PROGRAMMING 2 PROJECT

CRYPTOCURRENCY SIMULATION

TABLE OF CONTENTS

1. Overview

1.1 Goal

The aim of this project is the creation of a simulation of a network of cryptocurrency exchange based on the blockchain technology and, specifically, the model of Bitcoin, in Java.

Some specific features of this model have to be: the lack of a central entity for the control and distribution of cryptocurrency; the possibility for entities ("users") to try to spend more currency than they own; the validation of transactions by the validators ("miners"); the miners work in parallel on separate threads.

1.2 Result

In the end, I created a real-time simulation of such transaction network as described above. In addition, this network was accompanied by a visual representation of the most interesting features: a chart of the live balances of the users; the miners' working status; the transaction ledger/queue (up to 5 elements); the blockchain (up to 5 elements).

There is no central entity for the control and distribution of the currency. The users can spend up to their balance, however, there exists the possibility for them to spend multiple times from this balance before their transactions have been validated, making for a double-(or more)-spending scenario.

The miners work in parallel on multiple threads and validate the transactions by checking two factors:

1. the legitimacy of the transaction, that is, that the transaction has been signed by the sender user's private key, by checking against their public key;
2. whether the sender user can afford the transaction (avoiding in this way double-spending).

In addition, the miners have mechanisms to check that the transactions they are currently validating have not been already validated by other miners (that is, are not already in the blockchain). In this way, we avoid blockchain branching.

1.3 Assumptions

Some of the assumptions employed by the network are:

- a user cannot have a balance of 0 currency or less;
- there are no conflicting branching blockchains. Because the miners make extensive checks to assure the avoidance of conflicts between the block they are mining and the current blocks on the chain, it cannot happen for two or more blocks to contain the same transaction, therefore such a case is assumed as unreachable;

2. The Network

2.1 A high-level description

2.1.a Network

```
#Attributes:

public static User[] Users;
public static Miner[] Miners;

#Methods:

//Initializes Users and Miners
public static void Start(int _userCount, int _minerCount)

//Network update
//Updates the miners
//Every 1 to 3 seconds triggers a user to perform a transaction
public static void Update()

//Broadcasts a new block throughout the network
public static void BroadcastBlock(Block _block)

//Broadcasts a new transaction to the miners
public static void BroadcastTransaction(Transaction _transaction)

//Broadcasts a list of removed transations to miners
//Used to remove invalid transactions or completed transactions
public static void BroadcastRemovedTransactions(LinkedList<Transaction>
_transactions)

//Returns a random user from Users
public static User RandomUser()
```

2.1.b Transaction

```
#Attributes:

public User Sender;
public User Receiver;
public int Quantity;
public String Signature;
public long Timestamp;

#Methods:

//Creates a transaction
public Transaction(User _sender, User _receiver, int _quantity)
```

2.1.c User

```
#Attributes:

public String Username;
public int Balance = 30;
public String PublicKey;
private String privateKey;
public LinkedList<Block> BlockChain = new LinkedList<>();
private final int KEY_LENGTH = 10;

#Methods:

//Initialization
public User()

//Generate public/private keys
private void GenerateKeys()

//Generates digital signature
//Used to verify the signature using PublicKey
public String GenerateSignature(User _receiver, int _quantity, long
_timestamp, String _key)

//Generates digital signature
//Used to initially generate the signature, using privateKey
public String GenerateSignature(User _receiver, int _quantity, long
_timestamp)

//Creates a random transaction
public Transaction CreateTransaction()
```

2.1.d Miner

```
Miner extends User implements Runnable

#Attributes:

public LinkedList<Transaction> TransactionQueue = new LinkedList<>();
public RunningState State = RunningState.Idle;
public boolean JustMined = false;
private final int BLOCK_SIZE = 3;

#Methods:

//Running State Machine
public void Update()

//region Mining
@Override
public void run()

//Try to verify transactions and if any isn't valid, discard it and pick some
other transaction
private  LinkedList<Transaction>
VerifyAllTransactions(LinkedList<Transaction> _transactions)

//Sometimes, all transactions on the list will be valid on their own but they
add up to an invalid sum. This function checks for such cases
private LinkedList<Transaction>
VerifyDoubleSpendTransactions(LinkedList<Transaction> _transactions)

//Checks that the sender can afford the transaction and verifies its
signature
private boolean VerifyTransaction(Transaction _transaction)

//Verifies sender signature
private boolean VerifySignature(Transaction _transaction)

//Picks the next transaction in queue
private LinkedList<Transaction> PickTransactions(int _count)

//Picks the next transaction in queue, skipping the invalid ones
private LinkedList<Transaction> PickTransactions(int _count,
LinkedList<Transaction> _ignoreTransactions)

//Calculates nonce for the block
private String CalculateNonce(Transaction[] _transactions)

//Miner working state
public enum RunningState
```

2.1.e Block

```
#Attributes:

public String PreviousBlockHash;
public Transaction[] TransactionList;
public String TransactionsHash;
public String Nonce;
public String CurrentHash;
public long Timestamp;
private final int MINER_REWARD = 3;

#Methods:

//Creates a block
public Block(String _previousBlockHash, Transaction[] _transactionList,
String _nonce, long _timestamp, Miner _miner)

//Generates the hash of the block using MerkleTree
private String GenerateHash()
```

2.1.e MerkleTree

```
#Method:

//Generates hashing for a list of transactions by recursively splitting it
down to a binary tree and calculating the hashes of the nodes and returning
the hash of the root
public static String GenerateHash(Transaction[] _transactions, long
_timestamp)
```

2.1.f Main

```
#Attributes:
//Assets for the visualization

#Methods:

//Creates assets and initializes the Network
@Override
public void create ()

//Renders the content and updates the Network
@Override
public void render ()

//Disposes of the assets
@Override
public void dispose ()
```

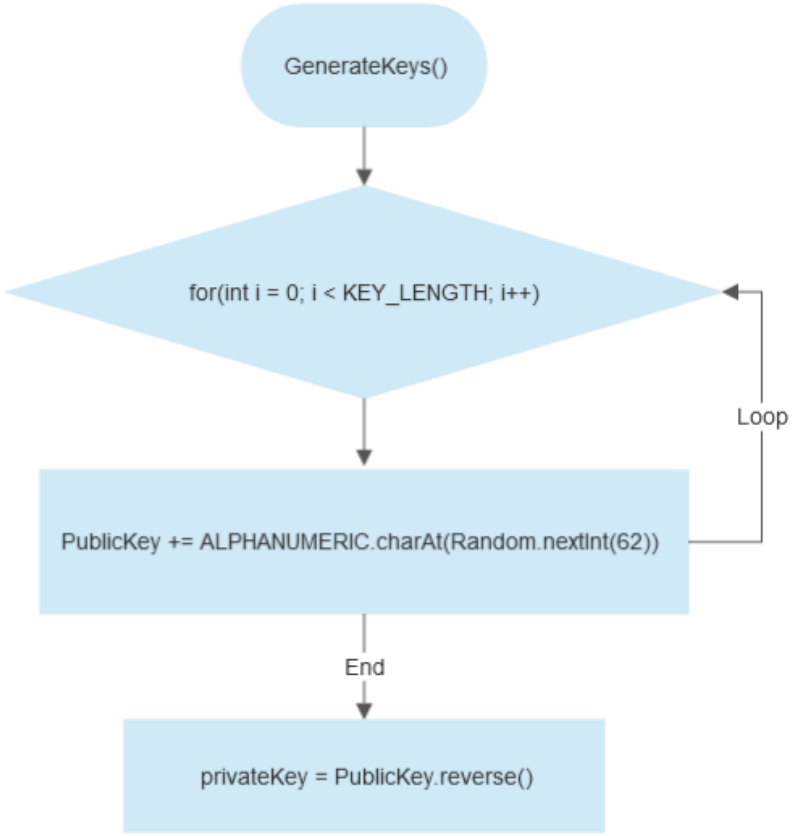
2.2 Implementation

Below are some of the more important and simple-enough-to-describe systems of the program.

2.2.a User Initialization

The User() constructor calls GenerateKeys() which generates the PublicKey and privateKey based on the KEY_LENGTH attribute. The characters for the keys are randomly picked from a string which contains numbers 0-9 and characters a-z and A-Z called ALPHANUMERIC.

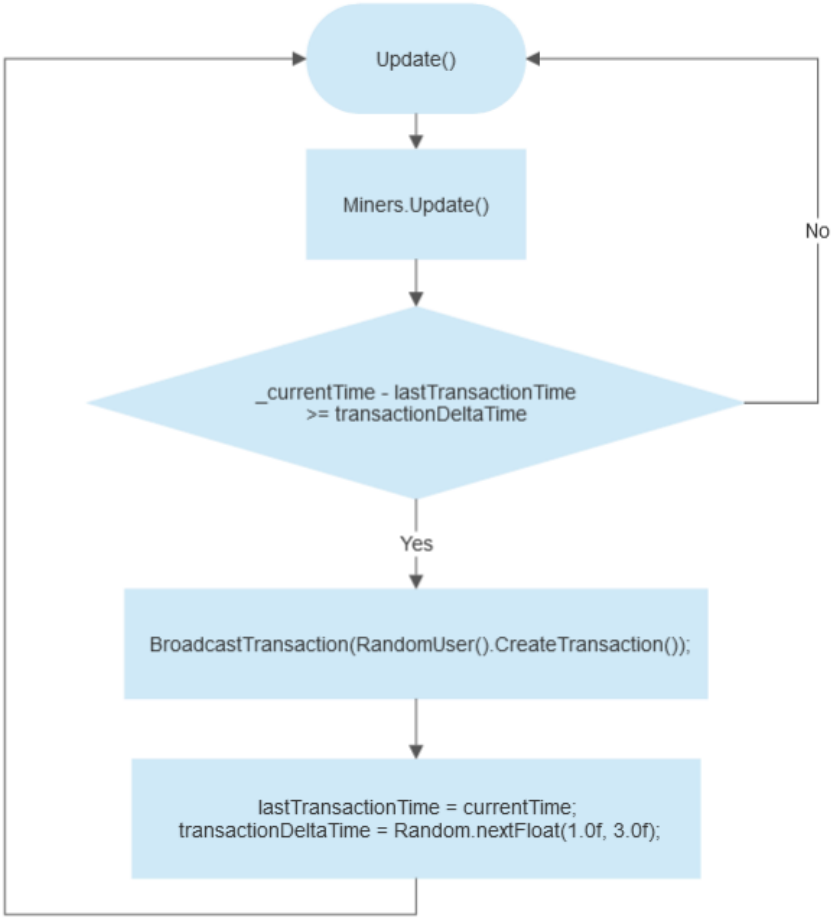
privateKey is set as reversed PublicKey in order to help with the implementation of hashing for the Signature system which can be used to both sign transactions with the privateKey and verify them with the PublicKey.



2.2.b Network Update

Network's Update() function performs two main tasks:

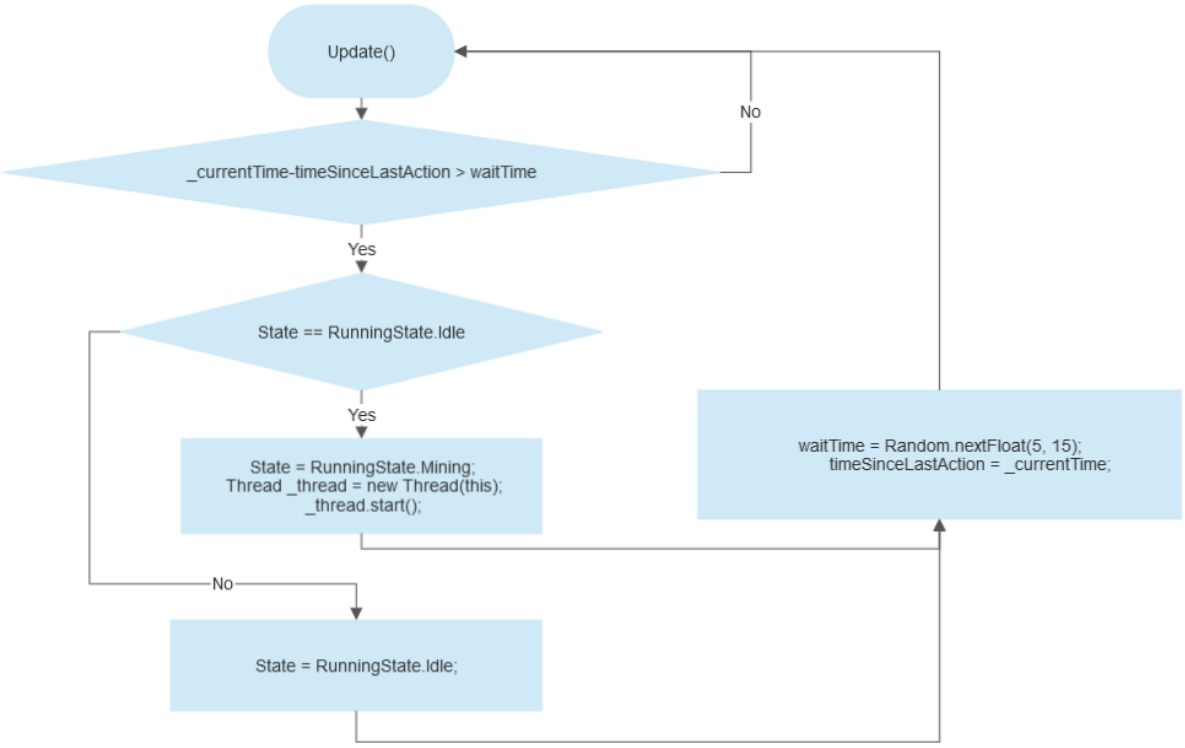
- calls Miners' Update() function for each miner in Miners;
- every 1 to 3 seconds triggers a random User to perform a Transaction and broadcasts this transaction to the miners. Normally this task would be performed on the User's side and Network would simply call their Update() function, but for practical purposes of our simulation, having this task performed by the Network gives us, as viewers, the opportunity to control and balance the network.



2.2.c Miner State Machine

Every 7 to 15 seconds, the Miner switches state, from Idle to Mining or vice-versa. Initially, every Miner is initialized in the Idle state and the wait time is 20 seconds, in order to allow some time for Transactions to happen and fill the ledger before mining begins.

If a Miner enters the Mining state, a new Thread is created with the Miner itself as a parameter since Miner implements Runnable and this Thread is started.



2.2.d Miner run()

The run() method of the Miners contains the code that runs in parallel on a separate thread.

This method begins by waiting for waitTime, which is the same waitTime as described above in the state machine, so a random number of seconds between 5 and 15 seconds. The reason that this waiting is performed is because, despite the fact that the mining process consists in many verification steps, it is still almost instantaneous. In order to make the process more clear for the viewers and keep the simulation running in real time, this waiting is performed.

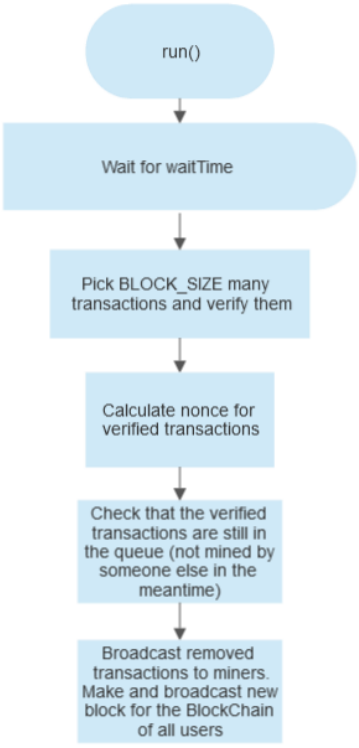
The method then begins the real work. BLOCK_SIZE is a constant which determines the number of transactions per block in the blockchain (in our case it has been set to 3).

Then, the verification process begins. As seen in point 2.1.d, this process consists in several methods. In short, a set of transactions is verified about these main points:

- all senders can afford each transaction;
- the transactions are valid (that is, their signature is legitimate);
- all senders can afford multiple transactions of the set in the case they have multiple transactions in the set. For example: if user A has a balance of 10 and the transaction set contains these transactions: user A pays user B 5 coins; user A pays user C 7 coins;, then both of these transactions are valid on their own but together they are not. So we have to pick only one in this case.

After the verification, the miner calculates the nonce of the potential block to be made from the current transactions. I say "potential" because there exists the possibility that in the meantime another miner has mined some of the transactions that the miner has selected. So a final check is made to make sure that the transactions are still on the ledger.

Finally, a new block is created using the nonce calculated earlier and the selected transactions.



2.2.e Hash functions

Everything on the network end was programmed from scratch and this includes the hash functions. There is a total of 3 new hash functions in this program:

- User.GenerateSignature() – this function takes as parameters a User (the receiver of the transaction), the quantity of the transaction as well as the timestamp of the transaction and, using the privateKey of the sender, generates a unique signature for the transaction. In addition, it is also used later to verify that this signature is valid by calling it using the same parameters and, this time, the sender's PublicKey, and checking that the result is the same as the existing signature;
- MerkleTree.GenerateHash() – this function takes as parameters the transaction set of a block as well as its timestamp, and, using recursion to create a binary tree out of these transactions, as described in the Bitcoin whitepaper, generates their hash, as well as a hash for each node of the tree and returns the hash of the node. In our case there are only 3 leaves to this tree but the function works with as many as necessary;
- Block.GenerateHash() – this function generates the final hash of the block, taking into consideration all its data.

2.2.f Graphics with libGDX

The graphical features needed for this project were basic: rectangle rendering, image rendering, text rendering. There were a couple of libraries that were considered for this purpose. Below is a breakdown of their features and why they were (not) picked:

- JavaFX

When looking for a graphics library, one of the most common suggestions I found on the Internet was JavaFX. It seemed to be well-documented and have an active community, so I decided to try it. It was capable of rendering rects, text and images. However, it had a major "flaw".

My project runs in real time and, like most real time programs (such as games), it needs an Update() function that runs every frame as described above. JavaFX unluckily doesn't work like that. It has a very event-driven structure, which makes it suitable for input-output applications such as, for example, a calculator. But our simulation needs to run all the time, with no input from the user so this option was not suitable as a graphics library.

- lwjgl

Knowing that my project needed a game-like system, the Light Weight Java Game Library seemed like the logical next step. I started following a tutorial on how to render images with this library. It seemed a lot more complicated than it should have been and, turns out that this library was merely a wrapper for OpenGL, so all the rendering had to be done manually. Manual rendering was absolutely out of the scope of the project, so I did not want to do that. Luckily, I found a library based on lwjgl which handled the rendering, libGDX.

- libGDX

As mentioned above, libGDX handles the rendering of rectangles, text and images and is fairly straightforward to use. In addition, it does have a function that runs every frame, render(), so it was perfect for the project. Using it did have small side effects though, because it was based on an older version of Java which does not support lambdas and, since they were used in a couple of places around the project, I had to make small changes to the code of the project.

The way graphics are made in libGDX for the project is fairly simple:

- create() – this function runs once at the beginning of the program. It initializes the assets such as the fonts, textures and renderers, as well as calls the initializer of the network;
- render() – this function runs every frame. First of all, it calls Network's Update() function. Then, it clears the window/canvas/frame and draws all the rectangles, text and images for the simulation, based on the data fetched from the network;
- dispose() – this function runs at the end of the program and disposes of the assets used;

3. A specific example

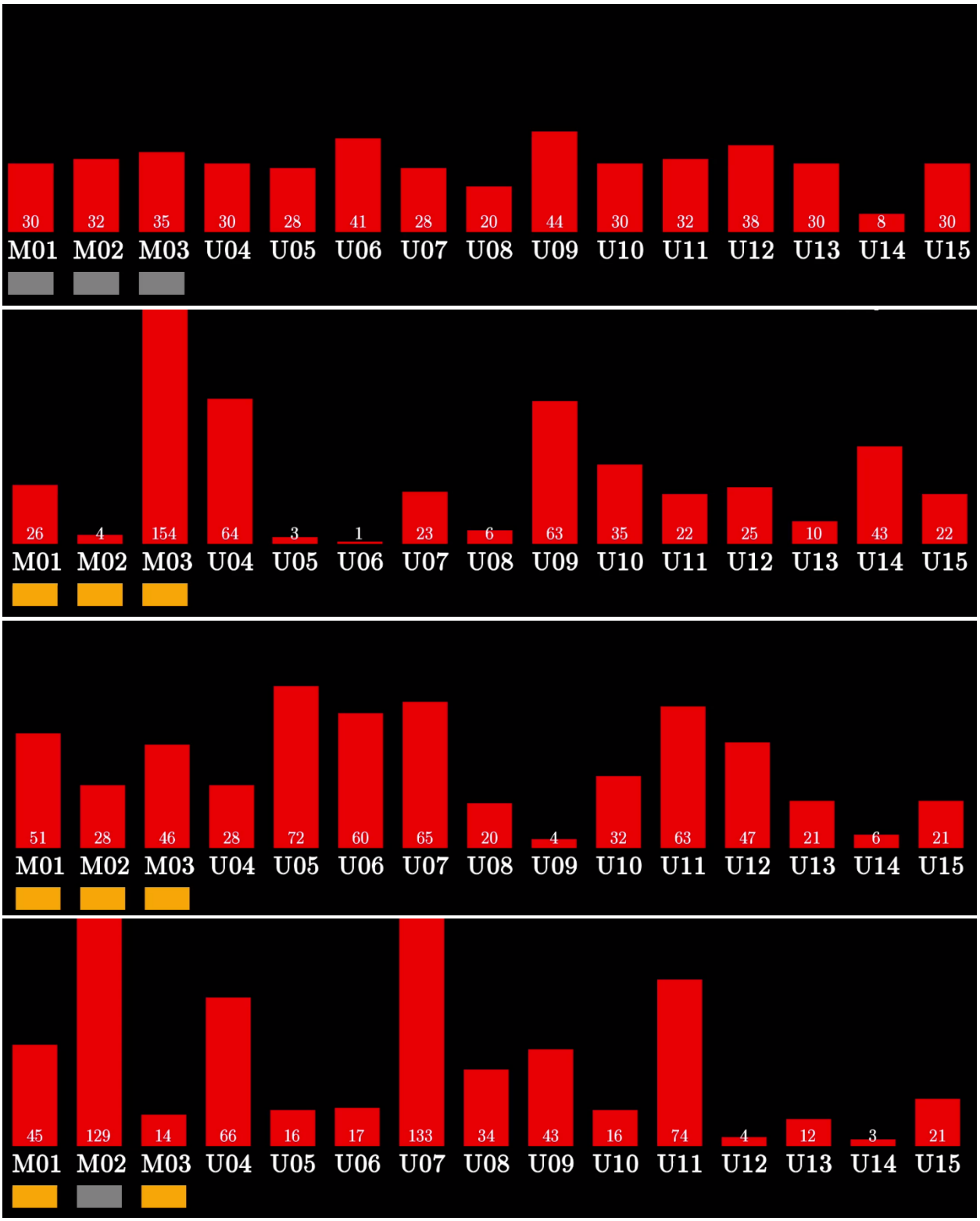
There are several variables which can be set to define the model:

- user count;
- miner count;
- initial user balance;
- transaction delta time (the time between two transactions);
- how much a user spends per transaction;
- time miners spend in each state;
- block size;
- miner reward for mining a block;

Despite the validation work and the coverage of most edge cases, if the initial variables create a highly disbalanced network, there is the chance for race conditions between threads and, as a result, errors. Furthermore, considering the miners run on different threads in real time, it makes sense to not have more threads than cores available, otherwise miners would be waiting each other in order to have access to computational resources (cores) which is not very realistic.
In addition, having some balanced and fixed values helps with the creation of a visual representation of the network and also makes the network easier to observe and analyze. Here are the values of the specific example used in this project:

- 15 users;
- 3 miners (included in the users);
- initial balance of 30 (arbitrary value);
- 1-3 seconds between each transaction;
- The amount of coins a user spends per transaction is determined by the function f(x) = Random.nextInt(1, (1 + Math.ceil(Balance/Random.nextInt(1, 4)))) ; Dividing the upper bound by a random number between 1 and 4 means that users are more likely to spend a low amount of coins rather than a high amount, which is more realistic than spending just a random amount of money.
- Miners spend 7-15 seconds in each state. They start by spending 20 seconds in the Idle state so as to allow the ledger to be filled with some transactions before they start mining;
- 3 transactions per block;
- miners receive 3 coins per block mined;

After a lot of tests with the model, the results are that, as expected by the high use of randomness, the results are also very random. It is very normal for there to be users with lots of coins and users with very few coins, and, after a few seconds, the balance might get completely changed because the “rich” users may randomly spend a lot of their coins. It is also possible for the system to reach a balanced state where users have similar balances of coins but most of the time there are spikes and valleys. Below are snapshots of the system in several moments of the same run of the program, which illustrate what already described. In the images, almost all users have considerable fluctuations, especially those in the first half of the group.


30 32 35 30 28 41 28 20 44 30 32 38 30 8 30
M01 M02 M03 U04 U05 U06 U07 U08 U09 U10 U11 U12 U13 U14 U15
26 4 154 64 3 1 23 6 63 35 22 25 10 43 22
M01 M02 M03 U04 U05 U06 U07 U08 U09 U10 U11 U12 U13 U14 U15
51 28 46 28 72 60 65 20 4 32 63 47 21 6 21
M01 M02 M03 U04 U05 U06 U07 U08 U09 U10 U11 U12 U13 U14 U15
45 129 14 66 16 17 133 34 43 16 74 4 12 3 21
M01 M02 M03 U04 U05 U06 U07 U08 U09 U10 U11 U12 U13 U14 U15