer se podrazumijeva da je kod tog ugovora napravljen u cilju obuhvaćanja svih mogućih događaja u stvarnom svijetu.

U ovom diplomskom radu pokušat ćemo pokazati jednu moguću implementaciju i korisnost pametnih ugovora u stvarnom svijetu te ćemo pokušati pokazati kako se takva implementacija može natjecati sa sadašnjim modelom izvršavanja tih ugovora gdje velike korporacije ... TODO

Model i interakcija sa pametnim ugovorima na blockchainu je drugačiji nego današnje aplikacije koje u većini slučajeva komuniciraju sa svojim centralnim serverom pomoću API-ja (engl. Application programming interface).

Naša aplikacija je decentralizirana jer ne komunicira sa centralnim našim serverom nego se izvršava na svim kompjuterima na Ethereum blockchainu. Time naša aplikacija postaje „Dapp”, naziv koji se koristi za aplikacije tj. pametne ugovore koji se izvršavaju na Ethereum.

--- KRATAK OPIS APLIKACIJE I SVIH NJENIH DIJELOVA

# **Poglavlje 1**

# Uvod u blockchaina

## 1.1 Uvod

Kroz ovo poglavlje ukratko ćemo opisati glavne karakteristike i dizajn blockchain kao i njegovu povijest. Također značajna će nam biti i njegova motivacija i kakve pogodnosti blockchain donosi.

## 1.2 Povijest

Nakon izuma računala 1980-tih ljudi su se počeli pitati kakve su sve još inovacije moguće pomoću tog stroja. Ljudi još nisu shvaćali sve mogućnosti koje će im taj stroj omogućiti u budućnosti. Tada su još to bila izolirana računala koja su slabo bila povezana. Ali i već tada su neki istaknuti stručnjaci počeli uviđati kako da povežu ta računala u jedan koherentan sustav ili mrežu. *David Chaum* je bio pionir u tom području, i on je već 1982. u svojoj doktorskoj disertaciji [1] počeo razmišljati kako organizacije koje međusobno ne vjeruju jedna drugoj, mogu zajedničkim snagama izgraditi i održavati siguran računalni sistem kojem svi mogu vjerovati. Uvidio je veliku potrebu u privatnom i javnom sektoru za takve sisteme. Kriptografskim tehnikama bilo bi moguće takav sistem ostvariti praktičnim, gdje bi se spremljeni i podaci u opticaju mogli biti zaštićeni mehanizmom sefa (*eng. vault*) Kasnije, 1991. *Stuart Haber* i *W. Scot. Sornetta* iznose prvi dizajn kriptografski osiguranog lanca blokova u kojem se ne može manipulirati vremenskim oznakama. Godinu dana nakon u svoj dizajn unose novu inovaciju binarno hash stablo (*eng. Merkle Tree*). Ono je omogućilo jednostavnu i sigurnu verifikaciju sadržaja velikih struktura podataka. Također je omogućilo da više certifikata dokumenata budu uključeni u blokove.

## Bitcoin

Iako je bilo mnoštvo akademski radove i napretka u kriptografiji do 2008. nije postojala nikakva veća i značajnija implementacija blockchain u svijetu. Tada je

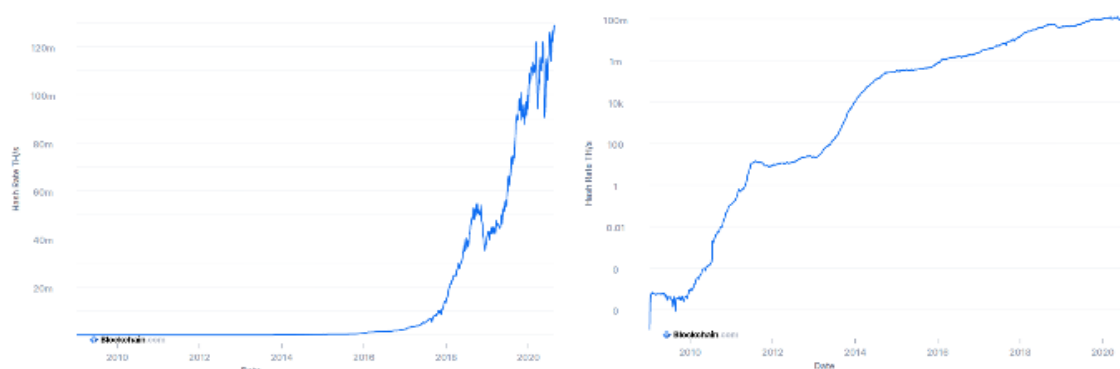
u jeku najveće financijske krize skupina ljudi ili jedan čovjek (što do danas nije poznato) pod pseudonimom Satoshi Nakamoto objavljuje članak eng. *Bitcoin: A Peer-to-Peer Electronic Cash System*. [2] U njemu iznosi osnovne teze i potrebe za neovisnim i decentraliziranim novčanim sustavom:

- Potpuno neovisan elektronički novac bi omogućio direktno slanje novca od jednog sudionika do drugog bez prisustva neke financijske institucija.
- Takav sustav koji se temelji na povjerenju u jedan centralni entitet ima svoje nedostatke. Neke od njih su mogućnost poništavanja transakcija i blokiranja računa sudionika te time izgon iz financijskog sustava i nemogućnosti sudjelovanja u financijskim uslugama.
- Transakcijski troškovi su povećani prisustvom posredovatelja

U tom članku je preporučeno elektronički naplatni sustav koji se temelji na kriptografiji, a ne na povjerenju. Dopuštajući da bilo koja dva sudionika sudjeluju u financijskoj transakciji bez potrebe da vjeruju jedan drugome.

Problem dvostruke potrošnje ( eng. *double-spending problem* ) već potrošenog novca riješen je na način da se umrežena računala na Bitcoin mreži slože oko toga da postoji jedan jedan opći vremenski lanac gdje bi se generirao kriptografski dokaz o kronološkom poretku transakcija.

Takav sustav je siguran dok većina poštenu čvorova na Bitcoin mreži kontrolira više procesorske moći nego napadačka skupina čvorova. Takav sustav je robustan u svojoj nestrukturiranoj jednostavnosti



Slika 1.1: Hash rate Bitcoin mreže kroz vrijeme (linearna, logaritamska skala)

Na slici 1.1

Iako u početku odbačen i ne hvaljen od strane akademika i finansijskih institucija koji su u njemu vidjeli pokušaj promijene starih vrijednosti. Mreža je aktivna i rastuća, te bez prestanka niti kakvih smetnji pouzdano izvršava svoju funkciju.

Kasnije 2014. godine izlazi članak i dizajn novog tipa blockchaina po imenu Ethereum. On je naslijedio opću funkcionalnost slanja novčane valute od Bitcoin protokola, ali je u svojoj implementaciji dodao još jednu jako važno novinu. A to je da čvorovi na Ethereum mreži još mogu i izvršavati posebno napisanu programsku logiku u programskom jeziku nazvanom Solidity. Time je otvoren put pametnim ugovorima te je i svaki takav programski isječak na Ethereumu i prozvan *eng. Smart Contract* . U budućim poglavljima ćemo pobliže opisati unutrašnji dizajn Ethreumu, osnovne značajke programskog jezika Solidity te kako je moguće da se programski kod paralelno izvršava na svim računalima.

## 1.3 Osnovne karakteristike

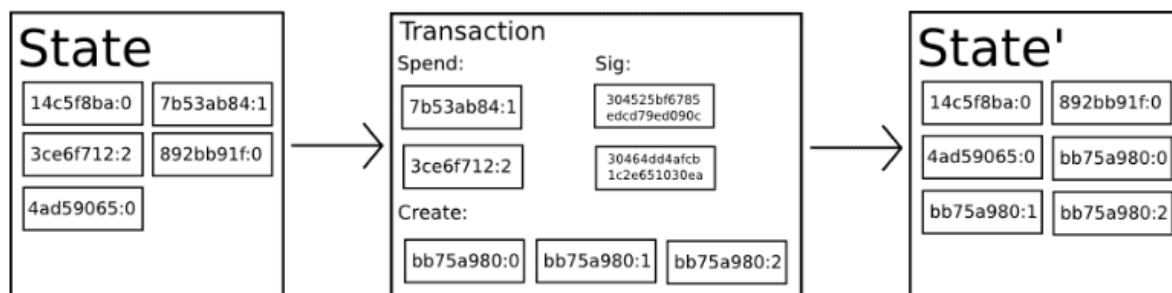


U ovom poglavlju ćemo ukratko opisati glavne dijelove i karakteristike blockchaina fokusirajući se na inicijalni dizajn Bitcoin protokola.

Njegov doprinos i značaj je ogroman jer je istovremeno riješio dva velika problema:

1. Omogućio je jednostavan i efektivan konsenzus algoritam, dajući računalima (*eng. nodes*) na mreži svojstvo da se kolektivno slože oko trenutnog stanje Bitcoinove knjige salda (*eng. ledger*) te mehanizma kako će se ona mijenjati.
2. Riješio je veliki implementacijski problem tko odlučuje o konsenzusu na mreži, i time dao mogućnost jednostavnog ulaska i sudjelovanja novih računala na mreži.

Sa tehničkog stajališta, knjiga salda elektronske valute Bitcoin, je zapravo sistem promijene stanja (*eng. „state transition system”*) gdje se stanje vlasništvo na elektronskom valutom te funkcija promijene stanja (*eng. „state transition function”*).



Slika 1.2: Bitcoin blockchain kao sustav promijene stanja

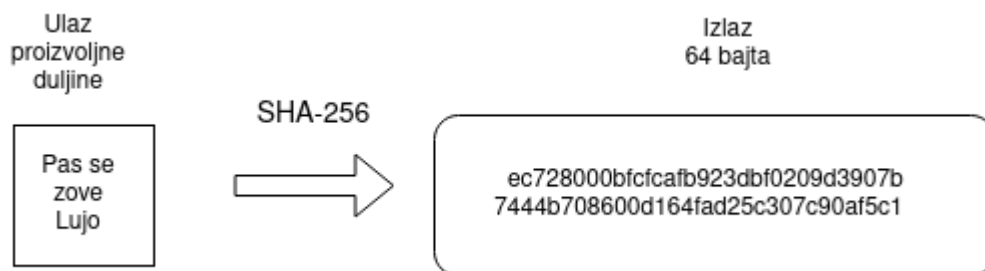
Sada ćemo opisati sve dijelove za implementaciju ovakvog sustav.

## Kriptografska hash funkcija

U pozadini blockchaina je posebna klasa kriptografske funkcije koji ima svojstvo da je ulazni nezavisna varijabla proizvoljne veličine, a rezultat te funkcije je fiksne veličine. Takva funkcija je jednosmjerna te joj se ne može izračunati inverz.

Time je jedini mogući algoritam za pronalaska ulaznog parametra za zadanu vrijednost funkciji, onaj u kojem se eng. brute-force algoritmom provjeravaju sve vrijednosti. U ovom slučaju koristi je posebna hash funkcija imenom

SHA256 koja vraća 256 bitni broj koji je prikazan kao heksadecimalni broj sa 64 znamenke radi lakše čitljivosti.



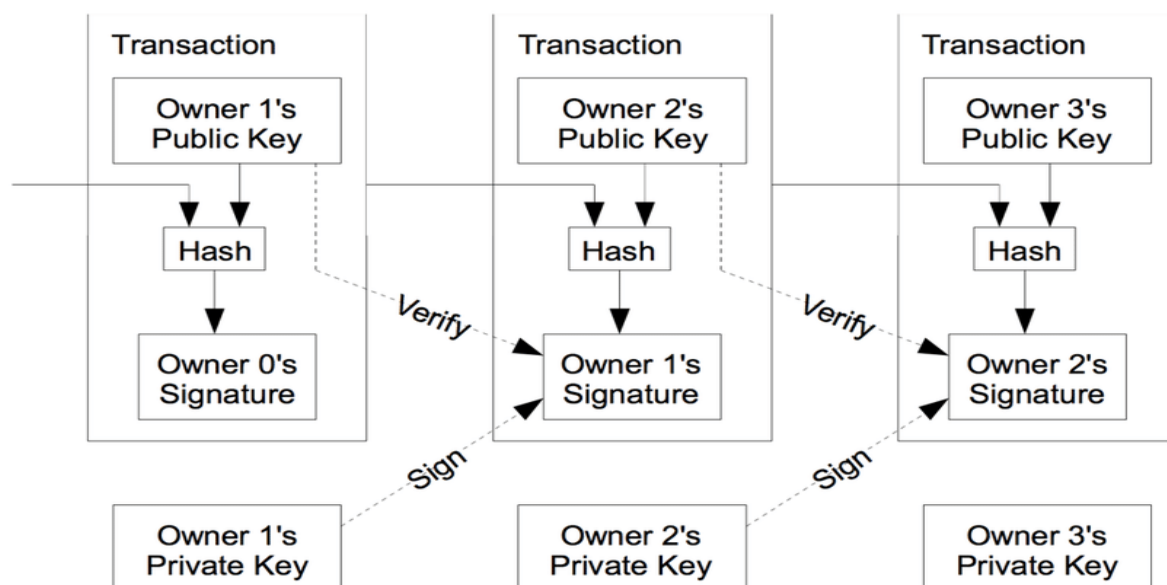
Slika 1.3 SHA-256 hash funkcija

## Transakcije

Na blockchainu elektronska valuta se definira kao lanac digitalnih kriptografski potpisa. Svaki sudionika transferira željenu količinu digitalnog novca koji on posjeduje tako da pomoću svog privatnog ključa potpisuje skup koji sadrži sljedeće informacije:

- Hash posljednje vlastite transakcije
- Javnih ključ primatelja te digitalne transakcije

Transakcije nisu privatne, već javne. Time je omogućeno da se svaka transakcija vidi i provjeri je li ispravna.



Slika 1.4: Dizajn Bitcoin transakcije

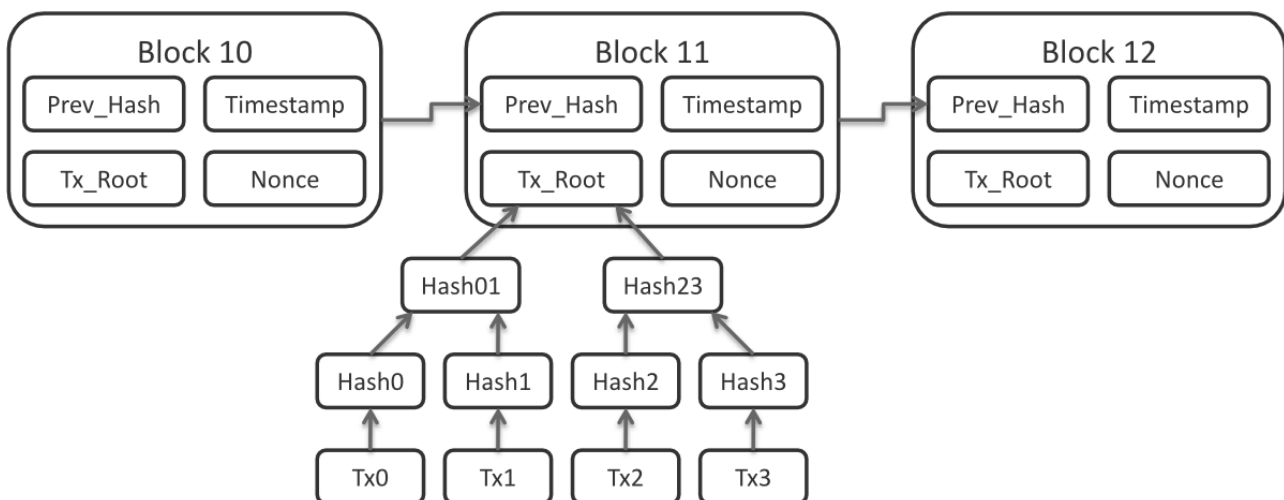
Svaki vlasnik elektronske valute transferira željenu vrijednost tako da digitalnog potpisuje hash prijašnje transakcije i javni ključ primatelja kojemu želi poslati te digitalne novce. Da bi primatelj potvrdio vlasništvo, on jednostavno uz pomoć javnog ključa pošiljatelja verificira digitalni potpis.

Ovim dizajnom se postiže da digitalna valuta nije fiksni iznos koji svaki sudionika ima (vrijednost na računu), već se sistem temelji na prijenosu prava sa prijašnjeg vlasnika na budućeg koji onda ima pravo potpisati i prenijeti ta prava gdje god on želi.

## Blokovi

U prijašnjem objašnjenju o transakcijama na blockchainu, postoji problem dvostruke potrošnje (*eng. double-spending*). Primatelj digitalne valute nemože sa sigurnošću ustvrditi da pošiljatelj već negdje druge nije potrošio tu količinu valute. Jedan od rješenja tog problema bi bio stvaranjem jednog centralnog entiteta koju bi svi vjerovali, te bi on provjeravao da transakciju već nisu negdje drugdje potrošene. Ali time bi se potkopala cijela ideja decentralizacije i neovisnosti.

Umjesto toga implementiran je dizajn lanca blokova gdje se svi elementi blok *hashiraju* te se vremenski obilježavaju te se propagiraju kroz cijelu mrežu.



Slika 1.5: Povezivanje blokova u lanac

Trenutni blok se puni nepotvrđenim transakcijama koji se onda uz pomoć **binarnog hash stabla** povezuju i stvara se krajnji hash ( *Tx\_Root* ) svih transakcija. Time se postiže da su transakcije vremenski obilježene te da su morale postojati inače ne bi mogle biti unesene u hash bloka.

## Algoritam konsenzusa

Da bi se uspio implementirati ovakav distribuiran vremenski lanac blokova na većem broju računala (*eng. peer-to-peer*), potrebno je utvrditi mehanizam kako će sva računala koja održavaju mrežu složiti oko toga koji je glavni lanac i kako se on povezuje. Implementiran je algoritam imena *eng. Proof-of-Work*.

On je prenio moć odlučivanja sa formalnog demokratskog odlučivanja, gdje je jedinstveno registriran entitet ima pravo na jedan glas (*eng. one-IP-address-one-vote*) na ekonomsku barijeru, gdje je moć jednog računala na mreži u direktnoj proporciji na procesorsku snagu koju on donosi na mrežu (*eng. one-CPU-one-vote*).

Time je riješeno pitanje odabira glavnog lanca oko kojeg će se većina računala na mreži složiti, a to je onaj koji je najduži. Na njegovo stvaranje je potrošeno najviše procesorske moći, i uz njegov odabir kao glavnog lanca postiže se i najveća sigurnost protokola. Ako je većina CPU snage u poštenim računalima na mreži, tada će taj pošten lanac raste najviše i premašiti ostale po veličini.

Taj algoritam je implementiran tako da se uvećava *engl. Nonce* u zaglavlju bloka dok hash tog bloka nebude manji od unaprijed zadane složenosti. Time se želio postići da je prosječno vrijeme izrade bloka (*eng mining*) oko 10 minuta.

Kako se mijenja procesorska moći cijele mreže, tako se i mijenja razina složenosti da bi se postiglo prosječno vrijeme izrade bloka. Mreža svakih 2016 blokova provjerava koliko je bilo prosječno vrijeme izrade blokova. Ako je manje od ciljanje 10-minutne, onda se složenost povećava tj. hash bloka će morati biti još manje tj. morat će imati više vodećih nula.

## Scripting

U pozadini cijelog protokola je i jednostavan programski jezik na stogu (*eng. stack*) koji služi kao scripting sustav za transakcije. „Script” je zapravo lista instrukcija upisana u svaku transakciju koji opisuje kako će sljedeća osoba dobiti pristup svojem elektroničkom novcu. Provjera i izvršavanje ovog jednostavnog jezika je povjerena čvorovima na mreži, koji je izvršavaju i upisuju ispravne transakcije u novi blok, tj. transakcije u kojima se skripta uspješno izvršila. Koliko god ovaj sistem bio jednostavan i efikasan u održavanju Bitcoin mreže i protokola, on ima i neke svoje nedostatke:

- **Izostanak Turingove konačnosti**

Iako sadrži veliki skup operacija koje se mogu izvršiti, taj programski jezik ne podržava petlje. To je implementirano sa svrhom da se izbjegnu beskonačno izvršavanje petlji dok se verificiraju transakcije.

- **Nemogućnost pamćenja vrijednosti**

Ne postoji mogućnost da se tim transakcijskim skriptama da veća kontrola nad isplatama i iznosima transakcija. Time se efektivno ne mogu izvršavati nikakve složenije transakcije koji imaju neke uvjete.

- **Nepostojanje stanja**

Sve nepotrošene transakcijske vrijednosti ( *eng UTXO – Unspent transaction output*) su ili potrošeni ili nepotrošeni. Nema nikakve šanse da one ulaze u neke višestruke ugovore ili skripte koji čuvaju unutrašnje stanje

- **Blockchain slijepoća**

UTXO ostaju slijepe na ostale podatke u blockchainu kao što su nonce, vrijeme ili prijašnjih blok hash. Time je scripting programski jezik višestruko limitiran i nedostaju mu važni izvori slučajnosti.

## 1.3 Ethereum

Zbog sve već navedenih nedostataka, pojavila se potreba za drugačijom i sofisticiranijim tipom blockchaine koji bi uspijeva izvršavati i kompliciraniju programsku logiku nego samo slanje elektroničke valute. U tom cilju razvio se Ethereum koji je imao cilj sagraditi alternativni tip blockchaine koji ne kompromitira ekonomski model i sigurnost koju nudi blockchain.

Implementiran je na ovim idejama:

- **Jednostavnost**

Ethereumov protokol treba biti što jednostavniji za korištenje i interakciju sa njim. Svaka optimizacija koja dodaje neku količinu kompleksnosti treba bi biti izbjegnuta jedino ako daje vrlo veliku korisnost.

- **Univerzalnost**

Fundamentalni dizajn Ethereuma je da nudi potpunu slobodu pri izradi

aplikacija time što nudi interni Turing-konačan jezik imena *Solidity* što nudi programeru opću slobodu u izradi svakog tipa pametnog ugovora i svih operacija koje je moguće matematički izvršiti

- **Modularnost**

Dijelove i implementacije Ethereum protokola trebali bi biti modularni i neovisni te bi trebali biti implementirani kao zasebni moduli koje i drugi protokoli mogu koristiti.

- **Svestranost (Agility)**

Detalji protokola nisu fiksirani, i svako poboljšanje koje pridonosi skalabilnijem i sigurnijem radu su dobrodošli i bit će detaljno razmotreni

- **Bez cenzure i diskriminacije**

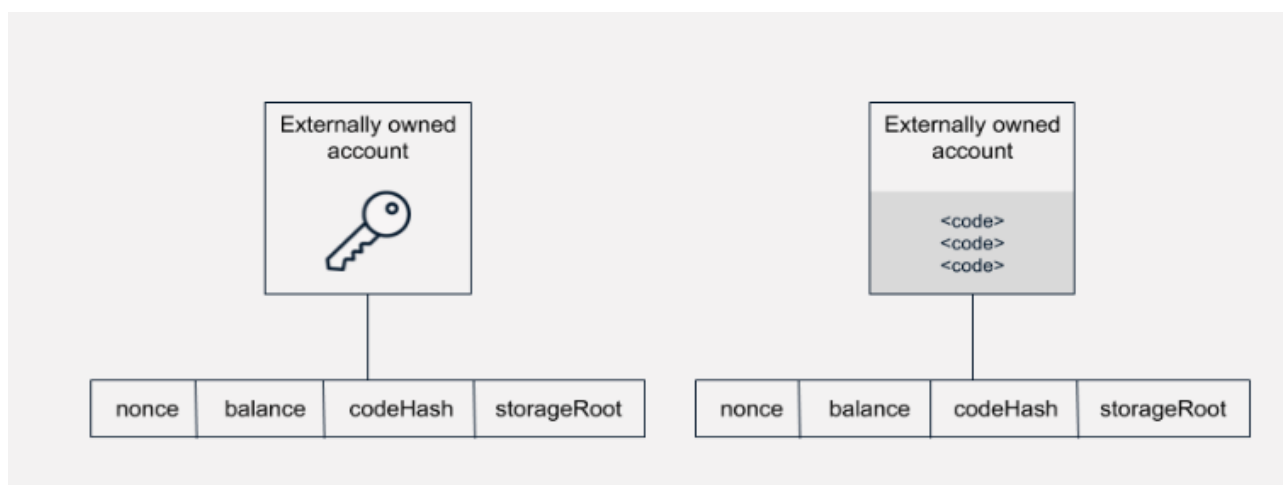
Protokol ne bi smio nikako cenzurirati ili diskriminirati specifične kategorije upotrebe ili sudionike

## Tip računa (account)

Na Ethereum protokolu stanje sustava je prikazano uz pomoć objekata pod imenom *eng. „accounts”*. Svaki od njih ima svoju osobnu 20-bajtnu adresu, a funkcija promijene stanja (*eng. state transition function*) između njih je direktan transfer vrijednosti i informacija. Digitalna valuta na Ethereumu je Ether(ETH)

Postoje dva tipa računa na Ethereum protokolu

1. *Externally owned accounts* – kontrolirani uz privatni ključ
2. *Contract Accounts* – kontroliran od strane koda pametnog ugovora



Slika 1.7: Tip računa na Ethereum protokolu

Također svaki račun na Etheremu sadrži 4 važna polja:

- **Nonce** : brojač, koji služi da transakcije budu procesirane samo jednom
- **Stanje računa**: količina digitalne valute Ether koju račun posjeduje
- **Kod programa**: ako postoji
- **Memorija računa**: početna vrijednost je prazna

Razlika u ta dva tipa računa je također u tome što *externally owned account* nema nikakvog programskog koda, s toga računa su mogu slati i primati poruke potpisivanjem transakcija. S druge strane, *contract account* ima drugačija svojstva te on na svaku primljenu poruku aktivira određeni dio koda, koja onda onda izvršava sve potrebne operacije npr. zapisivanja u internu memoriju, slanje poruka ostalim računima ili kreiranje novih pametnih ugovora.

Takvim dizajnom se postiže to da pametni ugovora na Ethereumu ne bi trebali biti viđeni kao fiksne strukture koje bi trebali biti izvršene ili s kojima je potrebno surađivati. Već više kao autonomne agente koji žive unutar Ethereum protokola i čekaju na poruke ili transakcije, imaju direktnu kontrolu nad stanjem računa i svim unutrašnjim varijablama.

## Transakcije i poruke

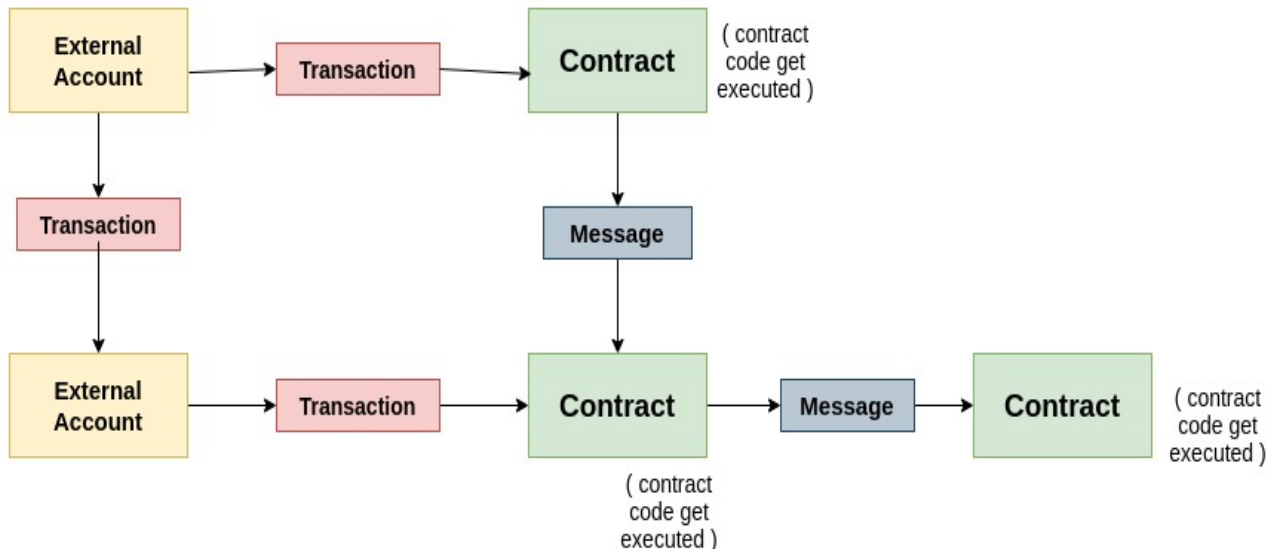
Uz različite tipova računa, na Ethereum blockchainu postoje i različiti tipovi poruku koje sudionici ili pametnih ugovori mogu koristiti.

### 1. Transakcije

Skup podataka potpisan od strane vanjskog agenta (*eng. External Actor*). Može reprezentirati poruku ili novi autonomni objekt, tj. novostvoreni pametni ugovor. Transakcije su eksplicitno zapisane na blockchain.

### 2. Poruke (*eng. Messages*)

Pametni ugovori imaju mogućnost slati poruke jedan drugome. Poruke su u tom slučaju virtualni objekti koji nisu direktno upisani u blockchain već postoje jedino u Ethereumovom unutrašnjem okruženju (*eng. Ethereum execution environment*) i tamo se izvršavaju. Ukratko poruka je kao transakcija, ali je napravljena od strane pametnog ugovora, a ne vanjskog sudionika.



Slika 1.8: Poruke i transakcije

Svaka transakcije upisana u blok sadrži poruku, ali poruke koje šalju pametni ugovori nisu upisani u blok kao podaci, već su implicitno uključene u izvršavanje pametnog ugovora.

Formalno, kod pametnih ugovora je napisan u program jeziku niže razine te pretvoren u *eng. bytecode* koji onda Ethereumov virtualni kompjuter (*eng. Ethereum virtual machine -EVM*) izvršava dio po dio uz pomoć stoga.

Value	Mnemonic	$\delta$	$\alpha$	Description
0x00	STOP	0	0	Halts execution.
0x01	ADD	2	1	Addition operation. $\mu'_s[0] \equiv \mu_s[0] + \mu_s[1]$
0x02	MUL	2	1	Multiplication operation. $\mu'_s[0] \equiv \mu_s[0] \times \mu_s[1]$
0x03	SUB	2	1	Subtraction operation. $\mu'_s[0] \equiv \mu_s[0] - \mu_s[1]$
0x04	DIV	2	1	Integer division operation. $\mu'_s[0] \equiv \begin{cases} 0 & \text{if } \mu_s[1] = 0 \\ \lfloor \mu_s[0] \div \mu_s[1] \rfloor & \text{otherwise} \end{cases}$

Slika 1.9: Primjer nekih operacija i njihovih kodnih vrijednost

Taj programski jezik niže razine i sve njegove operacije pridonose raznolikosti operacija koje je moguće ukodirati u svoj pametni ugovor.

## Transakcijski troškovi



Osnovni jedinični računalni korak naziva se *eng. GAS* te svaka *bytdecode* operacija na EVM sadrži neku količinu koraka za izvršavanje, i oni u izraženi u količini *GAS* a potrebnih za njegovo izvršavanje.

U svakoj transakciji postoje dva polja koja pobliže označavaju koliko najviše je platitelj računalnih koraka spreman platiti i po kojoj jediničnoj cijeni za osnovnu transakciju:

## **1. STARTGAS**

Najveći broj računalnih koraka kojih je moguće izvršiti

## **2. GASPRICE**

Naknada koju pošiljatelj transakcije plaća po računalnom koraku (*eng. GAS*)

Ovim ekonomski mehanizmom postiže se korisnost koji računala(čvorovi) na Ethremovom blockchainu dobivaju za održavanje mreže i protkola.

U analogiji s automobilom, GAS predstavlja količinu goriva koja je potrebna, a GASPRICE jediničnu cijenu po litri, a umnožak oba je zapravo kraljni i ukupni trošak.

Problem je u tome što računala na Ethereum mreži koja rudare blokove i izvršavaju kod u transakcijama i porukama pametnih ugovora ne mogu ,unaprijed prije izvršavanje koda, znati hoće li biti u mogućnosti izvršiti za zadane resurse koji su primili u transakcije (*eng. halting problem*). Ukoliko neće računalo (*eng. miner*) ,koje je izvršavalo taj kod, će zadržati sve Ethere, te diskvalificirati transakciju podnoseći engl. „*Out of Gas Exception*”

## **Poglavlje 2**

### **Pametni ugovori**

#### **2.1 Uvod**

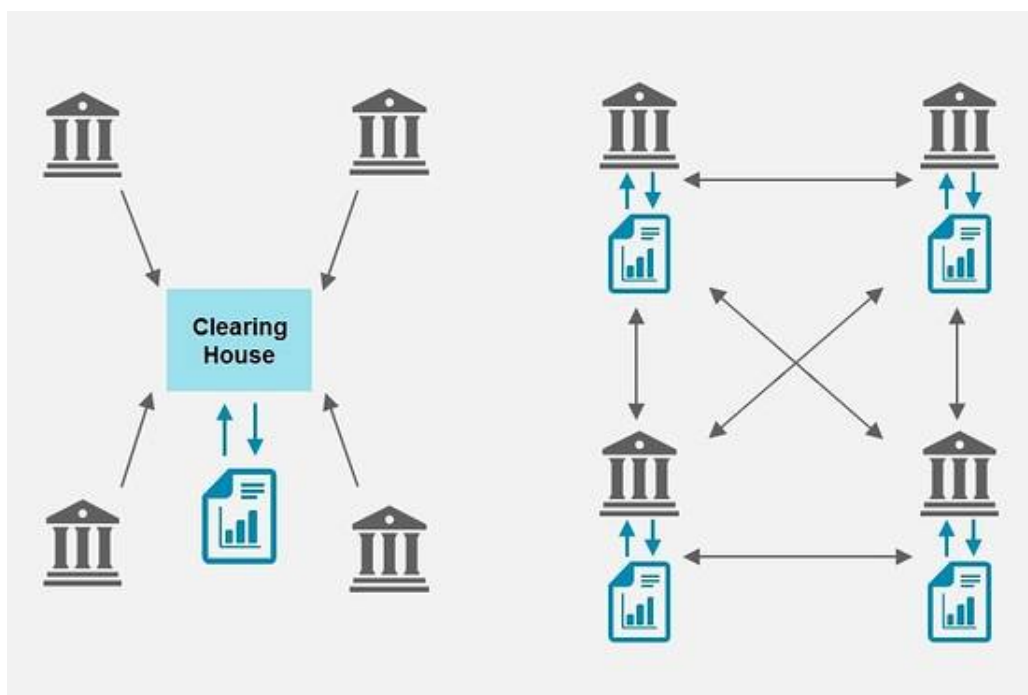
Pametni ugovori naziv su za bilo kakav kompjutorski program ili transakcijski protokol koji automatski izvršava i kontrolira izvršavanje nekog ugovora i svih njegovih dijelova. Ideja je da su uvjeti ugovora kao i njegova etape izvršavanja potpuno prepušteni kompjutorskom kodu.

Glavna korisnost pametnih ugovora je u tome što se oni ne ovise o nikakvim sigurnim posrednicima ili arbitražama i svim troškovima koje takvo posredstvo nosi. Svi ostali problemi i prevare koji mogu nastati u izvršavanju ugovora također su izostavljeni jer se podrazumijeva da je kod tog ugovora napravljen u cilju obuhvaćanja svih mogućih događaja u stvarnom svijetu.

Još jedan važna odluka kod pametnih ugovora jest gdje će se taj program tj. ugovor izvršavati, postoje više solucija ovog problema:

1. Pametni ugovor se izvršava na računalnoj infrastrukturi jednog od sudionika (centralizirani model)
2. Neovisan treći promatrač i njegova infrastruktura je zadužena za izvršavanje pametnog ugovora (centraliziran model)
3. Pametni ugovor se izvršava na decentraliziranoj infrastrukturi, najčešće blockchain (decentralizirani model)

Blockchain ,posebno Ethereum, se nametnuo kao idealno mjesto za decentralizirano izvršavanje pametnih ugovora radi svoje sigurnost i neovisnost. Nijedan sudionik ne može prekinuti ili zabraniti izvršavanje tog ugovora.



Slika 2.1: Centralizirani i decentralizirani model pametnih ugovora

Takvi pametni ugovori koji se izvršavaju na blockchainu nude mnoge druge benefite nego današnji digitalni ugovori. Neki su od njih:

- **Sigurnost**

Izvršavanje pametnog ugovora na decentraliziranoj infrastrukturi osigurava da ne postojanje jednog vektora opasnosti, ili nekog centralnog entita koji će odlučivati što koji sudionik ili pametni ugovor smije raditi

- **Pouzdanost**

Procesiranje i izvršavanje našeg pametnog ugovora povjereno je mnoštvu računala(čvorova) na mreži što daje veliku sigurnost da će se naš ugovor sigurno izvršiti

- **Ravnopravnost**

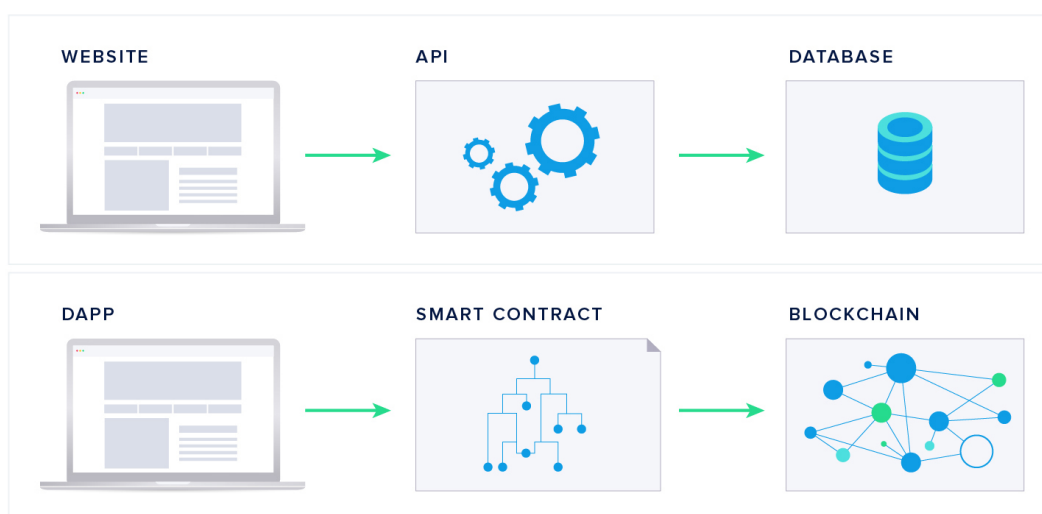
Koristeći decentralizirano mrežu ravnopravnih partnera svi sudionici imaju jednaka prava i smanje se mogućnost manipulacije centralnih profitnih posrednika

- **Efikasnost**

Automatizacija procesiranja ugovora u pozadini i izostajanje papirologije, nijedan sudionik ne mora čekati ni ručno unositi podatke

Pametni ugovor na blockchainu bismo mogli usporediti tradicionalnom web arhitekturom gdje ta aplikacija komunicira sa svojim centralnim serverom i bazom podataka pomoću API-ja (engl. Application Programming Interface). Vlasnik te infrastrukture ima potpuno kontrolu i privatnost, nitko ne autoriziran ne može pristupiti podacima.

S druge strane aplikacije na blockchainu imaju svoj API u obliku pametnog ugovora, te blockchain kao bazu podataka. Time su te aplikacije decentralizirane te imaju svoj ustaljeni naziv imena **DApp** da bi se razlikovale od tradicionalne web arhitekture. Tada se ta web aplikacija spaja preko blockchain klijenta sa pametnim ugovorom, i sva interakcije ide preko njega.

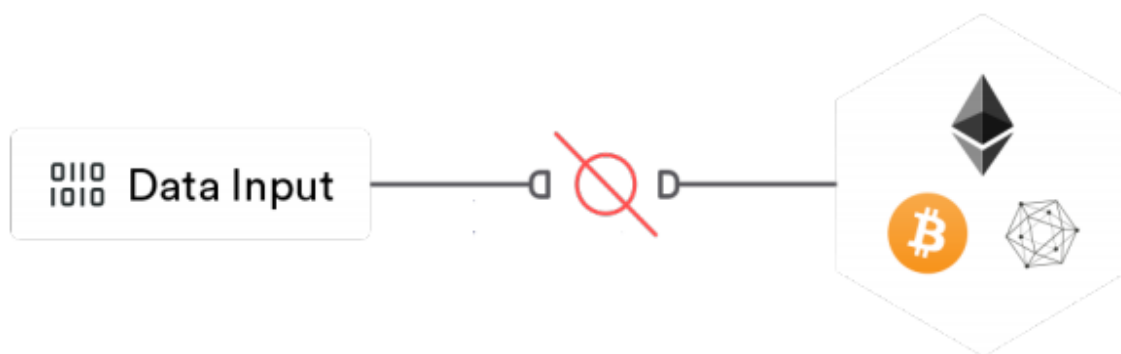


Slika 2.2: Tradicionalna web aplikacija vs. Aplikacija na blockchainu (DApp)

## 2.2 Oracle problem

Pametni ugovori blockchainu imaju prilike revolucionizirati mnoge sektore gdje bi zamijenili današnju potrebu za tradicionalnim legalnim ugovorima i centralnom izvršavanju takvih ugovora. Ali zbog njihovog konsenzus mehanizma, blockchain ne nudi podršku za jednostavno i prirodno komuniciranje sa vanjskim svijetom. A time ni siguran dotok podataka na blockchain.

Pametni ugovori ne mogu imati interakcije sa vanjskim podacima (*eng. off-chain*) bez mogućnosti narušavanja konsenzus mehanizma i blockchaina kao jedinstvenog izvora istine.



Slika 2.2: Problem povezanosti blockchaina sa vanjskim svijetom

Danas, solucija za rješavanje ovog problema i povezivanje sa vanjskim svijetom se nudi u obliku entiteta po imenom eng. „*Oracle*” , aludirajući na mudru i sveznajuću proročicu iz antičke Grčke.

Takav entitet bi trebao biti u mogućnosti riješiti dva tipa problema povezana interakcijom sa vanjskim svijetom:

- Nemogućnost validacije i sigurnog dotoka podataka iz vanjskog svijeta
- Nemogućnost iznosa podataka u druge vanjske sistem niti interakcija sa njima

U praksi to znači ne možemo sigurno donijeti nikakve vanjske podatke na blockchain bez da ne kompromitiramo sigurnost i decentralizaciju( npr. cijene financijskih instrumenta, podatke iz senzora itd.) . Te ne možemo izvršiti nikakve interakcije sa drugim vanjskim sistemima (npr. prijenos novca za bankovnih računa ).

Također time je i korištenje blockchain jako limitirano i usko primjenjivo samo na određeni broj aplikacija koje nemaju interakciju sa vanjskim svijetom.

Projekt **Chainlink** se postavio kao vodeće rješenje i standard za rješavanje problema vanjskih podataka na pametnim ugovorima (eng. Oracle problem).



Slika 1.10: Chainlink Oracle sistem

On nudi inovatnu i efikasnu soluciju za rješavanje tog problema kroz sljedeće solucije:

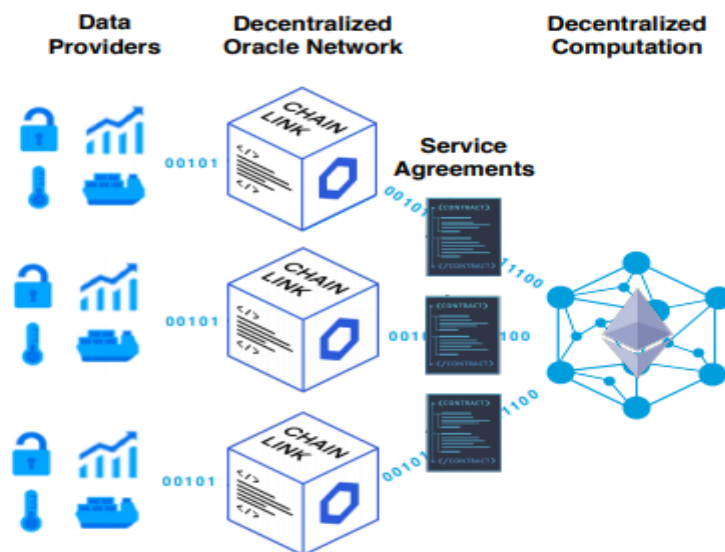
- **Decentralizacija**

Postoji više neovisnih računala koji vrše dužnost dohvata podataka te se oni međusobno spajaju u mrežu *Oracle* čvorove

- **Obvezujući Ugovor o uslugama (eng. Binding Service Agreement)**

Oracle Chainlink čvorovi stupaju u obvezujući ugovor o dotoku podataka kojeg su dužni izvršavati po unaprijed upisanim uvjetima. Svako neizvršavanje utjecaj će na reputaciju te će taj čvor teže kasnije dobivati zadatke za nove usluge

- **Sigurnost kroz individualne Chainlink čvorove i podatke**
- **Mnoštvo obrambenih slojeva** (eng. „*defense in depth*”) koji pridonose sigurnosti i pouzdanosti podataka i obrade tih podataka (npr. „trusted execution enviroment” u kojem svi Chainlink čvorovi izvode obrade podataka)
- Reputacijski sloj te provjerana i odana zajednica Chainlink operatora, koji su ispitani i testirani u različitim uvjetima



Slika 2.2: Chainlink model za siguran dotok podataka na blockchain

## 2.3 Opis projekta

### Motivacija

U ovom poglavlju ćemo opisati projekt imena „**Movement**” koji je smišljen za ovaj diplomski rad te sve njegove dijelove. Kako je pametni ugovor i njegova povezanost sa vanjskim svijetom bio glavna tematika ovog diplomskog rada., autor rada je htio osmisliti jedan mali ekosistem gdje će središnju riječ imati pametni ugovor koji će biti izvršavan na Ethreumovom blockchainu. Ostali dijelovi imati će također važnu ulogu i služiti će pametnim ugovoru kao konekcija za vanjskim svijetom.

Cilj je bio napraviti neku vrstu decentraliziranog platforme koja će koristiti lokacijske podatke sa android telefona za neku svoju namjenu.

Htjeli smo da pametni ugovor, koji nema informacije o događajima iz vanjskog svijeta, sazna i provjeri je li neki put od početne do krajnje točke napravljen. Namjena ovog projekta nisu točno specificirana, ali zbog generalnosti funkcija koju ona ima, može se koristiti u svakakve svrhe dostava pošiljaka, hrane i bilo čega drugog ili kao potvrda da je korisnik prešao neku udaljenosti.

### Opis

Implementacija tog projekta na prvi pogled izgleda

Također će se spajanjem svih tih dijelova htjeti smo pokazati kako više različitih tehnologija komunicira i međusobno daje veliku korisnost za sve sudionike.

Opišimo dijelove infrastrukture:

#### 1. Pametni ugovor



Pametni ugovor imena „Movement” bit će kreiran na Ethereum testnetu Kovan te će služiti kao glavni izvor komunikacija sa Dapp web aplikacijom te android aplikacijom. On će služiti kao glavna baza podataka te ćemo u njega upisivati sve relevantne informacije

## 2. Web aplikacija (Dapp)

Frontend web aplikacija napisana uz pomoć ReactJs paketa.

Služi za interakciju sa pametnim ugovorom na Ethreumu.

## 3. Mobilna android aplikacija

Android aplikacija napisana u kotlinu, služiti će registriranim korisnicima za prelaženje ruta i GPS senzor za lokacijske podatke. Služit će kao digitalna blizanač korisnika koji će slati podatke na Chainlink čvorove



Slika 2.4: Dijelovi projekta „Movement” za diplomski rad

Uz sagrađenu infrastrukturu vežemo i sudionike i entitete koji će ju koristiti i čije će međusobne interakcije sačinjavati ovaj projekt (uključili smo engleska imena jer će ona biti prisutna u aplikacijama).

### 1. Ruta (eng. Route)

Rute će biti zadane udaljenosti od početne do krajnje lokacije koji će korisnici moći preći. Rute će se zadavati preko Dapp web aplikacije te će sadržavati i naplatu u digitalnoj valuti Ether. Pametni ugovor će spremati rute, te brinuti za njihov status

### 2. Korisnik (eng. User)

Korisnici će biti sudionice ekosistema koji će izvršavati rute. Oni će se registrirati preko Dapp web aplikacije da bi imali pristup mobilnoj

aplikacija koja će im služiti za obavljanje ruta i dobivanje informacije o njihovom napretku

### 3. Čvor (eng. Nodes)

Čvorovi će biti računalni serveri koja se sadržavati više procesa složenih u Docker kontejnere među kojima su najvažniji **Data Adapter** (služi za obradu i spremanje lokacijskih podataka koji korisnici naprave na android aplikaciji) i **Chainlink node** koji će se služiti svojim *data adapterom* kako bi mogao točno i sigurno provjeriti je li korisnik stvarno napravio tu rutu. On će kao naplatu za izvršavanje svojih poslova (eng. Job ) biti naplaćen u digitalnoj valuti **LINK** koja predstavlja Chanlink-ovo sredstvo plaćanja na blockchainu.



Slika 2.4: Sudionici i entiteti u projektu

## 2.4 Implementacija pametnog ugovora

### Uvod

U ovom poglavlju opisati ćemo pametni ugovor koji će biti izvršavan na testnet Ethereum blockchainu Kovan. Ukratko ćemo opisati njegov proces izrade i alate, njegove dijelove te funkcionalnost.

### Solidity

Pametni ugovor „*Movement*” napisan je u objektno orijentiranom programskom jeziku više vrste **Solidity** koji služi za izradu pametnih ugovora na Ethereumu. Taj programski jezik je bio utjecan od strane C++, *Pythona* i *Javascripta* te je namijenjen da služi pisanju koda pametnih ugovora na bilo kakvom blockchainu koji koristi EVM (eng. Ethereum Virtual Machine).

*Solidity* je statički pisan programski jezik koji u sebi posjeduje mogućnost nasljeđivanja, rad sa ostalim modulima, kompleksno definirane tipove podataka i ostale stvari.

Također svaki pametni ugovor posjeduje eng. Application Binary Interface (ABI) koji opisuje funkcije, variable i tipove podataka koji je univerzalni za pametne ugovore. ABI uvelike olakšava i konekciju za drugim pametnog ugovora sa drugim programskim jezicima koje ćemo i mi koristiti za našu mobilnu aplikaciju u Kotlinu, a i web dapp aplikaciju u Javascriptu

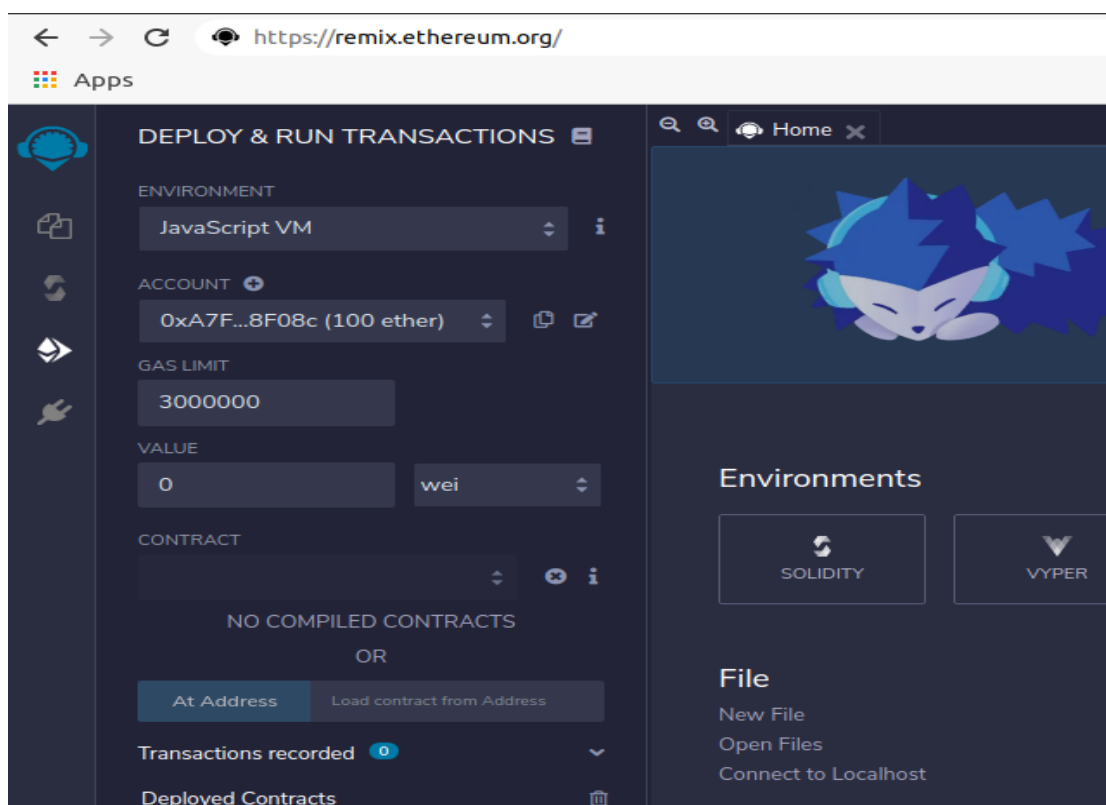
Pametni ugovor napisan u *Solidity*-ju prvo bude kompiliran u byte kod koji će se izvršavati na *EVM* (eng. *Ethereum Virtual Machine*).

## Alati

Nekoliko alata će nam biti potpuno neophodna za interakciju sa pametnim ugovorom i njihovu izradu. Ukratko ćemo ih nabrojiti i opisati njihovu funkcionalnost:

- **Remix IDE**

Glavni online alat za pisanje i *debugiranje* pametnih ugovora u Solidity koji onda i automatski kreira zadani pametni ugovor na blockchainu koji mi želim te izvršava funkcije ili čita stanje pojedinih varijabli.



Slika 2.4: Remix IDE za rad sa pametnim ugovorima na Ethereumu

Naš pametni ugovor je na lokalnoj memoriji računala i da bi bio povezan s web alatom Remix IDE -om potrebno mi je dati dopuštenje da iščitava sadržaj foldera u kojem se nalazi naš pametni ugovor.

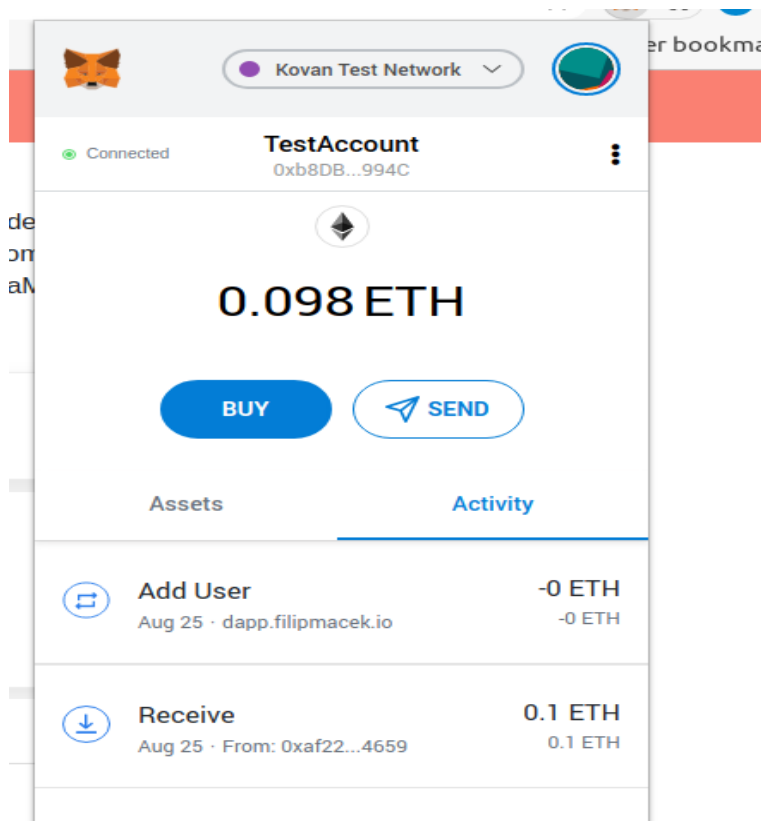
Također da bi ga se moglo spojiti sa blockchainom i pokretati funkcije pametnog ugovora potreban je web novčanik MetaMask koji će onda služiti kao medij kroz koji će ići sve upisne transakcije koje zahtijevaju digitalnu valutu Ether.

- **MetaMask Wallet**

Metamask je *web Browser* koja služio kao digitalni novčanik, a ima i funkcionalnost da komunicira sa pametnim ugovorima ili Dapp web aplikacijama. Kako svaka transakcija i upis podataka u pametni ugovor treba neku količinu digitalne valute Ether ovo je neophodan alat za izradu pametnih ugovora na blockchainu.

On je nam biti glavna veza i spona sa blockchainom jer je potreban i za rad sa Remix IDE -om te za konekciju Dapp web aplikacija sa blockchainom jer Metamask unosi u DOM (eng. Document Object Model) svake stranice web3 API koji će nam biti potpuno nužan za interakciju sa blockchainom.

Web3 API je zapravo protokol kojim komuniciramo sa Ethereum klijentom održavanim od strane MetaMaska. Time nam se pojednostavljuje izrada Dapp web aplikacija jer ne moramo brinuti o održavanju Ethereum klijenta. Dok kod ostalih dijelova infrastrukture, točnije mobilne aplikacija i Chainlink čvora, ćemo morati sami osigurati Ethereum klijenta. To nas dovodi do sljedećeg alata.



Slika 2.5: Metamask novčanik

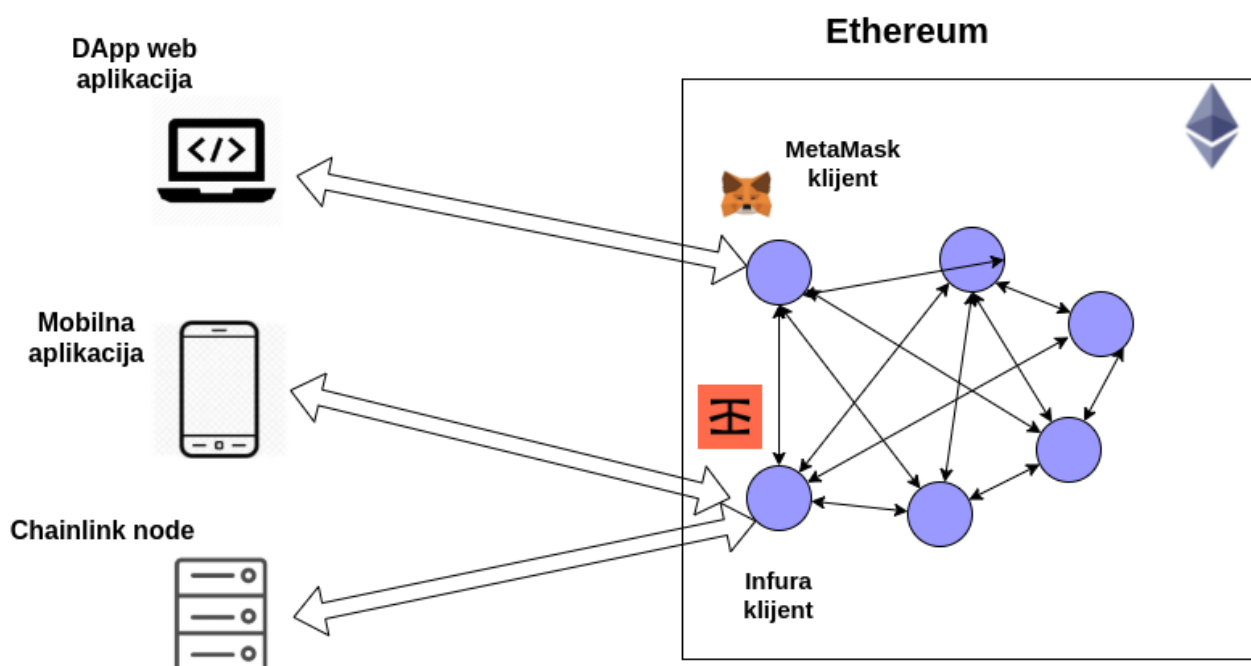
- **Ethereum Infura klijent**

Infura je blockchain kompanija i platforma koja nudi profesionalni usluge povezivanja sa Ethereum-om kroz svoje klijente koji su na blockchainu.

Oni nude besplatnog klijenta do 100,000 zahtijeva (*eng. requests*), što će nam biti dovoljno za ovu demo aplikaciju.

On će nam biti potreban kod konekcija mobilne aplikacije i Chainlink čvora. Također bit će nam i potreban različita tehnologija povezivanja sa Ethereum klijentom. Chainlink čvor će zahtijevati neprekidnu konekciju sa blockchainom da bi mogao slušati za potrebne poslove (*eng. Jobs*) dohvata vanjskih podataka koje su mu potrebni. To je postignuto uz pomoć *eng. WebSocket* tehnologije, dok će mobilna aplikacija zahtijevati HTTP konekcija, jer neće biti potrebna da je osluškuje Ethereum blockchain neprekidno već samo kada hoće upisati neke podatke.

Dijagram svih konekcija infrastrukture sa Ethereum blockchainom izgleda ovako:



# Implementacija

Pametni ugovor napisan u Solidity se ne razlikuje previše od objektnog programiranja i sam pametni ugovor možemo gledati kao jednu veliku klasu koji sadrži svoje varijable i metode/funkcije. Svaki pametni ugovor deklaracije ovih mogućih tipova:

- Varijable stanja
- Funkcija
- Funkcijskih modifikatora - određuju uz koje uvjete se funkcija izvršava
- Događaji
- Struct tipovi – kompleksni tipovi podataka koji sadrže više različitih varijabli

## Struct tipovi podataka

Opišimo prvo sve kompleksne tipove podatke ( eng. Struct) koje ćemo koristiti i koje ćemo spremati u niz da bi pametni ugovor imao uvid u sve događaje koji su mu bitni:

- **Korisnik**

```
struct User {
    uint userId;
    string username;
    string password;
    address addr;
    bool isExist;
    uint routesStarted;
    uint routesCanceled;
    uint routesFinished;
    uint routesCompleted;
}
// mapping from user address to userId
mapping(string =>bool) userExistsByUsername;
mapping(address => bool) userExistsByAddress;
mapping(string => uint) userIndexByUsername;
// Users array
User[] public users;

struct Route {
    uint routeId;
    address maker;
    string taker;
    string startLocation;
    string endLocation;
    bool isStarted;
    bool isFinished;
    bool isCompleted;
    string description;
}
// Routes array
Route[] public routes;
```

- **Ruta**

- **Node**

```
struct Node {
    uint nodeId;
    string nodeName;
    string ip;
    string data_endpoint;
    address oracleContractAddress;
    uint routesChecked;
    string jobIdDistance;
    string jobIdTime;
    string jobIdStatus;
}
// Nodes array
Node[] public nodes;
```

- **RouteStartEvent**

Služio kao podatak da je korisnik prihvatio i započeo izvršavanje rute preko mobilne aplikacije

```
struct RouteStartEvent{
    uint routeStartId;
    uint routeId; // which route
    string username; // who started this route
    uint timestamp; // when the event has been published
    bool node1Status;
    bool node2Status;
}
RouteStartEvent[] public routeStartEvents;
```

- **RouteEndEvent**

Upisuje se kad korisnik završio rutu na mobilnoj aplikaciji i prije nego što Chainlink čvor upiše na blockchain status rute, tj. je li ona završena ili nije

```
struct RouteEndEvent{
    uint routeEndId;
    uint routeId;
    string username;
    uint timestamp; // when the event has been published
    uint dataPoints; // how many coordinate data points the app recorded
    uint node1DataPoints;
    uint node2DataPoints;
    uint userStatus; // state in which route finished - did user cancel route, or submitted it
}
RouteEndEvent[] public routeEndEvents;
```



- **CheckStatusEvent**

Sadrži sve informacije o izvršavanju Chainlink čvorovi i njihovih odgovora o stanju rute, tj. je li korisnik završio rutu i u kojem vremenu i distanci

```
struct CheckStatusEvent{
    uint checkStatusId;
    uint routeId;
    string username; // for which user we are checking if he completed the route;
    uint timestamp;
    uint node1Distance;
    uint node1Time;
    bool node1Status;
    uint node2Distance;
    uint node2Time;
    bool node2Status;
}
CheckStatusEvent[] public checkStatusEvents;
```

## Događaji

Događaji (eng. Events) su još jedan važan član pametnih ugovora na Ethereumu. Oni su apstrakcija evidentiranja (*eng. logging*) na blockchainu, gdje svaki događaj kada bude emitiran biva upisan u transakcijski dnevnik svoj pripadajućeg pametnog ugovora, te ih različiti klijenti mogu iščitati ili se pretplatiti na novonastale događaje o kojima će biti obaviješteni.

S obzirom na pokrenute funkcije pametnog ugovora, odašiljat će se i različiti događaji. Ovo je njihov popis:

```
//===== Events =====
event UserCreated(uint userIndex);
event RouteCreated(uint routeIndex);
event StartRouteEvent(uint routeStartId);
event EndRouteEvent(uint routeEndId);
event StatusCheckEvent(uint checkStatusId);
event StatusCheckFulfilled(uint checkStatusId);
event CompletedRouteEvent(uint routeCompletedId);
```

Slika 2.5: Događaji pametnog ugovora

## Globalne varijable

- **Adresa agenta**

Agent je entitet koji bilježi i snima korisnikove poteze i događaje na mobilnoj aplikaciji gdje taj korisnik izvršava zadane rute. To nam je bitno kako bismo upisali sve korisnikove poteze na blockchain. Agent nije zadužen za provjeru je li korisnik napravio rutu već samo motri ono što mu se zadane.

- **Adresa LINK digitalne valute**

Chainlink čvorovi i njegovi pripadajući pametni ugovori koje koristi pri izvršavanju zadataka trebaju znati točnu adresu *LINK* digitalne valute koja im je standardizirana valuta naplate.

- **Količina naplate po izvršenom poslu**

Za koliku količinu *LINK* digitalne valute će *Chainlink* čvorovi biti naplaćeni po izvršenom poslu

- Konstante vezane za stanje je li korisnik prekinuo rutu tokom izvršavanja (ili zatvorio aplikaciju iako nije dovršio rutu) ili je korisnik uredno slijedio proces i dovršio rutu do kraja

```
address private agent;

uint constant private ORACLE_PAYMENT = 1*LINK;

address constant LINK_TOKEN_ADDRESS = 0xa36085F69e2889c224210F603D836748e7dC0088;

// Constants related to states that can happen with started route
uint constant USER_CANCELED = 1;
uint constant USER_SUBMITTED_ROUTE = 2;
```

Slika 2.6: Globalne varijable

## Funkcije

Funkcija ili metoda pametnog ugovora su osnova za njegov rad i izvršavanje različitih potrebnih funkcionalnost. Opisat ćemo sve najbitnije za rad pametnog ugovora, i da bi olakšali razumijevanje i uloge svake funkcije u projektu raspodijeli smo funkcije po dijelu infstrukture koji ju poziva. To može biti mobilna android aplikacija ili Dapp web frontend aplikacija.

- **addUser**

Pametni ugovor pomoću ove funkcije registrirati novog korisnika i daje mu pristup mobilnoj aplikacija gdje će se korisnik pomoću korisničkog imena i zaporku moći se ulogirati. (Kako je blockchain i svi njegovi podaci javni, na zaporku će biti primijenjena dvostruka SHA-256 funkcija te će taj rezultat biti upisan u blockchain)

```
function addUser(string memory _username, string memory _password) public {
    users.push(User(users.length+1, _username, _password, msg.sender, true, 0, 0, 0, 0));
    userExistsByUsername[_username]=true;
    userExistsByAddress[msg.sender]=true;
    userIndexByUsername[_username]=users.length-1;
    emit UserCreated(users.length-1);
}
```

Slika 2.6: Funkcija addUser

- **addRoute**

Ova funkcija registrira novu rutu koja je neki sudionik zadao na Dapp web aplikacija. Da bi ruta bila zadana i upisana potrebno joj je startna lokacija i krajnja lokacija te kratki opis rute. Svaka lokacija mora biti zapisana sadržavati geografsku širinu (eng., *latitude*) i dužinu (engl. *longitude*). Npr. „45.2322, 15.32112”

```
function addRoute(string memory _startLocation, string memory _endLocation, string memory _description) public {
    string memory taker = '';
    routes.push(Route(routes.length+1, msg.sender, taker, _startLocation, _endLocation, false, false, false, _description));
    emit RouteCreated(routes.length-1);
}
```

Slika 2.7: Funkcija addRoute

- **addNode**

Registrira novog Chainlink čvora te upise sve relevantne podatke, kao što su njegova ip adresa i eng. „DataEndpoint” koja služe da bi mobilna android aplikacija (kada korisnik krene izvršavati rutu), znala na koju adresu da šalje lokacije podatke. (Npr. format „ip:endpoint” – 53.21.123.22/api)

Također sadrži identifikacijsku oznaku za Chainlink čvor, da bi znao koji posao (eng. Job) dohvata podatka na blockchain treba izvršiti.

```

function addNode(string memory _name,string memory _ip,string memory _data_endpoint,
                address _oracleContractAddress,
                string memory _jobIdDistance,
                string memory _jobIdTime,
                string memory _jobIdStatus) private {
    nodes.push(Node(nodes.length+1,_name,_ip,_data_endpoint,_oracleContractAddress,0,_jobIdDistance,_jobIdTime,_jobIdStatus));
}

```

Slika 2.8: Funkcija addNode

- **startRouteEvent**

Izvršava se kada korisnik započne izvršavanje rute na mobilnoj aplikaciji. Bitno nam jer onda mijenjamo status rute i tu informaciju upisujemo u u blockchain. Također funkciju sadrži *modifier* „onlyAgent” što znači da ju jedino može izvršiti adresa agenta.

```

// Agents functions
function startRouteEvent(uint _routeId,string memory _username,bool _node1Status,bool _node2Status) public onlyAgent{
    // require that user is registered and exists
    require(userExistsByUsername[_username],"User doesnt exists");

    // Change route status
    // Dont forget to add -1 because its index of array
    routes[_routeId-1].isStarted = true;
    routes[_routeId-1].taker = _username;
    users[userIndexByUsername[_username]].routesStarted+=1;

    // Add info to routesStartEvents
    routeStartEvents.push(RouteStartEvent(routeStartEvents.length+1,_routeId,_username,now,_node1Status,_node2Status));

    emit StartRouteEvent(routeStartEvents.length-1);
}

function getRouteStartEventCount() public view returns(uint){
    return routeStartEvents.length;
}

```

Slika 2.9: Funkcija startRouteEvent

- **endRouteEvent**

Izvršava se kada korisnik prekine ili završi izvršavanje rute na mobilnoj aplikacija. Moramo uračunati oba slučaja kada korisnik prekine izvršavanje rute ili zatvori aplikaciju) te dovrši rutu do kraja.

Upisujemo još neke metapodatke koje je mobilna aplikacija zabilježila kao što su količina lokacijski podataka koji su postojali te koliko su ih Chainlink čvorovi primili.

```

function endRouteEvent(uint _routeId,string memory _username,uint user_event,uint _dataPoints,uint _node1DataPoints,uint _node2DataPoints)public onlyAgent{
    uint len = routeEndEvents.length;
    if(user_event == USER_CANCELED){
        // User canceled so we dont care about any other data
        // we init both dataPoints and node data to 0

        routeEndEvents.push(RouteEndEvent(len+1,_routeId,_username,now,0,0,0,USER_CANCELED));
        routes[_routeId-1].isStarted = false;
        routes[_routeId-1].taker = '';
        users[userIndexByUsername[_username]].routesCanceled+=1;
        emit EndRouteEvent(routeEndEvents.length-1);
    }
    else if(user_event == USER_SUBMITTED_ROUTE){
        // User submitted route
        routeEndEvents.push(RouteEndEvent(len+1,_routeId,_username,now,_dataPoints,_node1DataPoints,_node2DataPoints,USER_SUBMITTED_ROUTE));
        routes[_routeId-1].isFinished = true;
        users[userIndexByUsername[_username]].routesFinished+=1;

        // User submitted route - call requestRouteStatus to ask chainlink nodes about route status
        requestRouteStatus(_routeId,_username);

        emit EndRouteEvent(routeEndEvents.length-1);
    }
}
}

```

Slika 2.10: funkcija endRouteEvent

- **requestRouteStatus**

To je privatna funkcija koje će biti pozvana iz funkcije endRouteEvent ukoliko korisnik pravilno završi rute. Time će je efektivno agent pozvati i on će snositi transakcijske troškove te funkcija.

Ova funkcija je važna sto šalje preko registrirane adrese (*eng. „Oracle Address”*) zahtjev *eng. ChainlinkRequest* svim upisanim čvorovima na blockchain da izvrše svoje poslove i dohvate joj tražene podatke.

```

// Request route status by user on dapp
function requestRouteStatus(uint _routeId,string memory _username) private {

    string memory _routeIdString = uintToString(_routeId);

    // Init CheckStatusEvent data -- we will write to this variable all our results
    checkStatusEvents.push(CheckStatusEvent(checkStatusEvents.length+1,_routeId,_username,now,0,0,false,0,0,false));
    emit StatusCheckEvent(checkStatusEvents.length);

    //-----Node 1 -----
    // Distance Data Request
    Chainlink.Request memory req1 = buildChainlinkRequest(stringToBytes32(nodes[0].jobIdDistance),address(this),this.fulfillDistanceRequest1.selector);
    req1.add("extPath",_routeIdString);
    sendChainlinkRequestTo(nodes[0].oracleContractAddress,req1,ORACLE_PAYMENT);

    // // Time Data Request
    Chainlink.Request memory req2 = buildChainlinkRequest(stringToBytes32(nodes[0].jobIdTime),address(this),this.fulfillTimeRequest1.selector);
    req2.add("extPath",_routeIdString);
    sendChainlinkRequestTo(nodes[0].oracleContractAddress,req2,ORACLE_PAYMENT);

    // Status Data Request
    Chainlink.Request memory req3 = buildChainlinkRequest(stringToBytes32(nodes[0].jobIdStatus),address(this),this.fulfillStatusRequest1.selector);
    req3.add("extPath",_routeIdString);
    sendChainlinkRequestTo(nodes[0].oracleContractAddress,req3,ORACLE_PAYMENT);

    // ----- Node 2 -----
    // Distance Data Request
    Chainlink.Request memory req4 = buildChainlinkRequest(stringToBytes32(nodes[1].jobIdDistance),address(this),this.fulfillDistanceRequest2.selector);
    req4.add("extPath",_routeIdString);
    sendChainlinkRequestTo(nodes[1].oracleContractAddress,req4,ORACLE_PAYMENT);

    // // Time Data Request
    Chainlink.Request memory req5 = buildChainlinkRequest(stringToBytes32(nodes[1].jobIdTime),address(this),this.fulfillTimeRequest2.selector);
    req5.add("extPath",_routeIdString);
    sendChainlinkRequestTo(nodes[1].oracleContractAddress,req5,ORACLE_PAYMENT);

    //Status Data Request
    Chainlink.Request memory req6 = buildChainlinkRequest(stringToBytes32(nodes[1].jobIdStatus),address(this),this.fulfillStatusRequest2.selector);
    req6.add("extPath",_routeIdString);
    sendChainlinkRequestTo(nodes[1].oracleContractAddress,req6,ORACLE_PAYMENT);

    emit StatusCheckFulfilled(checkStatusEvents.length);
}

```

- Slika 2.13: Funkcija requestRouteStatus

## • Fulfill skup funkcija

To su engl. *callback* funkcije koje se izvršavaju koda *Chainlink* čvorovi prime potvrdu da su im uplaćeni *LINK* digitalne valuta za izvršavanje poslove. One unose na blockchain u tip podatka „*struct*” *CheckStatusEvent* ono su oba *Chainlink* čvora izračunala od lokacijski podataka. Kako imamo dva *Chainlink* čvora i tri podatka koja nas zanimaju (1. je li korisnik prešao rutu, 2. u koje vrijeme ,3. u kojem periodu) imati ćemo šest takvih funkcija

```
// Status Callbacks for node1 && node2

function fulfillStatusRequest1(bytes32 _requestId, bool _status) public recordChainlinkFulfillment(_requestId){
    checkStatusEvents[checkStatusEvents.length-1].node1Status = _status;
}

function fulfillStatusRequest2(bytes32 _requestId, bool _status) public recordChainlinkFulfillment(_requestId){
    checkStatusEvents[checkStatusEvents.length-1].node2Status = _status;
}
}
```

Slika 2.12: Primjer dvije *callback* funkcije oba Chainlink čvora o stanju rute

- **GetCount skup funkcija**

Ovaj skup funkcija samo vraća duljina svih inicijaliziranih nizova za već definirane tipove podataka. Potreban nam je kod poziva sa Dapp web aplikacije da bismo znali koliko API poziva je potrebno napraviti da dostavimo podatke sa blockchaina

```
// Get Count funkcije

function getUsersCount() public view returns(uint) {
    return users.length;
}

function getRoutesCount() public view returns(uint) {
    return routes.length;
}

function getNodesCount() public view returns(uint){
    return nodes.length;
}

function getRouteStartEventCount() public view returns(uint){
    return routeStartEvents.length;
}

function getRouteEndEventCount() public view returns(uint){
    return routeEndEvents.length;
}

function getCheckStatusEventCount () public view returns(uint){
    return checkStatusEvents.length;
}
}
```

Slika 2.13: „Get Count” tip funkcija

## Poglavlje 3

### Mobilna i web aplikacija

#### 3.1 Uvod

U ovom poglavlju ćemo opisati druge dvije važne komponente našeg projekta „Movement”. To su mobilna android aplikacija te web Dapp frontend aplikacija. Obje aplikacije komuniciraju sa pametnim ugovorom na Ethereumu. Kako implementacija ovih aplikacija i njihovih tehnologija nije tema ovog diplomskog rada nego pametni ugovori i njihova povezanost sa vanjskim svijetom, nećemo stavljati kod tih aplikacije, nego će se on moći pronaći na *Github* računu autora ovog diplomskog rada (<https://github.com/filipmacek>)

#### 3.2 Tehnologije



# Poglavlje 4

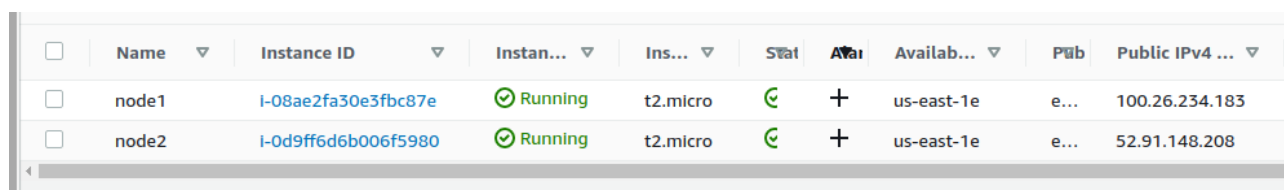
## Chainlink čvor

### 4.1 Uvod

U ovom poglavlju ćemo opisati sve dijelove *Chainlink* čvora. Kao važna napomena je da kada kažemo *Chainlink* čvor (eng. Node) mislimo i na ostale pomoćne procese koje su mu potrebni da bi izvršavao svoju funkciju (npr. *DataAdapter*, *Mongo* baza podataka itd). Opisati ćemo te ostale procese i njihov odnos.

### 4.2 Opis tehnologija i procesa

Odlučili smo se u ovom projektu koristiti sa dva *Chainlink* čvora, tj, koristi ćemo dva Linux Ubuntu servera uz pomoć Amazon web usluga (eng. *Amazon Web Services*). Ti Linux serveri će imati potpuno istu konstituciju, iste procese i istu funkciju. Time će i oba čvora vraćati iste podatke na blockchain o stanju ruta koje je korisnik napravio kada budu zatražene od pametnog ugovora. Cilj je bio pokazati kako pametni ugovor može decentralizirati svoj izvor podataka i ne ovisiti samo o jednom čvoru i njegovom odluci.



<input type="checkbox"/>	Name ▾	Instance ID ▾	Instan... ▾	Ins... ▾	Stat	Avai	Availab... ▾	Pub	Public IPv4 ... ▾
<input type="checkbox"/>	node1	i-08ae2fa30e3fbc87e	Running	t2.micro	🟢	+	us-east-1e	e...	100.26.234.183
<input type="checkbox"/>	node2	i-0d9ff6d6b006f5980	Running	t2.micro	🟢	+	us-east-1e	e...	52.91.148.208

Slika 4.1: Linux serveri na AWS oblaku koji služe kao *Chainlink* čvorovi

## Docker

Kao glavnu tehnologiju za povezivanje svi procesa koristili smo ideju izoliranje procesa u lagane kontejnere imena *Docker* koji se jednostavno mogu pokrenuti, zaustaviti ili ponovno pokrenuti. *Docker* koristi mogućnost virtualizacije u operacijskom sustavu da bi zapakirao to procese te uvelike smanjio količinu potrebnih resursa nego kao kod tradicionalnijih virtualnih mašina (*eng. Virtual Machine*). Također koristili smo i pomoćnu konfiguraciju imena *docker-compose* koje nam omogućava da lako i jednostavno spojimo više *Docker* procesa u jednu mrežu. Ti kontejneri se slažu tako da se prvotno specificira početna slika (tj. operacijski sustav) od kojeg će taj kontejner biti složen, a onda dodavaju dodatni i pomoćni paketni, te se na kraju kopira naš kod te mu se daje informacija kako da pokrene taj kontejner i koje da pristupne točke otvori (*eng. ports*)

## PostgreSQL i MongoDB

Koristit ćemo relacijsku (*eng. SQL*) bazu podataka *Postgres* koja je potrebna Chainlink čvoru kako bi upisao sve njemu važne i relevantne podatke (npr. poslove koje izvršava, njihove konfiguracije itd.).

Također druga baza podataka nerelacijsku (*eng. NoSQL*) koristimo u cilju da bi naš Data Adapter bio u mogućnosti spremati i čitati lokacijske podatke.

## Redis

Redis je baza podataka u koja sprema tražene strukture podataka u memoriju i nama služi kao red (*eng. queue*) u koji upisujemo lokacijski podataka koji je pristigao, da ga u pozadini može obraditi radnik (*eng. worker*) napisan uz pomoć Javascript paketa „bull”.

## Data Adapter

Data Adapter je web server napisan u *Javascriptu* uz pomoć paketa „express” i „mongoose” koji ga spaja sa *Mongo* bazom podataka. Cilj ovog procesa je da ima otvorene rute i prihvća podatke sa mobilne aplikacije kada korisnik na mobilnoj aplikaciji započne izvršavanje rute. Kada ovaj proces primi lokacijski podataka, ta informacija se upisuje u *redis* i naš radnik započinje obradu tog lokacijskog podataka. Time naš server ne blokira, jer obrađuje podatke u pozadini.

## Nginx

Kako bi osigurali veću sigurnost našeg servera, otvorit ćemo samo *port 80* kroz koji će prolaziti svi naši lokacijski podaci. Time nam web server *nginx* služi kao eng. „Reverse proxy” koji samo prosljeđuje dobivene podatke na ostale procese po unaprijed zadanim pravilima.

## Chainlink čvor

Sam Chainlink čvor je napisan u *Golangu*, i služi zapravo kao eng. *middleware* između blockchaina i vanjskog svijeta. Cilj je da se njihovom standardizacijom u samom čvoru, i definiciji poslova i njihovih specifikacija, formira decentralizirana *Chainlink oracle* mreža. Neka od najvažnijih svojstva ovog programa su:

- jednostavna povezanost sa pametnim ugovorima na blockchainu sa bilo kojim vanjskim sustavom ili API-jem
- različite metode dohvata i obrade podataka za potrebe pametnog ugovora
- automatska briga o troškovima upisa na blockchain
- translacija različitih tipova podataka u pripadajuće tipove na Ethereumu blockchainu
- stvaranje notifikacija prilikom svake promjene stanje na pametnom ugovoru, koji prati pomoću transakcijskog dnevnika na Ethereumu

Chainlink čvor također prilikom inicijalizacije stvara svoj privatnih i javnih ključ kako bi imao vlastitu adresu na blockchainu te bi pomoću nje mogao upisivati dohvaćene tražene podatke kada oni budu potrebni.

Komunicirat ćemo sa Chainlink čvorom pomoću njegovog web sučelja koji se otvara na *portu 6688*

Sve procese na serveru možemo izlistati pomoću naredbe **docker ps**

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
fc6abf7c8fd	dataadapter_data_adapter	"docker-entrypoint.s..."	5 days ago	Up 5 days	0.0.0.0:3000->3000/tcp	data_adapter
8db3fd25b97b	mongo:latest	"docker-entrypoint.s..."	5 days ago	Up 5 days	0.0.0.0:27017->27017/tcp	mongo
13971e8e3ee4	redis:latest	"docker-entrypoint.s..."	7 days ago	Up 7 days	0.0.0.0:6379->6379/tcp	redis
9dd9f9d39a99	dataadapter_nginx	"nginx"	7 days ago	Up 6 days	0.0.0.0:80->80/tcp	nginx
f336faccb75e	smartcontract/chainlink	"chainlink local n"	7 days ago	Up 7 days		chainlink
2eaabc40a66a	postgres	"docker-entrypoint.s..."	7 days ago	Up 7 days	0.0.0.0:5432->5432/tcp	postgres

Slika 4.2: Svi docker procesi na serveru koji služi kao Chainlink čvor

# Bibliografija

- [1] David Lee Chaum, *Computer Systems Established, Maintained and Trusted by Mutually Suspicious Groups*, (1982.) University of California, Berkley
- [2] Satoshi Nakamoto, Bitcoin: A Peer-to-Peer Electronic Cash System, (2008.) <https://bitcoin.org/bitcoin.pdf>

