

Universidad Pontificia de Comillas
ICAI

PROYECTO FINAL

Fundamentos de los Sistemas Operativos

María González Gómez, Lydia Ruiz Martínez

BLOCKCHAIN



Curso 2023-2024

Resumen

En este informe se explica cómo abordamos el Proyecto Final de la asignatura de Fundamentos de los Sistemas Operativos, realizado por María González Gómez y Lydia Ruiz Martínez, alumnas de segundo curso del Grado en Ingeniería Matemática e Inteligencia Artificial en la Universidad Pontificia de Comillas.

Primero, proporcionamos una breve introducción teórica en la que explicamos tres conceptos básicos para entender el proyecto: la arquitectura descentralizada, Bitcoin y Blockchain.

En relación a los archivos `Blockchain.py` y `Blockchain app.py`, :

- Detallamos cómo implementamos su funcionalidad, siguiendo exhaustivamente el enunciado proporcionado por los profesores de la asignatura.
- Añadimos pantallazos con nuestra resolución de cada apartado, acompañados de una descripción de por qué optamos por esa solución.

Además, incluimos pruebas para justificar el correcto funcionamiento de la aplicación en la máquina virtual con un sistema operativo Ubuntu.

También forman parte de este proyecto los ficheros `pruebas_requests.py` y `requirements.txt`.

En el primero hemos creado un programa que contiene código para enviar peticiones a nuestra api blockchain haciendo uso de la librería de requests. El objetivo es que ese programa realice diferentes pruebas a los nodos que hay en la red (tanto al localhost como a la máquina virtual), para comprobar que todos los nodos están correctamente sincronizados.

En el segundo se han especificado las librerías que se han instalado y sus versiones.

Bibliografía

Índice

Índice	2
1. Introducción	3
1.1. Arquitectura centralizada vs descentralizada	3
1.2. Bitcoin	4
1.3. Blockchain	4
2. Proyecto Blockchain	5
2.1. Transacciones y bloques	5
2.2. Blockchain	7
2.3. Prueba de trabajo	8
3. Aplicación web básica	10
3.1. Métodos de la app	10
3.2. Pruebas	14
3.2.1. Nueva transacción	14
3.2.2. Minar un bloque	15
3.2.3. Comprobación de la cadena hasta el momento	16
4. Aplicación web descentralizada	17
4.1. Pruebas	21
4.2. Resolución de conflictos	23
4.3. Protocolo ICMP	24
4.3.1. Pruebas del protocolo ICMP	26
4.4. Pruebas en la máquina virtual	27
5. Bibliografía	36

1. Introducción

1.1. Arquitectura centralizada vs descentralizada

Los sistemas centralizados son sistemas en los que uno o más nodos clientes se conectan a un servidor que es el que les proporciona el servicio que quieren o que necesitan. Por ejemplo, cuando se accede a Twitter, los usuarios acceden al servidor de Twitter que es el que les proporciona toda la información que necesitan (lo mismo ocurre con acceder a la Wikipedia o a un periódico en internet). Por lo tanto, hay dos componentes principales, el/los nodos maestros (servidores) y los nodos clientes (los usuarios que acceden al servidor). Esta arquitectura es muy popular hoy en día debido a que es una arquitectura simple donde toda la información va a pasar por un equipo central. Esto implica, entre otras cosas, que solo sea necesario mantener actualizado dicho servidor (no tiene responsabilidad sobre los clientes), facilitando el rastreo de la información y haciendo que sea más fácil estandarizar las interacciones entre el servidor central y los nodos cliente. No obstante, esta arquitectura también tiene algunos inconvenientes importantes. Por ejemplo, cuando el servidor falla, éste deja de proveer el servicio que necesitan los usuarios (por ejemplo, cuando se cae el servidor de Twitter o WhatsApp).

Por otro lado, en una red descentralizada, toda la carga de trabajo del procesamiento de los datos recae entre los dispositivos que componen la red. Cada uno de estos nodos actúa de manera independiente de la red, por lo que si uno de estos nodos fallase, la carga se redistribuiría entre el resto de nodos de la red. También se les conoce como redes Peer-to-Peer. Se han vuelto muy populares en los últimos años debido a los avances tecnológicos que han permitido una mayor potencia de procesamiento en los ordenadores. Estas redes tienen algunas ventajas importantes, como la que hemos comentado de que no tienen un único punto de fallo. Si un nodo cae, la carga se puede redistribuir entre el resto de nodos de la red. De la misma manera, son fácilmente escalables, ya que pueden agregarse más elementos a la red (es más complicado saturar la red que saturar un servidor). También ofrecen más privacidad a los usuarios, al no tener que pasar toda la información por un servidor central. Como puntos negativos está la necesidad de que se requiere de una mayor cantidad de dispositivos para que la red funcione bien. Por otro lado, es un sistema más complejo por lo que los nodos deben ser configurados y sincronizados de manera correcta. Además, pueden ocurrir algunos problemas de seguridad, ya que al haber más equipos, todos ellos deben estar correctamente actualizados (es probable que haya algún nodo obsoleto). Por último, al viajar la información por varios dispositivos implica que estas redes sean menos eficientes en términos de energía. No obstante, para algunas aplicaciones, este tipo de arquitectura es preferible a una arquitectura de cliente-servidor, como es el caso de BitTorrent y Bitcoin.

1.2. Bitcoin

En 2009 salió un artículo titulado Bitcoin: un sistema de dinero electrónico peer-to-peer, publicado por un autor (o grupo de autores) bajo el seudónimo de Satoshi Nakamoto. En dicho artículo se proponía un algoritmo para crear una red peer-to-peer para el intercambio de pagos entre personas sin que hubiese ninguna institución central (bancos) que se encargase de hacer dicho trabajo. Nació así el Bitcoin. En dicho artículo, además, se propuso un sistema distribuido para almacenar información (conocido como blockchain). Aunque inicialmente esta tecnología de blockchain se propuso para transacciones monetarias, puede tener otras aplicaciones como los contratos inteligentes (smart contracts), la sanidad o en el sector energético.



Figura 1: Moneda digital descentralizada Bitcoin

1.3. Blockchain

El Blockchain hace referencia a un mecanismo para almacenar información digital. Esta información puede ser desde transacciones monetarias como en Bitcoin a cualquier otra cosa (por ejemplo, archivos). La información se almacena en bloques que están enlazados (o encadenados) empleando hashes criptográficos. De ahí recibe el nombre de blockchain. Es decir, si almacenamos transacciones, cada bloque se compondrá de una o más transacciones. En esencia, el blockchain es una lista enlazada que contiene información ordenada, con las siguientes restricciones:

- Los bloques no se pueden modificar una vez que se añaden (inmutables).
- La información debe añadirse a los bloques cumpliendo determinadas reglas.
- Debe seguir una arquitectura distribuida (no hay un solo punto de control).

La información que vamos a almacenar en los bloques son distintas transacciones que puedan ocurrir (en cada transacción, una persona envía una cantidad de dinero específica a otra). No obstante, dependiendo de la aplicación, estas transacciones pueden estar formadas por diversos tipos de mensajes.

2. Proyecto Blockchain

En este proyecto hemos desarrollado una aplicación Blockchain donde diversos nodos (procesos asociados a un ordenador y a un puerto) han agrupando transacciones en bloques y se sincronizan entre ellos. Para ello, hemos construido la aplicación paso a paso, primero con un solo nodo (un proceso) y luego permitiendo a otros nodos adicionales integrarse en la red.

El proyecto se ha realizado de forma incremental. Primeramente, desarrollamos la parte de back-end, es decir, la lógica interna de la aplicación, donde se ha programado el código necesario para crear los bloques, la cadena de bloques (Blockchain), transacciones monetarias asociadas a cada bloque, etc. Dado que cada bloque tiene un hash asociado (serie de caracteres que identifican al bloque), también se define el mecanismo necesario para calcular dichos hashes y poder construir cadenas de bloques correctas. De la misma manera, se ha diseñado un mecanismo para evitar que otros nodos de la red suplanten nuestra cadena de bloques. Posteriormente, se ha hecho uso de una librería para poder implementar un servicio web que sea accesible mediante peticiones GET y POST del protocolo HTTP. Dicho servicio web se despliega en todos los nodos de nuestra red y podemos establecer una red sincronizada de nodos manteniendo toda la información asociada a los bloques y a las distintas transacciones.

2.1. Transacciones y bloques

La forma en la que se han almacenado los datos en nuestro proyecto de Blockchain es empleando en formato JSON (JavaScript Object Notation). JSON no deja de ser más que un formato de texto para almacenar datos. Se utiliza habitualmente para la transmisión de datos en las aplicaciones web (por ejemplo, el envío de algunos datos desde el servidor al cliente, para que puedan mostrarse en una página web, o viceversa, que sería para lo que se ha utilizado en el proyecto).

Por lo tanto, una transacción se ve de la siguiente forma en formato JSON:

```
{
  "origen": "origen transaccion",
  "destino": "destino transaccion",
  "cantidad": "cantidad de dinero enviada",
  "timestamp": "momento del tiempo en el que se realizo la transaccion"
}
```

Las transacciones se han agrupado en bloques. Por lo tanto, un bloque está compuesto de una o más transacciones. Los bloques que contienen las transacciones se van generando y añadiendo al blockchain. Dado que puede haber múltiples bloques, cada uno tiene un identificador único. Por lo tanto, creamos una clase para almacenar los bloques en el fichero `Blockchain.py`, donde se han creado las clases que representan a los bloques y el blockchain. La clase de bloque queda determinada de esta forma:

```
class Bloque:
    def __init__(
        self,
        indice: int,
        transacciones: list,
        timestamp: int,
        hash_previo: str,
        hash: str,
        prueba: int,
    ):
        """
        Constructor de la clase `Bloque`.
        :param indice: ID unico del bloque.
        :param transacciones: Lista de transacciones.
        :param timestamp: Momento en que el bloque fue generado.
        :param hash_previo hash previo
        :param prueba: prueba de trabajo
        """

        self.indice = indice
        self.transacciones = transacciones
        self.timestamp = timestamp
        self.hash_previo = hash_previo
        self.hash = None
        self.prueba = prueba

    def calcular_hash(self):
        """
        Metodo que devuelve el hash de un bloque
        """

        block_string = json.dumps(self.__dict__, sort_keys=True)
        return hashlib.sha256(block_string.encode()).hexdigest()

    def toDict(self):
        dict_bloque = {
            "hash": self.hash,
            "hash_previo": self.hash_previo,
            "indice": self.indice,
            "prueba": self.prueba,
            "timestamp": self.timestamp,
            "transacciones": self.transacciones,
        }
        return dict_bloque
```

Figura 2: Clase bloque

El hash previo indica la clave criptográfica del anterior bloque. Los bloques deben estar encadenados uno detrás de otro y la forma de garantizar la integridad de dichos bloques es precisamente mediante dicho hash. Para poder calcular el hash de un bloque, hemos hecho uso del método dentro de calcular hash de la clase de Bloque.

Como se puede observar, en ese código se devuelve el hash de un bloque, configurando el contenido del bloque en forma de diccionario. Dicho método nos permite obtener una cadena de 64 caracteres (hash) codificando toda la información del bloque. Es decir, dicha función, coge un bloque y lo transforma a una cadena de 64 caracteres (256 bits).

2.2. Blockchain

Una vez definida la clase de Bloque, debemos definir ahora la clase de Blockchain. Como hemos comentado antes, el blockchain debe ser capaz de mantener una lista de bloques, y además debe almacenar en otra lista aquellas transacciones que todavía no están confirmadas para ser introducidas en un bloque (el siguiente bloque que sería introducido en la cadena).

```
class Blockchain(object):
    def __init__(self):
        self.dificultad = 4
        self.transacciones_no_confirmadas = []
        self.bloques = [self.primer_bloque()]

    def primer_bloque(self):
        """
        Crear un bloque vacío cuyo índice sea 1, sin transacciones y un hash_previo de 1.
        El objetivo de este primer bloque es ser simplemente el primero de la cadena para que luego puedan añadirse más bloques a dicha cadena.
        """
        bloque = Bloque(1, [], time.time(), "1", None, 0)
        bloque.hash = bloque.calcular_hash()
        return bloque

    def last_block(self):
        """
        Devuelve el último bloque de una lista de bloques.
        El objetivo de este último bloque es ser simplemente el último de la cadena.
        """
        return self.bloques[-1]

    def nuevo_bloque(self, hash_previo: str) -> Bloque:
        """
        Crea un nuevo bloque a partir de las transacciones que no están confirmadas
        :param hash_previo: el hash del bloque anterior de la cadena
        :return: el nuevo bloque
        """
        nuevo_bloque = Bloque(
            len(self.bloques) + 1,
            self.transacciones_no_confirmadas,
            time.time(),
            hash_previo,
            None,
            0,
        )
        self.bloques.append(nuevo_bloque)
        self.transacciones_no_confirmadas = []
        return nuevo_bloque
```

Figura 3: Primera parte de la Clase Blockchain

En esta clase hemos creado un método que se llama primer bloque que lo que hace es crear un bloque vacío (cuyo índice es 1), sin transacciones y un hash previo de 1. El objetivo de este primer bloque es ser simplemente el primero de la cadena para que luego puedan añadirse más bloques a dicha cadena. También hemos añadido un método que devuelve el último bloque de una lista de bloques (last block) y un método para crear un nuevo bloque donde debemos introducir el conjunto de transacciones no confirmadas de la red Blockchain en dicho bloque, indicando como tiempo la hora actual con `time.time()`.


```
def nueva_transaccion(self, origen: str, destino: str, cantidad: int) -> int:
    """
    Crea una nueva transaccion a partir de un origen, un destino y una cantidad y la incluye en las listas de transacciones
    :param origen: <str> el que envia la transaccion
    :param destino: <str> el que recibe la transaccion
    :param cantidad: <int> la cantidad
    :return: <int> el indice del bloque que va a almacenar la transaccion
    """
    # [...] Codigo a completar
    nueva_transaccion = {
        "origen": origen,
        "destino": destino,
        "cantidad": cantidad,
        "tiempo": time.time(),
    }
    self.transacciones_no_confirmadas.append(nueva_transaccion)
    return len(self.bloques) + 1
```

Figura 4: Segunda parte de la Clase Blockchain

A la hora de crear una nueva transacción, hemos guardado los datos del origen, el destino, la cantidad, y el timestamp de dicha transacción. Una vez creada, se añade a la lista de transacciones no confirmadas.

2.3. Prueba de trabajo

Antes hemos comentado que los bloques deben almacenar el hash anterior y aunque eso añade más seguridad a la cadena de bloques, siempre puede pasar que alguna persona con intenciones maliciosas modifique algún bloque de la cadena y recalculé todos los hashes de los bloques posteriores para manipular la cadena. Para ello se ha añadido una prueba de trabajo que hace que en lugar de calcular cualquier hash para el bloque, calcula un hash que comienza con al menos tantos ceros (0s) como el valor indicado de dificultad de blockchain (en este caso, 4).

```
def prueba_trabajo(self, bloque: Bloque) -> str:
    """
    Algoritmo simple de prueba de trabajo:
    - Calculara el hash del bloque hasta que encuentre un hash que empiece por tantos ceros como dificultad.
    - Cada vez que el bloque obtenga un hash que no sea adecuado, incrementara en uno el campo de prueba del bloque
    :param bloque: objeto de tipo bloque
    :return: el hash del nuevo bloque (dejara el campo de hash del bloque sin modificar)
    """
    bloque.prueba = 0
    hash_bloque = bloque.calcular_hash()
    while not self.prueba_valida(bloque, hash_bloque):
        bloque.prueba += 1
        hash_bloque = bloque.calcular_hash()
    return hash_bloque
```

Figura 5: Tercera parte de la Clase Blockchain

Este método de prueba trabajo inicializa el campo de prueba del bloque recibido por argumento a 0 y calcula el hash del bloque hasta que encuentra un hash que comienza con tantos 0s como el campo de dificultad del blockchain. Cada vez que devuelva un hash que no cumpla la condición, incrementa el valor de prueba del bloque en 1 y vuelve a calcular el hash. Por lo tanto, el campo de prueba no deja de ser un contador del número de veces que ha habido que calcular el hash del bloque.

hasta cumplir con la restricción de los ceros.

Por último, necesitamos otros dos métodos relacionados con la prueba de trabajo. El primero se llama prueba valida y comprueba que el valor de prueba es correcto. El segundo siguiente es integra bloque, que introduce el bloque en la cadena del blockchain si es un bloque correcto.

```
def prueba_valida(self, bloque: Bloque, hash_bloque: str) -> bool:
    """
    Metodo que comprueba si el hash_bloque comienza con tantos ceros como la dificultad estipulada en el blockchain
    Ademas comprueba que hash_bloque coincide con el valor devuelto del metodo de calcular hash del bloque.
    Si cualquiera de ambas comprobaciones es falsa, devolvera falso y en caso contrario, verdadero
    :param bloque:
    :param hash_bloque:
    :return:
    """
    hash = bloque.calcular_hash()
    if ( hash == hash_bloque and hash_bloque[: self.dificultad] == "0" * self.dificultad ):
        return True
    else:
        return False

def integra_bloque(self, bloque_nuevo: Bloque, hash_prueba: str) -> bool:
    """
    Metodo para integrar correctamente un bloque a la cadena de bloques.
    Debe comprobar que hash_prueba es valida y que el hash del bloque ultimo de la cadena coincida con el hash_previo del bloque que se va a integrar.
    Si pasa las comprobaciones, actualiza el hash del bloque nuevo a integrar con hash_prueba, lo inserta en la cadena y
    hace un reset de las transacciones no confirmadas (vuelve a dejar la lista de transacciones no confirmadas a una lista vacia)
    :param bloque_nuevo: el nuevo bloque que se va a integrar
    :param hash_prueba: la prueba de hash
    :return: True si se ha podido ejecutar bien y False en caso contrario (si no ha pasado alguna prueba)
    """

    if (self.last_block().hash == bloque_nuevo.hash_previo):
        nuevo = Bloque(
            bloque_nuevo.indice,
            bloque_nuevo.transacciones,
            bloque_nuevo.timestamp,
            bloque_nuevo.hash_previo, bloque_nuevo.hash,
            bloque_nuevo.prueba
        )

        nuevo.hash = nuevo.calcular_hash()
        if nuevo.hash == hash_prueba:
            self.bloques.append(nuevo)
            self.transacciones_no_confirmadas = []
            return True
        else:
            return False
    else:
        return False
```

Figura 6: Cuarta parte de la Clase Blockchain

3. Aplicación web básica

Para esta parte de la práctica, ha sido necesario instalar los siguientes paquetes de Python que nos han permitido hacer una pequeña aplicación web:

- Flask
- requests

Además, hemos instalado la aplicación de Postman en el sistema operativo host, que nos ha permitido realizar tests a nuestra aplicación web empleando el formato JSON.

3.1. Métodos de la app

Para crear la aplicación web, creamos un fichero llamado `Blockchain_app.py` con todo el código necesario para el despliegue de nuestra aplicación.

Hemos añadido varios métodos para tener una funcionalidad básica de nuestro proyecto blockchain:

- Un método para recibir transacciones.
- Un método para minar los bloques (construir bloques haciendo uso de las transacciones que tenemos).
- Un método para consultar la cadena creada hasta este momento.
- Un método para obtener los detalles del nodo actual.

Además de esos tres métodos, el proceso principal, antes de lanzar la aplicación, crea un hilo para que cada 60 segundos, dicho hilo haga una copia de seguridad del blockchain actual.

A continuación se muestra cual el código para el método de añadir una transacción y el método para consultar la cadena creada.

En el código de añadir una nueva transacción, se emplea un método de POST. Esto nos va a permitir enviar datos en formato JSON para que ese método lo lea como si fuese un diccionario (variable values) y a partir de ahí acceder a los datos de cada uno de esos campos llamando al método de nueva transaccion del blockchain. Como respuesta, enviamos el mensaje de que la transacción se incluirá en un bloque con un índice en particular y devolveremos el mensaje junto con el código 201, que indica que el resultado de la operación ha sido satisfactorio y se ha registrado la petición. Si por ejemplo, los valores de origen, destino y cantidad, no se incluyen es que faltan valores, es decir, la transacción no está bien construida y por lo tanto no se puede registrar. Dado que eso es un error del cliente (el que ha hecho la petición), se devuelve el código de error de 400.

```
@app.route("/transacciones/nueva", methods=["POST"])
def nueva_transaccion():

    mutex_1.acquire()

    values = request.get_json()
    # Comprobamos que todos los datos de la transaccion estan
    required = [
        "origen",
        "destino",
        "cantidad",
    ]
    if not all(k in values for k in required):
        return "Faltan valores", 400
    # Creamos una nueva transaccion
    indice = blockchain.nueva_transaccion(
        values["origen"], values["destino"], values["cantidad"]
    )
    response = {
        "mensaje": f"La transaccion se incluirea en el bloque con indice {indice}"
    }

    mutex_1.release()

    return jsonify(response), 201
```

Figura 7: Método de añadir una transacción

```
@app.route("/chain", methods=["GET"])
def blockchain_completa():
    """
    Endpoint para obtener la cadena completa de bloques mediante una solicitud GET.

    :return: Una respuesta JSON con la cadena de bloques y su longitud.
    """
    mutex_2.acquire()

    # Construimos la respuesta JSON que incluye la cadena de bloques y su longitud
    response = {
        "chain": [b.toDict() for b in blockchain.bloques if b.hash is not None],
        "longitud": len(blockchain.bloques),
    }
    mutex_2.release()

    # Respondemos con la cadena de bloques y un código de éxito 200
    return jsonify(response), 200
```

Figura 8: Método de consultar la cadena

En el caso del método de blockchain completa, lo que hace es transformar cada bloque de la cadena en formato diccionario (método toDict() de la clase de Bloque) y devolver la lista. En este caso, devolvemos una salida al usuario con código 200 que indica que la petición es correcta. En este caso es un método GET. Con GET, no estamos enviando datos desde el cliente (no vamos a enviar un JSON como a la hora de registrar las transacciones), sino que solamente obtenemos una consulta del servidor sin modificar ninguna información.

El siguiente método añadido es el de minar un bloque (método de GET, dado que no enviamos nada). Cuando se procede a hacer el minado, se tiene que añadir una transacción adicional (ya que el minero debe recibir un pago por integrar el bloque en el blockchain). Dicha transacción tiene como origen el carácter 0, el destino es la ip del ordenador (obtenida mediante la variable de mi ip), y la cantidad de dicho dinero será 1.

```
@app.route("/minar", methods=["GET"])
def minar():

    mutex_1.acquire()
    # No hay transacciones
    if len(blockchain.transacciones_no_confirmadas) == 0:
        response = {
            "mensaje": "No es posible crear un nuevo bloque. No hay transacciones"
        }
    else:
        # Hay transaccion, por lo tanto ademas de minar el bloque, recibimos recompensa

        blockchain.nueva_transaccion("0", mi_ip, 1)

        conflictos = resuelve_conflictos()

        if conflictos == True:
            # Borro las transacciones actuales no confirmadas
            blockchain.transacciones_no_confirmadas = []
            response = {
                "mensaje": "Ha habido un conflicto. Esta cadena se ha actualizado con una version mas larga"
            } # Obligando al nodo a volver a volver a capturar transacciones si quiere crear él un nuevo bloque
        else:
            nuevo_bloque = Bloque(
                len(blockchain.bloques) + 1,
                blockchain.transacciones_no_confirmadas,
                time.time(),
                blockchain.last_block().hash,
                "0",
                0,
            )

            nuevo_hash = blockchain.prueba_trabajo(nuevo_bloque)
            validez = blockchain.integra_bloque(nuevo_bloque, nuevo_hash)

            if validez == True:
                nuevo_bloque.hash = nuevo_hash
                response = {
                    "hash_bloque": nuevo_bloque.hash,
                    "hash_previo": nuevo_bloque.hash_previo,
                    "indice": nuevo_bloque.indice,
                    "mensaje": "Nuevo bloque minado",
                    "prueba": nuevo_bloque.prueba,
                    "timestamp": nuevo_bloque.timestamp,
                    "transacciones": nuevo_bloque.transacciones,
                }

            mutex_1.release()

    return jsonify(response), 200
```

Figura 9: Método de minar un bloque

Primeramente creamos un nuevo bloque con el hash previo, procedemos a realizar la prueba de trabajo y finalmente integramos el bloque en el blockchain con la prueba de trabajo calculada. Una vez generado, crea un mensaje de respuesta (response) que contiene un mensaje.

Esta función se utiliza para obtener los detalles del sistema del nodo actual. Devuelve un JSON que contiene los siguientes datos: la máquina, el nombre del sistema y la versión.

```
@app.route("/system", methods=["GET"])
def detalles_nodo_actual():
    """
    Endpoint para obtener detalles específicos del nodo actual, como el sistema operativo, su versión y el tipo de procesador.

    :return: Una respuesta JSON con información sobre el sistema del nodo.
    """
    mutex_1.acquire()

    # Obtenemos detalles del sistema operativo
    response = {
        "maquina": platform.machine(),
        "nombre_sistema": platform.system(),
        "version": platform.version(),
    }

    mutex_1.release()

    # Respondemos con los detalles del sistema y un código de éxito 200

    return jsonify(response), 200

nodos_red = []
```

Figura 10: Método de obtener detalles del nodo

3.2. Pruebas

En esta sección realizamos algunas pruebas para comprobar que todo está correcto por ahora. Para ello ejecutamos la parte de la aplicación en `Blockchain_app.py`, que debe quedarse esperando hasta que recibamos conexiones en el puerto 5000. Es decir, la salida es la siguiente:

```
PS C:\Users\Lydia\Desktop\PROYECTO_FUSO> & C:/Users/Lydia/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/Lydia/Desktop/PROYECTO_FUSO/Blockchain_app.py"
Copia de seguridad creada en respaldo-nodo192.168.1.102-5000.json
* 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://192.168.1.102:5000
Press CTRL+C to quit
```

Figura 11: Salida de `Blockchain_app.py`

Ahora, abrimos Postman y procedemos a crear una nueva petición HTTP (HTTP request) desde nuestro ordenador host.

3.2.1. Nueva transacción

La primera acción que realizamos es añadir una transacción. Para ello, configuramos una petición de HTTP de tipo POST (porque estamos enviando datos al nodo), pasándole como cuerpo (Body, raw) un JSON con el formato de transacción que se ha discutido.

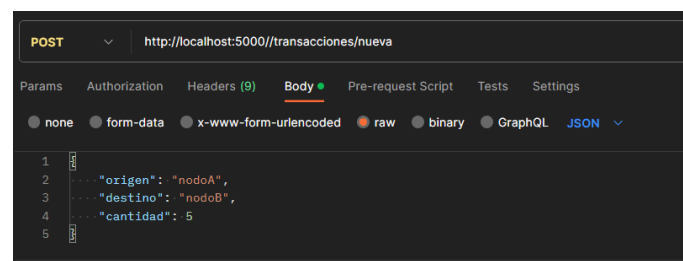


Figura 12: Transacción nueva

Obtenemos el siguiente mensaje:

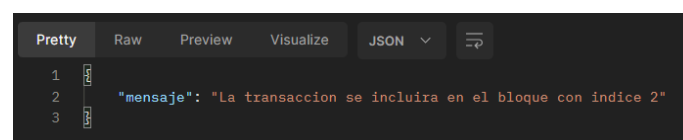


Figura 13: Respuesta a la petición anterior

3.2.2. Minar un bloque

En este caso, vamos a solicitar al nodo minar un bloque. Para ello, creamos otra petición HTTP de tipo GET (ya que en este caso, solo es de consulta). Por lo tanto replicando lo que se indicó en el apartado anterior (pero con la acción de get en la URL de minar), obtenemos lo siguiente :

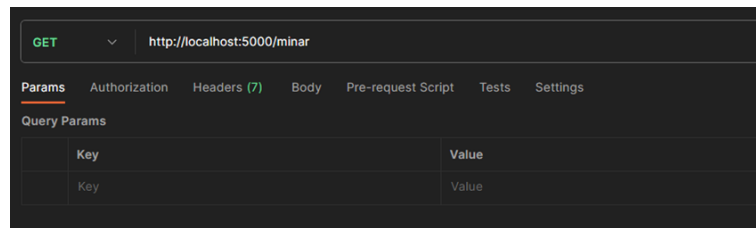


Figura 14: Minar el bloque

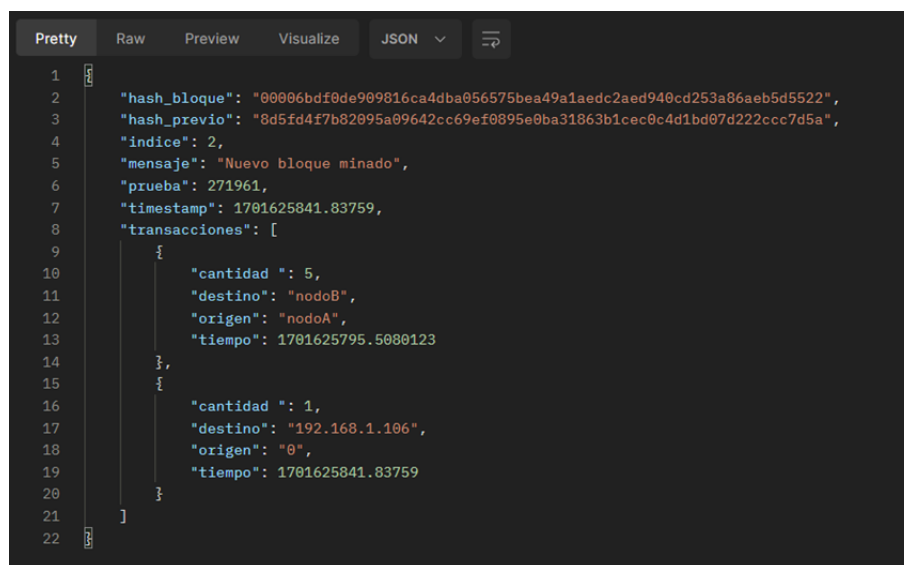
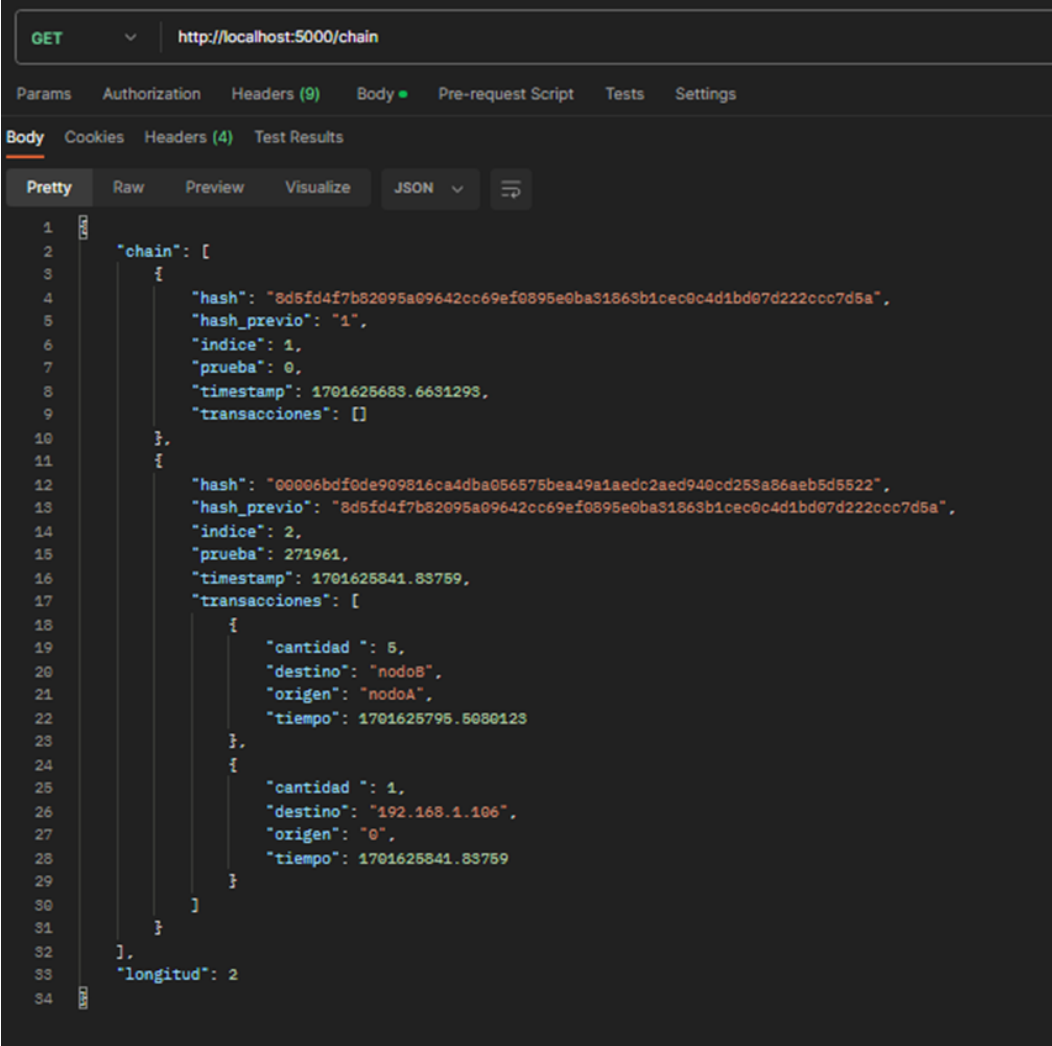


Figura 15: Resultado obtenido al minar

El hash del bloque empieza con 4 ceros como indicamos en nuestra prueba de trabajo y hay 2 transacciones, no una, debido a que nuestro nodo debe recibir un “pago” por minar el bloque.

3.2.3. Comprobación de la cadena hasta el momento

En la ruta de /chain, podemos comprobar la cadena del blockchain que tenemos hasta el momento. Por lo tanto, volviendo a realizar una acción de GET a la URL de chain, obtenemos lo siguiente:



```
GET http://localhost:5000/chain

Body
Pretty
Raw
Preview
Visualize
JSON
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
{
  "chain": [
    {
      "hash": "8d5fd4f7b82095a09642cc69ef0895e0ba31863b1cec0c4d1bd07d22ccc7d5a",
      "hash_previo": "1",
      "indice": 1,
      "prueba": 0,
      "timestamp": 1701625683.6631293,
      "transacciones": []
    },
    {
      "hash": "00006bdf0de909816ca4dba056575bea49a1aedc2aed940cd253a86aeb5d5522",
      "hash_previo": "8d5fd4f7b82095a09642cc69ef0895e0ba31863b1cec0c4d1bd07d22ccc7d5a",
      "indice": 2,
      "prueba": 271961,
      "timestamp": 1701625841.83759,
      "transacciones": [
        {
          "cantidad": 5,
          "destino": "nodoB",
          "origen": "nodoA",
          "tiempo": 1701625795.5080123
        },
        {
          "cantidad": 1,
          "destino": "192.168.1.106",
          "origen": "0",
          "tiempo": 1701625841.83759
        }
      ]
    }
  ],
  "longitud": 2
}
```

Figura 16: Resultado obtenido al comprobar la cadena

La cual nos devuelve que la longitud de la cadena es 2 (por el bloque inicial y el bloque previamente minado). Se puede ver, además, como el hash previo del segundo bloque coincide con el hash del primer bloque.

4. Aplicación web descentralizada

En esta parte del proyecto, vamos a permitir que otros nodos se registren en nuestra aplicación web descentralizada. Para pasar de un solo nodo a una red de pares “peer-to-peer” necesitamos crear un punto de acceso para permitirle a un nodo tener conciencia de otros compañeros en la red y sincronizarse con la cadena. Para realizar esto, junto con la variable global de blockchain, crearemos un conjunto (set) en el fichero `Blockchain_app.py`, llamado `nodos red` que almacenará todos los nodos de la red (menos el del programa actual).

Inicialmente, este conjunto está vacío. Además, hemos creado dos funciones adicionales en nuestra aplicación. La primera, registrar nodos completo (accesible desde `/nodos/registrar`), recibe mediante el método de POST una lista de URLs y lo que hace es almacenar los nodos recibidos en `nodos red` y enviar a dichos nodos la blockchain del nodo al que se han unido. Es decir, estos nuevos nodos tienen que tener una copia de la blockchain del nodo principal.

La primera función se utiliza para registrar nuevos nodos en la red. Recibe una lista de direcciones de los nodos de la red.

Para cada nodo en la lista, se prepara un JSON con la dirección del nodo y la blockchain actualizada, luego, se envía este JSON al nodo correspondiente utilizando el método POST.

En caso de que todas las solicitudes sean exitosas, la función devuelve un diccionario que un mensaje de éxito en la operación y la lista de todos los nodos en la red, sino devuelve un mensaje de error.

```
@app.route("/nodos/registrar", methods=["POST"])
def registrar_nodos_completo():
    """
    Endpoint para registrar nodos en la red y actualizar sus blockchains.
    :return: Una respuesta JSON con información sobre el resultado del registro de nodos.
    """
    mutex_1.acquire()

    values = request.get_json()

    global blockchain
    global nodos_red

    nodos_nuevos = values.get("direccion_nodos")

    if nodos_nuevos is None:
        return "Error: No se ha proporcionado una lista de nodos", 400

    all_correct = True

    nodo_local = "http://" + mi_ip_local + ":" + mi_puerto
    nodo_red_enviar = []
    nodo_red_enviar.append(nodo_local)

    for nodo in nodos_nuevos:
        nodo_red_enviar.append(nodo)

    for nodo in nodos_nuevos:
        if (
            (nodo not in nodos_red)
            and (nodo != ("http://" + mi_ip + ":" + mi_puerto))
            and (nodo != ("http://" + mi_ip_local + ":" + mi_puerto))
        ):
            nodos_red.append(nodo)
            nodo_enviados = nodo_red_enviar.copy()
            nodo_enviados.remove(nodo)
            data = {
                "nodos_direcciones": nodo,
                "blockchain": [
                    b.toDict() for b in blockchain.bloques if b.hash is not None
                ],
                "nodos_red": nodo_enviados,
            }
            # Enviar JSON al nodo para registrar y actualizar blockchain
            response = requests.post(
                nodo + "/nodos/registro_simple",
                data=json.dumps(data),
                headers={"Content-Type": "application/json"},
            )
            if response.status_code != 200:
                all_correct = False
    if all_correct:
        response = {
            "mensaje": "Se han incluido nuevos nodos en la red",
            "nodos_totales": nodos_red,
        }
    else:
        response = {
            "mensaje": "Error notificando el nodo estipulado",
        }

    mutex_1.release()

    return jsonify(response), 201
```

Figura 17: Método de registrar nodos nuevos en la red

Para notificar a los nodos que quieren unirse a la red, empleamos el segundo método llamado “registrar nodo actualiza blockchain”. Esta función se utiliza para registrar un nodo y actualizar la blockchain. Recibe el mismo diccionario inicial que la función anterior, pero en este caso, agrega la dirección del nodo a la lista de nodos en la red y actualiza la blockchain del nodo actual con la recibida.

En caso de que la blockchain pase la prueba de validez, devolvemos un mensaje de éxito.

```
@app.route("/nodos/registro_simple", methods=["POST"])
def registrar_nodo_actualiza_blockchain():
    """
    Endpoint para que un nodo registre y actualice su blockchain en la red.
    :return: Una respuesta JSON con información sobre el resultado del registro y actualización.
    """

    mutex_2.acquire()

    # Obtenemos la variable global de blockchain
    global blockchain
    global nodos_red

    blockchain_leida = Blockchain.Blockchain()

    read_json = request.get_json()
    lista_nodos = read_json.get("nodos_red")
    for nodo in lista_nodos:
        if (nodo not in nodos_red):
            nodos_red.append(nodo)
    block_data = read_json.get("blockchain", [])
    for block in block_data:
        bloque_leido = Bloque(
            block['indice'],
            block['transacciones'],
            block['timestamp'],
            block['hash_previo'],
            block['hash'],
            block['prueba']
        )
        hash_leido = block['hash']
        bloque_leido.hash = bloque_leido.calcular_hash()
        if (hash_leido == bloque_leido.hash):
            if block['indice'] == 1:
                blockchain_leida.bloques[0].indice = block['indice']
                blockchain_leida.bloques[0].transacciones = block['transacciones']
                blockchain_leida.bloques[0].timestamp = block['timestamp']
                blockchain_leida.bloques[0].hash_previo = block['hash_previo']
                blockchain_leida.bloques[0].hash = block['hash']
                blockchain_leida.bloques[0].prueba = block['prueba']
            else:
                blockchain_leida.bloques.append(bloque_leido)
                blockchain_leida.transacciones_no_confirmadas = []

    mutex_2.release()

    if blockchain_leida is None:
        return jsonify("El blockchain de la red esta corrupto"), 400
    else:
        blockchain = blockchain_leida
        return jsonify("La blockchain del nodo"
            + str(mi_ip)
            + ":"
            + str(puerto)
            + "ha sido correctamente actualizada"
        ), 200
```

Figura 18: Método de registrar nodo y actualizar la blockchain

Por lo tanto, el procedimiento de cara al registro de nodos, sería el siguiente:

- Cuando queramos registrar un nodo en la aplicación desplegada, le pasaremos un JSON a la dirección `http://<ip>:<puerto>/nodos/registrar` (por ejemplo `http://localhost:5000/nodos/registrar`) con este formato:

```
{
    "direccion_nodos": ["http://127.0.0.1:5001"]
}
```

- Una vez enviado, se ejecuta el código de `registrar_nodos_completo`, que, por cada uno de los nodos de la lista recibida, crea un JSON para ser enviado a cada uno de los nodos de la lista. Dicho JSON contiene lo siguiente:
 - Una copia del blockchain del nodo principal.
 - Una lista de todos los nodos que habría en la red.

Supongamos que tenemos un nodo principal en `http://localhost:5000`. Supongamos que le enviamos a la dirección `http://localhost:5000/nodos/registrar` el siguiente JSON:

```
{
    "direccion_nodos": ["http://localhost:5001", "http://localhost:5002"]
}
```

La función de `/nodos/registrar` debe enviar al nodo `http://localhost:5001` un JSON conteniendo una lista con los nodos `http://localhost:5000` y `http://localhost:5002` (nodos de la red que no son él mismo), y además, una copia del blockchain mediante POST a la dirección de `/nodos/registro_simple` (al nodo `http://localhost:5002` se le enviaría una lista con las direcciones de `http://localhost:5000` y `http://localhost:5001`). Para enviar ese JSON, se ha empleado el código:

```
response = requests.post(nodo + "/nodos/registro_simple",
                        data=json.dumps(data),
                        headers={'Content-Type': "application/json"})
```

Donde `data` es un diccionario conteniendo dos claves y `nodo` hace referencia al nodo al que queremos enviar la copia de la blockchain. La primera clave, llamada `"nodos_direcciones"`, contiene la lista de nodos a almacenar y la segunda clave, denominada `"blockchain"`, contiene una copia del blockchain del nodo actual pasado a JSON. Todo esto se ejecuta en la función de registro simple.

Por lo tanto, cuando el nodo en el puerto 5000 reciba una lista de nodos a registrar, en cada uno de esos nodos se ejecuta el método indicado con

/nodos/registro_simple". Si todos los nodos reciben la copia de la blockchain de manera correcta, entonces se devuelve un mensaje diciendo los nodos que se han añadido a la red.

Cuando el nodo que se quiere añadir a la red recibe la lista y la copia del blockchain, se ejecuta el método de `registrar_nodo_actualiza_blockchain` (el método que se ejecuta en `/nodos/registro_simple`). Este método debe hacer lo siguiente:

- Almacenar en su variable de `nodos_red`, la lista de nodos recibida.
- Si recibe el blockchain en formato JSON, debe construir bloque a bloque la cadena a partir del JSON. Para ello debe, por cada bloque de la blockchain del JSON, crear el bloque y comprobar que el hash del bloque es válido.

4.1. Pruebas

Una vez añadidas estas implementaciones, comprobamos el funcionamiento de la aplicación de la siguiente forma.

Primeramente, ejecutamos la aplicación en el puerto 5000, y lanzamos otra instancia en el puerto 5001. En la aplicación del puerto 5000, añadimos alguna transacción y minamos un bloque. Una vez hemos hecho eso, comprobamos que en la instancia del puerto 5001, tenemos una cadena vacía haciendo una petición get como se muestra en la siguiente imagen:

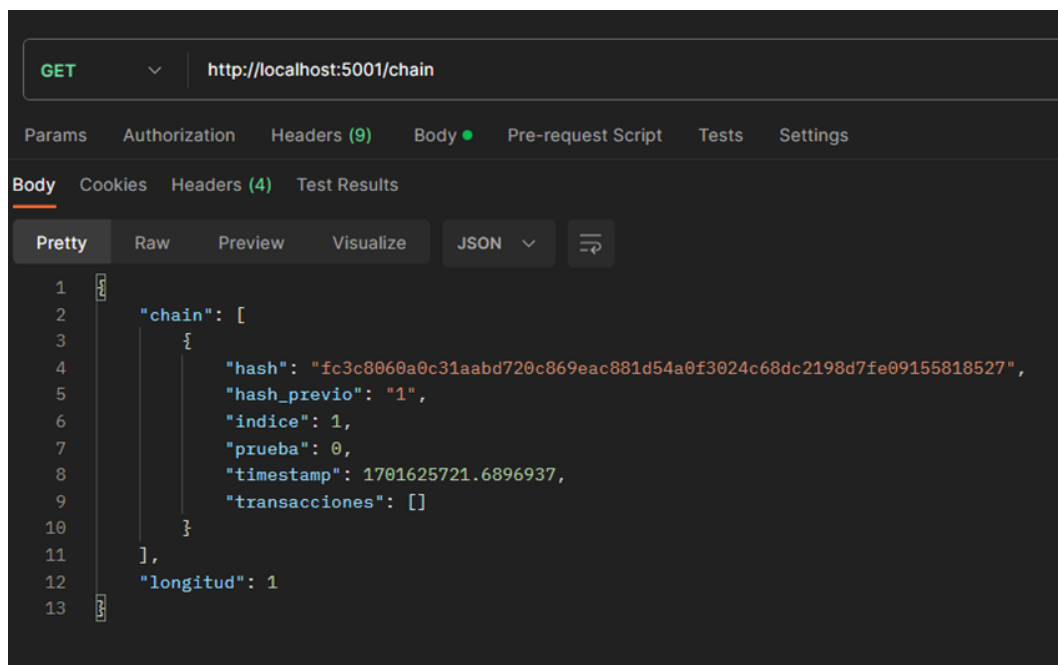


Figura 19: Resultado obtenido al comprobar la cadena

Una vez comprobado que el nodo 5001 no tiene la cadena actualizada, registramos dicho nodo en la aplicación desplegada en el puerto 5000, como se muestra en esta imagen:

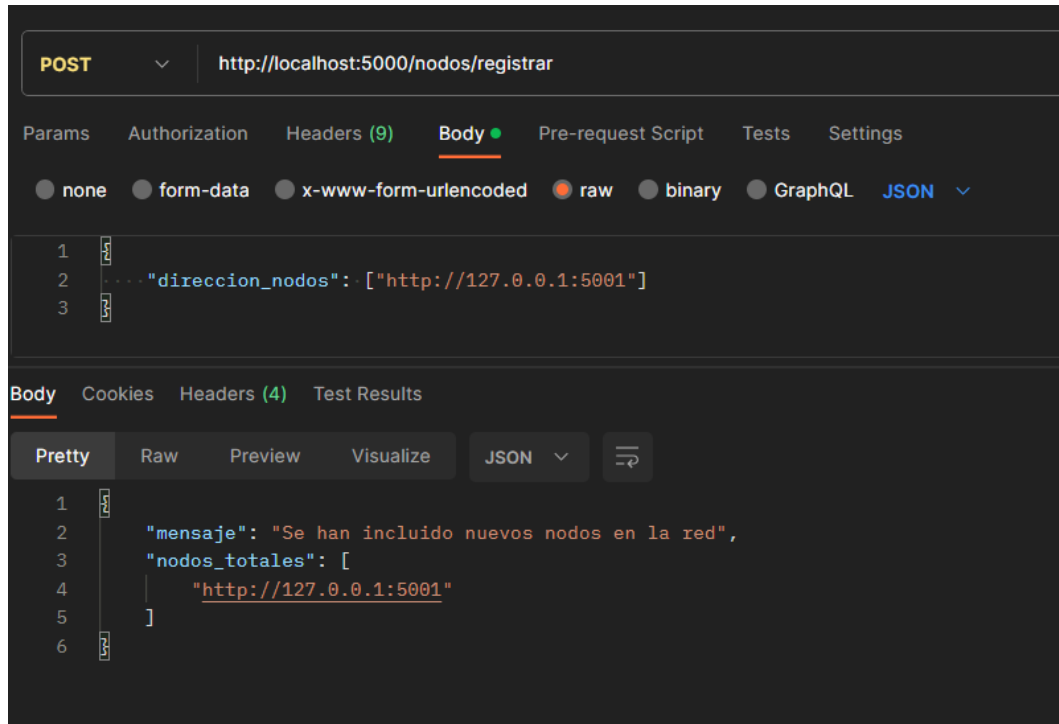


Figura 20: Resultado obtenido al comprobar la cadena

Si todo ha salido correcto, ahora volvemos a hacer una petición de get a la aplicación desplegada en el puerto 5001 y deberíamos obtener la misma cadena del puerto 5000.

```
GET http://localhost:5001/chain

Body
Pretty
Raw
Preview
Visualize
JSON
[
  {
    "hash": "8d5fd4f7b82095a09642cc69ef0895e0ba31863b1cec0c4d1bd07d222ccc7d5a",
    "hash_previo": "1",
    "indice": 1,
    "prueba": 0,
    "timestamp": 1701625683.6631293,
    "transacciones": []
  },
  {
    "hash": "00006bdf0de909816ca4dba056575bea49a1aedc2aed940cd253a86aeb5d5522",
    "hash_previo": "8d5fd4f7b82095a09642cc69ef0895e0ba31863b1cec0c4d1bd07d222ccc7d5a",
    "indice": 2,
    "prueba": 271961,
    "timestamp": 1701625841.83759,
    "transacciones": [
      {
        "cantidad": 5,
        "destino": "nodoB",
        "origen": "nodoA",
        "tiempo": 1701625795.5080123
      },
      {
        "cantidad": 1,
        "destino": "192.168.1.106",
        "origen": "0",
        "tiempo": 1701625841.83759
      }
    ]
  }
],
"longitud": 2
```

Figura 21: Resultado obtenido al comprobar la cadena

4.2. Resolución de conflictos

Tenemos ya varios nodos cooperando, pero cada uno puede estar minando bloques de forma que cada nodo puede tener cadenas completamente distintas cuando el objetivo es que haya una única cadena entre todos los nodos. Por esta razón necesitamos un mecanismo para resolver estas discrepancias. Para ello creamos un algoritmo en `Blockchain_app.py` que comprueba si nuestra cadena es la más larga del resto de nodos de la cadena (o al menos tiene la misma longitud). Si es así, no pasará nada, y si no, lo que hacemos es cambiar nuestra cadena actual por la más larga de la red. La función que creamos se llama “resuelve conflictos”. Esta función se utiliza para resolver los posibles conflictos que pueden ocurrir a lo largo de la ejecución del programa, además de para establecer un consenso en la red.


```
def resuelve_conflictos():  
    """  
    Mecanismo para establecer el consenso y resolver los conflictos  
    """  
    global blockchain  
    global nodos_red  
    conflicto = False  
    longitud_actual = len(blockchain.bloques)  
    for nodo in nodos_red:  
        response = requests.get(str(nodo) + "/chain") # Cadena en formato json  
        longitud_cadena = response.json()["longitud"]  
        if longitud_cadena > longitud_actual:  
            conflicto = True  
  
    if conflicto == True:  
        return True # Hay conflicto  
    else:  
        return False # No ha habido conflicto
```

Figura 22: Método de resolver conflicto

4.3. Protocolo ICMP

Cuando se establece una comunicación con cualquier host remoto, es importante conocer el protocolo ICMP, que sirve básicamente para comprobar que el host de destino tiene conexión. En esta aplicación blockchain hemos creado un servicio nuevo que sea accesible desde la ruta “/ping”. Cuando se acceda a dicho servicio, se envía un json a cada nodo de la red conteniendo los datos del host de origen (ip + puerto) que ha enviado el ping, un mensaje conteniendo la palabra “PING” y el momento en el que se envió dicho mensaje. El envío del ping, es recogido en cada nodo en otro servicio llamado “pong” que envía como respuesta al nodo que ha solicitado el ping sus datos de ip y puerto y un mensaje que contiene el mensaje original que envió el nodo solicitando el ping, y una respuesta indicando la ip y el puerto del host que responde junto con el mensaje PONG.

```
@app.route("/ping", methods=["GET"])
def ping():
    """
    Endpoint para enviar solicitudes de ping a nodos en la red y obtener respuestas.
    :return: Una respuesta JSON con información sobre los pings y las respuestas de los nodos en la red.
    """
    mutex_1.acquire()

    global nodos_red

    nodo_origen = "http://" + mi_ip_local + ":" + mi_puerto

    respuesta_final = ""

    respuestas = 0

    for nodo in nodos_red:
        data = {"nodos": nodo_origen, "mensaje": "PING ", "hora": time.time()}

        response = requests.post(
            nodo + "/pong",
            data=json.dumps(data),
            headers={"Content-Type": "application/json"},
        )

        read_json = response.json()
        nodes_destino = read_json.get("ip_puerto_destino")
        retardo = read_json.get("Retardo")
        respuesta_final += f"#PING de {nodos_destino[7:]}. Respuesta: PONG {nodo[7:]}. Retardo {retardo}. "
        if response.status_code == 200:
            respuestas += 1

    if respuestas == len(nodos_red):
        respuesta_final += " Todos los nodos responden"
    else:
        respuesta_final += " Algún nodo no responde"

    mutex_1.release()

    return jsonify({"respuesta_final": respuesta_final})
```

Figura 23: Método de PING

```
@app.route("/pong", methods=["POST"])
def pong():
    """
    Endpoint para recibir solicitudes de ping, calcular el retardo y enviar respuestas de pong.
    :return: Una respuesta JSON con información sobre el nodo de destino y el retardo.
    """
    mutex_2.acquire()

    read_json = request.get_json()
    nodes_destino = read_json.get("nodos")
    time_leido = read_json.get("hora")
    retardo = time.time() - time_leido

    mutex_2.release()

    return jsonify({"ip_puerto_destino": nodes_destino, "Retardo": retardo}), 200
```

Figura 24: Método de PONG

4.3.1. Pruebas del protocolo ICMP



Figura 25: Método de PING/PONG

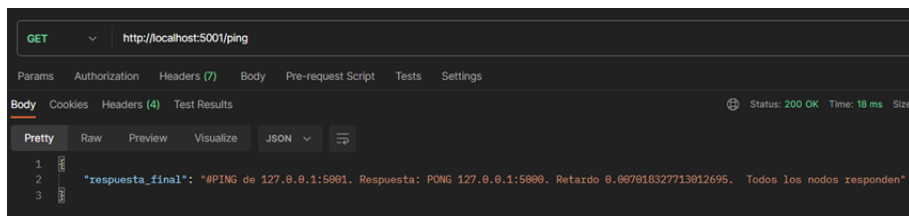


Figura 26: Método de PING/PONG

4.4. Pruebas en la máquina virtual

En este apartado se han realizado las modificaciones necesarias en la ejecución de la aplicación para que en lugar de ejecutarse todo en el entorno windows o mac (en nuestro caso Windows), que haya un nodo en el entorno windows/mac (host) y un nodo en una de las máquinas virtuales de Ubuntu (guest).

Los cambios realizados en el código consisten en utilizar la variable mi ip en lugar de mi ip local. Posteriormente, en la función de registrar nodos, definimos nodo local como `http:// + mi ip + : mi puerto`. Además, en la función que realiza el PING, se define nodo origen como `http:// + mi ip + : mi puerto`.

El nodo en el entorno Windows es `http://192.168.1.106:5000` y en la máquina virtual de Ubuntu es `http://192.168.1.114:5000`.

A continuación se muestran los pantallazos de manera análoga a las pruebas realizadas anteriormente.

```
/bin/python3 /home/lydiaRuiz/Escritorio/Software/Blockchain_app.py
lydiaRuiz@lydiaRuiz-VirtualVox:~$ /bin/python3 /home/lydiaRuiz/Escritorio/Software/Blockchain_app.py
* Serving Flask app 'Blockchain_app'
* Debug mode: off
Copia de seguridad creada en respaldo-nodo192.168.1.114-5000.json
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://192.168.1.114:5000
Press CTRL+C to quit
```

Figura 27: Pruebas en la máquina virtual en el entorno Linux

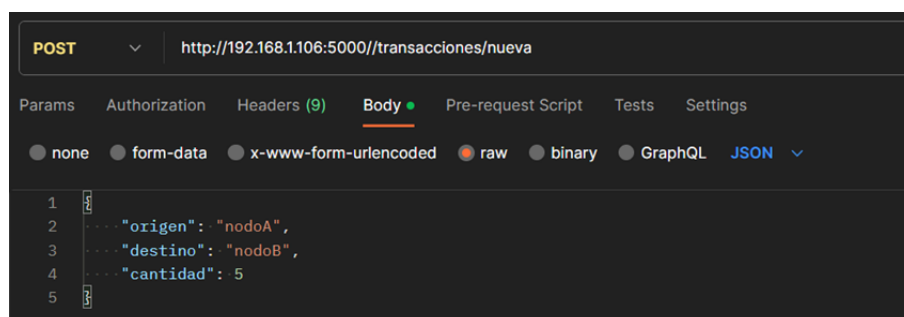


Figura 28: Transacción nueva

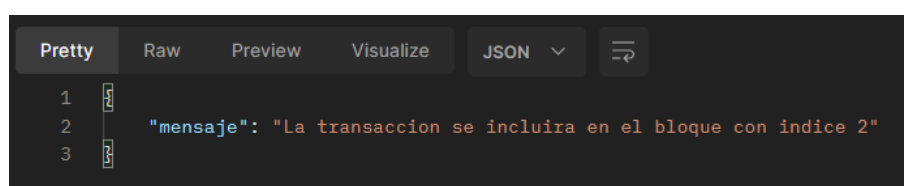


Figura 29: Respuesta a la petición anterior

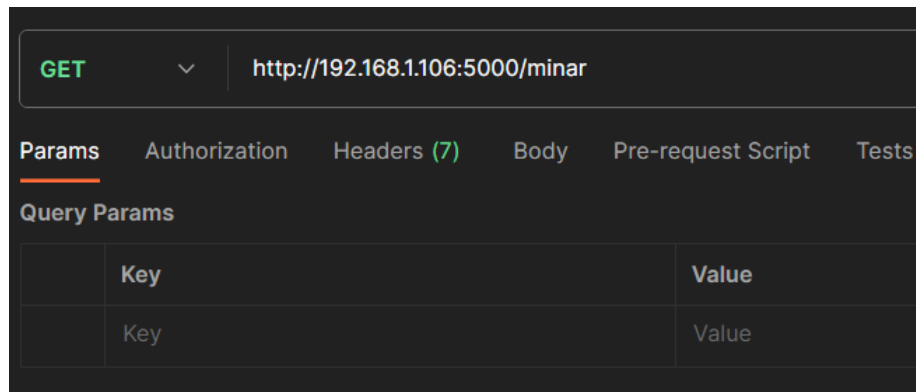


Figura 30: Minar el bloque

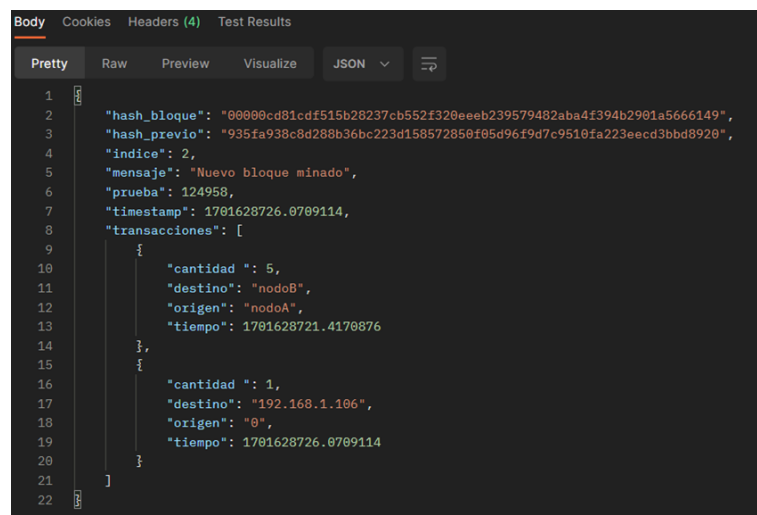
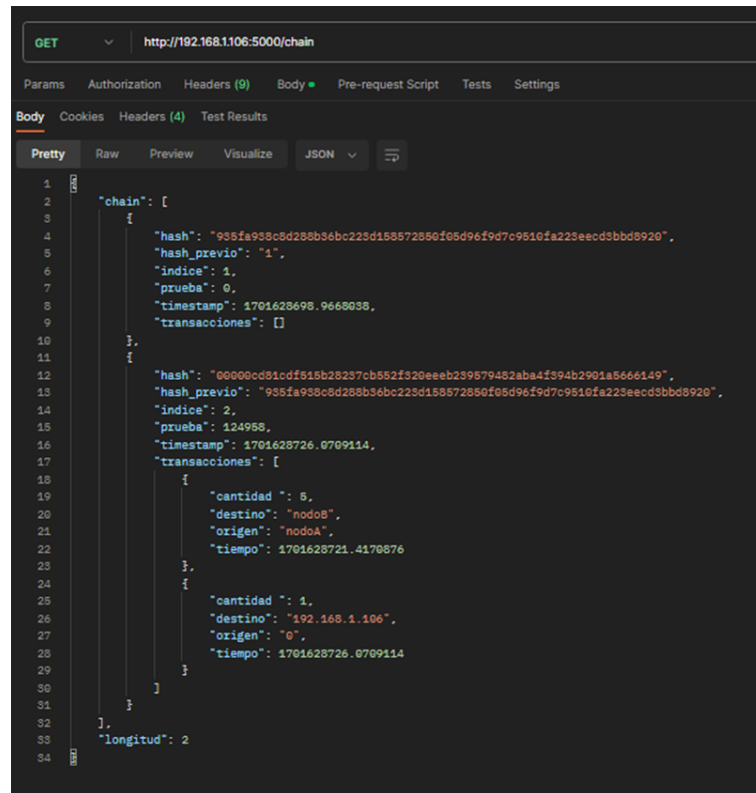
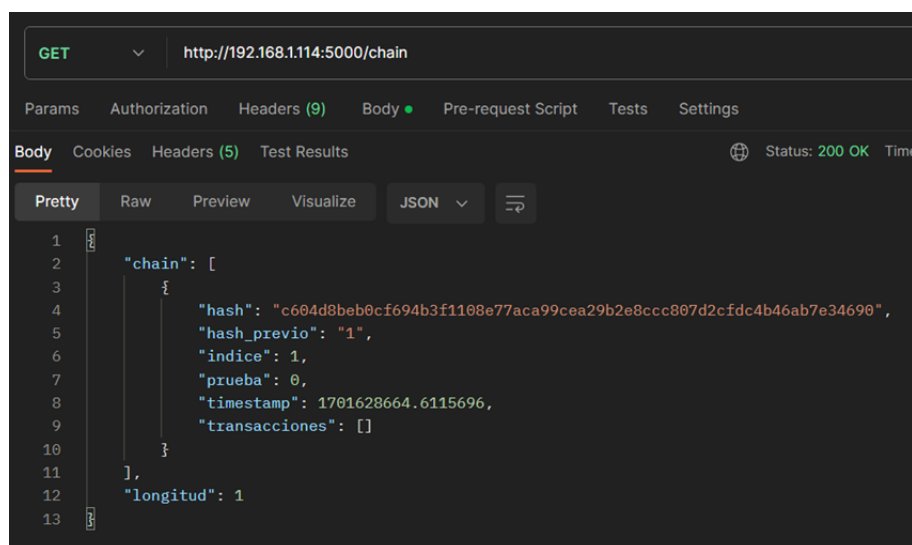


Figura 31: Resultado obtenido al minar



```
1 GET http://192.168.1.106:5000/chain
2
3 Params Authorization Headers (9) Body Pre-request Script Tests Settings
4 Body Cookies Headers (4) Test Results
5 Pretty Raw Preview Visualize JSON
6
7 1 {
8   "chain": [
9     {
10      "hash": "935fa938c8d288b36bc223d158572856f05d96f9d7c9510fa223eecd3bbd9920",
11      "hash_previo": "1",
12      "indice": 1,
13      "prueba": 0,
14      "timestamp": 1701628698.9668038,
15      "transacciones": []
16     },
17     {
18      "hash": "60006cd81cdf518b28237cb552f328eeeb239579482aba4f394b2901a5666149",
19      "hash_previo": "935fa938c8d288b36bc223d158572856f05d96f9d7c9510fa223eecd3bbd9920",
20      "indice": 2,
21      "prueba": 124958,
22      "timestamp": 1701628726.0709114,
23      "transacciones": [
24        {
25          "cantidad": 5,
26          "destino": "nodoB",
27          "origen": "nodoA",
28          "tiempo": 1701628721.4170876
29        },
30        {
31          "cantidad": 1,
32          "destino": "192.168.1.106",
33          "origen": "0",
34          "tiempo": 1701628726.0709114
35        }
36      ]
37     }
38   ],
39   "longitud": 2
40 }
```

Figura 32: Resultado obtenido al comprobar la cadena en Windows



```
1 GET http://192.168.1.114:5000/chain
2
3 Params Authorization Headers (9) Body Pre-request Script Tests Settings
4 Body Cookies Headers (5) Test Results Status: 200 OK Time
5 Pretty Raw Preview Visualize JSON
6
7 1 {
8   "chain": [
9     {
10      "hash": "c604d8beb0cf694b3f1108e77aca99cea29b2e8ccc807d2cfdc4b46ab7e34690",
11      "hash_previo": "1",
12      "indice": 1,
13      "prueba": 0,
14      "timestamp": 1701628664.6115696,
15      "transacciones": []
16     }
17   ],
18   "longitud": 1
19 }
```

Figura 33: Resultado obtenido al comprobar la cadena en Linux

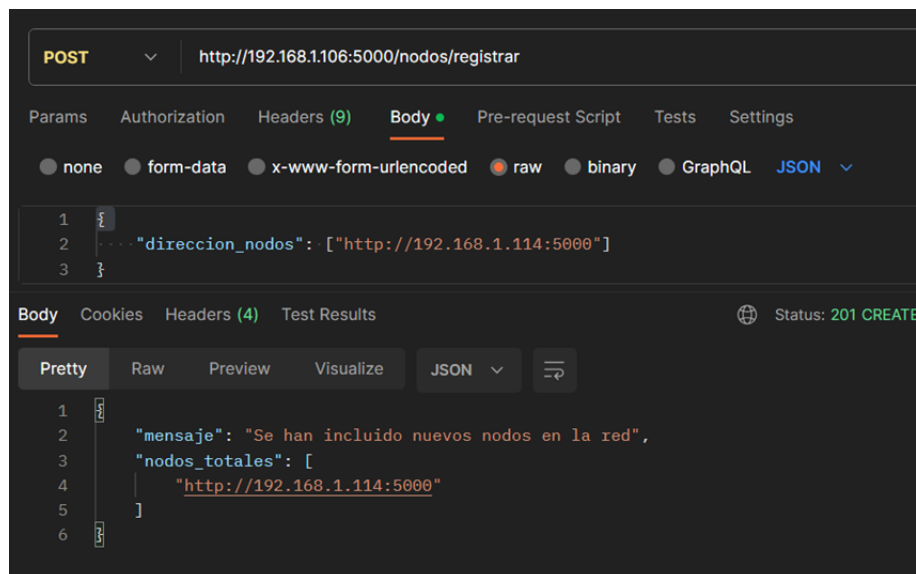


Figura 34: Registro de un nodo

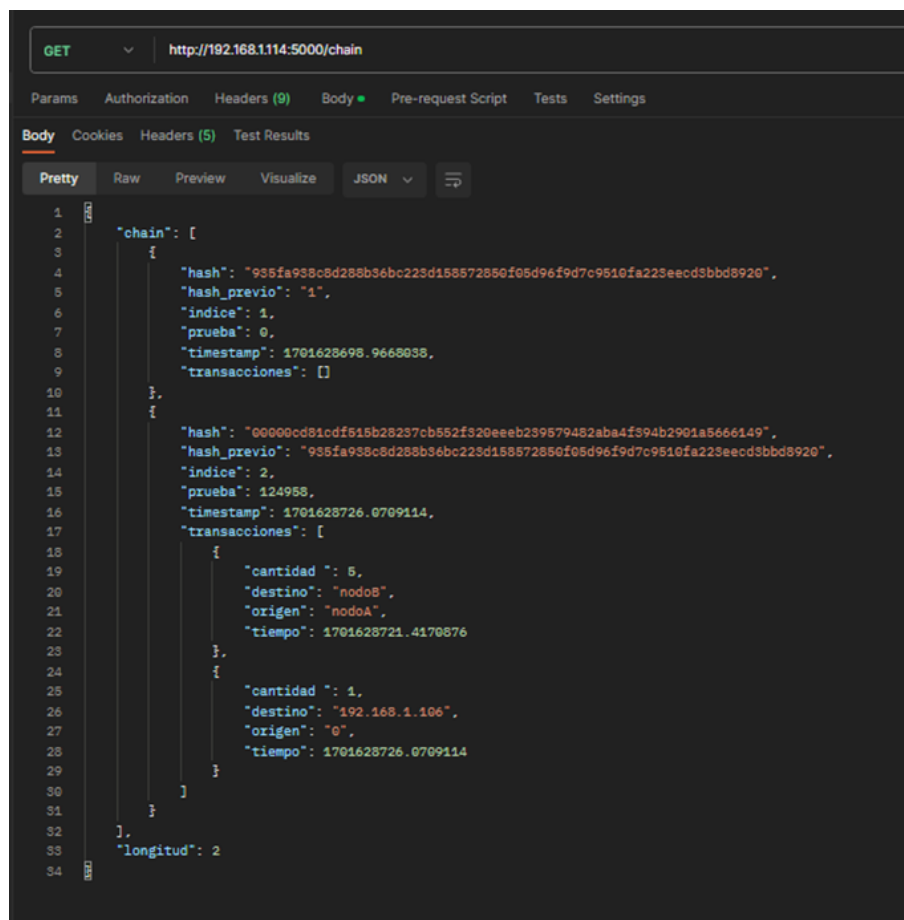


Figura 35: Comprobación de la cadena

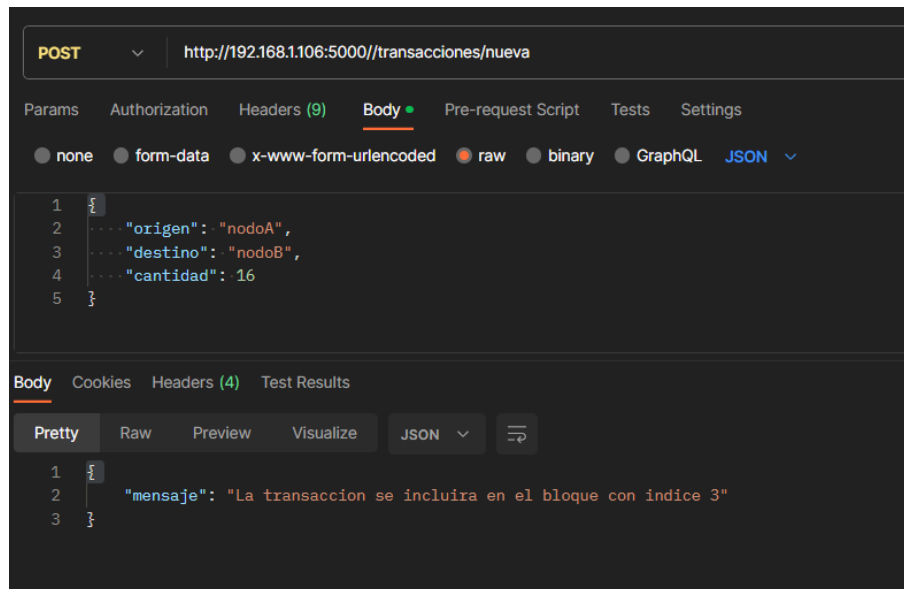


Figura 36: Nueva transacción en Windows

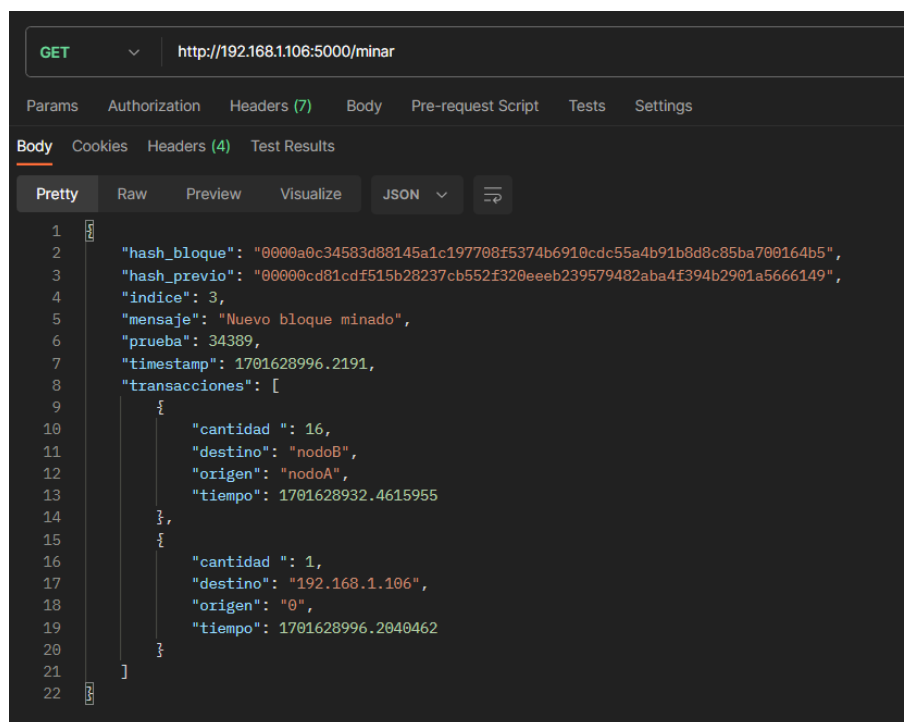


Figura 37: Minado de un nuevo bloque



```
1 2
2 3
3 4
4 5
5 6
6 7
7 8
8 9
9 10
10 11
11 12
12 13
13 14
14 15
15 16
16 17
17 18
18 19
19 20
20 21
21 22
22 23
23 24
24 25
25 26
26 27
27 28
28 29
29 30
30 31
31 32
32 33
33 34
34 35
35 36
36 37
37 38
38 39
39 40
40 41
41 42
42 43
43 44
44 45
45 46
46 47
47 48
48 49
49 50
50 51
51 52
52 53
53 54
54 55
```

```
{
  "chain": [
    {
      "hash": "935fa938c8d288b36bc223d158572850f85d96f9d7c9518fa223eecd3bbd8929",
      "hash_previo": "1",
      "indice": 1,
      "previo": 0,
      "timestamp": 1701628698.9668938,
      "transacciones": []
    },
    {
      "hash": "00000cd81cdf515b28237cb552f320eeeb239579482aba4f394b2901a5666149",
      "hash_previo": "935fa938c8d288b36bc223d158572850f85d96f9d7c9518fa223eecd3bbd8929",
      "indice": 2,
      "previo": 124958,
      "timestamp": 1701628726.0709114,
      "transacciones": [
        {
          "cantidad": 5,
          "destino": "nodoB",
          "origen": "nodoA",
          "tiempo": 1701628721.4174876
        },
        {
          "cantidad": 1,
          "destino": "192.168.1.106",
          "origen": "0",
          "tiempo": 1701628726.0709114
        }
      ]
    },
    {
      "hash": "0000a0c34583d88145a1c197788f5374b6918cdc55a4b91b648c85ba708164b5",
      "hash_previo": "00000cd81cdf515b28237cb552f320eeeb239579482aba4f394b2901a5666149",
      "indice": 3,
      "previo": 34389,
      "timestamp": 1701628996.2191,
      "transacciones": [
        {
          "cantidad": 16,
          "destino": "nodoB",
          "origen": "nodoA",
          "tiempo": 1701628932.4615955
        },
        {
          "cantidad": 1,
          "destino": "192.168.1.106",
          "origen": "0",
          "tiempo": 1701628996.2040462
        }
      ]
    }
  ],
  "longitud": 3
}
```

Figura 38: Comprobación de la cadena

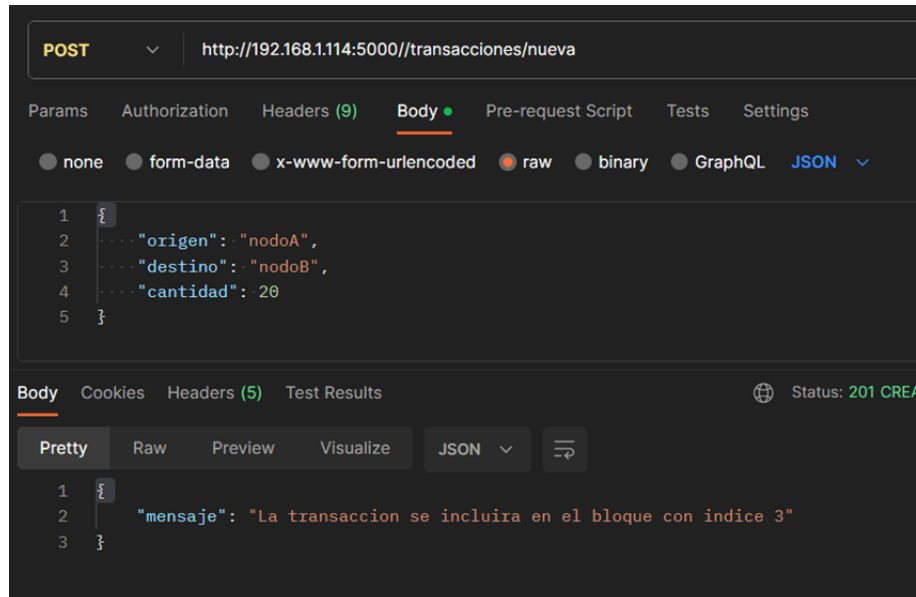


Figura 39: Nueva transacción en Linux

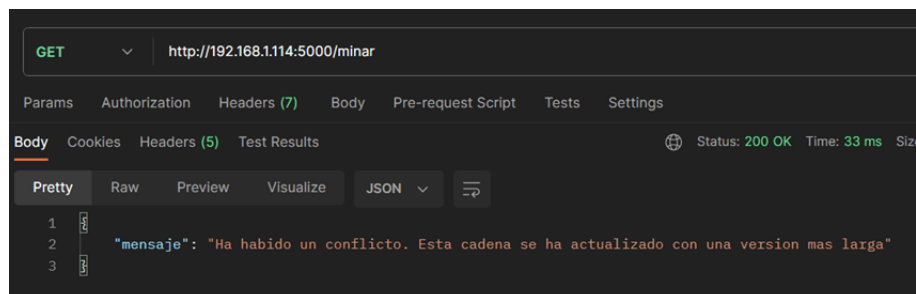


Figura 40: Conflicto al minar la nueva transacción

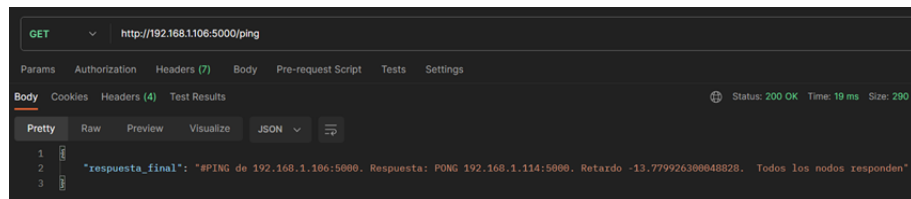


Figura 41: Método de PING/PONG

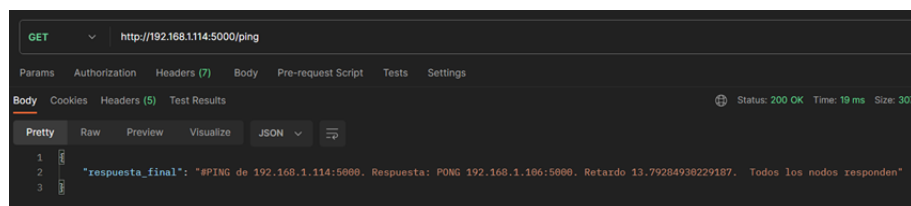


Figura 42: Método de PING/PONG

El retardo desde Windows hasta Linux es de -13.77 segundos y el retardo de Linux a Windows es de 13.79. Eso se debe a que en la máquina virtual (Linux) la hora está retrasada unos 13 segundos respecto a la hora de Windows.

No sabemos a qué se debe este desfase.

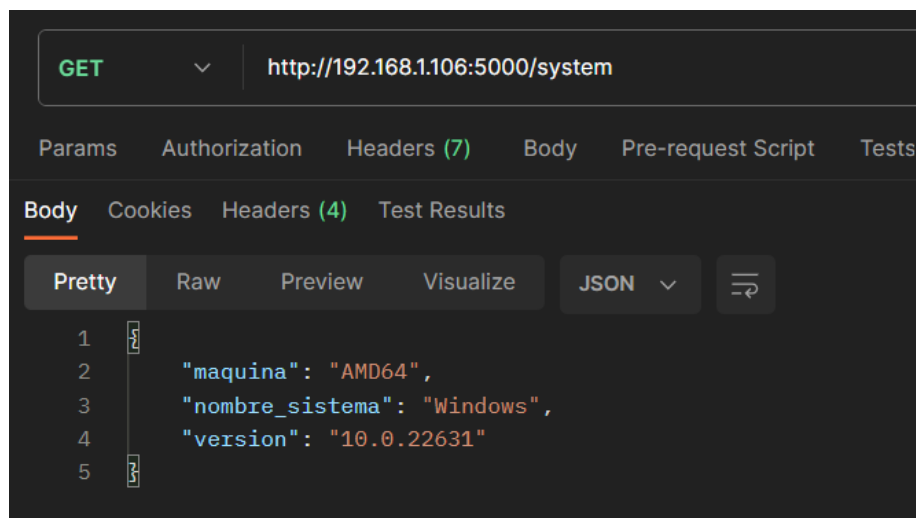


Figura 43: Información del sistema operativo, versión y tipo de procesador (Windows)

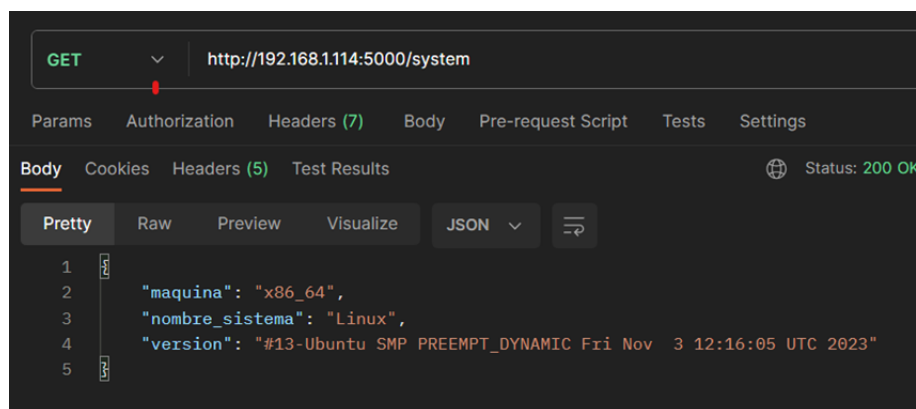


Figura 44: Información del sistema operativo, versión y tipo de procesador (Linux)

5. Bibliografía

Para realizar este proyecto se han consultado los apuntes de Fundamentos de los Sistemas Operativos realizados por los profesores de esta asignatura de la Universidad Pontificia de Comillas.

En cuanto a las explicaciones teóricas de esta memoria se han seguido de forma gradual los contenidos disponibles en el enunciado de este Proyecto Final.