Ingeniería Matemática e Inteligencia Artificial

Memoria – Práctica final

Blockchain

Asignatura de Fundamentos de Sistemas Operativos

Alejandro Martínez de Guinea García - 202113492
Lara Ocón Madrid - 202115710

Índice

1. Introducción 2

2. BLOCKCHAIN.PY 3

2.1. Clase Bloque: 3

2.2. Clase Blockchain : 3

3. BLOCKCHAIN_APP.PY 6

4. FUNCIONAMIENTO 9

11. CONCLUSIONES 16

1. Introducción

El sistema blockchain es una tecnología que ha evolucionado mucho en los últimos años y que consiste en una forma de mantener un registro seguro y descentralizado. Como su nombre indica, almacena una cadena de bloques enlazados entre sí, que a su vez contienen información acerca de diferentes cosas. Lo más común de encontrar en una blockchain son transacciones económicas, normalmente asociadas a dinero digital (por ejemplo, Bitcoin o Etherium). El objetivo de esta práctica es simular una aplicación que permita almacenar transacciones utilizando este sistema blockchain empleando las herramientas Python y Postman para ejecutar los programas.

Para ello, hemos desarrollado nuestra propia librería Blockchain, con todas las funciones necesarias para llevar a cabo la práctica, y hemos diseñado nuestro programa aplicación descentralizada que permite al usuario interactuar con el ordenador y almacenar los datos que requiera.

En este informe, vendrán detallados tanto la librería Blockchain como la aplicación y dos ejemplos de ejecución desde Postman y Python.

2. BLOCKCHAIN.PY

En la librería Blockchain.py hemos desarrollado todas las clases y métodos asociados a ellas, necesarios para formar la estructura de la Blockchain. Para ello, ha sido necesario importar las siguientes librerías: "json" (para el formato de los bloques y de la cadena), "hashlib" (para poder asignar un hash a cada bloque)," time" (para guardar los instantes de tiempo en los que se van creando los bloques) y finalmente de "typing" hemos importado "List" y "Dict" (lo usaremos para indicar los inputs y outputs de algunas funciones.

2.1. Clase Bloque:

Esta será la estructura dentro de la cual almacenaremos un conjunto de transacciones y toda la información relativa al bloque, para después introducirlo a la Blockchain.

A continuación, se muestran los métodos de dicha clase:

2.1.1. Método constructor de la clase (__init__) :

Este método recibe los parámetros indice (entero), transacciones (lista de diccionarios), timestamp (float), hash_previo (string) y prueba (entero que por defecto se inicializa a 0). Todos estos inputs se los hemos asigando como atributos al objeto Bloque que creamos.

2.1.2. Calcular_hash :

Este método transforma el bloque a un string mediante la función json.dumps(), transformando el objeto bloque de un diccionario a un json. Este json se lo pasamos a la funcion sha256 de la librería 'hashlib' para que nos devuelva un hash para dicho bloque.

2.1.3. toDict() :

Esta función transforma el objeto bloque a un diccionario ordenado.

2.2. Clase Blockchain :

Esta será la estructura que emplearemos para almacenar los bloques que vayamos creando y las transacciones que aún no hayan sido minadas. Tratando de que esta sea lo más rígida, segura y muy difícil de modificar (tratamos de hacer una simplificación de la Blockchain real).

A continuación, se muestran los métodos de dicha función:

2.2.1. Método constructor de la clase (__init__) :

Este método no recibe nada por argumentos, puesto que cada vez que se crea un objeto Blockchain, sus atributos, "cadena_bloques" y "transacciones_no_confirmadas", son listas vacías. Además, contiene un tercer atributo, "dificultad", que establecemos como 4, y será el número de 0 que debe contener delante cada hash de cada bloque que tratemos de minar a nuestra Blockchain.

2.2.2. *Primer_bloque:*

Esta función es la que establece el primer bloque de la Blockchain (al implementar el programa Blockchain_app, la llamaremos justo después de inicializar la cadena para que así inicialmente, siempre tenga un bloque el mismo bloque.

Para ello, dentro de la función creamos un objeto bloque con los siguientes atributos: índice = 1, transacciones = [] , timestamp = 0, hash_previo = "1" y prueba = 0. Tras esto, calculamos el hash de dicho bloque con la función "calcular_hash" de la clase bloque, ya definida anteriormente. En caso de que dicho hash no contenga delante tantos "0" como indique el atributo "dificultad" de la clase, irá sumándole 1 al atributo "prueba" del bloque, hasta que el hash calculado cumpla con la condición. Una vez se encuentre dicho hash, añadiremos a "cadena_bloques" el bloque.

2.2.3. *Nuevo_bloque():*

Este método debe recibir por argumento un string con el hash previo al nuevo bloque que vamos a crear. De esta forma, devuelve un objeto de la clase "bloque" cuyo índice será la última posición de la cadena, sus transacciones serán todas aquellas acumuladas en el atributo "transacciones_no_confirmadas" de la Blockchain, su timestamp será el instante en el que creamos dicho objeto (lo obtenemos introduciendo time.time()), el hash_previo será el que nos hayan pasado por argumento, y finalmente, prueba será 0.

2.2.4. *Nueva_transaccion()*

Este método, recibirá como argumentos origen y destino (string ambos), y cantidad (entero). La función creará un diccionario con la información de la transacción y la añadirá a la lista 'self.transacciones_no_confirmadas'. Como output devolverá el índice del último bloque de la cadena.

2.2.5. *Prueba_trabajo()*

Esta función calculará el hash de un bloque introducido por argumento, y modificará su atributo 'prueba', hasta que el hash del bloque contenga 4 ceros a la izquierda. Una vez se cumpla dicha condición, devolverá el hash final del bloque.

2.2.6. *Prueba_valida()*

El método prueba_valida, recibirá por argumentos un bloque y un hash, y lo que hará será : 1. Comprobar que el hash introducido por comienza con tantos ceros como el atributo 'dificultad' de la Blockchain indique. En caso de no pasar esta comprobación devolverá False y saldrá de la función. En caso contrario, pasaríamos a la comprobación 2. Que el hash introducido por argumentos sea el mismo que se obtiene al calcular el hash del bloque, que en caso de no pasar la comprobación devolverá False y en caso contrario True.

2.2.7. *Integra_bloque()*

Este método integrará correctamente los bloques a la cadena de bloques. Por ello recibirá como inputs : bloque_nuevo (objeto de la clase Bloque), y hash_prueba (string).

Antes de proceder a integrar el bloque, realizará una serie de comprobaciones : 1) deberá devolver True en prueba_valida (comprueba que el hash empieza por el número de ceros adecuado y coincide con el hash obtenido al llamar a calcular_hash del bloque). Además, realiza una comprobación adicional que consiste en ver que el hash del bloque anterior coincide con el atributo 'hash_previo' del bloque que queremos integrar (esta última condición es muy importante para lograr una Blockchain robusta y difícil de modificar). Si las anteriores comprobaciones no se superan devolverá False, y en caso contrario procederá a añadir el bloque. Antes de inserta el bloque en la cadena, establecerá que el hash del bloque es el hash_prueba que nos habían pasado por argumento, inserta el bloque en la lista 'self.cadena_bloques', y vuelve a poner la lista 'self.transacciones_no_confirmadas' como una lista vacía, hecho esto devolverá True finalizando así la ejecución del método.

2.2.8. Last_block()

Este último método de la clase nos devolverá el objeto bloque justo al final de la cadena, es decir, el último. Esto es algo que nos será útil más adelante en Blockchain_app.py a la hora de minar un bloque y añadir su hash previo.

3. BLOCKCHAIN_APP.PY

El fichero Blockchain_app.py era el que nos permitía ejecutar y desplegar la aplicación, y de esta forma definir un sistema blockchain en una red de nodos descentralizada. Para ello hemos requerido el uso de las siguientes librerías:

1. Blockchain: librería anteriormente definida
2. flask: de cara a definir el programa como aplicación y poder definir las rutas a las que hay que llamar para ejecutar las funciones
3. argparse: para definir el argumento de entrada que determina el puerto a usar
4. multiprocessing: de esta librería usamos los semáforos para poder ejecutar la copia de seguridad sin conflictos
5. threading: creamos el hilo de la copia de seguridad con esta librería
6. pandas: empleamos la función *to_datetime* para guardar la fecha en la copia de seguridad
7. time: para medir los 60 segundos, de cara a llamar al hilo de copia de seguridad
8. json: para darle formato a las cadenas cuando haya que manipularlas
9. platform: esta librería nos permite obtener información de la máquina y sistema empleados
10. requests: esta librería nos permite usar los métodos post y get que serán llamadas al ejecutar la aplicación

3.1. Funciones

3.1.1. nueva_transacción()

En esta función recibimos la transacción enviada al nodo en cuestión, y la almacenamos para en un futuro integrarlas en un bloque. Dicha transacción debe tener los argumentos "origen", "destino" y "cantidad". De no ser así, mostrará un error por pantalla definiendo que faltan valores por introducir. Una vez comprobado esto, llamamos a la función *nueva_transacción()* de la librería Blockchain y se muestra por pantalla en que bloque se introducirá dicha transacción.

El método empleado para llamar a esta función es el método POST, y la ruta a seguir es "/transacciones/nueva".

3.1.2. blockchain_completa()

El objetivo de esta función es mostrar por pantalla la blockchain almacenada hasta el momento por el respectivo nodo. Para ello, lee la cadena almacenada y cada bloque lo parsea utilizando la función *toDict()* de la librería Blockchain.

El método empleado para llamar a esta función es el método GET, y la ruta a seguir es "/chain".

3.1.3. minar()

El objetivo de esta función es incluir todas las transacciones colgadas en un bloque y minar dicho bloque para integrarlo en la blockchain. Para ello, primero llamamos a la función *resolver_conflictos()*, de manera que no mine el bloque en caso de haber un conflicto. Después de eso, comprueba que efectivamente existen transacciones que incluir. De no ser así, muestra un error por pantalla diciendo el siguiente mensaje: "No es posible crear un nuevo bloque. No hay transacciones". En el caso de superar ambas comprobaciones pasa a integrar el bloque. Define el *hash_previo*

del nuevo bloque y llama a las funciones *nuevo_bloque()* y *prueba_trabajo()* de la librería Blockchain para obtener los argumentos necesarios para integrar el bloque. El método empleado para llamar a esta función es el método GET, y la ruta a seguir es "/minar".

3.1.4. *hilo_copia_seguridad()*

Esta función es llamada por un hilo en el main, de manera que está constantemente ejecutándose mientras el programa esté activo. Lo que hace es crear una copia de seguridad de la blockchain en un fichero *.json* cada 60 segundos. Para ello, utilizamos la librería time, tomamos un tiempo inicial y luego tomamos un tiempo final en un bucle hasta que la diferencia entre estos sea mayor que 60. Cuando se llega a dicho punto, con un semáforo el resto del programa se queda paralizado, se ejecuta la copia de seguridad y se escribe en el fichero json, con nombre "respaldo-nodo<IP del nodo>-<puerto usado en dicho nodo>.json".

3.1.5. *detalles_nodo_actual()*

Esta función muestra información del nodo al que se llama. Usando la librería platform, obtenemos información acera de la máquina, el sistema y la versión del nodo.

El método empleado para llamar a esta función es el método GET, y la ruta a seguir es "/system".

3.1.6. *registrar_nodos_completo()*

Esta función recibe una lista de nodos para registrar en la red. Una vez recibida, añade al set *nodos_red* dichos nodos y les envía uno a uno (llamando con el método POST a la función *registrar_nodo_actualiza_blockchain()*) una copia de dicho set y de la blockchain almacenada en la red hasta el momento.

El método empleado para llamar a esta función es el método POST, y la ruta a seguir es "/nodos/registrar".

3.1.7. *registrar_nodo_actualiza_blockchain()*

Esta función se encarga de almacenar una copia de la blockchain de la red en el nodo en cuestión. Esto se usa cuando registramos nuevos nodos en la red de cara a que almacenen una copia de la blockchain. Cuando recibe dicha copia, llama a la función *integrar_cadena()* para integrar los bloques de dicha cadena.

El método empleado para llamar a esta función es el método POST, y la ruta a seguir es "/nodos/registro_simple".

3.1.8. *resuelve_conflictos()*

Esta función se encarga de comprobar si al minar un bloque en un nodo, dicho nodo contiene una versión de la blockchain actualizada. En un bucle, va comprobando nodo por nodo si la longitud de la blockchain es la mayor de todas. De no ser así, se queda con la mayor longitud y hace una copia de dicha cadena y la integra llamando a la función *integrar_cadena()*.

3.1.9. *integrar_cadena()*

Esta función se encarga de integrar una cadena de bloques recibida al registrar un nuevo nodo o al encontrar un conflicto entre nodos de la red. Recibe una copia de la cadena en cuestión e integra bloque a bloque dentro de una blockchain vacía. Si encuentra algún error, muestra por pantalla "Error: La blockchain recibida no es válida".

3.2. Main

Para empezar, se definen en el programa las variables globales. Estas son: app (define la aplicación en la que se van a recibir las rutas para llamar a funciones), blockchain (objeto de la clase Blockchain de la librería Blockchain que almacenará la blockchain), nodos_red (set en el que se guardan el resto de los nodos de la red), mi_ip (variable que almacena la IP del nodo en el que se encuentra) y semaforo_copia_seguridad (semáforo que parará el programa entero cuando se esté ejecutando una copia de seguridad).

Después, lo primero que se hace es llamar al hilo de copia de seguridad para que empiece a ejecutarse durante el resto del programa. Finalmente, se recibe el argumento que define el puerto que se va a utilizar y se empieza a correr la aplicación a la espera de instrucciones.

4. FUNCIONAMIENTO

4.1. Postman

Para mostrar el funcionamiento del programa vamos a ejecutar una serie de instrucciones tanto en Postman (conectando el ordenador con la máquina virtual) como en un fichero denominado “fichero_requests.py” (enlazando dos puertos distintos con una misma IP). Las instrucciones son las siguientes:

1. http://<IP1>:<puerto1>/system
2. http://<IP1>:<puerto1>/transacciones/nueva
3. http://<IP1>:<puerto1>/minar
4. http://<IP1>:<puerto1>/chain
5. http://<IP2>:<puerto2>/chain
6. http://<IP1>:<puerto1>/nodos/registrar
7. http://<IP2>:<puerto2>/transacciones/nueva
8. http://<IP2>:<puerto2>/minar
9. http://<IP1>:<puerto1>/transacciones/nueva
10. http://<IP1>:<puerto1>/minar

Para empezar con la demo, ejecutamos Blockchain_app.py en los respectivos nodos con su IP y puerto.

```
alexmgg@alexmgg-VirtualBox:/media/sf_PracticaFinal$ python3 Blockchain_app.py -p 5000
puerto 5000
 * Serving Flask app 'Blockchain_app'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
```

Figura 1.- *Ejecución en máquina virtual*

```
PS C:\Users\Alex MGG\Documents\ICAI\2ºiMAT\Fundamentos de Sistemas Operativos\Practicas\PracticaFinal> python .\Blockchain_app.p
y -p 5001
puerto 5001
 * Serving Flask app 'Blockchain_app'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5001
 * Running on http://192.168.1.82:5001
Press CTRL+C to quit
```

Figura 2.- *Ejecución en ordenador local*

Una vez hecho esto, empezamos a ejecutar las instrucciones anteriormente mencionadas. Como aclaración, la IP empleada en la máquina virtual será “192.168.56.101” y la del ordenador local “192.168.56.1”.

http://192.168.56.1:5001/system

Como podemos ver, nos da información del nodo que se ha solicitado. En este caso, se trata de una máquina Windows 10.

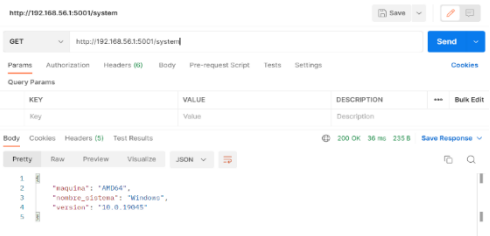

Figura 3.- *Información del sistema nodo 1*

http://192.168.56.1:5001/transacciones/nueva

Creamos una transacción nueva. Esta tendrá como origen “nodoA”, como destino “nodoB” y como cantidad “5”. Al ser una transacción válida, el mensaje mostrado por pantalla es que se incluría en el siguiente bloque de la blockchain (la primera transacción se incluye en el segundo bloque debido a que el primer bloque es de prueba).

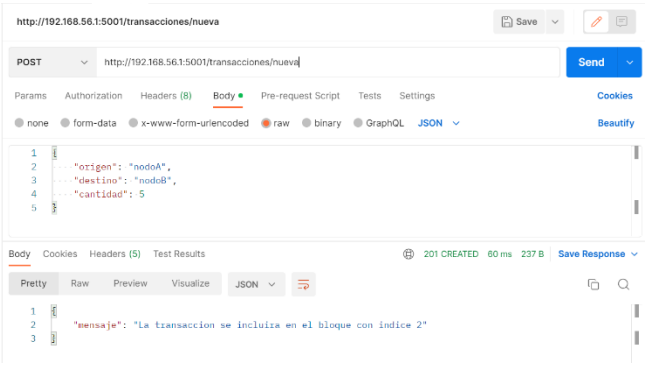

Figura 4.- *Información del sistema nodo 1*

http://192.168.56.1:5001/minar

Minamos el bloque de manera que se incluyen todas las transacciones que quedaban sueltas aún. Genera un bloque con hash, hash_previo y transacciones.

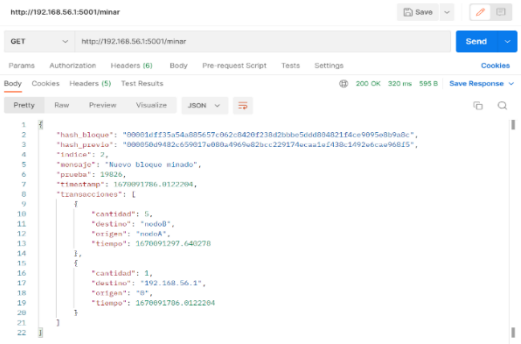

Figura 5.- *Información del sistema nodo 1*

http://192.168.56.1:5001/chain & http://192.168.56.101:5000/chain

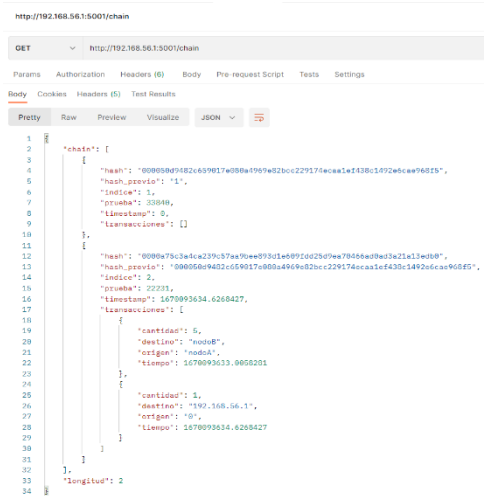

Figura 6.- *Cadena del nodo 1*

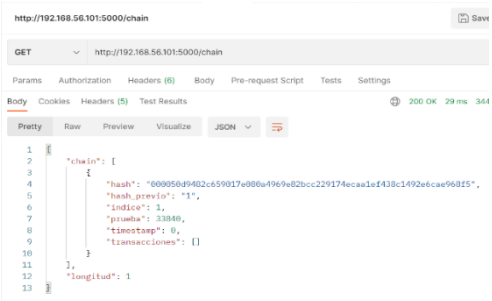

Figura 7.- *Cadena del nodo 2*

Como podemos apreciar, en el nodo que hemos minado un bloque, tenemos una blockchain más larga (con el bloque minado) mientras que en el otro tenemos solo el bloque de prueba.

http://192.168.56.1:5001/nodos/registrar

En este caso, registraremos el nodo 2 en la red formada por el nodo 1. Como no sucede ningún error se muestra el mensaje de que se han incluido los nodos nuevos en la red. Para comprobar que efectivamente se han integrado bien, podemos ejecutar de nuevo el comando *http://192.168.56.101:5000/chain* y veremos que efectivamente la blockchain de dicho nodo ya es correcta.

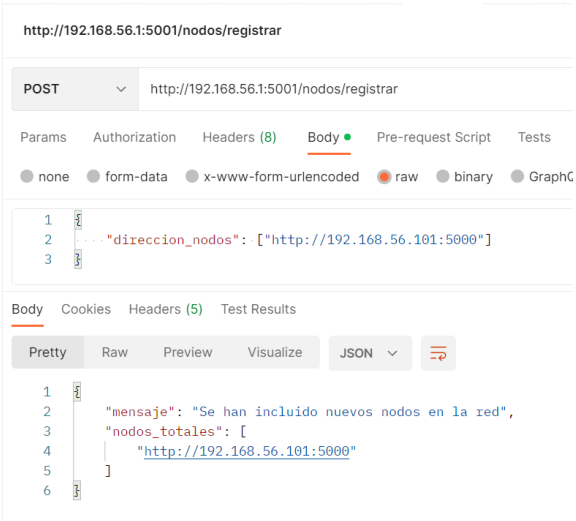

Figura 8.- *Registramos nodo 2 en el nodo 1*

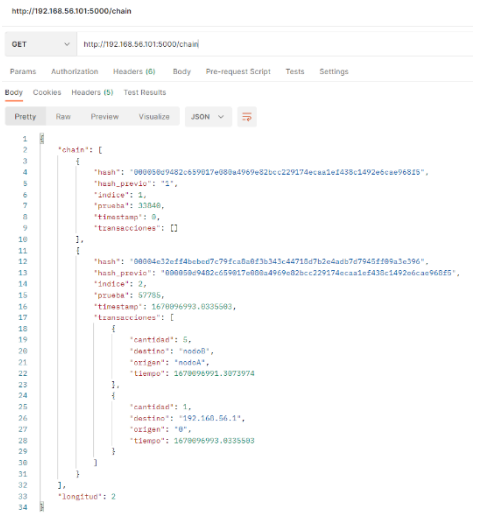

Figura 9.- *Cadena actualizada en nodo 2*

http://192.168.56.101:5001/transacciones/nueva http://192.168.56.101:5001/minar

Para comprobar la función de resuelve conflictos, minamos un nuevo bloque en uno de los nodos y luego intentamos minar uno en el otro nodo.

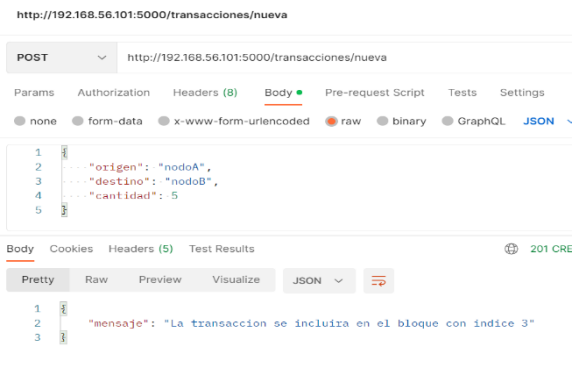

Figura 10.- *Introducimos transacción en nodo 2*

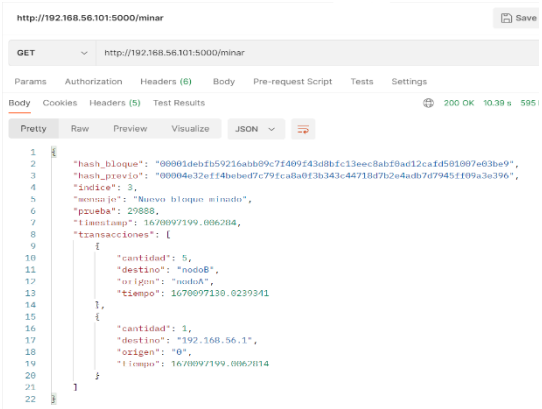

Figura 11.- *Minamos el bloque*

http://192.168.56.1:5001/transacciones/nueva **http://192.168.56.1:5001/minar**

Ahora intentamos minar un bloque en el nodo con la blockchain menos actualizada. Vemos que efectivamente muestra un error por no estar actualizada.

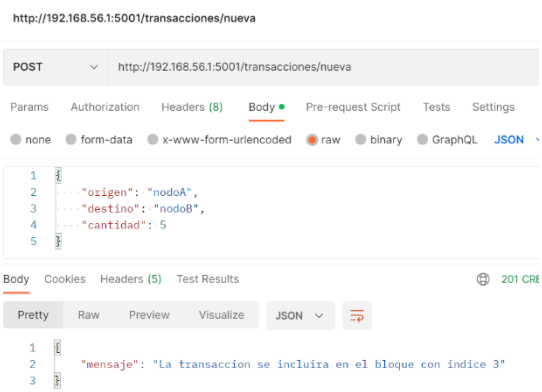

Figura 12.- *Introducimos transacción en nodo 1*

Figura 13.- *Intentamos minar el bloque.*

http://192.168.56.1:5001/chain

Comprobamos que efectivamente la cadena se ha actualizado tras el conflicto.

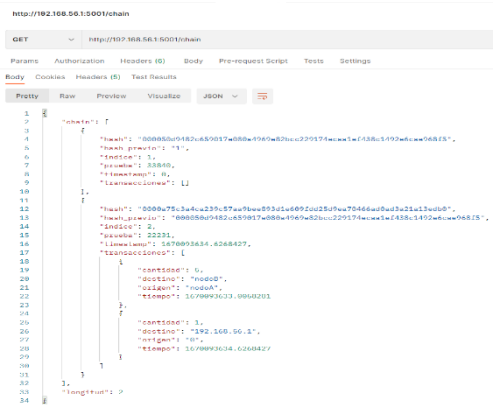

Figura 14.- *Cadena antes del conflicto*

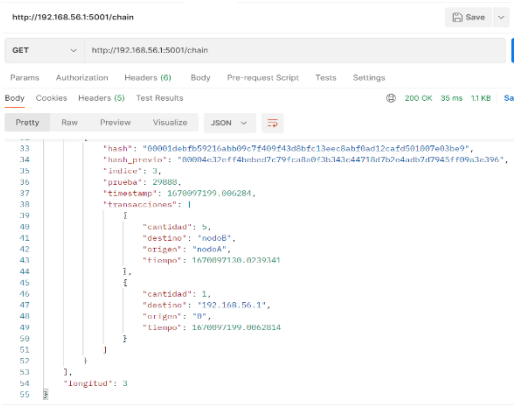

Figura 15.- *Cadena después del conflicto*

4.2. Fichero_requests.py

En este programa hemos querido simular una secuencia de peticiones distintos nodos de la Blockchain y así mostrar un ejemplo de cómo es su funcionamiento.

Para ejecutar dicho fichero hemos importado las siguientes librerías:

- Requests: para poder lanzar las peticiones a los nodos.
- Json: para darle el formato adecuado al body de las requests.

Una vez importadas las librerías, hemos desarrollado una secuencia de peticiones similares a las descritas anteriormente en el apartado 5.1 (postman), pero esta vez lanzando el programa Blockchain_app.py desde 2 nodos en la maquina host, pero con puertos distintos (en este caso 5000 y 5001).

La secuencia de peticiones que vamos a realizar será:

1) Añadir transacciones al nodo en el puerto 5000.
2) Minar un bloque en la cadena del nodo en el puerto 5000.
3) Obtener la cadena del nodo.
4) Pedimos al nodo 5000 que registre al 5001.
5) Mostramos la cadena del nodo 5001.
6) Generamos transacciones en el nodo 5000.
7) Minamos sus transacciones a un nuevo bloque.
8) Mostramos la cadena del nodo 5000.
9) Generamos transacciones en el nodo 5001.
10) Minamos sus transacciones a un nuevo bloque.
11) Mostramos la cadena del nodo 5001.

Lo que queremos con dicha secuencia de peticiones es registrar 2 nodos en la red, y mostrar como a medida que van añadiendo transacciones y minando bloques, se generan y se resuelven conflictos entre ellos, de forma que todos los nodos registrados acaben sincronizando la Blockchain.

A continuación, mostramos como sería la ejecución del programa:

En primer lugar, lanzamos “Blockchain_app.py” desde dos puertos distintos:

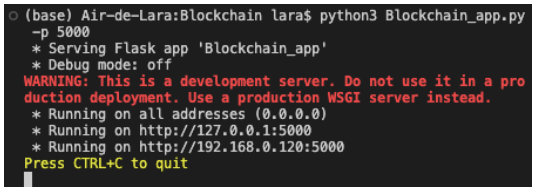

Figura 16.- *Ejecución nodo 1*

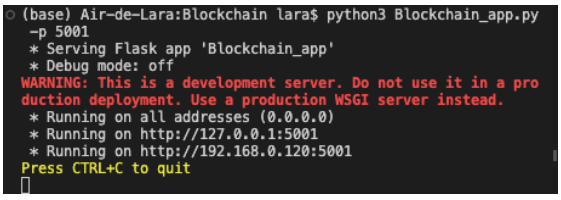

Figura 17.- *Ejecución nodo 2*

Ahora ya podemos lanzar “fichero_requests.py”.

Vamos a ir mostrando lo que se ha impreso por la terminal de los nodos 5000 y 5001 (donde se verán las peticiones) y a continuación lo que imprimirá por pantalla “fichero_requests.py” (respuesta a dichas peticiones).

1) Añadir transacciones al puerto 5000:

```
127.0.0.1 - - [04/Dec/2022 00:40:22] "POST /transacciones/nueva HTTP/1.1" 201 -
```

```
Generamos nueva transacción en el puerto 5000.
{"mensaje":"La transaccion se incluira en el bloque con indice 2"}
```

2) Minamos un bloque en el puerto 5000:

```
127.0.0.1 - - [04/Dec/2022 00:40:23] "GET /minar HTTP/1.1" 200 -
```

```
Minamos un bloque en el puerto 5000.
{"hash_bloque":"0000bf5531a7416ff43a086ea6ea5c2d4ec3d116686870ba88fc096a564a8da9","hash_previo":"000050d9482c659017e080a4969e8
2bcc229174ecaa1ef438c1492e6cae968f5","indice":2,"mensaje":"Nuevo bloque minado","prueba":30212,"timestamp":1670110822.223027,"
transacciones":[{"cantidad":10,"destino":"nodoB","origen":"nodoA","tiempo":1670110822.1993759},{"cantidad":1,"destino":"192.16
8.0.120","origen":"0","tiempo":1670110822.2230232}]}
```

3) Mostramos la cadena del puerto 5000:

```
127.0.0.1 - - [04/Dec/2022 00:40:23] "GET /chain HTTP/1.1" 200 -
```

```
Obtenemos la cadena del puerto 5000.
{"chain":[{"hash":"000050d9482c659017e080a4969e82bcc229174ecaa1ef438c1492e6cae968f5","hash_previo":"1","indice":1,"prueba":338
40,"timestamp":0,"transacciones":[]},{"hash":"0000bf5531a7416ff43a086ea6ea5c2d4ec3d116686870ba88fc096a564a8da9","hash_previo":
"000050d9482c659017e080a4969e82bcc229174ecaa1ef438c1492e6cae968f5","indice":2,"prueba":30212,"timestamp":1670110822.223027,"tr
ansacciones":[{"cantidad":10,"destino":"nodoB","origen":"nodoA","tiempo":1670110822.1993759},{"cantidad":1,"destino":"192.168.
0.120","origen":"0","tiempo":1670110822.2230232}]}],"longitud":2}
```

4) Registramos al nodo 5001:

```
192.168.0.120 - - [04/Dec/2022 00:40:24] "POST /nodos/registro_simple HTTP/1.1" 200 -
```

```
127.0.0.1 - - [04/Dec/2022 00:40:25] "POST /nodos/registrar HTTP/1.1" 201 -
```

Veamos como el puerto 5000 hace una petición a su vez a registro_simple del puerto 5001, para que este sincronice su cadena y se añada a la red de nodos.

```
127.0.0.1 - - [04/Dec/2022 00:40:25] "POST /nodos/registro_simple HTTP/1.1" 200 -
```

Podemos ver ahora que el puerto 5000 tiene registrado al nodo en el 5001.

```
Registramos al nodo 5001 en el puerto 5000.
{"mensaje":"Se han incluido nuevos nodos en la red","nodos_totales":["http://127.0.0.1:5001"]}
```

Ahora veamos a ver la cadena del 5001 para ver si el registro se ha hecho correctamente.

5) Mostramos la cadena del 5001.

```
127.0.0.1 - - [04/Dec/2022 00:40:25] "GET /chain HTTP/1.1" 200 -
```

```
Obtenemos la cadena del puerto 5001.
{"chain":[{"hash":"000050d9482c659017e080a4969e82bcc229174ecaa1ef438c1492e6cae968f5","hash_previo":"1","indice":1,"prueba":338
40,"timestamp":0,"transacciones":[]},{"hash":"0000bf5531a7416ff43a086ea6ea5c2d4ec3d116686870ba88fc096a564a8da9","hash_previo":
"000050d9482c659017e080a4969e82bcc229174ecaa1ef438c1492e6cae968f5","indice":2,"prueba":30212,"timestamp":1670110822.223027,"tr
ansacciones":[{"cantidad":10,"destino":"nodoB","origen":"nodoA","tiempo":1670110822.1993759},{"cantidad":1,"destino":"192.168.
0.120","origen":"0","tiempo":1670110822.2230232}]}],"longitud":2}
```

Es exactamente igual a la del 5000, luego ya está bien registrados.

Ahora vamos a tratar de generar un conflicto, para ello haremos que el 5001 mine un bloque y después el 5000 trate de hacerlo también, generando así un conflicto y obligando al 5000 sincronizar su cadena a la del 5001.

6) Generamos transacciones en el nodo 5000.

```
127.0.0.1 - - [04/Dec/2022 00:40:25] "POST /transacciones/nueva HTTP/1.1" 201 -
```

```
Primero, generamos transacciones en el puerto 5001 y las minamos.
{"mensaje":"La transaccion se incluira en el bloque con indice 3"}
```

7) Minamos un bloque en el 5001.

```
127.0.0.1 - - [04/Dec/2022 00:40:32] "GET /minar HTTP/1.1" 200 -
```


```
{"hash_bloque":"00007f7441551cd4d47c3524ed9f26226dfe9377b79e8678be187da1341315fb","hash_previo":"0000bf5531a7416ff43a086ea6ea5
c2d4ec3d116686870ba88fc096a564a8da9","indice":3,"mensaje":"Nuevo bloque minado","prueba":174493,"timestamp":1670110825.1945379
,"transacciones":[{"cantidad":10,"destino":"nodoB","origen":"nodoA","tiempo":1670110825.146055},{"cantidad":1,"destino":"192.1
68.0.120","origen":"0","tiempo":1670110825.1945322}]}
```


8) Vemos la cadena del 5001.

```
127.0.0.1 - - [04/Dec/2022 00:40:32] "GET /chain HTTP/1.1" 200 -
```


```
{"chain":[{"hash":"000050d9482c659017e080a4969e82bcc229174ecaa1ef438c1492e6cae968f5","hash_previo":"1","indice":1,"prueba":338
40,"timestamp":0,"transacciones":[]},{"hash":"0000bf5531a7416ff43a086ea6ea5c2d4ec3d116686870ba88fc096a564a8da9","hash_previo":
"000050d9482c659017e080a4969e82bcc229174ecaa1ef438c1492e6cae968f5","indice":2,"prueba":30212,"timestamp":1670110822.223027,"tr
ansacciones":[{"cantidad":10,"destino":"nodoB","origen":"nodoA","tiempo":1670110822.1993759},{"cantidad":1,"destino":"192.168.
0.120","origen":"0","tiempo":1670110822.2230232}]},{"hash":"00007f7441551cd4d47c3524ed9f26226dfe9377b79e8678be187da1341315fb",
"hash_previo":"0000bf5531a7416ff43a086ea6ea5c2d4ec3d116686870ba88fc096a564a8da9","indice":3,"prueba":174493,"timestamp":167011
0825.1945379,"transacciones":[{"cantidad":10,"destino":"nodoB","origen":"nodoA","tiempo":1670110825.146055},{"cantidad":1,"des
tino":"192.168.0.120","origen":"0","tiempo":1670110825.1945322}]}],"longitud":3}
```


Ahora repetimos los 3 últimos pasos con el nodo 5000.

9) Generamos transacciones en el 5000.

```
127.0.0.1 - - [04/Dec/2022 00:40:32] "POST /transacciones/nueva HTTP/1.1" 201 -
```

```
Ahora, generamos transacciones en el puerto 5000 y las minamos.
{"mensaje":"La transaccion se incluira en el bloque con indice 3"}
```

10) Minamos sus transacciones:

```
127.0.0.1 - - [04/Dec/2022 00:40:33] "GET /minar HTTP/1.1" 200 -
```

```
{"mensaje":"Ha habido un conflicto. Esta cadena se ha actualizado con una version mas larga"}
```

Vemos como en efecto, se ha producido el conflicto, por lo que el nodo 5000 ha debido de sincronizar su cadena a la del 5001. Confirmemos esto accediendo a la cadena del 5000 y viendo si es igual.

11) Accedemos a la cadena del 5000:

```
127.0.0.1 - - [04/Dec/2022 00:40:33] "GET /chain HTTP/1.1" 200 -
```


```
{"chain":[{"hash":"000050d9482c659017e080a4969e82bcc229174ecaa1ef438c1492e6cae968f5","hash_previo":"1","indice":1,"prueba":338
40,"timestamp":0,"transacciones":[]},{"hash":"0000bf5531a7416ff43a086ea6ea5c2d4ec3d116686870ba88fc096a564a8da9","hash_previo":
"000050d9482c659017e080a4969e82bcc229174ecaa1ef438c1492e6cae968f5","indice":2,"prueba":30212,"timestamp":1670110822.223027,"tr
ansacciones":[{"cantidad":10,"destino":"nodoB","origen":"nodoA","tiempo":1670110822.1993759},{"cantidad":1,"destino":"192.168.
0.120","origen":"0","tiempo":1670110822.2230232}]},{"hash":"00007f7441551cd4d47c3524ed9f26226dfe9377b79e8678be187da1341315fb",
"hash_previo":"0000bf5531a7416ff43a086ea6ea5c2d4ec3d116686870ba88fc096a564a8da9","indice":3,"prueba":174493,"timestamp":167011
0825.1945379,"transacciones":[{"cantidad":10,"destino":"nodoB","origen":"nodoA","tiempo":1670110825.146055},{"cantidad":1,"des
tino":"192.168.0.120","origen":"0","tiempo":1670110825.1945322}]}],"longitud":3}
```


Así es, el nodo 5000 ha actualizado su cadena a la del 5001 y ahora sí podrá minar sus transacciones.

11. CONCLUSIONES

A lo largo de la práctica, nos hemos encontrado varios conflictos para los cuales hemos tenido que buscar soluciones alternativas.

Por otro lado, también nos topamos con otros obstáculos que nos hicieron ver que nuestras soluciones propuestas no eran del todo correctas. Por ejemplo, al integrar bloques a la Blockchain nos saltaba un error debido a que el primer bloque cumplía la condición del hash (algo que conseguimos arreglar evitando la comprobación de este cada vez que se tuviese que integrar la cadena, ya que dicho primer bloque era para todos los nodos el mismo).

Una vez conseguimos que el programa funcionase correctamente desde el local host, pasamos a probarlo desde otro nodo: la máquina virtual de Ubuntu. Lo que nos llevó a encontrarnos con un gran fallo: al registrar desde el local host al nodo en la máquina virtual, saltaba un fallo de conexión. Después de investigar acerca del fallo y analizar bien las direcciones IP que se estaban introduciendo en loa variable nodos_red, descubrimos que esto era por una simple razón: la dirección IP con la que se identificaba el local host, no era la dirección creada por la máquina virtual para comunicarse con este, de forma que cuando la máquina virtual trataba de conectarse con el local host se producía un fallo de conexión. Para ver si estábamos en lo cierto, probamos a introducir las IPs correctas manualmente y vimos que entonces ya funcionaba. Una vez localizado cual era el error, pensamos en distintas formas de arreglarlo, entre ellas crear un bridge entre la IP creada por la máquina virtual y la IP del localhost, sin embargo, debido a que cada ordenador tiene diferentes IPs y diferentes formas de identificarlas, esto era muy complejo de generalizar para todos los ordenadores (aunque lo consideramos como posible mejora para futuros proyectos).

Gracias a estos obstáculos, hemos logrado entender mejor cómo funcionan muchos de los conceptos vistos en clase, como sockets, requests, semáforos, hilos, etc. Además de aprender a utilizar la máquina virtual de Ubuntu y formatos de ficheros como json que no estamos acostumbrados a emplear.

A modo de resumen, a pesar de los impedimentos encontrados, hemos logrado no solo afianzar conceptos vistos en clase, sino que además hemos obtenido una visión más cercana de lo que sería una versión simplificada del sistema Blockchain, despertando nuestra curiosidad por llegar a entenderlo mejor en un futuro próximo.