Универзитет у Крагујевцу

Факултет инжењерских наука

Пројектни задатак – документација

Тема:

Фер игра на срећу имплементирана коришћењем blockchain технологије

Студент:
Ђорђе Гачић 626/2018

Предметни професор:
др. Владимир Миловановић

Крагујевац, август 2021. године

Садржај

1. Увод

У новије вријеме све више се говори о blockchain технологији која има огроман потенцијал за примјену у разним софтверским пројектима. Најраспрострањенија примјена до сада је примјена у криптовалутама. Поред тога све чешће се помиње увођење blockchain технологије у коцкарску индустрију јер омогућава фер и провјерљиве резултате клађења. Управо таква примјена (на неком нижем нивоу) је имплементирана у овом пројекту.

2. Опис основних дијелова имплементације пројекта

Да би схватили начин функционисања апликације потребно је упознати се са неким основним технолошким имплементацијама на којима почива и овај пројекат. Најважније су:

- peer-to-peer мрежа
- hash функција
- “public key” криптографија и дигитални потпис
- Blockchain технологија

Peer-to-peer мрежа је кључна за децентрализацију како би се уклонио централни ауторитет и тако повећало повјерење учесника који ће на тај начин учествовати у генерисању случајног резултата заједно са другим учесницима и валидатором, чији блок ће бити додат на основу „договора“ са другим валидаторима.

Hash функција која се користи у овом пројекту је SHA-256 која враћа 256-битни излаз и представља детерминистичку функцију која је коришћена и у Биткоин имплементацији. Хеш функције су такве да обично нису инвертибилне што значи да не постоји функија која из задатог излаза враћа улаз hash функције на чему се и заснива доказана насумичност излаза из hash функције.

“Public key” криптографија користи се за генерисање пара који се састоји од тајног и јавног кључа чиме се омогућава идентификација како учесника тако и валидатора. Генерисање ових кључева заснива се на RSA алгоритму за асиметричну криптографију у коме кључну улогу имају велики прости бројеви као и сложеност факторизације тих бројева. Укратко, потписивање поруке функционише на следећи начин: Нека имамо генерисан пар тајни/јавни кључ и поруку коју желимо потписати. За потпис нам је неопходан тајни кључ и порука, док нам је за верификацију потписа довољан јавни кључ, потпис и порука, што значи да тајни кључ није потребан при верификацији.

signature = sign(privateKey, message)

verify(publicKey, message, signature)

Blockchain је заправо компјутерски фајл који се састоји од блокова података који су међусобно повезани. Сваки блок садржи везу са претходним блоком и на тај начин се формира ланац. У зависности од употребе, блокови могу садржати различите податке. Blockchain је у основи осмишљен да буде јавно доступан тј. да свако може да види историју трансакција (у нашем случају добитака) уграђених у блокове, али може да их посједује само онај чији се тајни кључ поклапа са јавним кључем који је повезан са добитком.

Структура blockchain-а у овом пројекту:


```
"ab72b7628dc058b0247945f3bc8b11399ea2d10c75e99901cbb385fb3dedd91c":
    {
     "validatorPK": "-----BEGIN PUBLIC KEY-----\nMIA...QAB\n-----END PUBLIC KEY-----",
     "blockTimestamp": 1628778304.960544,
     "prevBlockHash": "367af80012b1c93...64d9c880f0dd",
     "bets":
       [
        {"44e41c57c4ac3ecafbb5520784c64f91df27248a861b772e7590593b82cb61f6":
            {
             "gamblerPK": "-----BEGIN PUBLIC KEY--...b3DAB\n-----END PUBLIC KEY-----",
             "numForProbability": "8",
             "sequenceChoice": "010",
             "betTimestamp": 1628778302.678402,
             "betSignature": "64aca8a442b5...c8fd03"
            }
        },
        {"ecb8bba30b4183243acfc4709a5260187c238a236848ddcb554a1ad23c765f28":
            {
             "gamblerPK": "-----BEGIN PUBLIC KEY--...QAB\n-----END PUBLIC KEY-----",
             "numForProbability": "2",
             "sequenceChoice": "1",
             "betTimestamp": 1628778302.729225,
             "betSignature": "9471d5720dbd98ed...d7b"
            }
        },
        {"e6af2e8a8cdcbc7476430ab5b0ab539d142c9b20e1d657ee0aa75bb6f836d7cb":
            {
             "gamblerPK": "-----BEGIN PUBLIC KEY-----\nMIGf...--END PUBLIC KEY-----",
             "numForProbability": "4",
             "sequenceChoice": "10",
             "betTimestamp": 1628778302.753308,
```

```
            "betSignature": "1a154c161...7b7822f"
           }
       },
       ...
      ],
   "blockSignature": "424660eef715a9aff08b00...561f7"
  }
```

Blockchain је у овом пројекту формиран као Пајтон објекат типа dict тј. ријечник података а чува се као .json фајл. По потреби се из тог фајла чита вриједност blockchaina или се у њега додаје нови блок.

Најгорња хеш вриједност је хеш читавог блока добијена из свих преосталих података у блоку и та вриједност представља један у низу кључева Пајтон објекта dict који представља blockchain. Вриједност везана за један такав кључ представља податке блока у које спадају редом јавни кључ валидатора чији блок је додат, timestamp тј. тренутак у коме је генерисан блок, хеш вриједност претходног блока, низ опклада и на крају валидаторов потпис блока.

Свака опклада је представљена као кључ:вриједност податак гдје је кључ заправо хеш вриједност свих података опкладе међусобно конкатенираних, а вриједност је скуп података опкладе и састоји из јавног кључа коцкара тј. учесника у извлачењу, одабране вјероватноће на коју је учесник играо, одабране секвенце за коју учесник претпоставља да ће се појавити на крају бинарне репрезентације хеш вриједности блока, timestamp-а када је генерисана опклада и потписа учесника.

Све поменуте функционалности и њихова имплементација у овом пројекту ће бити детаљније појашњени у наставку.

3. Начин функционисања апликације

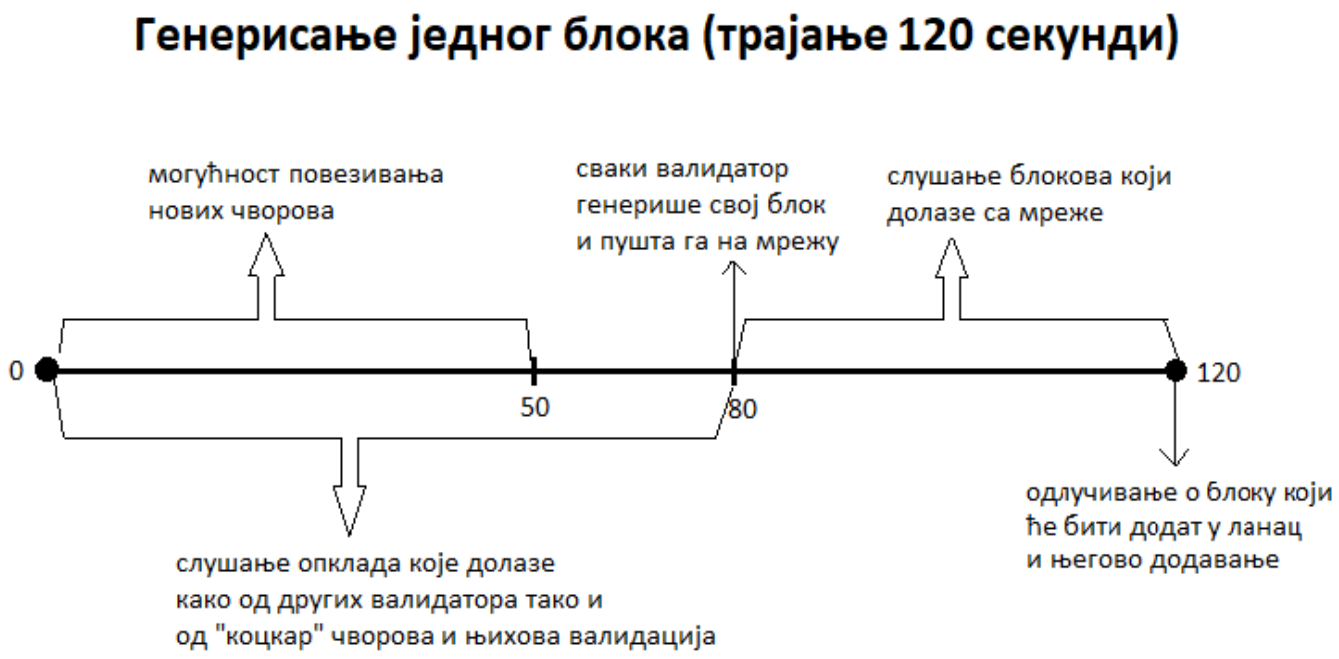
Постоје 3 врсте чворова p2p мреже имплементиране у овом пројекту и то су:

- Рутер (routerNode)
- Валидатор (validatorNode)
- Коцкар (gamblerNode)

Што се тиче „рутер“ чвора он у овом пројекту представља мало одступање од чисте децентрализације. Он се покреће први и мора бити стално доступан на свима познатој адреси. У суштини представља сервер који служи да се њему јаве сви чворови који желе у мрежу. Потреба за рутер чвором настаје јер остали чворови иницијално не знају који су чворови активни у мрежи да би се на њих повезали. Оно што тачно ради рутер чвор јесте

то да у сваком моменту посједује списак тренутно активних „валидатор“ чворова и обезбјеђује прослијеђивање тог списка оном чвору који се јави на опште познату адресу рутер чвора. Из тог списка, чвор који се јавио бира насумично три чвора и на њих се повезује. За активне чворове рутер чвор сматра све оне који су се једном послали захтјев за повезивање и од чијег последњег оглашавања није прошло више од 5 минута или који нису послали захтјев за напуштање мреже. Оно што је битно напоменути јесте да је то једина улога овог чвора, те да он ни на који начин не утиче на генерисање резултата извлачења и нема никаквог додира са blockchain-ом. То је препуштено осталим чворовима.

„Валидатор“ чворови раде највећи дио посла у мрежи. Они слушају опкладе, провјеравају исправност примљених опклада и блокова, додају блокове у ланац и служе „коцкар“ чворовима као сервери за повезивање. При иницијализацији мреже потребно је да постоје најмање четири „валидатор“ чвора јер се сваки од њих мора повезати на још три. Иницијални „валидатор“ чворови преузимају иницијални blockchain фајл у коме је почетни блок, док ће „валидатор“ чворови који се касније јаве у мрежу преузимати blockchain од једног од чворова на које се повежу како би имали најновију верзију фајла. Нови блок ће бити уграђен у ланац на свака 2 минута, што значи да то и интервал између два кола у клађењу. Оно што се дешава у току та 2 минута илустративно је приказано на слици 1:



Слика 1: Процес генерисања блока

У првих 80 секунди сваки валидатор слуша опкладе које долазе од „коцкар“ чворова повезаних на њега и од других валидатора. Сваки валидатор провјерава исправност опкладе коју прими и под условом да је валидна и да је већ нема на списку примљених опклада прослиједјује је даље валидаторима са којима је повезан. Могућност повезивања нових чворова ограничена је на 50 секунди, а не на 80 секунди из разлога да би се оставио сасвим довољан интервал од 30 секунди како би све опкладе дошле до сваког валидатора. То је битно јер на крају сви валидатори морају имати код себе све опкладе за које је чуо било који активан валидатор. Примјера ради, уколико би вријеме за повезивање нових чворова тј. учесника било продужено на 80 секунди, учесник би могао послати опкладу у последњем моменту и тако постоји вјероватноћа да она не би обишла све валидаторе у мрежи прије него се пређе на процес генерисања и слања блокова.

Након 80 секунди, сваки валидатор прелази на генерисање сопственог блока који ће садржати све опкладе за које је валидатор чуо у току претходних 80 секунди. После генерисања тај блок се шаље валидаторима на које је повезан, али након провјере код „рутер“ чвора да ли су ти валидатори још активни (ако нису врши се конектовање на друге - активне валидаторе у мрежи). Затим сваки валидатор прелази на слушање блокова који долазе са мреже. На крају интервала од 120 секунди битно је да сви поштени чворови угради у ланац исти блок, а за такво нешто потребан је консензус између валидатора. То је имплементирано на начин да блок са најмањом хеш вриједношћу треба бити уграђен у ланац. Стога приликом слушања блокова са мреже, сваки валидатор врши провјеру да ли је блок валидан и да ли му је хеш вриједност мања од вриједности претходног блока који је примио. Ако јесте, тај блок се прослиједјује даље и оставља се као исправан за уградњу у blockchain под условом да после њега не стигне блок са мањом хеш вриједношћу. За пропагацију блокова кроз мрежу остављен је период од 40 секунди. О начину провјере валидности блокова и опклада биће биће више ријечи при проласку кроз само код

Након истека 120 секунди тј. након уградње новог блока у blockchain, прелази се на нови круг тј. на поновно слушање опклада и слушање захтјева учесника. Такође, треба напоменути да ће поруке бити игнорисане од стране чвора примаоца уколико стигну кад им није вријеме (нпр. случај ако се појави малициозни чвор који би такву поруку послао)

„Коцкар“ чвор тј. учесник у клађењу се повезује преко „рутер“ чвора на три насумична „валидатор“ чвора у периоду предвиђеном за повезивање (првих 50 секунди). Тим валидаторима се шаље опклада генерисана од стране учесника и затим учесник има опцију да сачека док истекне остатак времена до краја тог извлачења (тј. до генерисања блока у коме ће бити његова опклада) када он може провјерити да ли је његова секвенца добитна. Ако то не провјери тада, може и накнадно повезивањем у мрежу само са захтјевом за blockchain фајл (тј. без слања опкладе). У том случају добиће све опкладе које су повезане са јавним кључем који је учесник (коцкар) претходно унио.

4. Структура пројекта и опис дијелова кода

Овај пројектни задатак имплементиран је у потпуности у програмском језику Пајтон и састоји се из следећих фајлова:

- routerNode.py
- validatorNode.py
- gamblerNode.py
- routerApp.py
- validatorApp.py
- gamblerApp.py
- node.py
- nodeconnection.py

Оно што је потребно напоменути јесте да су фајлови node.py и nodeconnection.py дио уграђене Пајтон библиотеке p2pnetwork која може бити инсталирана употребом pip алата на следећи начин:

pip install p2pnetwork

Међутим, због потребе пројекта да се пар ситница измјени у поменутим фајловима нисам користио инсталирани пакет него сам фајлове пронашао и преузео са следеће github странице: https://github.com/macsnoeren/python-p2p-network и исправио жељене дијелове. Класа из фајла node.py увезена је у остале фајлове гдје је наслијеђена од стране класа имплементираних за потребе наше апликације. Садржај датотека node.py и nodeconnection.py неће бити детаљно обрађен у овој документацији јер је то дио неког другог пројекта чији се дијелови овде користе ради једноставније имплементације.

4.1. Датотека routerNode.py

```
from datetime import datetime
import socket
import sys
import time
from node import Node
from random import randint
import ast

class RouterNode(Node):
    def __init__(self, host, port, id=None, callback=None, max_connections=0):

        self.arrayOfConnected = []
        self.dictOfConnected = {}
```

```
        super(RouterNode, self).__init__(host, port, id, callback, max_connections)
        print("RouterNode: Started")

    def inbound_node_connected(self, node):
        print("inbound_node_connected: (" + str(self.port) + "): " + str(node.port))

    def inbound_node_disconnected(self, node):
        print("inbound_node_disconnected: (" + str(self.port) + "): " + str(node.port))

    def node_message(self, node, data):
        print("node_message (" + str(self.port) + ") from " + str(node.port) + ": " + s
tr(data))

        message = str(data)
        address = (node.host, int(node.port))

        if message == 'connect':
            if address not in self.arrayOfConnected:
                self.arrayOfConnected.append(address)
                self.dictOfConnected[address] = int(datetime.timestamp(datetime.now()))
                print("Address: "+str(address)+" connected")
        elif message == 'connectGambler':
            print("Address gambler: "+str(address)+" connected")
        elif message == 'disconnect':
            self.arrayOfConnected.remove(address)
            del self.dictOfConnected[address]
            print("Address: "+str(address)+" disconnected")
        elif message == 'getNodes':
            if len(self.arrayOfConnected)==0:
                strToSend = 'nodes:nothing'
            else:
                strToSend = 'nodes:'+str(self.arrayOfConnected)
            self.send_to_node(node, strToSend)
        elif 'checkValidators:' in message[:16]:
            arrToCheck = ast.literal_eval(message[16:])
            ok = True
            for i in arrToCheck:
                if i not in self.arrayOfConnected:
                    ok = False
                    break
            self.send_to_node(node, 'ok:'+str(ok))
        elif message == 'hi':
            self.dictOfConnected[address] = int(datetime.timestamp(datetime.now()))
            print("Address: "+str(address)+" said hi")
```

```
    def check_timestamps(self):
        for i in self.dictOfConnected.copy():
            if (int(datetime.timestamp(datetime.now())) -
self.dictOfConnected[i]) > 300:
                self.arrayOfConnected.remove(i)
                del self.dictOfConnected[i]
                print("Address: "+str(i)+" disconnected because of timestamp")
```

Датотека routerNode.py садржи имплементацију класе RouterNode која наслиједује класу Node из node.py датотеке. При иницијализацији класе RouterNode формира се низ у који ће бити смјештене адресе свих активних валидатора у мрежи и један ријечник података (објекат типа dict) у коме ће уз сваку адресу активног валидатора бити вриједност timestamp-а када се он последњи пут јавио на мрежу.

Функције `inbound_node_connected, и inbound_node_disconnected` служе за исписивање података о чвору који се јавља или прекида везу.

Функција node_message је редефинисана и позива се када стигне порука било ког типа. Као параметре прима конекцију чвора који је послао поруку и саму поруку.

- Порука „connect“ значи да пошиљалац захтјева придруживање мрежи валидатора и његову адресу ће рутер чвор убацити у низ активних валидатора као и у ријечник при чему му додијељује timestamp који му говори када се повезао.
- Порука „connectGambler“ то значи да неки „коцкар“ чвор захтјева приступ списку активних валидатора како би се повезао у мрежу, и његова адреса се не уврштава у списак активних валидатора пошто он то није, него се само исписује податак о његовом јављању у мрежу.
- Порука „disconnect“ је сигнал рутер чвору да уклони из листе активних валидатора адресу онога који је ту поруку послао.
- Порука „getNodes“ значи да треба послати низ активних валидатора ономе ко шаље ту поруку тј. ко иницира повезивање у p2p мрежу.
- Порука која почиње са „checkValidators:“ у наставку има низ адреса валидатора на које је пошиљалац повезан и то је сигнал рутер чвору да све адресе валидатора из тог примљеног низа треба провјерити да ли су још увијек активни. Ако јесу шаље поруку „ok:True“, а ако нису „ok:False“
- Порука „hi“ значи да се неки од валидатора јавља као још увијек активан и потребно је промијенити његов timestamp на тренутак његовог јављања.

Функција check_timestamps служи за провјеру да ли је прошло више од 5 минута тј. 300 секунди од јављања неког од валидатора. Ако јесте он се уклања из низа активних валидатора. Провјера се врши тако што се одузме тренутни timestamp од timastamp-а валидатора за кога се врши провјера и ако је разлика већа од 300 то значи да тај валидатор више није активан. Иначе, треба напоменути да је за сваког валидатора програмирано да се јавља рутер чвору сваких 120 секунди.

4.2. Датотека „routerApp.py“

```
from routerNode import RouterNode
from datetime import datetime
import socket
import sys
import time
from random import randint
import ast

print("Access http://localhost:9000")
routerNode = RouterNode('127.0.0.1', 9000, 'RouterNode: '+str(100))
routerNode.start()
while True:
    if len(routerNode.arrayOfConnected) == 0:
        time.sleep(1)
        continue
    else:
        print('\nValidators in the network:')
        for i in routerNode.arrayOfConnected:
            print(i)
        time.sleep(5)
        routerNode.check_timestamps()
```

На почетку се увози класа RouterNode из датотеке routerNode.py што значи да ће бити имплементиран рутер чвор. Даље се формира објекат класе RouterNode чиме се инстанцира рутер чвор и то на адреси локалног сервера 127.0.0.1, порту 9000, са идентификатором RouterNode:100.

Напомена: Комуникација између чворова тестирана је само на локалном серверу и сви покренути чворови имали су исту адресу хоста 127.0.0.1, али различите портове што значи да су при тестирању апликације портови кључни за адресирање порука у имплементираној мрежи.

Након стварања објекта класе RouterNode рутер чвор се покреће и поред својих главних функционалности примања и слања порука приказаних у опису датотеке

„ruoterNode.py“ дешава се и испис свих тренутно активних валидатора све док је рутер чвор активан, као и провјера активности валидатора сваких 5 секунди.

4.3. Датотека „validatorNode.py“

Због обимности датотеке и касније прегледности описа функционалности, образлагање ће ићи редом и постепено после одређених дијелова кода.

Неки закоментарсани дијелови кода неће бити приказани у опису јер служе као помоћ при тестирању функционалности кода.

```
import socket
import sys
import time
from node import Node
from random import randint
from datetime import datetime
from Crypto.PublicKey import RSA
from Crypto.Hash import SHA256
from Crypto.Signature.pkcs1_15 import PKCS115_SigScheme
import binascii
import os
import json
import ast

class ValidatorNode(Node):
    def __init__(self, host, port, id=None, callback=None, max_connections=0):
        super(ValidatorNode, self).__init__(host, port, id, callback, max_connections)
        print("ValidatorNode: Started")
        self.arrNodes = []
        self.blockchain = {}
        self.tempArrOfHash = []
        self.signalIfWaited = 0
        self.blockchainFileName = ''
        self.sentRequest = 0
        self.receivedBets = []
        self.validBlock = None
        self.allBlockchainBets = []
        self.addedBlock = 0
        self.downloadedBlockchain = 0
```

У овој датотеци имплементирана је класа ValidatorNode која наслиједјује такође Node класу и представља имплементацију валидатор чвора у мрежи.

При иницијализацији класе формирају се следеће промјењиве а то су:

- „arrNodes“ којој се додјељује вриједност празног низа и у њега ће бити смјешетене адресе валидатор чворова које су примљене од рутер чвора и од којих ће се насумично изабрати три за повезивање на њих.
- „blockchain“ којој се додијељује вриједност празног ријечника података и у коју ће бити смјештен blockchain преузет од једног од повезаних валидатор чворова.
- „tempArrOfHash“ у коју ће бити смјештене хеш вриједности последњег блока које се примају од повезаних валидатор чворова, а то се дешава прије преузимања blockchain-а. Разлог за слање тих хеш вриједности јесте тај да се иста хеш вриједност мора добити од три повезана валидатор чвора како бисмо били сигурнији да сви имају идентичан blockchain и да би се више заштитили ако неки чвор посједује погрешну или застарјелу вриједност blockchain-а. Ако се од 3 примљене хеш вриједности једна разликује врши се повезивање на друге валидаторе и тако све док се не добију три исте хеш вриједности.
- „signalIfWaited“ којој је додијељена вриједност 0 и мијења се у 1 ако је валидатор чвор био један од 4 иницијална, а служи да се у „blockchain“ промјенљиву смјести иницијална вриједност blockchain-а а не да се она преузима од повезаних валидатора.
- „blockchainFileName“ која ће заправо бити стринг са именом фајла у који је сачуван blockchain
- „sentRequest“ која је иницијално 0 али се мијења на 1 ако је послат захтјев за слање blockchain-а
- „receivedBets“ којој се на почетку додијељује вриједност празног низа, а у њему ће касније бити примљене опкладе које треба да се угради у блок
- „validBlock“ промјенљива у којој ће бити смјештан блок за уградњу у ланац тј. онај блок чија је хеш вриједност најмања од до тада примљених и провјерених блокова.
- „allBlockchainBets“ је заправо низ у коме ће бити смјештене све опкладе које постоје до тада у blockchain-у
- „addedBlock“ је заправо сигнал промјенљива која има вриједност 0 ако у току једне партије још није додат блок, а 1 ако јесте. Већ у следећој партији/кругу се вриједност враћа на 0, а служи да се игноришу поруке које носе блок ако је он већ додат у ланац у тој партији.
- „downloadedBlokchian“ је такође сигнал промјенљива има вриједност 0 ако blockchain још није преузет, а 1 ако јесте. Служи за то да се не примају опкладе док се не преузме blockchain

```
    def node_message(self, node, data):
        #print("node_message (" + str(self.port) + ") from " + str(node.port) + ": " +
str(data))
        if (node.host == '127.0.0.1') and (node.port == 9000):
            if 'nodes:' in data[:6]:
                try:
                    nodesArray = ast.literal_eval(data[6:])
                    #print(nodesArray)
                    self.arrNodes = nodesArray
                except:
                    print("Invalid message: "+data[6:])
            elif 'ok:' in data[:3]:
                if data[3:] == 'False':
                    self.reconnect_with_peers()
        elif data == 'sendBlockchainHash':
            #if (not self.time_for_wait_connection()):
            if self.signalIfWaited:
                f = open('blockchain_initial.json', 'r')
            else:
                f = open(self.blockchainFileName, 'r')
            blockchainToSendHash = str(list(json.load(f))[-1])
            f.close()
            self.send_to_node(node, "blckHash:"+str(blockchainToSendHash))
        elif 'blckHash:' in data[:9]:
            self.tempArrOfHash.append(data)
            if (len(self.tempArrOfHash) >= 3) and (self.sentRequest == 0):
                if self.tempArrOfHash.count(data) >= 3:
                    self.send_to_node(node, 'sendBlockchain')
                    time.sleep(1)
                    self.send_to_node(node, 'sendBets')
                    self.sentRequest = 1
                else:
                    self.tempArrOfHash = []
                    self.reconnect_with_peers()
                    for i in self.peers_validators():
                        self.send_to_node(i, 'sendBlockchainHash')
                        time.sleep(0.3)
        elif data == 'sendBlockchain':
            if (datetime.timestamp(datetime.now()) % 120 < 55):
                if self.signalIfWaited:
                    f = open('blockchain_initial.json', 'r')
                else:
                    f = open(self.blockchainFileName, 'r')
                blockchainToSend = json.load(f)
                f.close()
```

```
                self.send_to_node(node, str(blockchainToSend))
        elif data == 'sendBets':
            listBetsWithoutDups = [i for n, i in enumerate(self.receivedBets) if i not
in self.receivedBets[:n]]
            self.send_to_node(node, 'bets:'+str(listBetsWithoutDups))
        elif 'bets:' in data[:5]:
            try:
                self.receivedBets = ast.literal_eval(data[5:])
                dBets = self.receivedBets
                for i in dBets:
                    if (not self.valid_bet(i)):
                        self.receivedBets.remove(i)
                    else:
                        print('Received valid bet: ' + str(list(i.keys())[0]))
            except:
                print("Invalid bets message: " + str(data))
        elif 'bet:' in data[:4]:
            if (self.time_for_listen_bets() and self.downloadedBlockchain):
                try:
                    receivedBet = ast.literal_eval(data[4:])
                    if (receivedBet not in self.receivedBets) and (self.valid_bet(recei
vedBet)):
                        print('Received bet: '+str(list(receivedBet.keys())[0]))
                        self.receivedBets.append(receivedBet)
                        self.send_to_validators(data)
                    #else:
                        #print('\n\n\nBet is invalid or already in list\n\n\n')
                except:
                    print("Invalid bet message: " + str(data))
        elif 'block:' in data[:6]:
            if (not self.addedBlock) and (self.time_for_listen_blocks()):
                try:
                    receivedBlock = ast.literal_eval(data[6:])
                    if self.valid_block(receivedBlock):
                        print('Received block: '+str(list(receivedBlock.keys())[0]))
                        if self.validBlock == None:
                            self.validBlock = receivedBlock
                            #print('\n\n\n'+str(self.validBlock)+'\n\n\n')
                            time.sleep(1)
                            self.send_to_validators('block:'+str(receivedBlock))
                        elif list(receivedBlock.keys())[0] < list(self.validBlock.keys(
))[0]:
                            self.validBlock = receivedBlock
                            #print('\n\n\n'+str(self.validBlock)+'\n\n\n')
                            time.sleep(1)
```

```
                    self.send_to_validators('block:'+str(receivedBlock))
                #else:
                    #print('\n\n\nReceived invalid block\n\n\n')
            except:
                print("Invalid block message: " + str(data))
        elif type(ast.literal_eval(data)) == dict:
            self.download_blockchain_file(ast.literal_eval(data))
            self.sentRequest = 0
            self.signalIfWaited = 0
```

Функција node_message је редефинисана и позива се када стигне порука било ког типа. Као параметре прима конекцију чвора који је послао поруку и саму поруку.

- Ако је адреса пошиљаоца 127.0.0.1:9000 то значи да је стигла порука од рутер чвора и постоје двије врсте порука које се примају од њега. Прва која почиње са „nodes:“ а у наставку има листу активних валидатора, и друга која може бити „ok:True“ ако су валидатори на које смо повезани још увијек активни или „ok:False“ ако нису и у том случају врши се понвно повезивање на чворове.
- Порука „sendBlockchainHash“ значи да требамо послати хеш последњег блока у blockchain-у који је уједно и хеш читавог blockchain-а.
- Порука која садржи „blckHash:“ у наставку садржи хеш последњег блока у blockchain-у који нам шаље валидатор након што смо му упутили захтјев поруком „sendBlockchainHash“. Разлог за слање тих хеш вриједности јесте тај да се иста хеш вриједност мора добити од три повезана валидатор чвора како бисмо били сигурнији да сви имају идентичан blockchain и да би се више заштитили ако неки чвор посједује погрешну или застарјелу вриједност blockchain-а. Ако се од 3 примљене хеш вриједности једна разликује врши се повезивање на друге валидатора и тако све док се не добију три исте хеш вриједности.
- Порука „sendBlockchain“ је индикатор да требамо послати последњу верзију blockchain-а пошиљаоцу те поруке.
- Порука „sendBets“ је индикатор да се нови валидатор појавио у мрежи и да тражи до тада примљене опкладе у том кругу како би генерисао исправну вриједност блока.
- Порука која седржи „bets:“ у наставку садржи листу опклада коју нам шаље повезани валидатор након што смо му поруком „sendBets“ упутили захтјев. Опкладе из те листе се провјеравају да ли су исправне и оне које јесу додају се на властити списак опклада.
- Порука која садржи „bet:“ у наставку садржи вријадност једне опкладе и под условом да је вријеме за слушање опклада, да је опклада валидна и да је већ немамо на списку за тај круг, она се додаје на списак.

- Порука која садржи „block:“ у наставку садржи вриједност блока и под условом да је вријеме за слушање блокова, да је блок валидан и да блок већ није додат у ланац у том кругу, провјерава се да ли је хеш блока мањи од до тада најмање хеш вриједности примљеног блока и ако јесте тај новопримљени блок постаје кандидат за додавање у blockchain (ланац блокова).
- Порука која је читава типа dict (ријечник података) је заправо blockchain који нам шаље повезани валидатор након што смо му упутили захтјев поруком „sendBlockchain“.

```
    def download_blockchain_file(self, dct):
        self.blockchain = dct
        print("\nBlockchain file will be located in directory "+os.getcwd()+".\nDON'T r
eplace or remove it!")
        currTimestamp = int(datetime.timestamp(datetime.now()))
        self.blockchainFileName = "blockchain_"+str(self.port)+"_"+str(currTimestamp)+"
.json"
        print("Downloaded blockchain in file with name " + self.blockchainFileName+"\n"
)
        f = open(self.blockchainFileName, 'w')
        json.dump(dct, f)
        f.close()
        self.allBlockchainBets = self.list_of_bets_in_blockchain()
        self.downloadedBlockchain = 1
```

Функција download_bockchain_file као параметар прима ријечник података и формира се .json фајл чији ће назив имати формат „blockchian_назив порта_тренутни timestamp“ и у њега ће бити уграђен објекат типа dict тј. ријечник података прослијеђен као аргумент. Ова функција се позива кад се од једног од повезаних валидатора прими вриједност blockchain-а. У овој функцији се такође подешавају вриједности промјенљивих „self.allBlokchainBets“ и „self.downloadedBlockchain“.

```
    def add_block_to_blockchain(self):
        print('\n\nStarted adding block')
        print('\nHash of valid block for adding in blockchain is:\n'+str(list(self.vali
dBlock.keys())[0])+'\n\n')
        blockHash = list(self.validBlock.keys())[0]
        blockData = self.validBlock[blockHash]
        self.blockchain[blockHash] = blockData
        f = open(self.blockchainFileName, "w")
        json.dump(self.blockchain, f)
```

```
        f.close()
        self.allBlockchainBets = self.list_of_bets_in_blockchain()
        #self.copiedValidBlock = self.validBlock.copy()
        self.validBlock = None
        #print('\n\n\n'+str(self.validBlock)+'\n\n\n')
        self.receivedBets = []
        self.addedBlock = 1
```

Функција „add_block_to_blockchain“ ажурира вриједност промјенљиве „blockchain“ тако што се у њу додаје блок и такође се ажурира садржај .json фајла са blockchain-ом тако што се у њега уграђује нова вриједност промјенљиве „blockchain“. Такође се ажурира промјенљива „allBlockchinBets“, „validBlock“ се враћа на None вриједност да би се припремили за следећи круг, „receivedBets“ постаје опет празан низ како не би остале опкладе из претходног круга, а „addedBlock“ се сетује на 1 јер је блок управо додат у овом кругу.

```
    def init_connect_to_nodes(self, host, port):
        nodesIndicesToConnect = []
        i = 3
        while (i > 0):
            index = randint(0, len(self.arrNodes)-1)
            if (index not in nodesIndicesToConnect) and (index != self.arrNodes.index((
host, port))):
                nodesIndicesToConnect.append(index)
                i = i - 1
            else:
                continue
        nodesToConnect = []
        for i in nodesIndicesToConnect:
            nodesToConnect.append(self.arrNodes[i])
        self.nodes_inbound = []
        self.nodes_outbound = []
        for i in nodesToConnect:
            self.connect_with_node(i[0],i[1])
```

Функција „init_connect_to_nodes“ као параметре прима властиту хост и порт адресу и служи да се из низа примљених адреса активних валидатора насумично издвоје три и на њих да се повежемо и тако постанемо чвор p2p мреже. Прослијеђени аргументи служе да се не бисмо случајним одабиром покушали повезати на властиту адресу што не би било изводљиво.

```
def join_p2p(self):
    self.connect_with_node('127.0.0.1', 9000)

    sNode = None
    for i in self.nodes_outbound:
        if i.host=='127.0.0.1' and i.port==9000:
            sNode = i
            break
    self.send_to_node(sNode, 'connect')
    #time.sleep(2)
    self.send_to_node(sNode, 'getNodes')
    while (len(self.arrNodes)<=3):
        self.signalIfWaited = 1
        self.send_to_node(sNode, 'getNodes')
        time.sleep(2)

    self.disconnect_with_node(sNode)
    self.init_connect_to_nodes(self.host, self.port)
    time.sleep(2)
```

Функција „join_p2p“ како јој име каже служи да би се придружили p2p мрежи. Прво се јављамо рутер чвору са поруком „connect“, а затим и „getNodes“ како бисно добили адресе активних валидатора на које је могуће повезати се (значење ових порука објашњено је изнад када је обрађен фајл „routerNode.py“), онда прекинамо везу са рутер чвором, а затим се позива претходно објашњена функција „init_connect_to_nodes“ чиме се придружујемо p2p мрежи.

```
def check_connected_validators(self):
    if len(self.nodes_inbound) + len(self.nodes_outbound) < 3:
        self.reconnect_with_peers()
    arrToSend = []
    for n in self.nodes_outbound:
        arrToSend.append((str(n.host), int(n.port)))

    self.connect_with_node('127.0.0.1', 9000)

    sNode = None
    for i in self.nodes_outbound:
        if i.host=='127.0.0.1' and i.port==9000:
            sNode = i
```

```
                break
        self.send_to_node(sNode, 'checkValidators:'+str(arrToSend))
        time.sleep(1)
        self.send_to_node(sNode, 'hi') #say hi to main node every 2 min
        self.disconnect_with_node(sNode)
```

Функција „check_connected_validators“ служи да провјеримо да ли су валидатори са којима смо повезани још увијек активни. Прво се провјерава да ли постоји веза са три или више чворова, и ако не постоји поново се повезујемо. Затим шаљемо рутер чвору листу адреса валидатора на које смо повезани да би рутер чвор провјерио да ли су још увијек активни на мрежи и враћа нам поруку о томе да ли јесу или нису. Ако јесу примићемо поруку „ok:True“, а ако нису „ok:False“ и тада ћемо се поново повезати на нове валидаторе као што је претходно објашњено када смо говорили о порукама. Такође се ка рутер чвору шаље „hi“ порука како би нас завео као још увијек активне на мрежи.

```
    def valid_bet(self, bet):
        betHashStr = list(bet.keys())[0]

        currTimestamp = datetime.timestamp(datetime.now())

        strBetPK = bet[betHashStr]['gamblerPK']
        betPK = RSA.importKey(strBetPK.encode())

        numForProbability = bet[betHashStr]['numForProbability']
        sequenceChoice = bet[betHashStr]['sequenceChoice']

        betTimestamp = bet[betHashStr]['betTimestamp']

        betSignature = bet[betHashStr]['betSignature']
        betSgn = binascii.unhexlify(betSignature.encode())

        hashOfBet = SHA256.new((str(strBetPK)+str(numForProbability)+str(sequenceChoice
)+str(betTimestamp)+str(betSignature)).encode()).hexdigest()

        hashOfBetData = SHA256.new((str(strBetPK)+str(numForProbability)+str(sequenceCh
oice)+str(betTimestamp)).encode())

        betVerifier = PKCS115_SigScheme(betPK)

        prevBlockTimestamp = float(self.blockchain[list(self.blockchain.keys())[-
1]]['blockTimestamp'])
```

```
        if (float(betTimestamp) < prevBlockTimestamp):
            return 0

        if currTimestamp <= float(betTimestamp):
            return 0

        if (betHashStr != hashOfBet):
            return 0

        for b in self.allBlockchainBets:
            if b == betHashStr:
                return 0
        try:
            betVerifier.verify(hashOfBetData, betSgn)
            return 1
        except:
            return 0
```

Функција „valid_bet“ прима као параметар опкладу и враћа 1 ако је опклада исправна, а 0 ако није. Исправност опкладе се провјерава на основу више критеријума:

- Ако је timestamp опкладе мањи од timestamp-а последњег блока у blockchain-u (тј. ако је опклада млађа од последњег блока у ланцу) тада опклада неће бити валидна.
- Ако је тренутни timestamp мањи од timestamp-а опкладе (што би значило да је опклада генерисана „у будућности“) тада опклада неће бити валидна.
- Ако је хеш вриједност која је прослијеђена уз опкладу различита од хеш вриједности над стварним подацима опкладе тада опклада неће бити валидна.
- Ако се хеш вриједност опкладе поклапа са неком опкладом која већ постоји у blockchain-у тада опклада неће бити валидна.
- Ако се потпис опкладе не поклапа са јавним кључем у опклади и подацима опкладе тада опклада неће бити валидна.
- У супротном опклада је валидна.

```
    def make_block(self, keyPair):

        pk = keyPair.publickey().exportKey().decode()

        currTimestamp = datetime.timestamp(datetime.now())

        prevBlockHash = list(self.blockchain.keys())[-1]
```

```
        bets = [i for n, i in enumerate(self.receivedBets) if i not in self.receivedBet
s[:n]]

        myBlockValue = {}
        myBlockValue['validatorPK'] = pk
        myBlockValue['blockTimestamp'] = currTimestamp
        myBlockValue['prevBlockHash'] = prevBlockHash
        myBlockValue['bets'] = bets

        hashToSign = SHA256.new((str(pk)+str(currTimestamp)+str(prevBlockHash)+str(bets
)).encode())

        signer = PKCS115_SigScheme(keyPair)

        blockSignature = signer.sign(hashToSign)
        strBlockSgn = binascii.hexlify(blockSignature).decode()

        myBlockValue['blockSignature'] = strBlockSgn

        hashOfMyBlock = SHA256.new((str(pk)+str(currTimestamp)+str(prevBlockHash)+str(b
ets)+str(strBlockSgn)).encode()).hexdigest()

        myBlock = {hashOfMyBlock:myBlockValue}

        return myBlock
```

Функција „make_block“ служи да се генерише блок на крају слушања опклада и као параметар прослиједјује се пар тајни/јавни кључ јер је потребно потписати блок. У блок се уграђују следећи подаци:

1. Хеш вриједност блока која настаје конкатенирањем свих даље наведених података у блоку и позивањем хеш функције sha_256 над тако конкатенираним подацима
2. Јавни кључ валидатора тј. онога ко генерише блок
3. Тренутак у коме је блок генерисан (тренутни timestamp)
4. Хеш вриједност претходног блока
5. Листу опклада за које је валидатор који генерише блок чуо у протеклом интервалу слушања опклада
6. Потпис блока тајним кључем гдје се потписује порука настала као излаз хеш функције sha_256 чији су улаз конкатенирани претходни подаци под бројем 2, 3, 4 и 5.

Излаз функције је објекат типа dict који садржи један кључ:вриједност податак, гдје је кључ хеш блока, а вриједност је такође објекат типа dict који садржи вриједности претходно излистане под бројевима 2, 3, 4, 5 и 6 чији су кључеви стрингови који говоре која врста вриједности се чува.

```
    def send_block_to_peers(self, blockDict):

        strToSend = 'block:'+str(blockDict)
        self.send_to_validators(strToSend)
```

Функција „send_block_to_peers“ прима као параметар објекат типа dict који представља блок и обично је тај параметар излаз функције „make_block“. Функција „send_block_to_peers“ служи да пошаљемо генерисани блок ка повезаним валидатор чворовима.

```
    def valid_block(self, block):

        prevBlockHash = list(self.blockchain.keys())[-1]

        currTimestamp = datetime.timestamp(datetime.now())

        prevBlockTimestamp = float(self.blockchain[list(self.blockchain.keys())[-
1]]['blockTimestamp'])

        blockHashStr = list(block.keys())[0]

        pk = block[blockHashStr]['validatorPK']
        blockOwnerPK = RSA.importKey(pk.encode())

        blockTimestamp = block[blockHashStr]['blockTimestamp']

        blockPrevHash = block[blockHashStr]['prevBlockHash']

        blockBets = block[blockHashStr]['bets']

        blockSignature = block[blockHashStr]['blockSignature']
        blockSgn = binascii.unhexlify(blockSignature.encode())

        hashToValidateBlock = SHA256.new((str(pk)+str(blockTimestamp)+str(blockPrevHash
)+str(blockBets)+str(blockSignature)).encode()).hexdigest()
```

```
        hashToValidateSignature = SHA256.new((str(pk)+str(blockTimestamp)+str(blockPrev
Hash)+str(blockBets)).encode())

        blockVerifier = PKCS115_SigScheme(blockOwnerPK)

        if (currTimestamp <= float(blockTimestamp)) or (float(blockTimestamp) <= float(
prevBlockTimestamp)):
            return 0

        if (hashToValidateBlock != blockHashStr):
            return 0

        listOfBets = blockBets

        for i in listOfBets:
            betTs = float(i[list(i.keys())[0]]['betTimestamp'])
            if (not self.valid_bet(i)) or (betTs > float(blockTimestamp)) or (betTs > c
urrTimestamp):
                return 0

        hashBetsRecv = []
        hashBetsFromBlock = []
        listOfBetsWithoutDups = [i for n, i in enumerate(self.receivedBets) if i not in
 self.receivedBets[:n]]
        for i in listOfBetsWithoutDups:
            hashBetsRecv.append(list(i.keys())[0])
        for j in listOfBets:
            hashBetsFromBlock.append(list(j.keys())[0])

        if sorted(hashBetsRecv) != sorted(hashBetsFromBlock):
            return 0

        try:
            blockVerifier.verify(hashToValidateSignature, blockSgn)
            return 1
        except:
            return 0
```

Функција „valid_block“ служи да се провјери исправност блока, а као параметар прима блок који чији је формат исти као излаз функције „make_block“. Функција враћа 1 ако је блок исправан, а 0 ако није. Исправност опкладе се провјерава на основу више критеријума:

- Ако је timestamp блока мањи од timestamp-а последњег блока у blockchain-u (тј. ако је блок млађи од последњег блока у ланцу) тада блок неће бити валидан.
- Ако је тренутни timestamp мањи од timestamp-а блока (што би значило да је блок генерисан „у будућности“) тада блок неће бити валидан.
- Ако је хеш вриједност која је прослијеђена уз блок различита од хеш вриједности над стварним подацима блока тада блок неће бити валидан.
- Ако се опкладе у блоку који се провјерава разликују од опклада за које је валидатор који провјерава блок чуо у протеклом интервалу слушања опклада, тада блок неће бити валидан јер се опкладе морају поклопити.
- Ако се потпис блока не поклапа са јавним кључем онога ко је генерисао блок и подацима блока тада блок неће бити валидан.
- У супротном блок ће бити валидан

```
def list_of_bets_in_blockchain(self):
    lst = []
    for block in self.blockchain:
        bData = self.blockchain[block]
        lstBlockBets = bData['bets']
        for bt in lstBlockBets:
            lst.append(list(bt.keys())[0])
    return lst
```

Функција „list_of_bets_in_blockchain“ враћа низ свих опклада које се налазе у најновијој верзији blockchain-а.

```
def leave_p2p(self):
    self.connect_with_node('127.0.0.1', 9000)
    sNode = None
    for i in self.nodes_outbound:
        if i.host=='127.0.0.1' and i.port==9000:
            sNode = i
            break
    self.send_to_node(sNode, 'disconnect')
    self.disconnect_with_node(sNode)
    for i in self.nodes_outbound:
        self.disconnect_with_node(i)
```

Функција „leave_p2p“ се позива ако желимо да нестанемо из мреже. То се постиже тако што се повезујемо на рутер чвор и шаљемо му захтјев (поруку „disconnect“) да нас уклони из листе активних валидатора. Такође, прекидамо везу са свим валидаторима на које смо се повезали.

```
def peers_validators(self):
    peersValidators = []
    for n in self.nodes_inbound:
        if (n not in peersValidators) and ('validator:' in str(n.id)[:10]):
            peersValidators.append(n)
    for n in self.nodes_outbound:
        if (n not in peersValidators) and ('validator:' in str(n.id)[:10]):
            peersValidators.append(n)

    arr = []
    for i in peersValidators:
        arr.append((i.host, i.port, i.id))

    arrWithoutDups = [j for k, j in enumerate(arr) if j not in arr[:k]]

    listWithoutDups = []
    for n in arrWithoutDups:
        for m in peersValidators:
            if m.host == n[0] and m.port == n[1] and m.id == n[2]:
                listWithoutDups.append(m)
                break

    return listWithoutDups
```

Функција „peers_validators“ враћа листу конекција које представљају валидатор чворове на које смо повезани или који су повезани на нас и то без дупликата.

```
def send_to_validators(self, data):

    for node in self.peers_validators():
        self.send_to_node(node, data)
```

Функција „send_to_validators“ као параметар прима податак који треба послати, а служи да се пошаље податак свим валидатор чворовима са којима имамо конекцију.

```
    def reconnect_with_peers(self):
        self.signalIfWaited = 0
        self.arrNodes = []

        for i in self.nodes_outbound:
            self.disconnect_with_node(i)

        self.connect_with_node('127.0.0.1', 9000)

        sNode = None
        for i in self.nodes_outbound:
            if i.host=='127.0.0.1' and i.port==9000:
                sNode = i
                break
        self.send_to_node(sNode, 'getNodes')
        while (len(self.arrNodes)<=3):
            self.signalIfWaited = 1
            self.send_to_node(sNode, 'getNodes')
            time.sleep(2)

        self.disconnect_with_node(sNode)
        self.init_connect_to_nodes(self.host, self.port)
```

Функција „reconnect_with_peers“ служи да се повежемо на друге валидаторе. То се постиже тако што прво прекинемо везу са постојећим повезаним валидаторима, затим се повезујемо на рутер чвор и тражимо листу активних валидатора. Затим прекидамо везу са путер чвором и позивамо функцију „init_connect_to_nodes“ како би наш чвор повезали са новим валидатор чворовима.

```
    def time_for_wait_connection(self):

        return (datetime.timestamp(datetime.now()) % 120 > 50)
```

Функција „time_for_wait_connection“ враћа True ако је у тренутном кругу прошло 50 секунди и даје нам тиме сигнал да требамо чекати нови круг да би се повезали у мрежу.

```
    def time_for_listen_bets(self):

        return (datetime.timestamp(datetime.now()) % 120 < 80)
```

Функција „time_for_listen_bets“ враћа True ако је у току период првих 80 секунди текућег круга када можемо слушати опкладе које долазе са мреже, у супротном враћа False.

```
    def time_for_listen_blocks(self):

        return ((datetime.timestamp(datetime.now()) % 120 < 120) and (datetime.timestam
p(datetime.now()) % 120 > 80))
```

Функција „time_for_listen_blocks“ враћа True ако је у току период од 80 до 120 секунди текућег круга када можемо слушати блокове које долазе са мреже, у супротном враћа False.

4.4. Датотека „validatorApp.py“

Ова датотека представља имплементацију свега из „validatorNode.py“ датотеке и представља програм који се покреће када желимо да покренемо валидатор чвор. Опис ћемо подијелити у двије цјелине.

1. цјелина)

```
from validatorNode import ValidatorNode
import socket
import sys
import time
from random import randint
from datetime import datetime
from Crypto.PublicKey import RSA
from Crypto.Hash import SHA256
from Crypto.Signature.pkcs1_15 import PKCS115_SigScheme
import binascii
import os
import json
import ast

host = str(sys.argv[1])
port = int(sys.argv[2])

while True:
```

```
    print("Do you want to generate new key pair? [y/n]\n(If answer is yes, program will
 create key in .pem file at location of your choice.\nIf answer is no, it means that yo
u already have the key generated by this application and you have to load it.)")
    answer1 = input("Answer: ")
    if answer1 in ['y', 'Y', 'yes']:
        while True:
            print("\nInput directory path where key file will be located:")
            currTimestamp = int(datetime.timestamp(datetime.now()))
            keyPath = input("Path: ")
            try:
                f = open(keyPath+'/mykey_'+str(port)+'_'+str(currTimestamp)+'.pem', 'wb
')
            except:
                print("Invalid path! Try again...")
                continue
            break
        key = RSA.generate(1024)
        f.write(key.exportKey('PEM'))
        f.close()
        print('Only PK:\n'+str(key.publickey().exportKey().decode())+'\n')
        break
    elif answer1 in ['n', 'N', 'no']:
        while True:
            print("\nInput correct name of .pem key file with entire path:")
            keyPath = input("Path: ")
            try:
                f = open(keyPath, 'r')
                key = RSA.importKey(f.read())
            except:
                print("Invalid path or file! Try again...")
                continue
            break
        f.close()
        print('Only PK:\n'+str(key.publickey().exportKey().decode())+'\n')
        break
    else:
        print("Incorrect answer! Try again...\n")
        continue
```

Прво требамо да знамо да се валидатор чвор покреће тако што укуцамо нпр:

python validatorApp.py 127.0.0.1 8000

гдје је потребно да нам први аргумент буде хост адреса, а други порт адреса и сходно томе ти аргументи постају вриједности промјенљивих „host“ и „port“ и служиће да се додјели адреса валидатор чвору када га покренемо.

Затим се улази у петљу у којој нам се поставља питање да ли желимо да генеришемо нови пар тајни/јавни кључ. Ако потврдно одговоримо програм ће нас питати да унесемо путању до директоријума у који ће смјестити кључ креираће 1024-битни кључ коришћењем RSA алгоритма у .pem фајлу и ако га сачувамо можемо га користити сваки следећи пут. Ако је одговор не, то значи да већ имамо генерисан кључ и програм ће нас питати да унесемо читаву путању до кључа заједно са називом .pem фајла у коме се кључ налази. Затим се из пара тајни/јавни кључ који је претходно генерисан/учитан издваја и исписује само јавни кључ.

2. цјелина) наставак кода:

```
myKeyPair = key
myPublicKey = key.publickey()
pkHash = SHA256.new(myPublicKey.exportKey()).hexdigest()
myID = 'validator:'+str(pkHash)
myValidatorNode = ValidatorNode(host, port, myID)
myValidatorNode.start()

#if myValidatorNode.time_for_wait_connection():
print('waiting for connection ...')
while myValidatorNode.time_for_wait_connection():
    #print('waiting for connection ...')
    time.sleep(1)
    continue

myValidatorNode.join_p2p()

print('\nConnected validators:')
for i in myValidatorNode.peers_validators():
    print(str(i.port)+' - '+str(i.id))
print('\n')


try:

    for i in myValidatorNode.peers_validators():
        myValidatorNode.send_to_node(i, 'sendBlockchainHash')
        time.sleep(0.3)
    time.sleep(2)

    while True:
```

```
        time.sleep(1)
        if myValidatorNode.time_for_listen_bets():
            print('listening for bets ...')
        while myValidatorNode.time_for_listen_bets():
            myValidatorNode.addedBlock = 0
            #print('listening for bets ...')
            time.sleep(1)
            continue
        #print('\n\n'+str(myValidatorNode.receivedBets)+'\n\n')
        myValidatorNode.check_connected_validators()
        myBlock = myValidatorNode.make_block(myKeyPair)
        myValidatorNode.send_block_to_peers(myBlock)
        if myValidatorNode.time_for_listen_blocks():
            print('listening for blocks ...')
        while myValidatorNode.time_for_listen_blocks():
            #print('listening for blocks ...')
            time.sleep(1)
            continue
        time.sleep(2)
        myValidatorNode.add_block_to_blockchain()

        continue

except:
    sys.exit(1)
```

У наставку изнад формира се ID који ће бити стринг који се састоји од секвенце „validator:“ послије које слиједи хеш вриједност јавног кључа. Послије тога се ствара објекат класе ValidatorNode и покреће се валидатор чвор. Даље се провјерава да ли је потребно чекати на повезивање и ако јесте чека се и врти се петља док буде вријеме за повезивање и онда се наш чвор преко функције „join_p2p“ повезује у p2p мрежу. Затим се исписују идентификатори валидатора са којима смо се повезали и шаље им се порука „sendBlockchainHash“ чије значење је већ објашњено. Након тога се улази у петљу кроз коју један пролазак значи један круг извлачења који траје 2 минута тј. 120 секунди.

Круг почиње са чекањем опклада и то траје првих 80 секунди тј. док је вријеме за слушање опклада (функција „time_for_listen_bets“), а затим се прелази редом на провјеру активности повезаних валидатора (функција „check_connected_validators“), прављење блока (функција „make_block“) и слање блока валидаторима (функција „send_block_to_peers“). Послеије тога се прелази на слушање блокова (слушање порука формата „block: …“) што траје 40 секунди (од 80. до 120.) и у току тог периода одлучује се чији ће блок бити уграђен у blockchain (а биће уграђен онај блок који је стигао на вријеме, који је валидан и који има најмању хеш вриједност). После истека тог периода додаје се

блок у blockchain(функција „add_block_to_blockchain“) и то онај који је окарактерисан као исправан за додавање и тај исти блок ће бити додат и од стране свих других валидатора ако раде по правилима мреже и сви ће на крају сваког круга имати идентичан blockchain. Након додавања блока прелази се на нови пролазак петље тј. нови круг (поново се чекају опкладе).

4.5. Датотека „gamblerNode.py“

Због обимности датотеке и касније прегледности описа функционалности, образлагање ће ићи редом и постепено после одређених дијелова кода.

Неки закоментарсани дијелови кода неће бити приказани у опису јер служе као помоћ при тестирању функционалности кода.

```
import socket
import sys
import time
from node import Node
from random import randint
from datetime import datetime
from Crypto.PublicKey import RSA
from Crypto.Hash import SHA256
from Crypto.Signature.pkcs1_15 import PKCS115_SigScheme
import binascii
import os
import json
import ast
import math

class GamblerNode(Node):

    def __init__(self, host, port, id=None, callback=None, max_connections=0):
        super(GamblerNode, self).__init__(host, port, id, callback, max_connections)
        print("GamblerNode: Started")
        self.arrNodes = []
        self.blockchain = {}
        self.tempArrOfHash = []
        self.blockchainFileName = ''
        self.sentRequest = 0
```

У овој датотеци имплементирана је класа GamblerNode која наслиједјује такође Node класу и представља имплементацију „коцкар“ чвора у мрежи односно учесника (у клађењу) како ћемо га називати у наставку.

При иницијализацији класе формирају се следеће промјењиве а то су:

- „arrNodes“ којој се додјељује вриједност празног низа и у њега ће бити смјешетене адресе валидатор чворова које су примљене од рутер чвора и од којих ће се насумично изабрати три за повезивање на њих.
- „blockchain“ којој се додијељује вриједност празног ријечника података и у коју ће бити смјештен blockchain преузет од једног од валидатор чворова на које учесник буде повезан.
- „tempArrOfHash“ у коју ће бити смјештене хеш вриједности последњег блока које се примају од повезаних валидатор чворова, а то се дешава прије преузимања blockchain-a. Разлог за слање тих хеш вриједности јесте тај да се иста хеш вриједност мора добити од три повезана валидатор чвора како бисмо били сигурнији да сви имају идентичан blockchain и да би се више заштитили ако неки чвор посједује погрешну или застарјелу вриједност blockchain-a. Ако се од 3 примљене хеш вриједности једна разликује врши се повезивање на друге валидаторе и тако све док се не добију три исте хеш вриједности.
- „blockchainFileName“ која ће заправо бити стринг са именом фајла у који је сачуван blockchain
- „sentRequest“ која је иницијално 0 али се мијења на 1 ако је послат захтјев за слање blockchain-a

```
    def node_message(self, node, data):
        #print("node_message (" + str(self.port) + ") from " + str(node.port) + ": " +
str(data))
        if (node.host == '127.0.0.1') and (node.port == 9000):
            if 'nodes:' in data[:6]:
                #try:
                nodesArray = ast.literal_eval(data[6:])
                #print(nodesArray)
                self.arrNodes = nodesArray
                #except:
                    #print(data[6:])
            elif 'ok:' in data[:3]:
                if data[3:] == 'False':
                    self.reconnect_with_peers()
        elif 'blckHash:' in data[:9]:
            self.tempArrOfHash.append(data)
            if (len(self.tempArrOfHash) >= 3) and (self.sentRequest == 0):
                if self.tempArrOfHash.count(data) >= 3:
                    self.send_to_node(node, 'sendBlockchain')
                    self.sentRequest = 1
```

```
                else:
                    self.tempArrOfHash = []
                    self.reconnect_with_peers()
                    for i in self.peers_validators():
                        self.send_to_node(i, 'sendBlockchainHash')
                        time.sleep(0.3)
        elif type(ast.literal_eval(data)) == dict:
            self.download_blockchain_file(ast.literal_eval(data))
            self.sentRequest = 0
```

Функција node_message је редефинисана и позива се када стигне порука било ког типа. Као параметре прима конекцију чвора који је послао поруку и саму поруку.

- Ако је адреса пошиљаоца 127.0.0.1:9000 то значи да је стигла порука од рутер чвора и постоје двије врсте порука које се примају од њега. Прва која почиње са „nodes:“ а у наставку има листу активних валидатора, и друга која може бити „ok:True“ ако су валидатори на које смо повезани још увијек активни или „ok:False“ ако нису и у том случају врши се понвно повезивање на чворове.
- Порука која садржи „blckHash:“ у наставку садржи хеш последњег блока у blockchain-у који нам шаље валидатор након што смо му упутили захтјев поруком „sendBlockchainHash“. Разлог за слање тих хеш вриједности јесте тај да се иста хеш вриједност мора добити од три повезана валидатор чвора како бисмо били сигурнији да сви имају идентичан blockchain и да би се више заштитили ако неки чвор посједује погрешну или застарјелу вриједност blockchain-а. Ако се од 3 примљене хеш вриједности једна разликује врши се повезивање на друге валидатора и тако све док се не добију три исте хеш вриједности.
- Порука која је читава типа dict (ријечник података) је заправо blockchain који нам шаље повезани валидатор након што смо му упутили захтјев поруком „sendBlockchain“.

```
    def download_blockchain_file(self, dct):
        self.blockchain = dct
        print("\nBlockchain file will be located in directory "+os.getcwd()+".")
        currTimestamp = int(datetime.timestamp(datetime.now()))
        self.blockchainFileName = "blockchain_"+str(self.port)+"_"+str(currTimestamp)+"
.json"
        print("Downloaded blockchain in file with name " + self.blockchainFileName+"\n"
)
        f = open(self.blockchainFileName, 'w')
        json.dump(dct, f)
        f.close()
```

Функција download_bockchain_file као параметар прима ријечник података и формира се .json фајл чији ће назив имати формат „blockchian_назив порта_тренутни timestamp“ и у њега ће бити уграђен објекат типа dict тј. ријечник података прослијеђен као аргумент. Ова функција се позива кад се од једног од повезаних валидатора прими вриједност blockchain-a.

```
def init_connect_to_nodes(self):
    nodesIndicesToConnect = []

    i = 3
    while (i > 0):
        index = randint(0, len(self.arrNodes)-1)
        if (index not in nodesIndicesToConnect):
            nodesIndicesToConnect.append(index)
            i = i - 1
        else:
            continue

    nodesToConnect = []
    for i in nodesIndicesToConnect:
        nodesToConnect.append(self.arrNodes[i])
    self.nodes_inbound = []
    self.nodes_outbound = []

    for i in nodesToConnect:
        self.connect_with_node(i[0],i[1])
```

Функција „init_connect_to_nodes“ служи да се из низа примљених адреса активних валидатора насумично издвоје три и на њих да се повежемо и тако постанемо чвор p2p мреже.

```
def join_p2p(self):
    self.connect_with_node('127.0.0.1', 9000)
    sNode = None
    for i in self.nodes_outbound:
        if i.host=='127.0.0.1' and i.port==9000:
            sNode = i
            break
    self.send_to_node(sNode, 'getNodes')
    while (len(self.arrNodes)<=3):
```

```
            self.send_to_node(sNode, 'getNodes')
            time.sleep(2)

        self.disconnect_with_node(sNode)
        self.init_connect_to_nodes()
        time.sleep(1)
```

Функција „join_p2p“ како јој име каже служи да би се придружили p2p мрежи. Прво се јављамо рутер чвору са поруком „getNodes“ како бисно добили адресе активних валидатора на које је могуће повезати се (значење ових порука објашњено је изнад када је обрађен фајл „routerNode.py“), онда прекинамо везу са рутер чвором, а затим се позива претходно објашњена функција „init_connect_to_nodes“ чиме се придружујемо p2p мрежи.

```
    def check_connected_validators(self):
        if len(self.nodes_outbound) < 1:
            self.reconnect_with_peers()
        arrToSend = []
        for n in self.nodes_outbound:
            arrToSend.append((str(n.host), int(n.port)))

        self.connect_with_node('127.0.0.1', 9000)

        sNode = None
        for i in self.nodes_outbound:
            if i.host=='127.0.0.1' and i.port==9000:
                sNode = i
                break
        self.send_to_node(sNode, 'checkValidators:'+str(arrToSend))
        time.sleep(2)
        self.disconnect_with_node(sNode)
```

Функција „check_connected_validators“ служи да провјеримо да ли су валидатори са којима смо повезани још увијек активни. Прво се провјерава да ли постоји веза са једним или више чворова, и ако не постоји поново се повезујемо. Затим шаљемо рутер чвору листу адреса валидатора на које смо повезани да би рутер чвор провјерио да ли су још увијек активни на мрежи и враћа нам поруку о томе да ли јесу или нису. Ако јесу примићемо поруку „ok:True“, а ако нису „ok:False“ и тада ћемо се поново повезати на нове валидаторе као што је претходно објашњено када смо говорили о порукама.

```
    def make_bet(self, numForProbability, sequenceChoice, keyPair):

        pk = keyPair.publickey().exportKey().decode()
        currTimestamp = datetime.timestamp(datetime.now())

        hashOfBetData = SHA256.new((str(pk)+str(numForProbability)+str(sequenceChoice)+
str(currTimestamp)).encode())

        signer = PKCS115_SigScheme(keyPair)

        betSgn = signer.sign(hashOfBetData)
        strBetSgn = binascii.hexlify(betSgn).decode()

        dictWithData = {'gamblerPK':pk, 'numForProbability': numForProbability, 'sequen
ceChoice': sequenceChoice, 'betTimestamp':currTimestamp, 'betSignature': strBetSgn}

        hashOfBet = SHA256.new((str(pk)+str(numForProbability)+str(sequenceChoice)+str(
currTimestamp)+str(strBetSgn)).encode()).hexdigest()

        dictToSend = {hashOfBet:dictWithData}

        return dictToSend
```

Функција „make_bet“ прима три аргумента, а то су одабрана ознака за вјероватноћу, одабрана секвенца и пар тајни/јавни кључ који је потребан за потписивање опкладе. Функција служи да се од унесених података формира опклада у одређеном формату. У опкладу се уграђују следећи подаци:

1. Хеш вриједност опкладе која настаје конкатенирањем свих даље наведених података у блоку и позивањем хеш функције sha_256 над тако конкатенираним подацима
2. Јавни кључ учесника (коцкара) тј. онога ко генерише опкладу
3. Одабрана ознака за вјероватноћу добитка
4. Одабрана секвенца нула и јединица као предвиђање резултата извлачења
5. Тренутак у коме је опклада генерисана (тренутни timestamp)
6. Потпис блока тајним кључем гдје се потписује порука настала као излаз хеш функције sha_256 чији су улаз конкатенирани претходни подаци под бројем 2, 3, 4 и 5.

Излаз функције је објекат типа dict који садржи један кључ:вриједност податак, гдје је кључ хеш опкладе, а вриједност је такође објекат типа dict који садржи вриједности претходно излистане под бројевима 2, 3, 4, 5 и 6 чији су кључеви стрингови који говоре која врста вриједности се чува.

```
    def send_bet_to_peers(self, betDict):

        strToSend = 'bet:'+str(betDict)
        self.send_to_validators(strToSend)
```

Функција „send_bet_to_peers“ прима као параметар објекат типа dict који представља опкладу и обично је тај параметар излаз функције „make_bet“. Функција „send_bet_to_peers“ служи да пошаљемо генерисани опкладу ка валидатор чворовима на које је учесник повезан.

```
    def check_bet_in_block(self, pk, blockHash, lstBlockBets, binBlock):
        num = 0
        betsDictionary = {}
        for bet in lstBlockBets:
            if pk == bet[list(bet.keys())[0]]['gamblerPK']:
                num = num + 1
                betsDictionary[str(num)+"):"+str(blockHash)] = bet
                prob = bet[list(bet.keys())[0]]['numForProbability']
                seq = bet[list(bet.keys())[0]]['sequenceChoice']
                if seq != binBlock[-int(math.log(int(prob),2)):]:
                    status = "You lost this bet!"
                else:
                    status = "Congratulations! You won x"+prob+"!"
                print("\nBlock hexadecimal: " + str(blockHash) +'\nBlock binary: '+str(
binBlock) + '\nBet id: ' + str(list(bet.keys())[0]) +'\nSequence of last '+str(len(seq)
)+' bit(s) in binary block is: '+str(binBlock[-
int(math.log(int(prob),2)):]) + "\nYour predicted sequence was: " + str(seq) + "\n"+str
(status)+'\n')

        return betsDictionary
```

Функција „check_bet_in_block“ је функција која као параметре прима јавни кључ, хеш вриједност блока, низ опклада у блоку и хеш вриједност блока у бинарној репрезентацији. Функција служи да у датом низу опклада пронађе оне чији се јавни кључ поклапа са прослијеђеним и избаци резултат сваке опкладе која се поклапа тј. да испише да ли се десио добитак или губитак на основу прослијеђене бинарне репрезентације хеш вриједности блока. Функција враћа ријечник података гдје су кључеви хеш вриједност прослијеђена као аргумент али конкатенирана са редним бројем пронађене опкладе, а

вриједности представљају опкладе у истом формату као што су прослијеђене у низу као аргумент.

```
    def check_all_my_bets(self, pk):
        betsDict = {}
        for block in self.blockchain:
            binBlock = bin(int(block, 16))[2:].zfill(256)
            lstBlockBets = self.blockchain[block]['bets']
            dct = self.check_bet_in_block(pk, block, lstBlockBets, binBlock)
            betsDict.update(dct)
        if betsDict == {}:
            print("Nothing found")
        return betsDict
```

Функција „check_all_my_bets“ прима као параметар јавни кључ, а служи да пронађе опкладе у читавом blockchain-у чији се јавни кључ поклапа са јавним кључем прослијеђеним као аргумент. Претрага функционише тако што се пролази кроз блокове и за сваки блок и низ опклада у њему се позива функција „check_bet_in_block“ чија је функционалност већ објашњена. Функција враћа ријечник података у коме су исти елементи као и они који представљају излаз „check_bet_in_block“ функције.

```
    def check_last_block(self, pk):
        betsDict = {}

        lastBlockHash = list(self.blockchain.keys())[-1]
        lastBlockData = self.blockchain[lastBlockHash]

        binBlock = bin(int(lastBlockHash, 16))[2:].zfill(256)

        betsDict = self.check_bet_in_block(pk, lastBlockHash, lastBlockData['bets'], bi
nBlock)

        if betsDict == {}:
            print("Nothing found")

        return betsDict
```

Функција „check_last_block“ као параметар има јавни кључ, а служи да се провјери поклапање јавног кључа прослијеђеног као аргумент са јавним кључевима из низа опклада али само последњег блока у blockchain-у. И у овој функцији се такође позива функција „check_bet_in_block“ којој се као аргументи прослијеђују јавни кључ са којим се

тражи поклапање, хеш последњег блока, низ опклада последњег блока и бинарна репрезентација хеш вриједности последњег блока. Излаз функције је идентичан излазу функције „check_bet_in_block“.

```
def peers_validators(self):
    peersValidators = []
    for node in self.nodes_outbound:
        if (node.host == '127.0.0.1' and node.port == 9000):
            continue
        peersValidators.append(node)
    return peersValidators
```

Функција „peers_validators“ враћа низ конекција са валидатор чворовима не које је учесник повезан.

```
def send_to_validators(self, data):
    for node in self.nodes_outbound:
        if (node.host == '127.0.0.1' and node.port == 9000):
            continue
        self.send_to_node(node, data)
```

Функција „send_to_validator“ прима као параметар има податак који шаље, а служи да се тај податак пошаље валидатор чворовима на које је учесник повезан (три чвора).

```
def reconnect_with_peers(self):
    self.arrNodes = []

    for i in self.nodes_outbound:
        self.disconnect_with_node(i)
    self.join_p2p()
```

Функција „reconnect_to_peers“ служи да прекинамо везу са балидатор чворовима са којима смо повезани и да се поново придружимо p2p мрежи позивом функције „join_p2p“ која је на некој од претходних страница детаљније објашњена.

```
def time_for_wait_connection(self):
    return (datetime.timestamp(datetime.now()) % 120 > 50)
```

Функција „time_for_wait_connection“ враћа True ако је у тренутном кругу прошло 50 секунди и даје нам тиме сигнал да требамо чекати нови круг да би се повезали у мрежу.

```
    def time_for_listen_bets(self):

        return (datetime.timestamp(datetime.now()) % 120 < 80)
```

Функција „time_for_listen_bets“ враћа True ако је у току период првих 80 секунди текућег круга када валидатори слушају опкладе и тада учесник (коцкар) може послати своју опкладу без да чека следећи круг тј. блок. У супротном функција враћа „False“.

4.6. Датотека „gamblerApp.py“

Ова датотека представља имплементацију свега из „GamblerNode.py“ датотеке и представља програм који се покреће када желимо да покренемо „коцкар“ чвор тј. учесника у клађењу. Опис кода ћемо подијелити у три цјелине.

1. цјелина:

```
from gamblerNode import GamblerNode
import socket
import sys
import time
from random import randint
from datetime import datetime
from Crypto.PublicKey import RSA
from Crypto.Hash import SHA256
from Crypto.Signature.pkcs1_15 import PKCS115_SigScheme
import binascii
import os
import json
import ast
import math

host = str(sys.argv[1])
port = int(sys.argv[2])


while True:
    print("Do you want to generate new key pair? [y/n]\n(If answer is yes, program will
 create key in .pem file at location of your choice.\nIf answer is no, it means that yo
u already have the key generated by this application and you have to load it.)")
```

```
    answer1 = input("Answer: ")
    if answer1 in ['y', 'Y', 'yes']:
        while True:
            print("\nInput directory path where key file will be located:")
            currTimestamp = int(datetime.timestamp(datetime.now()))
            keyPath = input("Path: ")
            try:
                f = open(keyPath+'/mykey_'+str(port)+'_'+str(currTimestamp)+'.pem', 'wb
')
            except:
                print("Invalid path! Try again...")
                continue
            break
        key = RSA.generate(1024)
        f.write(key.exportKey('PEM'))
        f.close()
        print('Only PK:\n'+str(key.publickey().exportKey().decode())+'\n')
        break
    elif answer1 in ['n', 'N', 'no']:
        while True:
            print("\nInput correct name of .pem key file with entire path:")
            keyPath = input("Path: ")
            try:
                f = open(keyPath, 'r')
                key = RSA.importKey(f.read())
            except:
                print("Invalid path or file! Try again...")
                continue
            break
        f.close()
        print('Only PK:\n'+str(key.publickey().exportKey().decode())+'\n')
        break
    else:
        print("Incorrect answer! Try again...\n")
        continue
```

Прво требамо да знамо да се „коцкар“ чвор покреће тако што укуцамо нпр:

python gamblerApp.py 127.0.0.1 8010

гдје је потребно да нам први аргумент буде хост адреса, а други порт адреса и сходно томе ти аргументи постају вриједности промјенљивих „host“ и „port“ и служиће да се додјели адреса „коцкар“ чвору када га покренемо.

Затим се улази у петљу у којој нам се поставља питање да ли желимо да генеришемо нови пар тајни/јавни кључ. Ако потврдно одговоримо програм ће нас питати да унесемо путању до директоријума у који ће смјестити кључ креираће 1024-битни кључ коришћењем RSA алгоритма у .pem фајлу и ако га сачувамо можемо га користити сваки следећи пут. Ако је одговор не, то значи да већ имамо генерисан кључ и програм ће нас питати да унесемо читаву путању до кључа заједно са називом .pem фајла у коме се кључ налази. Затим се из пара тајни/јавни кључ који је претходно генерисан/учитан издваја и исписује само јавни кључ.

2. цјелина: наставак кода

```
myProbability = None
mySequenceChoice = None
requestedBlockchain = 0
requestedGambling = 0
onlyBlockchain = 0

while True:
    answer2 = input('\nDo you want to gamble or only download blockchain?\nEnter 0 for gambling\nEnter 1 for download blockchain\nYour answer: ')
    if answer2 in ['0', '1']:
        if int(answer2) == 0:
            requestedGambling = 1
            while True:
                probabilityChoice = input("\nBet probability options:\n2 for probability: 1/2\n4 for probability: 1/4\n8 for probability: 1/8\n16 for probability: 1/16\n32 for probability: 1/32\n64 for probability: 1/64\n128 for probability: 1/128\n256 for probability: 1/256\nEnter your choice: ")
                if (probabilityChoice in ['2', '4', '8', '16', '32', '64', '128', '256']):
                    myProbability = int(probabilityChoice)
                    break
                else:
                    print("Invalid input!")
                    continue
            while True:
                print("\nYou have to enter sequence with following number of ones or zeros: "+str(int(math.log(myProbability, 2))))
                mySequence = input("Enter your sequence: ")
                if len(mySequence) != int(math.log(myProbability, 2)):
                    print("Invalid input length!")
                    continue
                invalid = False
```

```
                for s in mySequence:
                    if (s not in ['0', '1']):
                        invalid = True
                        break
                if invalid:
                    print("Invalid sequence input!")
                    continue
                print("You predict that last "+str(int(math.log(myProbability, 2)))+" b
it(s) in hash of block with your transaction in binary representation will be "+mySeque
nce)
                mySequenceChoice = mySequence
                break
            while True:
                checkChoice = input("\nDo you want to download blockchain after game to
 check if you are winner or loser? [y/n]\nYour answer: ")
                if checkChoice in ['y', 'Y', 'yes']:
                    print("Blockchain file will be located in directory "+os.getcwd()+"
.")
                    requestedBlockchain = 1
                    break
                elif checkChoice in ['n', 'N', 'no']:
                    requestedBlockchain = 0
                    break
                else:
                    print("Incorrect answer! Try again...")
                    continue
            break
        elif int(answer2) == 1:
            onlyBlockchain = 1
            break
    else:
        print("Invalid input! Try again ...")
        continue
```

У овом наставку кода најприје се додјељује вриједност 5 промјенљивих:

- „myProbability“ која ће да садржи ознаку одабране вјероватноће
- „mySequenceChoice“ која ће да садржи секвенцу коју предвиђамо
- „requestedBlockchain“ која је иначе 0, а биће 1 ако затражимо провјеру добитка при чему смо прво одабрали опцију да се кладимо
- „requestedGambling“ које је иначе 0, а биће 1 ако прихватимо опцију да се кладимо
- „onlyBlockchain“ која је иначе 0, а биће 1 ако затражимо само провјеру наших добитака без опције клађења

Да би било јасније шта све учесник у игри треба да одабере потребно је прво упознти се са опцијама које учесник има у току клађења:

Постоје 2 основне опције:

1. Са клађењем:
    a. Само пуштање опкладе на мрежу
    b. Пуштање опкладе на мрежу и слање захтјева за blockchain на крају круга како би провјерили да ли смо добили или изгубили у том извлачењу
2. Без клађења: само слање захтјева за blockchain како бисмо провјерили да ли се и које опкладе у blockchain-у поклапају са јавним кључем који смо генерисали/учитали и испис података о тим опкладама

У наставку кода од корисника се прво тражи да одговори да ли жели да се клади или да само пошаље захтјев за blockchain. Ако корисник одговори са 0 сматра се да је одабрао опцију да се клади, а ако унесе 1 одабрао је опцију без клађења.

Ако је корисник одабрао опцију да се клади промјенљива „requestedGambling“ добија вриједност 1 и од корисника ће бити захтјевано следеће:

- Да унесе ознаку за вјероватноћу са којом жели да добије и то по следећем принципу
    - да унесе 2 за вјероватноћу 1/2 или
    - да унесе 4 за вјероватноћу 1/4 или
    - да унесе 8 за вјероватноћу 1/8 или
    - да унесе 16 за вјероватноћу 1/16 или
    - да унесе 32 за вјероватноћу 1/32 или
    - да унесе 64 за вјероватноћу 1/64 или
    - да унесе 128 за вјероватноћу 1/128 или
    - да унесе 256 за вјероватноћу 1/256

    Учеснику су дозвољени само претходно приказани уноси, а промјенљива „myProbability“ добија вриједност уноса.

- Да унесе секвенцу за коју сматра да ће се појавити на крају бинарне репрезентације хеш вриједности блока у коме ће бити његова опклада. На тај начин и функционише ова игра на срећу. Оно што учесник заправо погађа јесте секвенца од одређеног броја нула и јединица која ће да се појави на крају (битови најмање тежине) бинарне репрезентације хеш вриједности блока у коме је садржана опклада тог учесника. Дужина секвенце коју учесник погађа зависи од вјероватноће коју је претходно одабрао и то по следећем принципу:
    - За вјероватноћу 1/2 бира секвенцу дужине 1 јер тако постоје 2 комбинације (0 или 1)

- За вјероватноћу 1/4 бира секвенцу дужине 2 јер тако постоје 4 комбинације (00, 01, 10 или 11)
- За вјероватноћу 1/8 бира секвенцу дужине 3 јер тако постоји 8 комбинација (000, 001, 010, 011, 100, 101, 110 или 111)
- За вјероватноћу 1/16 бира секвенцу дужине 4 јер тако постоји 16 комбинација (0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, ... , 1111)
- За вјероватноћу 1/32 бира секвенцу дужине 5 јер тако постоје 32 комбинације (00000, 00001, 00010, 00011, 00100, ... , 11111)
- За вјероватноћу 1/64 бира секвенцу дужине 6 јер тако постоје 64 комбинације (000000, 000001, 000010, 000011, ... , 111111)
- За вјероватноћу 1/128 бира секвенцу дужине 7 јер тако постоји 128 комбинација (0000000, 0000001, 0000010, 0000011, ... , 1111111)
- За вјероватноћу 1/256 бира секвенцу дужине 8 јер тако постоји 256 комбинација (00000000, 00000001, 00000010, 00000011, ... , 11111111)

Учеснику је дозвољено да унесе само секвенце нула и јединица и то дугачке онолико колико то предвиђа претходни одабир вјероватноће добитка. Промјенљива „mySequenceChoice“ добија вриједнот уносене секвенце.

- Да ли жели да провјери на крају круга статус своје опкладе тј. да ли је добио или изгубио. Ако одговори потврдно програм ће остати на чекању док не истекну 2 минута предвиђена за један круг а затим ће захтјевати последњу верзију blockchain-а како би обавио провјеру након чега исписује статус опкладе. У случају потврдног одговора вриједност промјенљиве „requestedBlockchain“ постаје 1. Такође, треба напоменути да ако корисник одговори са не, његова опклада неће нестати ако је већ послата на мрежу, а корисник у сваком моменту може провјерити статус своје опкладе.

Ако је корисник одабрао опцију без клађења тј. ако је поднио само захтјев за blockchain, он ће добити последњу верзију blockchain-а и аутоматски ће се покренути провјера да ли посједује неке опкладе тј. да ли се његов претходно унесени јавни кључ поклапа са јавним кључем неке од опклада у blockchiain-у. Ако се поклапа биће исписан статус сваке опкладе посебно. Помоћу ове опције учесник може у сваком тренутку доказати своје власништво над опкладом (било да је у питању добитак или губитак). Такође, одабиром ове опције вриједност промјенљиве „onlyBlockchain“ постаје 1.

3. цјелина: наставак кода

```
myKeyPair = key
myPublicKey = key.publickey()
pkHash = SHA256.new(myPublicKey.exportKey()).hexdigest()
```

```
myID = 'gambler:'+str(pkHash)
myGamblerNode = GamblerNode(host, port, myID)
myGamblerNode.start()


#if myGamblerNode.time_for_wait_connection():
print('waiting for connection ...')
while myGamblerNode.time_for_wait_connection():
    #print('waiting for connection ...')
    time.sleep(1)
    continue
firstTimestamp = datetime.timestamp(datetime.now()) % 120
myGamblerNode.join_p2p()

print('\nConnected validators:')
for i in myGamblerNode.peers_validators():
    print(str(i.id))
print('\n')

try:
    if requestedGambling:
        time.sleep(1)
        if (not myGamblerNode.time_for_listen_bets()):
            print('waiting to send bet ...')
        while (not myGamblerNode.time_for_listen_bets()):
            #print('waiting to send bet ...')
            time.sleep(1)
            continue
        myBet = myGamblerNode.make_bet(str(myProbability), mySequenceChoice, myKeyPair)
        myGamblerNode.send_bet_to_peers(myBet)
        time.sleep(2)
        if requestedBlockchain:
            print('waiting for blockchain ...')
            time.sleep(float(120) - firstTimestamp + 2)
            myGamblerNode.check_connected_validators()
            for i in myGamblerNode.peers_validators():
                myGamblerNode.send_to_node(i, 'sendBlockchainHash')
                time.sleep(0.3)
            time.sleep(5)
            for n in myGamblerNode.peers_validators():
                myGamblerNode.disconnect_with_node(n)
            print("\nChecking bet:")
            myGamblerNode.check_last_block(myPublicKey.exportKey().decode())
            myGamblerNode.stop()
            time.sleep(2)
```

```
            sys.exit(0)
        else:
            for n in myGamblerNode.peers_validators():
                myGamblerNode.disconnect_with_node(n)
            myGamblerNode.stop()
            time.sleep(2)
            sys.exit(0)
    elif onlyBlockchain:
        for i in myGamblerNode.peers_validators():
            myGamblerNode.send_to_node(i, 'sendBlockchainHash')
            time.sleep(0.3)
        time.sleep(5)
        for n in myGamblerNode.peers_validators():
            myGamblerNode.disconnect_with_node(n)
        print("\nChecking bets ...")
        myGamblerNode.check_all_my_bets(myPublicKey.exportKey().decode())
        print('\n')
        myGamblerNode.stop()
        time.sleep(2)
        sys.exit(0)
except:
    myGamblerNode.stop()
    sys.exit(1)
```

У наставку изнад формира се ID који ће бити стринг који се састоји од секвенце „gambler:“ послије које слиједи хеш вриједност јавног кључа. Послије тога се ствара објекат класе GamblerNode и покреће се „коцкар“ чвор. Даље се провјерава да ли је потребно чекати на повезивање и ако јесте чека се и врти се петља док буде вријеме за повезивање и онда се наш чвор преко функције „join_p2p“ повезује у p2p мрежу. Затим се исписују идентификатори валидатора са којима је учесник повезан и шаље им се порука „sendBlockchainHash“ чије значење је већ објашњено.

Након тога слиједи провјера да ли је вриједност промјенљиве „requestedGambling“ једнака 1. Ако јесте ради се следеће:

- Провјерава се да ли је вријеме за слање опкладе и ако није чека се а ако јесте формира се опклада и шаље се валидаторима на које је учесник повезан.
- Провјерава се да ли је вриједност промјенљиве „requestedBlockchain“ једнака 1. Ако јесте чека се завршетак круга након чега се шаље захтјев за blockchain. Blockchain се преузме а учесник прекида везу за валидаторима и покреће функцију за провјеру статуса своје опкладе (функција „check_last_block“). Након тога се учесников „коцкар“ чвор стопира и завршава се извршавање програма. Ако није

учесник само прекине везу са валидаторима, чвор се стопира и завршава се извршавање програма.

Ако није вриједност промјенљиве „requestedGambling“ једнака 1 тада се провјерава да ли је вриједност промјенљиве „onlyBlockchain“ једнака 1, а у случају да вриједност промјенљиве „requestedGambling“ није једнака 1, онда ће вриједност промјенљиве „onlyBlockchain“ сигурно бити једнака 1. У том случају се одмах шаље захтјев за blockchain. Blockchain се преузме а онда учесник прекида везу са валидаторима и покреће функцију за провјеру власништва опклада у blockchain-у (функција „check_all_my_bets“). Након тога учесников чвор се стопира и завршава се извршавање програма.

5. Закључак

Након објашњења начина функционисања апликација у пројекту и описа саме идеје развоја коцкарске индустрије на blockchain технологији неизбјежно је закључити да овај вид технлогије у комбинацији са p2p мрежом и неким сајним криптографским рјешењима као што су то овде RSA алгоритам и SHA_256 хеш функција даје заиста широке могућности за примјену. Разлог зашто би такав вид технологије требао ући у употребу јесте тај што обичан корисник може имати пуно повјерење у такву технологију јер прије свега ради на принципу децентрализације преко p2p мреже тако да важећи blockchain фајл никад није у власништву једног сервера, већ његов власник може бити сваки корисник који покрене преузимање последње верзије blockchain-а и остане ажуран у даљем преузимању нових верзија. Још један битан разлог за повјерење јесте тај да сваки учесник дијелом утиче на генерисање резултата једног извлачења, тако да нпр. без учешћа једног од корисника, резултат једног извлачења би био сасвим другачији.

Такође, напомињем да у овом пројекту није имплементирана криптовалута па стога не постоје могућности трансакција, провјере да ли неки учесник има средства за клађење као ни могућност конкретног добитка. Овај пројекат пружа могућност учеснику да заједно са другим учесницима генерише резултат извлачења у једном кругу и представља моје схватање како се уз помоћ blockchain технологије, децентрализације и веома напредне криптографије може имплементирати фер игра на срећу. Свјестан сам да овај пројекат има доста недостатака али сам се трудио да кроз читав пројекат останем досљедан поменутом концепту фер игре на срећу.

6. Литература

1. https://www.coursera.org/learn/crypto
2. https://www.coursera.org/learn/cryptocurrency
3. https://github.com/macsnoeren/python-p2p-network
4. Др Владимир М. Миловановић, Компоновање рачунарских програма, Факултет инжењерских наука, Крагујевац, 2021
5. Dr. Charles R. Severance, Python for Everybody, University of Michigan, 2016
6. https://stackoverflow.com/
7. https://www.geeksforgeeks.org/
8. https://www.wikipedia.org/