Ingeniería Matemática e Inteligencia Artificial

Memoria Blockchain

Práctica Final
Fundamentos de los Sistemas Operativos

Sergio Herreros Pérez
Daniel Sánchez Sánchez

BlockChain.py

- **Clase Bloque**:

    - Forma los cimientos de la cadena. Cada bloque tiene unas ciertas transacciones, las cuáles se formalizan una vez que el bloque se une a la cadena. Para "enganchar" los bloques entre sí, se le da un hash generado según su propia estructura, además de tener un valor de prueba para que cumpla con la dificultad elegida (empiece con cierto número de ceros), y tiene el hash del bloque anterior en la cadena. Por último, tiene un índice y un registro de tiempo correspondiente a su creación.

    - Además, tiene tres métodos:

        - Calcular hash: Calcula el hash del bloque en función de sus parámetros, usando la función *toDict*. Como el hash depende exclusivamente de las características del bloque, se calcula con hash vacío, pues si también dependiese de él mismo, en cuanto se pusiese el hash calculado como atributo del bloque, éste dejaría de corresponder.

        - To dict: Convierte el bloque en un diccionario, siendo los nombres y los valores de sus parámetros los *keys* y los *values* del diccionario, respectivamente. Al ser las transacciones un set, éste no puede ser pasado como valor de un json (al no ser serializable), así que lo casteamos como una lista y la ordenamos (para que el hash no cambie para un mismo conjunto de transacciones).

        - From dict: Hace el proceso contrario; asigna a cada parámetro del bloque su valor correspondiente del diccionario, y castea de vuelta las transacciones a un set.

- **Clase Blockchain**:

    - Es la cadena. Sirve para conectar los bloques, y contiene la gran mayoría de métodos. Contiene una lista con los bloques añadidos, un set con las transacciones aun no añadidas, su nivel de dificultad y el número de bloques añadidos. Además, para empezar, se añade un nuevo bloque.

      Las transacciones están contenidas en un set porque las únicas operaciones que se hacen es agregar transacciones y calcular diferencias entre distintos grupos de transacciones, las cuáles son mucho más eficientes en un set que en una lista, y no nos importa el orden de las transacciones.

- Sus métodos son:

    a) Primer bloque: Crea el bloque inicial de la cadena. Este se llama siempre cuando se crea la instancia de Blockchain. Dicho bloque tiene hash previo igual al string 1 y transacciones vacías. Se le calcula su valor de prueba usando el método *prueba_trabajo* y se integra directamente a la cadena.

    b) Nuevo bloque: Crea un nuevo bloque dado un hash previo. El bloque tendrá índice igual al número de bloques más uno, transacciones igual a una copia de la *pool* (es una copia para evitar que se añadan más transacciones al bloque una vez haya sido creado), *timestamp* igual a la hora en la que fue creado y prueba iniciada a 0.

    c) Nueva transacción: Crea una nueva transacción con un origen, una cantidad y un destino y la añade a la *pool* como un string del diccionario creado. Hay que pasarla a string pues los diccionarios no son hasheables, así que un set no los puede contener.

    d) Prueba trabajo: Calculará el hash del bloque que cumpla con el nivel de dificultad establecido. Para ello, calculará el hash una y otra vez, aumentando el valor del parámetro prueba en uno cada iteración, hasta que lo cumpla, pues al depender el hash de los parámetros del bloque, modificar el parámetro prueba, aunque sea en uno, cambiará el hash drásticamente (el método de calcular el hash es tal, que un pequeño cambio en los parámetros modifica de manera total el hash. Esto es, principalmente, para que no se pueda adivinar qué valor de prueba cumplirá con la dificultad, y haya que usar fuerza bruta para minar el bloque).

    La dificultad se establece para que minar el bloque tarde una cantidad de tiempo medio fijo, al irla aumentando para que cada vez cueste más operaciones, según se van desarrollando y mejorando los procesadores de los ordenadores modernos.

    e) Prueba válida: Comprueba que un bloque tenga hash correspondiente al proporcionado a la función, y éste cumpla con el nivel de dificultad establecido. Si es así, devuelte True, y en caso contrario devuelve False.

    f) Integra bloque: Primero, comprueba que el hash previo del bloque coincida con el hash del último bloque integrado a la cadena, y, usando el método *prueba_valida*, comprueba que el hash proporcionado es correcto.

Si es así, establece el hash del bloque como el hash proporcionado y añade el bloque a la cadena. Además, quita de la *pool* las transacciones presentes en el bloque, y aumenta el número de bloques en uno.

g) Check chain: Comprueba que toda la cadena es correcta. Para ello, recorre la cadena de bloques comprobando que el hash de cada bloque es válido (usando el método *prueba_valida*) y que el hash previo del siguiente bloque de la cadena coincide con su hash.

h) toDict: Convierte la cadena en un diccionario, siendo los nombres y los valores de sus parámetros los *keys* y los *values* del diccionario, respectivamente. Al igual que con el bloque, casteamos la *pool* a una lista, aunque esta vez no hace falta ordenarla. También pasamos cada bloque de la cadena a diccionario usando el método *toDict* de la clase bloque.

i) fromDict: Hace lo contrario que el método *toDict*. Pasa cada bloque de la cadena a una instancia de la clase bloque, usando el método *fromDict* de dicha clase, y devuelve la *pool* a un set.

j) fromChain: Inicializa una instancia de la clase Blockchain a partir de una lista de bloques. La cadena será la lista de bloques pasada (casteado cada bloque como una instancia de la clase bloque, al igual que en el método *fromDict*), y la dificultad, la *pool* y *n_bloques* serán la dificultad elegida, un set vacío y la longitud de la cadena respectivamente.

El funcionamiento será el siguiente: se creará la blockchain cuando se inicialice el programa (con ello generándose el primer bloque). Se irán haciendo transacciones a medida que se hagan requests, y una vez que se mine un bloque (y no haya conflictos), se integrará ese bloque a la cadena con las transacciones que había en la *pool* cuando se creó.

Blockchain_app.py

En este fichero se incluye la funcionalidad de la aplicación de Blockchain. Utilizando Flask, creamos una app que se podrá ejecutar por cada nodo y permitirá la conexión con otros nodos, de manera que se cree una red peer-to-peer.

Aquí se creará una instancia de la clase Blockchain, y un set con los nodos de la red.

- **Nueva transacción:**
    - Asociamos la ruta “/transacciones/nueva” y el método POST a la funcion nueva_transaccion().
    - Recibe un json con el contenido de la transacción, comprueba que al menos contiene origen destino y cantidad, y la incluye al pool de la Blockchain.
    - Devolvemos un string con formato json con la respuesta y el HTTP status code, que será 200 en caso de haberla incluido correctamente, o 400 si no contenía los campos necesarios.
- **Blockchain completa:**
    - Asociamos la ruta “/chain” y el método GET a la función blockchain_completa().
    - Esta es una funcion para compartir la cadena con otros nodos, o poder ver la cadena en cualquier momento.
    - Devolvemos un string con formato json con la cadena y su longitud y el status 200 en caso de haberla incluido correctamente, o un mensaje de error y el status 500 en caso de haber habido algún fallo del lado del servidor. Esto nos ayuda mucho a la hora de debuggear la app.
- **Minar:**
    - Asociamos la ruta “/minar” y el método GET a la función minar().
    - En esta funcion, se sacan las transacciones no confirmadas de la pool y se crea un nuevo bloque para incluirlo en la cadena, calculando la prueba de trabajo correspondiente.
    - Pueden pasar varias cosas:
        - Si no había transacciones, devolvemos un mensaje y el status code 201
        - Si había, creamos un bloque con estas y comenzamos a trabajar en la prueba de trabajo. Una vez encontrado el nonce, procedemos a comprobar que no hay otra cadena de mayor longitud que la nuestra con la función resuelve_conflictos().
        - Si somos la cadena más larga, integramos el bloque en la cadena principal y quitamos del pool de transacciones las que acabamos de incluir. No la vaciamos del todo ya que en el tiempo que minamos el bloque (en bitcoin serian unos 10

mins) podrían llegar muchas nuevas transacciones a nuestro pool, y estas no habría que eliminarlas.

Si todo ha ido bien, recibimos un pago por minar creando una transacción a nosotros mismos, y devolvemos un json con el ultimo bloque minado, y el status code 200.

- En caso de conflicto, se actualizará nuestra cadena con la nueva más larga, se desechará el bloque que acabamos de minar, y se devolverá un mensaje diciendo que ha habido un conflicto, y el status code 202.

- Si ha habido algún problema integrando el bloque, se devolverá el status 203.

- **Regristrar nodos:**

  - Asociamos la ruta "/nodos/registrar" y el método POST a la función registrar_nodos_completo().

  - En esta funcion y en la siguiente se incluirá la funcionalidad de red peer-to-peer.

  - Primero comprobamos que nos ha llegado una lista con nuevos nodos, y en caso contrario devolvemos un 400.

  - Si tenemos la lista, comprobamos que cada nodo es un string con formato http://ip:puerto. Los que no lo sean, los quitamos y guardamos en un set.

  - Después, enviamos a cada nodo una copia de nuestra blockchain, y una copia de los nodos de la red, incluyéndonos a nosotros y excluyendo a dicho nodo. Los nodos a los que no es posible conectarse, los quitamos y guardamos en un set.

  - Finalmente añadimos a nuestra copia de nodos los nuevos nodos que se han podido conectar.

  - En caso de que todos los nodos se hayan conectado y estén bien formateados, devolvemos un mensaje satisfactorio y un 200.

  - En caso contrario, devolvemos un mensaje indicando que nodos han dado qué tipo de fallo, y el código 201.

- **Actualizar blockchain:**

  - Asociamos la ruta "/nodos/registro_simple"y el método POST a la función registrar_nodo_actualiza_blockchain()

- Esta función sirve para registrar a cada nodo desde la función mencionada anteriormente. Recibe la copia de la blockchain y los nodos de la red, actualizando su blockchain y los nodos para poder formar parte de la red.

- Si falta o la blockchain o los nodos, se devuelve un 400

- Si no, comprobamos que la blockchain recibida es válida, en cuyo caso simplemente actualizamos nuestra cadena, y si no lo es, devolvemos un mensaje avisando de que la blockchain es corrupta, con código 401.

- **Resolver conflictos:**

  - Esta función no va asociada a la app.

  - Simplemente se usa al minar para comprobar nodo por nodo si hay una cadena más larga antes de incluir nuestro bloque recién minado.

  - Si no se encuentra ninguna cadena más larga, devolvemos False, diciendo que no hay ningún conflicto.

  - En cambio, si ha habido conflicto, y existe una cadena valida más larga, procedemos a reemplazar nuestra cadena con dicha cadena, y a quitar de nuestro pool de transacciones las que ya han sido incluidas en dicha cadena, para así no repetir transacciones y crear bloques no válidos. En este caso se devuelve True, indicando que, si que ha habido un conflicto, y así desechar el bloque en la función de minar.

- **Copia de seguridad:**

  - Por último, tenemos una función que crea una copia de seguridad de la cadena cada 60 segundos. En primer lugar, usamos un semáforo para proteger la escritura en archivos. En segundo lugar, dividimos el time.sleep() en muchos intervalos pequeños, para que así podamos controlar la salida del programa con CTRL+C, ya que de no hacerlo se queda pillado en el sleep.

PROCESO

- ## Programar BlockChain.py

Esta es la parte más simple del proyecto, ya que está altamente guiada, y se ha implementado sin mucho problema.

- ## Programar blockchain_app.py

Esta parte es más complicada ya que debuguear una aplicación compleja requiere bastante esfuerzo.

Para hacerlo lo más ameno posible, hemos incluido los mensajes de error más detallados posibles, hemos distinguido cada tipo de respuesta en cada función, etc.… lo cual nos ha ayudado mucho.

- **Testeos básicos**

Para testear, nos hemos creado un fichero, prueba.py, donde poco a poco incorporábamos las nuevas funcionalidades. Esto nos permitía hacer más rápido la corrección de errores, ya que ejecutar todas las peticiones se hace de golpe, y no hay que ir una por una en Postman.

La primera parte es comprobar que funciona bien la app con un solo nodo. Para ello, hacemos varias transacciones, minamos y comprobar la cadena.

Ejecutamos la app en nuestro ordenador host.

```
PS Blockchain → & C:/Users/Gomi/AppData/Local/Programs/Python/Python310/python.exe "c:/Users/Gomi/OneDrive - Universida
damentos de Sistemas Operativos/Blockchain/blockchain_app.py" -p 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://192.168.1.45:5000
Press CTRL+C to quit
```

Hacemos varias transacciones y minamos un bloque.


```
MINAR NODO 1 (5000)
{
    "hash": "05708053da20063c90995e5352972e7949ace097a29241c9954c92c73ae9ec74",
    "hash_previo": "088003a8fa845e016a00e6c3a15c1dcf1b708f6a3f98d4934eaa4c5a722a02d6",
    "indice": 2,
    "mensaje": "Nuevo bloque minado",
    "timestamp": 1670093069.1559837,
    "transacciones": [
        "{'origen': 'A', 'destino': 'B', 'cantidad': 4}",
        "{'origen': 'B', 'destino': 'C', 'cantidad': 2}"
    ],
    "valor_prueba": 44
}
CHAIN NODO 1 (5000)
{
    "chain": [
        {
            "hash": "088003a8fa845e016a00e6c3a15c1dcf1b708f6a3f98d4934eaa4c5a722a02d6",
            "hash_previo": "1",
            "indice": 1,
            "prueba": 26,
            "timestamp": 1670093028.9165356,
            "transacciones": []
        },
        {
            "hash": "05708053da20063c90995e5352972e7949ace097a29241c9954c92c73ae9ec74",
            "hash_previo": "088003a8fa845e016a00e6c3a15c1dcf1b708f6a3f98d4934eaa4c5a722a02d6",
            "indice": 2,
            "prueba": 44,
            "timestamp": 1670093069.1559837,
            "transacciones": [
                "{'origen': 'A', 'destino': 'B', 'cantidad': 4}",
                "{'origen': 'B', 'destino': 'C', 'cantidad': 2}"
            ]
        }
    ],
    "longitud": 2
}
```

Por ahora la cadena es correcta, las transacciones se han inlcuido y el bloque se ha minado e incorporado correctamente.

Después, toca probar que podemos registrar otros nodos.

```
PS Blockchain → & C:/Users/Gomi/AppData/Local/Programs/Python/Python310/python.exe "c:/Users/Gomi/OneDrive - Universidad Pon
AT/Fundamentos de Sistemas Operativos/Blockchain/blockchain_app.py" -p 5001
 * Serving Flask app 'blockchain_app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://192.168.1.45:5001
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 599-461-942
```

Registramos el nuevo nodo al nodo principal.


```
RESPUESTA REGISTRAR NODO 2 (5001)
{
    "mensaje": "Se han incluido nuevos nodos en la red",
    "nodos_totales": [
        "http://192.168.1.45:5001",
        "http://192.168.1.45:5000"
    ]
}
CHAIN NODO 2 (5001)
{
    "chain": [
        {
            "hash": "0b50ddd349d3035b161fc4a9e082c87e8fc16f36399fb8d778b0e394e7d859fb",
            "hash_previo": "1",
            "indice": 1,
            "prueba": 13,
            "timestamp": 1670096165.9660144,
            "transacciones": []
        },
        {
            "hash": "036a33f5e0ee9dbedd5db8ff1b0b7d398e3670f06479af5103d794d9780a81bf",
            "hash_previo": "0b50ddd349d3035b161fc4a9e082c87e8fc16f36399fb8d778b0e394e7d859fb",
            "indice": 2,
            "prueba": 29,
            "timestamp": 1670096172.7629263,
            "transacciones": [
                "{'origen': 'A', 'destino': 'B', 'cantidad': 4}",
                "{'origen': 'B', 'destino': 'C', 'cantidad': 2}"
            ]
        }
    ],
    "longitud": 2
}
```


El nodo se incluye correctamente, y la cadena se clona sin problema.

Ahora hacemos más transacciones en el primer nodo, y minamos otro bloque.

```
CHAIN NODO 1 (5000)
{
    "chain": [
        {
            "hash": "0f824ec5a0796579dd4a538353b5b2252535382611bbb2b30aacc0fe138815cd",
            "hash_previo": "1",
            "indice": 1,
            "prueba": 12,
            "timestamp": 1670096376.4619658,
            "transacciones": []
        },
        {
            "hash": "0070f80558aaa2349faf8943b04a4c01540a9b2492145912ad7c847c7fd05611",
            "hash_previo": "0f824ec5a0796579dd4a538353b5b2252535382611bbb2b30aacc0fe138815cd",
            "indice": 2,
            "prueba": 23,
            "timestamp": 1670096381.496591,
            "transacciones": [
                "{'origen': 'A', 'destino': 'B', 'cantidad': 4}",
                "{'origen': 'B', 'destino': 'C', 'cantidad': 2}"
            ]
        },
        {
            "hash": "0f05f6abd1c81014d1ff38452cf8d4e2f5828213ce3b5706fadac1b87622e43c",
            "hash_previo": "0070f80558aaa2349faf8943b04a4c01540a9b2492145912ad7c847c7fd05611",
            "indice": 3,
            "prueba": 1,
            "timestamp": 1670096381.5109715,
            "transacciones": [
                "{'origen': 'F', 'destino': 'A', 'cantidad': 1}",
                "{'origen': 'H', 'destino': 'F', 'cantidad': 7}",
                "{'origen': 0, 'destino': '192.168.1.45', 'cantidad': 1}"
            ]
        }
    ],
    "longitud": 3
}
```


Ahora, vamos a comprobar que funciona nuestra resolución de conflictos. Si el nodo 2 recibe nuevas transacciones y mina el bloque, debería generar un conflicto, ya que el nodo 1 tiene una cadena más larga.


```
    "longitud": 3
}
MINAR NODO 2 (5001)
{
    "mensaje": "Conflicto: No es posible crear un nuevo bloque. Existe una cadena mas larga."
}
```


Efectivamente, se genera un conflicto

- **Pruebas desde VM**

Ahora hay que comprobar que funciona la red cuando hay nodos en una máquina virtual.

Para ello, se configura dicha máquina virtual con la red de only-host, y de esta forma, podremos comunicarnos con ella a través de la red a la que estemos conectados. En mi caso, la ip del host es 192.168.56.102 y la de la VM es 192.168.56.101.

ES NECESARIO MODIFICAR EL PROGRAMA PARA QUE SE INTRODUZCA LA IP DE TU MAQUINA A MANO EN BLOCKCHAIN_APP.PY

TAMBIEN HAY QUE INTRODUCIR A MANO LA IP DEL NODO1 Y EL NODO2 EN EL FICHERO DE PRUEBAS.PY

HAY QUE ASEGURARSE DE QUE LAS IPS SON LAS DEL EHTERNET QUE COUMUNICA EL HOST CON EL GUEST

Primero creamos el nodo principal en el host

```
PS Blockchain → & C:/Users/Gomi/AppData/Local/Programs/Python/Python310/python.exe "c:/Users/Gomi/OneDrive - Universidad
istemas Operativos/Blockchain/blockchain_app.py"
 * Serving Flask app 'blockchain_app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://192.168.56.102:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 599-461-942
```

Ahora creamos un nodo en la VM

```
gomi@kali: ~/Documents/Blockchain
┌──(gomi㉿kali)-[~/Documents/Blockchain]
└─$ python blockchain_app.py -p 5001
 * Serving Flask app 'blockchain_app' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployme
nt.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://192.168.56.101:5001/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 422-494-234
```

Veamos que el fichero de pruebas.py se ejecuta correctamente

```
CHAIN NODO 1 (192.168.56.102:5000)
{
    "chain": [
        {
            "hash": "083c5f8b1c7f402e059221f22d90d832bc69a72a383907304362f5de22eeb39f",
            "hash_previo": "1",
            "indice": 1,
            "prueba": 52,
            "timestamp": 1670155399.9349017,
            "transacciones": []
        },
        {
            "hash": "0a9d0018f06c548370c378d6387b8e7d0acd096b7fd10966ecf5eb47f29ff13c",
            "hash_previo": "083c5f8b1c7f402e059221f22d90d832bc69a72a383907304362f5de22eeb39f",
            "indice": 2,
            "prueba": 68,
            "timestamp": 1670155412.7193336,
            "transacciones": [
                "{'origen': 'A', 'destino': 'B', 'cantidad': 4}",
                "{'origen': 'B', 'destino': 'C', 'cantidad': 2}"
            ]
        }
    ],
    "longitud": 2
}
```

Las transacciones se han incluido correctamente en el nodo1


```
RESPUESTA REGISTRAR NODO 2 (192.168.56.101:5001)
{
    "mensaje": "Se han incluido nuevos nodos en la red",
    "nodos_totales": [
        "http://192.168.56.101:5001",
        "http://192.168.56.102:5000"
    ]
}
CHAIN NODO 2 (192.168.56.101:5001)
{
    "chain": [
        {
            "hash": "083c5f8b1c7f402e059221f22d90d832bc69a72a383907304362f5de22eeb39f",
            "hash_previo": "1",
            "indice": 1,
            "prueba": 52,
            "timestamp": 1670155399.9349017,
            "transacciones": []
        },
        {
            "hash": "0a9d0018f06c548370c378d6387b8e7d0acd096b7fd10966ecf5eb47f29ff13c",
            "hash_previo": "083c5f8b1c7f402e059221f22d90d832bc69a72a383907304362f5de22eeb39f",
            "indice": 2,
            "prueba": 68,
            "timestamp": 1670155412.7193336,
            "transacciones": [
                "{'origen': 'A', 'destino': 'B', 'cantidad': 4}",
                "{'origen': 'B', 'destino': 'C', 'cantidad': 2}"
            ]
        }
    ],
    "longitud": 2
}
```


La conexión con el nodo2 también se ha realizado sin problemas, y la cadena se ha sincronizado.

```
}
CHAIN NODO 1 (192.168.56.102:5000)
{
    "chain": [
        {
            "hash": "083c5f8b1c7f402e059221f22d90d832bc69a72a383907304362f5de22eeb39f",
            "hash_previo": "1",
            "indice": 1,
            "prueba": 52,
            "timestamp": 1670155399.9349017,
            "transacciones": []
        },
        {
            "hash": "0a9d0018f06c548370c378d6387b8e7d0acd096b7fd10966ecf5eb47f29ff13c",
            "hash_previo": "083c5f8b1c7f402e059221f22d90d832bc69a72a383907304362f5de22eeb39f",
            "indice": 2,
            "prueba": 68,
            "timestamp": 1670155412.7193336,
            "transacciones": [
                "{'origen': 'A', 'destino': 'B', 'cantidad': 4}",
                "{'origen': 'B', 'destino': 'C', 'cantidad': 2}"
            ]
        },
        {
            "hash": "04962b2d42f3f91515de75c55c0a7f8a6282eb5a59966cf3e96c77fa6c04c470",
            "hash_previo": "0a9d0018f06c548370c378d6387b8e7d0acd096b7fd10966ecf5eb47f29ff13c",
            "indice": 3,
            "prueba": 27,
            "timestamp": 1670155412.7615592,
            "transacciones": [
                "{'origen': 'F', 'destino': 'A', 'cantidad': 1}",
                "{'origen': 'H', 'destino': 'F', 'cantidad': 7}",
                "{'origen': 0, 'destino': '192.168.56.102', 'cantidad': 1}"
            ]
        }
    ],
    "longitud": 3
```

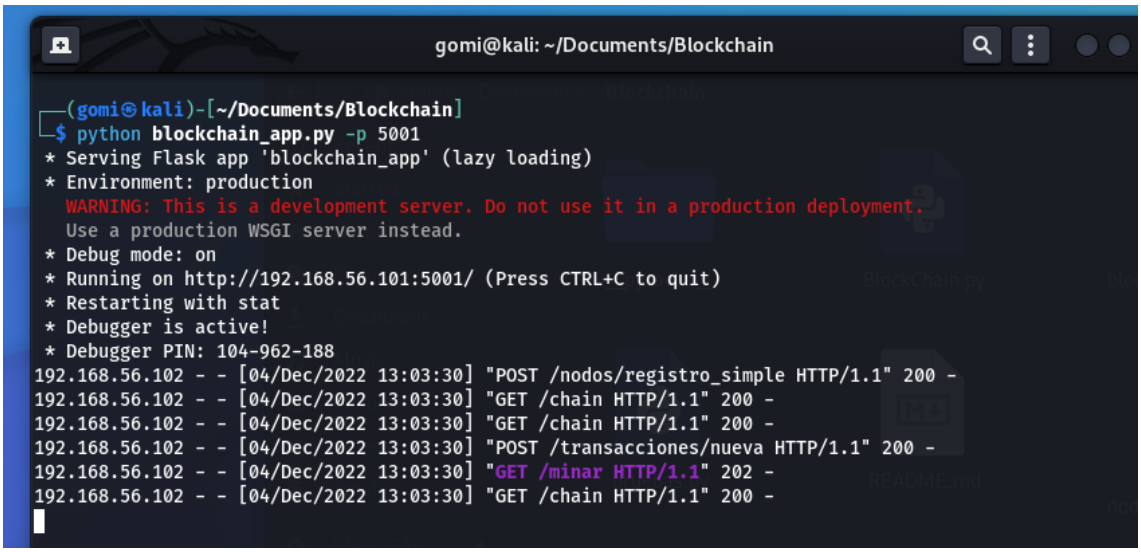

Las nuevas transacciones al nodo1 se han añadido, incluyendo también el pago por minar el anterior bloque. Ahora la cadena del nodo1 tiene longitud 3 y la del nodo2, 2.


```
}
RESPPUESTA MINAR NODO 2 (192.168.56.101:5001)
{
    "mensaje": "Conflicto: No es posible crear un nuevo bloque. Existe una cadena mas larga."
}
CHAIN NODO 2 (192.168.56.101:5001)
{
    "chain": [
        {
            "hash": "083c5f8b1c7f402e059221f22d90d832bc69a72a383907304362f5de22eeb39f",
            "hash_previo": "1",
            "indice": 1,
            "prueba": 52,
            "timestamp": 1670155399.9349017,
            "transacciones": []
        },
        {
            "hash": "0a9d0018f06c548370c378d6387b8e7d0acd096b7fd10966ecf5eb47f29ff13c",
            "hash_previo": "083c5f8b1c7f402e059221f22d90d832bc69a72a383907304362f5de22eeb39f",
            "indice": 2,
            "prueba": 68,
            "timestamp": 1670155412.7193336,
            ]
        },
        {
            "hash": "04962b2d42f3f91515de75c55c0a7f8a6282eb5a59966cf3e96c77fa6c04c470",
            "hash_previo": "0a9d0018f06c548370c378d6387b8e7d0acd096b7fd10966ecf5eb47f29ff13c",
            "indice": 3,
            "prueba": 27,
            "timestamp": 1670155412.7615592,
            "transacciones": [
                "{'origen': 'F', 'destino': 'A', 'cantidad': 1}",
                "{'origen': 'H', 'destino': 'F', 'cantidad': 7}",
                "{'origen': 0, 'destino': '192.168.56.102', 'cantidad': 1}"
            ]
        }
    ],
    "longitud": 3
}
```

Efectivamente, vemos que al intentar minar desde el nodo2 nos salta un conflicto, y la cadena se actualiza con la del nodo1, ya que esta es más larga.

```
 * Running on http://192.168.56.102:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 599-461-942
192.168.56.102 - - [04/Dec/2022 13:03:32] "POST /transacciones/nueva HTTP/1.1" 200 -
192.168.56.102 - - [04/Dec/2022 13:03:32] "POST /transacciones/nueva HTTP/1.1" 200 -
192.168.56.102 - - [04/Dec/2022 13:03:32] "POST /transacciones/nueva HTTP/1.1" 200 -
192.168.56.102 - - [04/Dec/2022 13:03:32] "GET /minar HTTP/1.1" 200 -
192.168.56.102 - - [04/Dec/2022 13:03:32] "GET /chain HTTP/1.1" 200 -
192.168.56.102 - - [04/Dec/2022 13:03:32] "POST /nodos/registrar HTTP/1.1" 200 -
192.168.56.102 - - [04/Dec/2022 13:03:32] "POST /transacciones/nueva HTTP/1.1" 200 -
192.168.56.102 - - [04/Dec/2022 13:03:32] "POST /transacciones/nueva HTTP/1.1" 200 -
192.168.56.102 - - [04/Dec/2022 13:03:32] "POST /transacciones/nueva HTTP/1.1" 200 -
192.168.56.102 - - [04/Dec/2022 13:03:32] "GET /minar HTTP/1.1" 200 -
192.168.56.102 - - [04/Dec/2022 13:03:32] "GET /chain HTTP/1.1" 200 -
192.168.56.101 - - [04/Dec/2022 13:03:32] "GET /chain HTTP/1.1" 200 -
```

Aquí podemos ver todas las comunicaciones con el nodo1.



Aquí podemos ver las comunicaciones con el nodo2 en la máquina virtual. Se ve que, al intentar minar, ha habido un conflicto y el status code es de 202.