Manual de sistema

RochiCoin 2.0

Integrantes:

Miguel Sierra

Khalil El Hage Kassem

Luis Daniel Fuentes Licero

Universidad del Norte

Barranquilla, Colombia

2021-II

Apartados

1. Sistema

1.1. Usuario
1.2. Billetera
1.3. Bloque
1.4. Transacción
1.5. Estado
1.6. Útil

2. Grafo

2.1. Vector
2.2. Vértice
2.3. Arista
2.4. Grafo

3. UI

3.1. Toast
3.2. Panel de billetera
3.3. Panel de historial de billetera
3.4. Panel de plano cartesiano

1. Sistema

En este apartado se van a explicar las clases sobre las cuales está constituida la base del programa; el funcionamiento de cada una de ellas y las formas en las cuales estas fueron implementadas.

1.1 Usuario

```java
public class User {

    private final String userID;
    private final String name;
    private final String surname;
    private final int identificationNumber;
    private final String email;
    private final String password;

    //Constructor
    public User(String name, String surname, int identificationNumber, String email, String password) {
        this.name = name;
        this.surname = surname;
        this.identificationNumber = identificationNumber;
        this.email = email;
        this.password = password;
        this.userID = calculateHash();
    }

    /*
    * Calcula el hash del usuario, se puede utilizar como ID unica ya que
    * nos aseguramos de que no pueda haber dos usuarios con el mismo correo
     */
    private String calculateHash() {
        return Util.applySha256(
                name
                + surname
                + identificationNumber
                + email
                + password
        );
    }
```

Esta clase sirve para guardar los datos de los usuarios que se registran en el programa. Su ID único se calcula mediante el uso de Sha256 y los datos del usuario. Se asegura que no se repita esa ID, pues verificamos que no se pueda crear un usuario con la misma dirección de correo electrónico que otro usuario existente.

1.2 Billetera

```
public class Wallet {

    private String ownerID;
    private PublicKey publicKey;
    private PrivateKey privateKey;
    private float balance;

    //Constructor normal
    public Wallet(String ownerID) {
        generateKeyPair();
        this.ownerID = ownerID;
        this.balance = 0;

    }
```

Esta clase representa un monedero para el usuario, donde se almacena su dinero y puede realizar transacciones con otros usuarios a través de su billetera. Ella contiene el ID de su dueño asociado, una llave pública con la cual realiza las transacciones y una llave privada con la cual firma las transacciones que se van a realizar por temas de seguridad.

```
//Constructor (abrir archivos)
public Wallet(String ownerID, String publicKeyString, String privateKeyString, float balance) {
    this.ownerID = ownerID;
    Security.addProvider(new org.bouncycastle.jce.provider.BouncyCastleProvider());
    KeyFactory factory = null;
    try {
        factory = KeyFactory.getInstance("ECDSA", "BC");
    } catch (NoSuchAlgorithmException ex) {
        Logger.getLogger(Wallet.class.getName()).log(Level.SEVERE, null, ex);
    } catch (NoSuchProviderException ex) {
        Logger.getLogger(Wallet.class.getName()).log(Level.SEVERE, null, ex);
    }
    byte[] publicKeyByte = Base64.decode(publicKeyString, Base64.NO_WRAP);
    byte[] privateKeyByte = Base64.decode(privateKeyString, Base64.NO_WRAP);
    try {
        publicKey = (PublicKey) factory.generatePublic(new X509EncodedKeySpec(publicKeyByte));
    } catch (InvalidKeySpecException ex) {
        Logger.getLogger(Wallet.class.getName()).log(Level.SEVERE, null, ex);
    }
    try {
        privateKey = (PrivateKey) factory.generatePrivate(new PKCS8EncodedKeySpec(privateKeyByte));
    } catch (InvalidKeySpecException ex) {
        Logger.getLogger(Wallet.class.getName()).log(Level.SEVERE, null, ex);
    }
    this.balance = balance;
}
```

Este segundo constructor es para cuando leemos las billeteras del archivo “billeteras.txt”. Debido a que toda la información está en

String, las llaves tienen que ser convertidas en public key y private key. Para ello, utilizamos una key factory, que utiliza la instancia de encriptación de ECDSA, que agarra el String convertido en Byte y nos devuelve las llaves de la billetera.

```
//Genera las llaves publicas y privadas
public void generateKeyPair() {
    Security.addProvider(new org.bouncycastle.jce.provider.BouncyCastleProvider());
    try {
        KeyPairGenerator keyGen = KeyPairGenerator.getInstance("ECDSA", "BC");
        SecureRandom random = SecureRandom.getInstance("SHA1PRNG");
        ECGenParameterSpec ecSpec = new ECGenParameterSpec("prime192v1");

        keyGen.initialize(ecSpec, random);
        KeyPair keyPair = keyGen.generateKeyPair();

        privateKey = keyPair.getPrivate();
        publicKey = keyPair.getPublic();
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

Esta función utiliza un key generator con la instancia de ECDSA para crear unas llaves aleatorias, una privada y una pública.

1.3 Bloque

```
*/
public class Block {

    private String hash;
    private String previousHash;
    private String merkleRoot;
    private long timeStamp;
    private int nonce;

    //Constructor
    public Block(String previousHash) {
        this.previousHash = previousHash;
        this.timeStamp = new Date().getTime();
        this.hash = calculateHash();
    }

    //Calcula el hash del bloque
    public String calculateHash() {
        String calculatedhash = Util.applySha256(
                previousHash
                + Long.toString(timeStamp)
                + Integer.toString(nonce)
                + merkleRoot
        );
        return calculatedhash;
    }
```

Esta clase representa un bloque del blockchain. Contiene la información con la que fue creada, como su tiempo de creación, una constante arbitraria, y el hash del bloque que le precede. Utiliza esta información para calcular el hash con el que se va a representar este bloque.

```
//Mina el bloque cuando esta lleno
public void mineBlock(int difficulty, ArrayList<Transaction> transactions) {
    merkleRoot = Util.getMerkleRoot(transactions);
    String target = new String(new char[difficulty]).replace('\0', '0');
    while (!hash.substring(0, difficulty).equals(target)) {
        nonce++;
        hash = calculateHash();
    }
}
```

Una vez llenadas las 3 transacciones del bloque, se mina el bloque para determinar el hash final con el merkle root, y con ello el bloque estaría completado.

1.4 Transacción

```
public class Transaction {

    private final String transactionId;
    private PublicKey sender;
    private PublicKey reciepient;
    private final float value;
    private long checksumAlpha;
    private long checksumBeta;
    private static int sequence = 0;

    //Constructor normal
    public Transaction(PublicKey from, PublicKey to, float value) {
        this.sender = from;
        this.reciepient = to;
        this.value = value;
        this.transactionId = calculateHash();

    }
```

Esta clase representa la transacción. Esta clase contiene el dinero transferido, las llaves públicas del remitente y del recipiente y un ID con el que se identifica. Además, tiene unos checksum que actúan como firmas para proteger la transacción.

```java
//Calcula el hash de la transaccion
private String calculateHash() {
    sequence++; //Asegura que no pueda haber 2 transacciones con el mismo hash
    return Util.applySha256(
            Util.getStringFromKey(sender)
            + Util.getStringFromKey(reciepient)
            + Float.toString(value) + sequence
    );
}
```

Esta función, tal como la función de calcular hash en usuario, nos da un ID único con el que se identifica la transacción. Se comprueba que no haya repitencia utilizando una variable estática para la clase transacción, que va aumentando a medida que se usa la clase.

```java
//Constructor (Abrir archivos)
public Transaction(String senderStringKey, String reciepientStringKey
        , float value, long checksumAlpha, long checksumBeta) {
    Security.addProvider(new org.bouncycastle.jce.provider.BouncyCastleProvider());
    KeyFactory factory = null;
    try {
        factory = KeyFactory.getInstance("ECDSA", "BC");
    } catch (NoSuchAlgorithmException ex) {
        Logger.getLogger(Wallet.class.getName()).log(Level.SEVERE, null, ex);
    } catch (NoSuchProviderException ex) {
        Logger.getLogger(Wallet.class.getName()).log(Level.SEVERE, null, ex);
    }
    byte[] senderKeyByte = Base64.decode(senderStringKey, Base64.NO_WRAP);
    byte[] reciepientKeyByte = Base64.decode(reciepientStringKey, Base64.NO_WRAP);
    try {
        this.sender = (PublicKey) factory.generatePublic(new X509EncodedKeySpec(senderKeyByte));
    } catch (InvalidKeySpecException ex) {
        Logger.getLogger(Wallet.class.getName()).log(Level.SEVERE, null, ex);
    }
    try {
        this.reciepient = (PublicKey) factory.generatePublic(new X509EncodedKeySpec(reciepientKeyByte));
    } catch (InvalidKeySpecException ex) {
        Logger.getLogger(Wallet.class.getName()).log(Level.SEVERE, null, ex);
    }

    this.value = value;
    this.checksumAlpha = checksumAlpha;
    this.checksumBeta = checksumBeta;
    this.transactionId = calculateHash();
}
```

Este constructor sirve para crear transacciones desde el archivo “transacciones.txt”. Se aplica el mismo método que en Billetera para convertir las llaves de tipo String a Key.

```java
//genera la firma de la transaccion
public void generateSignature(PrivateKey privateKey) {
    String data = Util.getStringFromKey(sender)
            + Util.getStringFromKey(reciepient)
            + Float.toString(value);
    byte[] dataB = data.getBytes();
    checksumBeta = Util.getCRC32Checksum(dataB);
    data = data + Util.getStringFromKey(privateKey);
    dataB = data.getBytes();
    checksumAlpha = Util.getCRC32Checksum(dataB);
}

//Verifica si hubo algun cambio en los datos de la transaccion
public boolean verifiySignature() {
    String data = Util.getStringFromKey(sender) +
            Util.getStringFromKey(reciepient) + Float.toString(value);
    byte[] dataB = data.getBytes();
    return checksumAlpha - checksumBeta == checksumAlpha - Util.getCRC32Checksum(dataB);
}
```

En la función generar firma, se usa la llave privada del remitente y se la sumamos a la información de la transacción. Posteriormente, se agarran los bytes del String completo y se le calcula el checksum, siendo este una suma de todos los bits del arreglo de Bytes.

1.5 Estado

```java
 */
public class State {

    private final float balanceSender;
    private final float balanceRecipient;

    public State(float b1, float b2) {
        this.balanceSender = b1;
        this.balanceRecipient = b2;
    }

    public float getBalanceSender() {
        return balanceSender;
    }

    public float getBalanceRecipient() {
        return balanceRecipient;
    }

}
```

Esta clase es usada por transacción para describir los balances de las cuentas del recipiente y remitente antes y después de realizada su transacción asociada.

1.6 Útil

Esta es una clase de utilidad con métodos utilizados en todo el sistema.

```
//Encripta la input con Sha256
public static String applySha256(String input) {
    try {
        MessageDigest digest = MessageDigest.getInstance("SHA-256");

        byte[] hash = digest.digest(input.getBytes("UTF-8"));
        StringBuffer hexString = new StringBuffer();
        for (int i = 0; i < hash.length; i++) {
            String hex = Integer.toHexString(0xff & hash[i]);
            if (hex.length() == 1) {
                hexString.append('0');
            }
            hexString.append(hex);
        }
        return hexString.toString();
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

Sha256 es un método que aplica la encriptación Sha256 a una entrada determinada, utilizando UTF-8.

```java
//Recibe un ArrayList de transacciones y calcula la merkleRoot correspondiente
public static String getMerkleRoot(ArrayList<Transaction> transactions) {
    int count = transactions.size();
    ArrayList<String> previousTreeLayer = new ArrayList<String>();
    for (Transaction transaction : transactions) {
        previousTreeLayer.add(transaction.getTransactionId());
    }
    ArrayList<String> treeLayer = previousTreeLayer;
    while (count > 1) {
        treeLayer = new ArrayList<String>();
        for (int i = 1; i < previousTreeLayer.size(); i++) {
            treeLayer.add(applySha256(previousTreeLayer.get(i - 1) + previousTreeLayer.get(i)));
        }
        count = treeLayer.size();
        previousTreeLayer = treeLayer;
    }
    String merkleRoot = (treeLayer.size() == 1) ? treeLayer.get(0) : "";
    return merkleRoot;
}
```

Se utilizan los conjuntos de transacciones para sumar sus encriptaciones y, posteriormente, se llega a la “Merkle root”. Esta es utilizada para comprobar que las transacciones pertenecen al bloque en cuestión.

```java
//Calcula el Checksum del array de bytes dado
public static long getCRC32Checksum(byte[] bytes) {
    Checksum crc32 = new CRC32();
    crc32.update(bytes, 0, bytes.length);
    return crc32.getValue();
}
```

Este método sumas los bits de un arreglo de Bytes, devolviendo el valor resultante (el checksum utilizado en transacción).

2. Grafo

En este apartado se va a explicar la estructura de datos implementada en el programa, con sus respectivas conexiones y dibujado.

2.1 Vector

```java
public class Vector {

    private double x, y;

    public Vector() {
        this.setX(0);
        this.setY(0);
    }

    public Vector(double x, double y) {
        this.setX(x);
        this.setY(y);
    }

    public Vector add(Vector v) {
        return new Vector(x + v.getX(), y + v.getY());
    }

    public Vector sub(Vector v) {
        return new Vector(x - v.getX(), y - v.getY());
    }

    public Vector mul(double m) {
        return new Vector(x * m, y * m);
    }

    public Vector div(double m) {
        return new Vector(x / m, y / m);
    }

    public double dot(Vector v) {
        return x * v.getX() + y * v.getY();
    }

    public double size() {
        return Math.sqrt(x * x + y * y);
    }
```

Esta clase utiliza como parámetros dos números dados para representar un vector bidimensional, en el cual se pueden realizar operaciones matemáticas para determinar diversos resultados.

2.2 Vértice

```
 */
public class Vertex {

    private Object o;
    private Vector pos, disp;
    private Rectangle area;

    public Vertex(Object o) {
        this.o = o;
        this.pos = new Vector((int) (100 + Math.random() * 1000)
                , (int) (100 + Math.random() * 1000));
        disp = new Vector(0, 0);
        area = new Rectangle((int) (pos.getX() - 20)
                , (int) (pos.getY() - 20), 40, 40);
    }

    public void Draw(Graphics2D g) {
        String name = "";
        if (o instanceof User) {
            name = "Usuario";
            g.setColor(Color.yellow);
        } else if (o instanceof Wallet) {
            name = "Billetera";
            g.setColor(Color.green);
        } else if (o instanceof Block) {
            name = "Bloque";
            g.setColor(Color.cyan);
        } else if (o instanceof Transaction) {
            name = "Transacción";
            g.setColor(Color.blue);
        } else if (o instanceof State) {
            name = "Estado";
            g.setColor(Color.pink);
        }
        g.fillOval((int) pos.getX() - 20, (int) pos.getY() - 20, 40, 40);
        g.setColor(Color.black);
        g.drawString(name, (int) (pos.getX() - 20), (int) (pos.getY() - 20));
    }
```

En esta clase actúa mayormente como un contenedor para los diversos objetos que actúan en el sistema. Para su dibujado, tenemos

2 vectores, uno de posición y otro de desplazamiento, y un área rectangular que engloba el vértice.

2.3 Arista

```
public class Edge {

    private final Vertex v;
    private final Vertex u;

    public Edge(Vertex v, Vertex u) {
        this.v = v;
        this.u = u;
    }

    public void Draw(Graphics2D g) {
        g.setColor(Color.black);
        g.drawLine((int) v.getPos().getX(), (int) v.getPos().getY()
                , (int) u.getPos().getX(), (int) u.getPos().getY());
    }

    public Vertex getV() {
        return v;
    }

    public Vertex getU() {
        return u;
    }

}
```

Esta clase representa una conexión entre 2 vértices. Es representada por medio de una línea recta.

2.4 Grafo

```java
public class Graph {

    private ArrayList<Vertex> vertices;
    private ArrayList<Edge> edges;
    String masterID;

    //Constructor del grafo, se encarga de cargar la forma inicial del grafo
    public Graph() {
        vertices = new ArrayList();
        edges = new ArrayList();
        masterID = Util.applySha256("Master" + "Admin" + 0000000000 + "admin" + "admin");

        File folder = new File("data");
        if (!folder.exists()) {
            try {
                folder.mkdir();

                User genesisUser = new User("Master", "Admin", 0000000000, "admin", "admin");
                Block genesisBlock = new Block("0");
                Wallet genesisWallet = new Wallet(genesisUser.getID());
                genesisWallet.setBalance(999999);

                Vertex v = new Vertex(genesisUser);
                Vertex u = new Vertex(genesisBlock);
                Vertex vl = new Vertex(genesisWallet);
                vertices.add(v);
                vertices.add(u);
                vertices.add(vl);
                edges.add(new Edge(v, u));
                edges.add(new Edge(v, vl));
            } catch (Exception e) {
                System.out.println("\n ***** ERROR ***** "
                        + "\nLocation: Graph class constructor"
                        + "\nError: " + e
                        + "\n ***************** \n");
            }
        } else {
            openFiles();
        }
    }
```

En nuestra implementación de la clase grafo utilizamos 2 ArrayList, uno de vértices, y otro de aristas. El constructor, en el caso que la carpeta data no exista, crea la carpeta y crea los usuario, bloque y billetera génesis, los cuales sirven como base para el resto de la estructura. En caso de que ya hayan datos existentes, se procede a abrir los archivos y crear la estructura de datos a partir de su información.

```java
//Abrre y escribe los datos del grafo
private void openFiles() {
    //Abre los datos de usuarios
    File file = new File("data/Usuarios.txt");
    if (!file.exists()) {
        try {
            file.createNewFile();
        } catch (IOException ex) {
            System.out.println("\n ***** ERROR ***** "
                    + "\nLocation: Graph.openFiles(), While opening users file"
                    + "\nError: " + ex
                    + "\n ***************** \n");
        }
    } else {
        try {
            Scanner in = new Scanner(file);
            String line = in.nextLine();
            String[] data = line.split(",");

            User u = new User(data[0], data[1], Integer.parseInt(data[2]), data[3], data[4]);
            vertices.add(new Vertex(u));

            while (in.hasNextLine()) {
                line = in.nextLine();
                data = line.split(",");
                u = new User(data[0], data[1], Integer.parseInt(data[2]), data[3], data[4]);
                insertUserVertex(u);
            }

        } catch (Exception ex) {
            System.out.println("\n ***** ERROR ***** "
                    + "\nLocation: Graph.openFiles(), While inserting users to graph"
                    + "\nError: " + ex
                    + "\n ***************** \n");
        }
    }
```

```
//Abre los datos de billeteras
file = new File("data/billeteras.txt");
if (!file.exists()) {
    try {
        file.createNewFile();
    } catch (IOException ex) {
        System.out.println("\n ***** ERROR ***** "
                + "\nLocation: Graph.openFiles(), While opening wallets file"
                + "\nError: " + ex
                + "\n ***************** \n");
    }
} else {
    try {
        Scanner in = new Scanner(file);
        while (in.hasNextLine()) {
            String line = in.nextLine();
            String data[] = line.split(",");

            Wallet w = new Wallet(data[0], data[1], data[2], Float.parseFloat(data[3]));
            Vertex v = searchUserVertex(w.getOwnerID());
            if (v != null) {
                insertWalletVertex(v, w);
            }

        }

    } catch (Exception ex) {
        System.out.println("\n ***** ERROR ***** "
                + "\nLocation: Graph.openFiles(), While inserting wallets to graph"
                + "\nError: " + ex
                + "\n ***************** \n");
    }
}
```

```
    //Abre los datos de transacciones
    file = new File("data/transacciones.txt");
    if (!file.exists()) {
        try {
            file.createNewFile();
        } catch (IOException ex) {
            System.out.println("\n ***** ERROR ***** "
                    + "\nLocation: Graph.openFiles(), While opening transactions file"
                    + "\nError: " + ex
                    + "\n ***************** \n");
        }
    } else {
        try {
            Scanner in = new Scanner(file);
            Block genesisBlock = new Block("0");
            Vertex v = new Vertex(genesisBlock);
            vertices.add(v);
            Edge e = new Edge(vertices.get(0), v);
            edges.add(e);

            while (in.hasNextLine()) {
                String line = in.nextLine();
                String data[] = line.split(",");
                Transaction t = new Transaction(data[0], data[1]
                        , Float.parseFloat(data[2])
                        , Long.parseLong(data[3]), Long.parseLong(data[4]));
                if (t.verifiySignature()) {
                    insertTransactionVertex(t);
                } else {
                }

            }

        } catch (Exception ex) {
            System.out.println("\n ***** ERROR ***** "
                    + "\nLocation: Graph.openFiles(), While inserting transactions to graph"
                    + "\nError: " + ex
                    + "\n ***************** \n");
        }
    }
    verifyBlocks();
    verifyWallets();
}
```

Al momento de abrir los archivos, revisa en cada uno de ellos su información, abriendo por orden “jerárquico” (usuarios tienen billeteras, desde las cuales se realizan transacciones). Esta información es procesada y se traslada a la estructura de datos del grafo. Al terminar este proceso, se verifica el blockchain en caso de existir algún error y, posteriormente, se verifican las billeteras por si alguien intenta manipular el archivo correspondiente.

```java
//Guarda los datos del grafo
public void saveFiles() {
    //Guarda los datos de los usuarios
    File file = new File("data/Usuarios.txt");
    try {
        file.createNewFile();
    } catch (IOException ex) {
        System.out.println("\n ***** ERROR ***** "
                + "\nLocation: Graph.saveFiles(), while creating a new users file"
                + "\nError: " + ex
                + "\n ***************** \n");
    }
    try (FileWriter fw = new FileWriter(file, false)) {

        BufferedWriter bw = new BufferedWriter(fw);
        for (Vertex v : vertices) {
            if (v.getO() instanceof User) {
                User o = (User) v.getO();
                bw.write(o.saveString());
                bw.flush();
            }
        }
        bw.close();
        fw.close();

    } catch (Exception e) {
        System.out.println("\n ***** ERROR ***** "
                + "\nLocation: Graph.saveFiles(), while saving data to users file"
                + "\nError: " + e
                + "\n ***************** \n");
    }
```

```
//Guarda los datos de las transacciones
file = new File("data/Transacciones.txt");
try {
    file.createNewFile();
} catch (IOException ex) {
    System.out.println("\n ***** ERROR ***** "
            + "\nLocation: Graph.saveFiles(), while creating a new transactions file"
            + "\nError: " + ex
            + "\n ****************** \n");
}
try (FileWriter fw = new FileWriter(file, false)) {

    BufferedWriter bw = new BufferedWriter(fw);
    for (Vertex v : vertices) {
        if (v.getO() instanceof Transaction) {
            Transaction o = (Transaction) v.getO();
            bw.write(o.saveString());
            bw.flush();
        }
    }
    bw.close();
    fw.close();

} catch (Exception e) {
    System.out.println("\n ***** ERROR ***** "
            + "\nLocation: Graph.saveFiles(), while saving data to transactions file"
            + "\nError: " + e
            + "\n ****************** \n");

}
```

```
    //Guarda los datos de las billeteras
    file = new File("data/Billeteras.txt");
    try {
        file.createNewFile();
    } catch (IOException ex) {
        System.out.println("\n ***** ERROR ***** "
                + "\nLocation: Graph.saveFiles(), while creating a new wallets file"
                + "\nError: " + ex
                + "\n ***************** \n");
    }
    try (FileWriter fw = new FileWriter(file, false)) {

        BufferedWriter bw = new BufferedWriter(fw);
        for (Vertex v : vertices) {
            if (v.getO() instanceof Wallet) {
                Wallet o = (Wallet) v.getO();
                bw.write(o.saveString());
                bw.flush();
            }
        }
        bw.close();
        fw.close();

    } catch (Exception e) {
        System.out.println("\n ***** ERROR ***** "
                + "\nLocation: Graph.saveFiles(), while saving data to wallets file"
                + "\nError: " + e
                + "\n ***************** \n");

    }
}
```

En estos métodos se guardan los datos del grafo en 3 distintos archivos; usuarios.txt; billeteras.txt y transacciones.txt. El método revisa vértice por vértice y guardar la información en su correspondiente archivo.

```
//Inserta un nuevo usuaro al grafo
public void insertUserVertex(User user) {
    int randomUser = randUserVertex();
    Vertex v = vertices.get(randomUser);
    Vertex u = new Vertex(user);
    vertices.add(u);
    edges.add(new Edge(v, u));
}
```

```java
//Inserta una nueva billetera al grafo
public void insertWalletVertex(Vertex v, Wallet w) {
    Vertex u = new Vertex(w);
    vertices.add(u);
    edges.add(new Edge(v, u));

}

//verifica si la transaccion es realizable y la inserta en el grafo
public boolean insertTransactionVertex(Wallet a, PublicKey b, float value) {
    verifyWalletBalance(a);

    if (!(a.getPublicKey().equals(b)) && a.getBalance() >= value && value >= 0) {
        Transaction t = new Transaction(a.getPublicKey(), b, value);
        t.generateSignature(a.getPrivateKey());
        insertTransactionVertex(t);
        return true;
    }else{
        return false;
    }
}

//Inserta un nuevo vertice transaccion al grafo (y sus vertices de estado)
private void insertTransactionVertex(Transaction t) {
    Vertex v = getLastBlock();
    Vertex u = new Vertex(t);
    vertices.add(u);
    edges.add(new Edge(v, u));

    Wallet a = searchWallet(t.getSender());
    Wallet b = searchWallet(t.getReciepient());
    Vertex u1 = new Vertex(new State(a.getBalance(), b.getBalance()));
    a.setBalance(a.getBalance() - t.getValue());
    b.setBalance(b.getBalance() + t.getValue());
    Vertex u2 = new Vertex(new State(a.getBalance(), b.getBalance()));
    vertices.add(u1);
    vertices.add(u2);
    edges.add(new Edge(u, u1));
    edges.add(new Edge(u, u2));
}
```

Cada uno de estos métodos inserta el objeto correspondiente en un vértice, el cual pasa a formar parte de la estructura de datos. En insertUserVertex, el nuevo vértice es conectado de manera aleatoria con otro vértice que contenga un objeto usuario. En insertWalletVertex, el nuevo vértice es conectado al vértice de tipo usuario proporcionado en los parámetros. En insertTransaccionVertex primero se comprueba que el remitente y el recipiente no sean la misma billetera. Posteriormente, se comprueba que el remitente si posea la cantidad de dinero que se desea

transferir. Una vez comprobadas las condiciones, se conecta el nuevo vértice de transacción en el último bloque de la blockchain y, a ese vértice, se le conectan dos vértices de estado, que indican el antes y el después en el balance de las dos billeteras.

```
//Busca el vertice que contiene el usuario con el ID dado
public Vertex searchUserVertex(String userID) {
    for (Vertex v : vertices) {
        if (v.getO() instanceof User) {
            User u = (User) v.getO();
            if (u.getID().equals(userID)) {
                return v;
            }
        }
    }
    return null;
}

//Busca el usuario con el correo y password correspondiente
public User searchLoginUser(String email, String password) {
    for (Vertex v : vertices) {
        if (v.getO() instanceof User) {
            User o = (User) v.getO();
            if (o.getEmail().equals(email) && o.getPassword().equals(password)) {
                return o;
            }
        }
    }
    return null;
}
```

```java
//Busca una billetera con la llave dada
public Wallet searchWallet(PublicKey key) {
    for (Vertex v : vertices) {
        if (v.getO() instanceof Wallet) {
            Wallet w = (Wallet) v.getO();
            if (w.getPublicKey().equals(key)) {
                return w;
            }
        }
    }
    return null;
}

//Busca las billeteras correspondientes al usuario con el ID dado
public ArrayList<Wallet> searchUserWallets(String uID) {
    ArrayList<Wallet> wallets = new ArrayList();
    for (Vertex v : vertices) {
        if (v.getO() instanceof Wallet) {
            Wallet w = (Wallet) v.getO();
            if (w.getOwnerID().equals(uID)) {
                wallets.add(w);
            }
        }
    }
    return wallets;
}

//Busca todas las transacciones relacionadas con la billetera con la llave dada
public ArrayList<Transaction> searchWalletTransactions(PublicKey key) {
    ArrayList<Transaction> transactions = new ArrayList();
    for (Vertex v : vertices) {
        if (v.getO() instanceof Transaction) {
            Transaction o = (Transaction) v.getO();
            if (o.getSender().equals(key) || o.getReciepient().equals(key)) {
                transactions.add(o);
            }
        }
    }
    return transactions;
}
```

En estos métodos se itera a través del Arraylist buscando los vértices con la información relacionada a los parámetros dados.

```
//Verifica si la cadena de bloques esta valida, sino la borra desde el error
private void verifyBlocks() {
    Vertex genb = getFirstBlock();
    verifyBlocks(genb);
}

private void verifyBlocks(Vertex v) {
    Edge edge = null;
    for (Edge e : edges) {
        if (e.getV() == v && e.getU().getO() instanceof Block) {
            edge = e;
            break;
        }
    }
    if (edge != null) {
        Block pB = (Block) edge.getV().getO();
        Block cB = (Block) edge.getU().getO();
        if (!pB.getHash().equals(cB.getPreviousHash())) {
            edges.remove(edge);
            deletefrom(edge.getU(), new Container());
        } else {
            verifyBlocks(edge.getU());
        }
    }
}

private Graph deletefrom(Vertex v, Container c) {
    Edge edge = null;
    for (Edge e : edges) {
        if (e.getV() == v) {
            edge = e;
            c.vertices.add(v);
            c.edges.add(e);

        }
    }
    if (edge == null) {
        vertices.removeAll(c.vertices);
        edges.removeAll(c.edges);
    } else {
        deletefrom(edge.getU(), c);
    }
    return null;
}
```

Estos métodos verifican que la blockchain esté correcta. Empieza en el primer bloque de la blockchain y se itera a través de cada arista de la blockchain, verificando que el bloque inicial de la arista tenga el mismo hash que el hashAnterior del bloque final de la arista. En caso que se encuentre una arista donde esa condición no se cumpla,

entonces se empieza a eliminar desde esa parte, en adelante, al blockchain.

```
//Verifica si todas las billeteras tienen su balance correcto
private void verifyWallets() {
    for (Vertex v : vertices) {
        if (v.getO() instanceof Wallet) {
            Wallet w = (Wallet) v.getO();
            verifyWalletBalance(w);
        }
    }
}

//Verifica que la billetera tiene el balance que dice que tiene, sino las corrige
private void verifyWalletBalance(Wallet a) {
    if (!a.getOwnerID().equals(masterID)) {
        float balance = 0;
        ArrayList<Transaction> transactions = searchWalletTransactions(a.getPublicKey());
        for (Transaction t : transactions) {
            if (a.getPublicKey().equals(t.getSender())) {
                balance -= t.getValue();
            } else if (a.getPublicKey().equals(t.getReciepient())) {
                balance += t.getValue();
            }
        }
        if (a.getBalance() != balance) {
            a.setBalance(balance);
        }
    }
}
```

verifyWallets itera a través de cada billetera, llamando a verifyWalletBalance para verificar todas las transacciones de esa billetera, comprobando que sí se tenga el balance que debería tener.

```
//Se encarga de dibujar el grafo
public void draw(Graphics2D g2) {

    for (Edge e : edges) {
        e.Draw(g2);
    }

    for (Vertex v : vertices) {
        v.Draw(g2);
    }
}
```

Este método llama a dibujar todos los vértices y aristas del grafo.

3. UI

En este apartado vamos a explicar las diversas clases que nos fueron de utilidad a la hora de implementar la interfaz de usuario.

3.1 Panel de billetera

```
/*
 * @Group #9
 */
public class WalletPanel extends javax.swing.JPanel {

    public WalletPanel(int n, String publicKey, String privateKey, Float balance) {
        initComponents();
        titleLabel.setText(titleLabel.getText() + n);
        jTextArea1.setText("Llave pública: " + publicKey
                + "\nLlave privada: " + privateKey
                + "\nBalance: " + balance);
    }
```

Este es un JPanel que instanciamos dinámicamente en nuestro programa a la hora de crear una nueva billetera. Este panel contiene la información de la billetera, siendo la llave pública, la llave privada y su balance.

3.2 Panel de historial de billetera

```
 */
public class WalletHistoryPanel extends javax.swing.JPanel {

    public WalletHistoryPanel(int n, float b, PublicKey pbk, ArrayList<Transaction> transactions) {
        initComponents();
        titleLabel.setText(titleLabel.getText() + n);
        balanceLabel.setText(balanceLabel.getText() + b);
        for (Transaction t : transactions) {
            if (t.getSender() == pbk) {
                transactionHistoryTA.setText(transactionHistoryTA.getText() + "\n- Enviado: " + t.getValue());
            } else {
                transactionHistoryTA.setText(transactionHistoryTA.getText() + "\n- Recibido: " + t.getValue());
            }
        }
    }
```

Al igual que el panel de billetera, este panel sirve para mostrar la información de la billetera, en este caso, el historial de todas las transacciones relacionadas a su llave pública y su balance total.

3.3 Panel de plano cartesiano

```
 */
public class TwoDPlane extends JPanel {

    final int WIDTH;
    final int HEIGHT;
    int xPan;
    int yPan;
    float scale;
    boolean drawBackground;
    Graph graph;

    int width;
    int height;
    int area;
    double k;
    double temp;
    double coolingRate;
    boolean equilibriumReached, running;
    int iteracion;
    Thread t;
```

```java
public TwoDPlane(JPanel placeHolder, Graph g) {
    initComponents();
    //placeHolder.setVisible(false);
    this.setSize(placeHolder.getSize());
    this.setBorder(placeHolder.getBorder());
    this.setBackground(Color.white);
    this.setVisible(true);

    WIDTH = this.getWidth();
    HEIGHT = this.getHeight();
    xPan = WIDTH / 2;
    yPan = HEIGHT / 2;
    this.drawBackground = true;
    scale = 1;
    this.graph = g;

    area = WIDTH * HEIGHT;
    temp = WIDTH / 10;
    coolingRate = 0.01;
    equilibriumReached = false;
    running = false;
    k = Math.sqrt(area / graph.getVertices().size());
    iteracion = 0;

    MouseAdapter ma = new MouseAdapter() {...57 lines };
    addMouseListener(ma);
    addMouseWheelListener(ma);
    addMouseMotionListener(ma);

}
```

Este panel es un un plano cartesiano bidimensional en el cual se dibuja el grafo. Dentro del constructor se inicializan todas las variables, se determinan las dimensiones del plano a través de un panel PlaceHolder y se agarra la instancia del grafo que se va a dibujar. Para interactuar con el plano, instanciamos un mouseAdapter que se encargará de las funciones de drag, zoom, y clickear para información del grafo.

```
private void showInfo(Object o) {
    if (o instanceof User) {
        User u = (User) o;
        new Toast.ToastSuccessful(
                "Usuario",
                "Información",
                u.toString(),
                Toast.LONG_DELAY
        );

    } else if (o instanceof Wallet) {
        Wallet w = (Wallet) o;
        new Toast.ToastSuccessful(
                "Billetera",
                "Información",
                w.toString(),
                Toast.LONG_DELAY
        );

    } else if (o instanceof Block) {
        Block b = (Block) o;
        new Toast.ToastSuccessful(
                "Bloque",
                "Información",
                b.toString(),
                Toast.LONG_DELAY
        );

    } else if (o instanceof Transaction
        Transaction t = (Transaction) o
        new Toast.ToastSuccessful(
                "Transacción",
                "Información",
                t.toString(),
                Toast.LONG_DELAY
        );

    } else if (o instanceof State) {
        State e = (State) o;
        new Toast.ToastSuccessful(
                "Estado",
                "Información",
                e.toString(),
                Toast.LONG_DELAY
        );
    }
}
```

El método showInfo es llamado cuando el usuario clickea un vértice en el grafo y llama un toast con la información del vértice.

```java
@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);

    Graphics2D g2 = (Graphics2D) g;
    g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
    g2.translate(xPan, yPan);
    g2.scale(scale, scale);

    if (drawBackground) {
        drawBackground(g2);
    }

    graph.draw(g2);

}
```

```java
private void drawBackground(Graphics2D g2) {
    int x1 = (int) ((-1 * (xPan - WIDTH / 2) - WIDTH / 2) / scale);
    int x2 = (int) ((-1 * (xPan - WIDTH / 2) + WIDTH / 2) / scale);
    int y1 = (int) ((-1 * (yPan - HEIGHT / 2) - HEIGHT / 2) / scale);
    int y2 = (int) ((-1 * (yPan - HEIGHT / 2) + HEIGHT / 2) / scale);

    for (int i = x1; i < x2; i++) {
        if (i % 10 == 0) {
            g2.setColor(new Color(224, 224, 224));
            g2.setStroke(new BasicStroke(1));
            g2.draw(new Line2D.Float(i, y1, i, y2));
        }
    }
    for (int w = y1; w < y2; w++) {
        if (w % 10 == 0) {
            g2.setColor(new Color(224, 224, 224));
            g2.setStroke(new BasicStroke(1));
            g2.draw(new Line2D.Float(x1, w, x2, w));
        }
    }
    for (int i = x1; i < x2; i++) {
        if (i % 50 == 0) {
            g2.setColor(new Color(192, 192, 192));
            g2.setStroke(new BasicStroke(2));
            g2.draw(new Line2D.Float(i, y1, i, y2));
        }
    }
    for (int w = y1; w < y2; w++) {
        if (w % 50 == 0) {
            g2.setColor(new Color(192, 192, 192));
            g2.setStroke(new BasicStroke(2));
            g2.draw(new Line2D.Float(x1, w, x2, w));
        }
    }

    g2.setStroke(new BasicStroke(2));
    g2.setColor(new Color(0, 0, 0));
    g2.draw(new Line2D.Float(x1, 0, x2, 0));
    g2.draw(new Line2D.Float(0, y1, 0, y2));

    //Reset config
    g2.setStroke(new BasicStroke(1));
}
```

En el paintComponent se utiliza traslación y escala para simular movimiento y zoom en el plano. Se dibuja el plano y encima de ello se llama a dibujar el grafo.

```
public void run() {
    running = true;
    t = new Thread(new Runnable() {
        @Override
        public void run() {
            while (!(equilibriumReached) && iteracion < 1000) {
                System.out.print("");
                if (running) {
                    try {
                        Thread.sleep(5);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    //Calcula la uerza repulsiva sobre cada vertice
                    for (Vertex v : graph.getVertices()) {
                        v.setDisp(new Vector(0, 0));
                        for (Vertex u : graph.getVertices()) {
                            if (v != u) {
                                Vector d = v.getPos().sub(u.getPos());
                                v.setDisp(v.getDisp().add((d.div(d.size())).mul(fr(d.size()))));
                            }
                        }
                    }
                    //Calcula la fuerza atractiva sobre cada vertice
                    for (Edge e : graph.getEdges()) {
                        Vector d = e.getV().getPos().sub(e.getU().getPos());
                        e.getV().setDisp(e.getV().getDisp().sub(
                                d.div(d.size()).mul(fa(d.size()))));
                        e.getU().setDisp(e.getU().getDisp().add(
                                d.div(d.size()).mul(fa(d.size()))));
                    }
                    //Mueve los vertices
                    equilibriumReached = true;
                    for (Vertex v : graph.getVertices()) {
                        if (v.getDisp().size() > 15) {
                            equilibriumReached = false;
                        }
                        v.setPos(v.getPos().add(v.getDisp().div(
                                v.getDisp().size()).mul(Math.min(
                                        v.getDisp().size(), temp))));
                    }
                    //Disminuye la temperatura del sistema
                    temp = Math.max(temp * (1 - coolingRate), 1);

                    repaint();
                    iteracion++;
                }
            }
        }
    });
    t.start();

}
```

```
private double fa(double x) {
    return (Math.pow(x, 2) / k);
}

private double fr(double x) {
    return (Math.pow(k, 2) / x);
}
```

En el método run() se llama un nuevo thread en el cual vamos a trazar el grafo en tiempo real. El algoritmo que utilizamos para plotear el grafo se llama algoritmo de grafo de fuerzas dirigidas de Fruchterman y Reingold [FR91]. Se trata de considerar los vértices del grafo como partículas cargadas positivamente y las aristas como resortes y utilizar sus interacciones para plotear el grafo mientras que baja la temperatura del sistema.

La fuerza atractiva fa y la fuerza repulsiva fr se calculan de la siguiente forma:

$$f_a(d) = d^2/k, \qquad f_r(d) = -k^2/d,$$

Siendo d la distancia entre los vértices y k siendo:

$$k = C\sqrt{\frac{area}{number\ of\ vertices}}.$$

Se calculan en el código de la siguiente forma, primera la fuerza repulsiva para cada par de vértices:

```
{calculate repulsive forces}
for v in V do begin
    {each vertex has two vectors: .pos and .disp
    v.disp := 0;
    for u in V do
        if (u ≠ v) then begin
        {δ is the difference vector between the positions of the two vertices}
            δ := v.pos − u.pos;
            v.disp := v.disp + (δ/|δ|) * f_r(|δ|)
        end
end
```

Después se calcula la fuerza atractiva entre cada par de vértices unidos por una arista:

```
{calculate attractive forces}
for e in E do begin
    {each edges is an ordered pair of vertices .vand.u}
    δ := e.v.pos − e.u.pos;
    e.v.disp := e.v.disp − (δ/|δ|) * f_a(|δ|);
    e.u.disp := e.u.disp + (δ/|δ|) * f_a(|δ|)
end
```

Y al terminar de calcular el vector de desplazamiento, simplemente calculamos como queda el vector posición de cada vértice y decm.

```
for v in V do begin
    v.pos := v.pos + (v.disp/|v.disp|) * min(v.disp, t);
    v.pos.x := min(W/2, max(−W/2, v.pos.x));
    v.pos.y := min(L/2, max(−L/2, v.pos.y))
end
{reduce the temperature as the layout approaches a better configuration}
t := cool(t)
```

Después se llama repaint() para que dibuje el grafo. Este ciclo sólo termina cuando el grafo llega a un equilibrio o la iteración se pasa de 1000 iteraciones.

```
public void stop() {
    running = false;
    iteracion = 0;
    equilibriumReached = false;
}
```

El método stop solo pausa el thread.