DICIEMBRE 2023

PROYECTO BLOCKCHAIN

PROYECTO FINAL: FUNDAMENTOS DE LOS SISTEMAS OPERATIVOS

ULISES DIEZ SANTAOLALLA
IGNACIO FELICES VERA
2º iMAT A

Tabla de contenido

Introducción Proyecto BlockChain

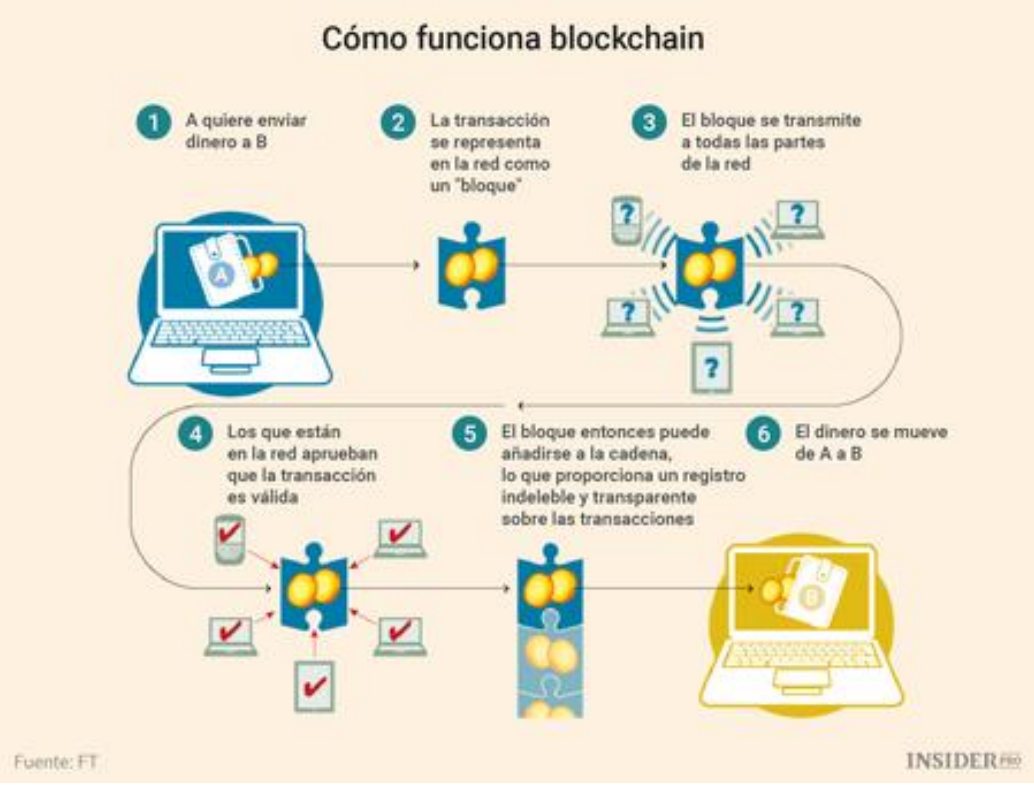
En este proyecto, nos embarcaremos en el desarrollo de una aplicación Blockchain. La idea principal es que múltiples nodos, que son procesos vinculados a un ordenador y un puerto específico, se encarguen de agrupar transacciones en bloques y sincronizarse entre sí. El plan es construir esta aplicación de manera progresiva: comenzaremos con un solo nodo, un proceso inicial, y luego permitiremos que otros nodos se integren a la red.

La estrategia de este proyecto es seguir un enfoque incremental. En primer lugar, nos enfocaremos en el desarrollo del backend, la parte interna y lógica de la aplicación. Aquí es donde se programará el código esencial para la creación de bloques, la estructura de la cadena de bloques (Blockchain) y la gestión de las transacciones monetarias asociadas a cada bloque. Un punto crucial será la definición del mecanismo para calcular los hashes asociados a cada bloque, asegurando la construcción precisa de cadenas de bloques.

Para proteger nuestra cadena de bloques de suplantaciones por otros nodos de la red, diseñaremos un mecanismo de seguridad específico. Luego, aprovecharemos una librería para implementar un servicio web que esté disponible mediante peticiones GET y POST del protocolo HTTP. Este servicio web será desplegado en todos los nodos de nuestra red, permitiéndonos establecer una red sincronizada de nodos que conservará toda la información relacionada con los bloques y las transacciones.

En resumen, nuestro objetivo es construir de manera progresiva y sólida esta aplicación Blockchain, comenzando desde la lógica interna hasta la implementación de servicios web en cada nodo, creando así una red sólida y sincronizada para gestionar la información de manera eficiente.

A continuación, se irá explicando de manera detallada y con imágenes cómo se ha ido desarrollando el código, su propósito, y cómo se ha ido comprobando su correcto funcionamiento.



Desarrollo del módulo “BlockChain.py”

Esta parte del programa contiene los elementos básicos de la Blockchain. Estos son la creación de bloques que formarán parte de la cadena, la inicialización de la cadena, y los diferentes métodos necesarios para incorporar nuevos bloques, crear nuevas transacciones o comprobar la correcta creación de los bloques, entre otras cosas.

En primer lugar, se importaron las librerías necesarias para el desarrollo del código:

- from typing import Tuple, List, Dict
- import json
- import hashlib
- import time

Clase Bloque:

Método __init__:

Este método permite inicializar los bloques que formarán parte de la cadena. Recibe varios parámetros: índice, transacciones, timestamp, hash_previo y prueba, la cual se inicializa en 0. Su objetivo es inicializar el bloque con la información proporcionada al momento de su creación, como el índice único del bloque, las transacciones incluidas, la marca de tiempo de su creación, el hash del bloque anterior en la cadena y la prueba de trabajo si se proporciona.

```python
def __init__(self, indice: int, transacciones: List[dict], timestamp: int, hash_previo: str, prueba: int =0):
    """
    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
    """
    # Codigo a completar (inicializacion de los elementos del bloque)
    self.indice = indice
    self.transacciones = transacciones
    self.timestamp = timestamp
    self.hash_previo = hash_previo
    self.prueba = prueba
    self.hash = None  # Inicialmente establecemos el hash como None
```

Método calcular_hash:

Este método calcula el hash del bloque utilizando el algoritmo de hashing SHA-256, devuelve el hash resultante como una cadena hexadecimal. Más adelante este hash se necesitará cambiar, ya que uno de los requisitos del programa es que los bloques se inicialicen con tantos ceros como el nivel de dificultad, pero eso se tendrá en cuenta más adelante.

```python
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()
```

Método toDict:

Este método convierte el objeto Bloque a un diccionario en Python. Su propósito es facilitar el manejo y almacenamiento del bloque convirtiéndolo en una estructura de datos más manipulable, cosa que será muy útil en el apartado “BlockChain_app.py”.

```python
def toDict(self):
    """
    Método que convierte un objeto Bloque a un diccionario.
    """
    bloque_dicc =  {
        "indice": self.indice,
        "transacciones": self.transacciones,
        "timestamp": self.timestamp,
        "hash_previo": self.hash_previo,
        "prueba": self.prueba,
        "hash": self.hash
    }

    return bloque_dicc
```

Clase Blockchain:

Método __init__:

Este método inicializa la cadena de bloques y sus atributos. Establece la dificultad inicial para la prueba de trabajo, que determinará por cuántos ceros tiene que empezar el hash de los bloques, crea una lista vacía para las transacciones y una lista que contendrá los bloques de la cadena. Además, llama al método primer_bloque() para generar el primer bloque en la cadena.

```
def __init__(self):
    """
    Inizializo la BlockChain con una lista de bloques vacía,
    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)
    """
    self.dificultad = 4
    self.lista_transacciones = []
    self.lista_bloques = []
    self.primer_bloque()  # Creo el primer bloque
```

Método primer_bloque:

Este método crea el primer bloque de la cadena. Establece el índice inicial con un índice de 1, sin transacciones (una lista vacía) y un hash previo, siendo este el punto de partida para la cadena de bloques.

```
def primer_bloque(self):
    """Función encargada de crear el primer bloque de la cadena
    para que luego puedan añadirse más bloques a dicha cadena.
    Returns:
        object: primer bloque inicializado con índice 1
    """
    # Crear el primer bloque
    bloque_inicio = Bloque(1, [], "0", "1")  # Índice 1, sin transacciones y hash_previo de 1
    bloque_inicio.hash = bloque_inicio.calcular_hash()   # Genero hash
    self.lista_bloques.append(bloque_inicio)    # Añado el bloque a la cadena
```

Método nuevo_bloque:

Este método crea nuevos bloques en la cadena a partir de las transacciones pendientes. Recibe el hash del bloque anterior y genera un nuevo bloque con un índice que es uno más que el número de bloques actuales. La prueba de trabajo se realiza para encontrar un hash válido que cumpla con la dificultad establecida antes de agregarlo a la cadena de bloques.

```
def nuevo_bloque(self, hash_previo: str) ->Bloque:
    """
    Crea un nuevo bloque a partir de las transacciones que no estan
    confirmadas
    :param hash_previo: el hash del bloque anterior de la cadena
    :return: el nuevo bloque
    """
    nuevo_indice = len(self.lista_bloques) + 1   # El índice será el uno más del número de bloques ya existentes
    nuevo_bloque = Bloque(nuevo_indice, self.lista_transacciones, time.time(), hash_previo)   # Genero el bloque
    nuevo_bloque.hash = self.prueba_trabajo(nuevo_bloque)    # Genero hashes hasta que empiece por 0000
    return nuevo_bloque
```

Método nueva_transaccion:

Este método sirve para crear una nueva transacción agregarla a la lista de transacciones pendientes, con el origen de la transacción, el destino, la cantidad y el tiempo actual como parámetros. Luego, añade esta transacción a la lista de transacciones pendientes y devuelve el índice del bloque que la almacenará.

```
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 candidad
    :return: <int> el indice del bloque que va a almacenar la transaccion
    """
    tiempo_actual = time.time()     # Tomo el tiempo en el momento que se crea la transacción
    # Creo la transacción con sus correspondientes valores
    nueva_transaccion = {
        'origen': origen,
        'destino': destino,
        'cantidad': cantidad,
        'tiempo': tiempo_actual
    }
    self.lista_transacciones.append(nueva_transaccion)      # Añado la transacción a la lista de transacciones
    indice_bloque = len(self.lista_bloques) + 1  # Devuelve el índice del bloque que almacenará la transacción
    return indice_bloque
```

Método prueba_trabajo:

Este método sirve para comprobar que el hash de los bloques comience por tantos ceros como el nivel de dificultad lo requiera. Para esto, genera el hash por defecto del bloque, y va aumentando el número del atributo “prueba” del bloque hasta que el hash de este cumpla con el requisito de empezar por los ceros.

```
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)
    """
    hash_bloque = bloque.calcular_hash()    # Calculo el hash inicial del bloque
    # Compruebo si tiene tantos 0 como dificultad, de lo contrario aumento el campo prueba
    while not hash_bloque.startswith('0' * self.dificultad):
        bloque.prueba += 1
        hash_bloque = bloque.calcular_hash()
    return hash_bloque
```

Método prueba_valida:

Este método comprueba si el hash del bloque comienza con tantos ceros como la dificultad de la blockchain. Además de esto, comprueba si el hash del bloque coincide con el valor que devuelve el método calcular_hash si lo vuelvo a ejecutar.

```
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 comprobara que hash_bloque coincide con el valor devuelvo del
    metodo de calcular hash del
    bloque.
    Si cualquiera de ambas comprobaciones es falsa, devolvera falso y en caso
    contrario, verdarero
    :param bloque:
    :param hash_bloque:
    :return:
    """
    empieza_ceros = hash_bloque.startswith('0' * self.dificultad)
    # Compruebo si empieza por los ceros correspondientes a la dificultad
    if empieza_ceros:
        # Compruebo si los hashes coinciden
        if hash_bloque == bloque.hash:
            return True
        # Si los hashes no coinciden
        else:
            return False
    # Si no comienza por los ceros correspondientes
    else:
        return False
```

Método integra_bloque:

Esta función se encarga de integrar un nuevo bloque en la cadena de bloques si pasa las pruebas de validez. Comprueba si el hash del bloque nuevo es válido y si el hash previo del bloque nuevo coincide con el hash del último bloque de la cadena. Si pasa estas comprobaciones, actualiza el hash del bloque, lo añade a la cadena y reinicia la lista de transacciones pendientes.

```python
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
    7
    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)
    """
    ultimo_bloque = self.lista_bloques[-1]
    # Compruebo si el bloque coincide con su hash
    if self.prueba_valida(bloque_nuevo, hash_prueba):
        # Compruebo que el hash_previo del bloque corresponda con el hash del último bloque de la lista
        if bloque_nuevo.hash_previo == ultimo_bloque.hash:
            # Actualizo el hash del bloque
            bloque_nuevo.hash = hash_prueba
            # Añado el bloque a la cadena
            self.lista_bloques.append(bloque_nuevo)
            # Reseteo la lista de transacciones
            self.lista_transacciones = []
            # Indico que se ha podido ejecutar correctamente
            return True
        # Si no coincide los hashes del bloque previo
        else:
            return False
    # Si no coincide el hash del bloque
    return False
```

En resumen, las funciones en la clase Blockchain se encargan de manejar toda la cadena de bloques. Esto implica crear nuevos bloques, agregar nuevas transacciones y asegurarse de que los bloques sean válidos antes de unirse a la cadena.

Por otro lado, la clase Bloque se concentra en cómo se representan y se calculan los códigos especiales (llamados hashes) para cada uno de los bloques que conforman la cadena.

Desarrollo del módulo “BlockChain_app.py”

Esta parte del programa contiene los elementos de la aplicación Blockchain. Se encarga de manejar las diferentes peticiones que puede recibir la cadena desde diferentes nodos, como puede ser la gestión de crear nuevas transacciones, minar bloques, consultar el estado de la cadena, incorporar nuevos nodos a la red, o realizar de manera continua copias de seguridad de la cadena, entre otras funciones.

En primer lugar, se importaron las librerías necesarias para el desarrollo del código:

- import BlockChain
- from uuid import uuid4
- import socket
- from flask import Flask, jsonify, request
- from argparse import ArgumentParser
- import threading
- import json
- import datetime
- import platform
- import requests

Al igual que esto, se comienza instanciando un nodo Flask, una cadena blockchain, un set de nodos de red, y se guarda en una variable la ip del usuario, variables que serán imprescindibles para el correcto funcionamiento de la app.

Funciones y sus objetivos

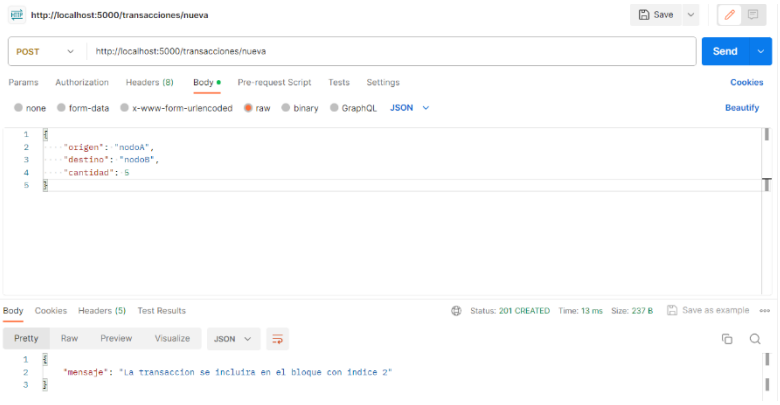
nueva_transaccion()

Objetivo: Este método maneja la creación de nuevas transacciones.

Funcionamiento: Recibe una solicitud POST con datos JSON que representan una transacción. A continuación, verifica si la solicitud contiene los campos requeridos ('origen', 'destino', 'cantidad'). Si la validación es exitosa, crea una nueva transacción y la agrega a la lista de transacciones pendientes en la cadena de bloques. Finalmente devuelve un mensaje con el índice del bloque en el que se incluirá la transacción.

```python
# Método para recibir transacciones
@app.route('/transacciones/nueva', methods=['POST'])
def nueva_transaccion():
    # Obtengo el json enviado con el POST
    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 incluira en el bloque con indice {indice}'}

    return jsonify(response), 201  # función de Flask que convierte objetos de Python en respuestas JSON
```

A continuación, se puede ver un ejemplo del funcionamiento de este método:



blockchain_completa()

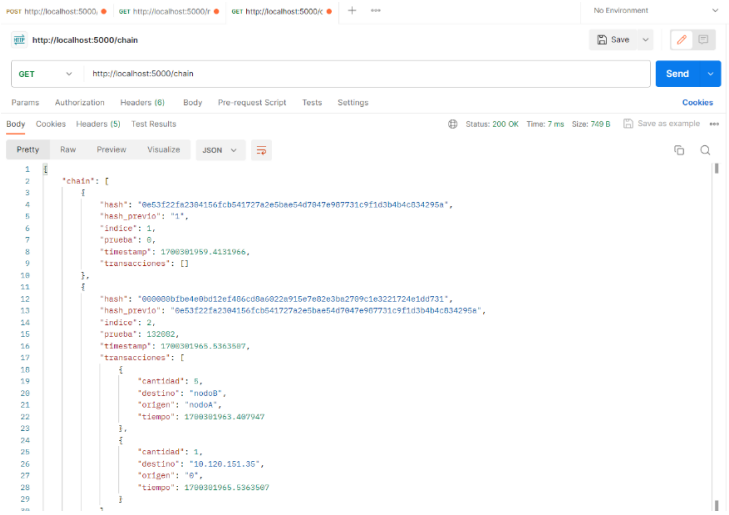
Objetivo: Este método proporciona la cadena de bloques completa.

Funcionamiento: Devuelve la cadena de bloques completa y su longitud en formato JSON.

```python
# Método para consultar la cadena completa
@app.route('/chain', methods=['GET'])
def blockchain_completa():
    response ={
        # Solamente permitimos la c                    s que tienen hash
        'chain': [b.toDict() for b                    .hash is not None],
        'longitud': len(blockchain.lista_bloques),
    }

    return jsonify(response), 200
```

(variable) lista_bloques: list

A continuación, se puede ver un ejemplo del funcionamiento de este método tras el minado de un bloque:



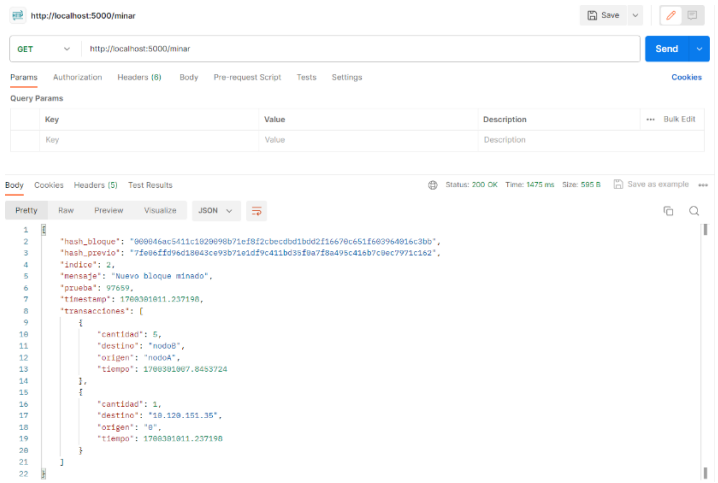
minar()

Objetivo: Maneja el proceso de minería de un nuevo bloque.

Funcionamiento: Verifica si existen transacciones pendientes para crear un nuevo bloque. A continuación, comprueba si hay conflictos de cadena y resuelve los conflictos si los hay. Crea una transacción de recompensa para el minero, con una recompensa de 1 en este caso, y genera un nuevo bloque, integrándolo en la cadena de bloques. Finalmente, devuelve detalles del bloque recién minado.

```python
@app.route('/minar', methods=['GET'])
def minar():
    if len(blockchain.lista_transacciones) == 0:
        response = {'mensaje': "No es posible crear un nuevo bloque. No hay transacciones"}
        return jsonify(response), 400
    else:
        # Aplico la resolución de conflictos antes de minar el bloque
        hay_conflicto = resuelve_conflictos()

        if hay_conflicto:
            response = {
                'mensaje': "Ha habido un conflicto. Esta cadena se ha actualizado con una versión más larga"
            }
            return jsonify(response), 409

        # Creo una transacción de recompensa para el minero
        recompensa = 1     # La recompensa es 1
        origen = "0"  # Origen de la recompensa
        destino = mi_ip  # Uso la IP

        # Creo la transacción de recompensa
        blockchain.nueva_transaccion(origen, destino, recompensa)

        # Mino el bloque
        anterior_bloque = blockchain.lista_bloques[-1]
        nuevo_bloque = blockchain.nuevo_bloque(anterior_bloque.hash)
        blockchain.integra_bloque(nuevo_bloque, nuevo_bloque.hash)

        # Mensaje de respuesta
        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
        }

    return jsonify(response), 200
```

A continuación, se puede ver un ejemplo del resultado tras el minado de un bloque:



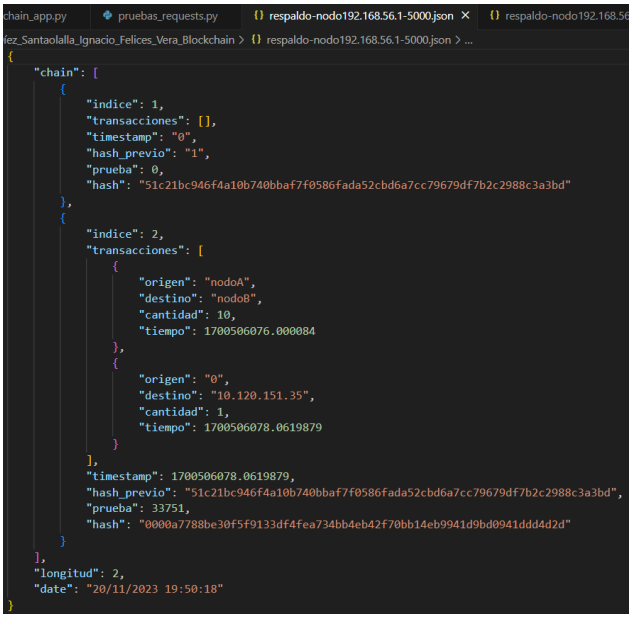
realizar_respaldo()

Objetivo: Realiza copias de seguridad periódicas de la cadena de bloques.

Funcionamiento: Cuando se inicializa un nodo se inicia un hilo que genera copias de seguridad de este nodo cada 60 segundos, bloqueando el acceso a otros hilos a la blockchain, y crea un respaldo de la cadena de bloques en un archivo JSON, incluyendo la longitud de la cadena y la fecha actual.

```python
# Método para realizar copia de seguridad
mutex = threading.Semaphore(1)      # Semáforo para controlar el acceso único a la blockchain, protegiéndola
@app.route('/respaldo', methods=['GET'])
def realizar_respaldo():
    while True:
        # Bloqueo el acceso a otros hilos
        mutex.acquire()
        try:
            # Obtengo la fecha y hora y la formateo
            fecha_actual = datetime.datetime.now().strftime('%d/%m/%Y %H:%M:%S')

            # Creo el respaldo de los datos
            respaldo_data = {
                "chain": [bloque.toDict() for bloque in blockchain.lista_bloques],
                "longitud": len(blockchain.lista_bloques),
                "date": fecha_actual
            }

            # Escribo el respaldo en un archivo JSON
            nombre_archivo = f"respaldo-nodo{socket.gethostbyname(socket.gethostname())}-{puerto}.json"
            with open(nombre_archivo, 'w') as file:
                json.dump(respaldo_data, file, indent=4)

        # Libero el mutex para permitir el acceso a otros hilos
        finally:
            mutex.release()

        # Espero 60 segundos antes de realizar el próximo respaldo
        threading.Event().wait(60)
```

A continuación se puede ver un ejemplo de un respaldo que ha ejecutado el programa durante la ejecución de un nodo:



obtener_detalles_nodo()

Objetivo: Devuelve detalles del sistema en el que se ejecuta el nodo.

Funcionamiento: Obtiene información del tipo de procesador, nombre y versión del sistema operativo en el que está corriendo el nodo y los devuelve en formato JSON.

```python
# Método para obtener detalles del nodo
@app.route('/system', methods=['GET'])
def obtener_detalles_nodo():
    sistema_operativo = platform.system()  # Nombre del sistema operativo
    version = platform.version()  # Versión del sistema operativo
    tipo_procesador = platform.machine()  # Tipo de procesador

    # Formateo la respuesta en formato JSON
    detalles_nodo = {
        "maquina": tipo_procesador,
        "nombre_sistema": sistema_operativo,
        "version": version
    }

    return jsonify(detalles_nodo), 200
```

registrar_nodos_completo()

Objetivo: Registra nodos nuevos en la red y actualiza su información con la cadena de bloques actual.

Funcionamiento: Recibe una solicitud POST con la lista de direcciones IP de nodos nuevos. Después de esto, actualiza la lista de nodos en la red, añadiendo la propia ruta del nodo actual. Una vez hecho esto, y realizar una copia de la blockchain del nodo actual, para cada nodo que hay en la red, menos a él mismo, hace la llamada registro_simple, donde envía un json con la lista de nodos y sus direcciones, y la copia de la blockchain que se ha realizado. Si la función registro_simple se lleva a cabo de manera exitosa, se devuelve un mensaje de que los nodos se han añadido a la red de manera exitosa.

```python
@app.route('/nodos/registrar', methods=['POST'])
def registrar_nodos_completo():
    # Obtengo el json enviado con el POST
    values = request.get_json()

    # Variables Globales
    global blockchain
    global nodos_red

    # Obtengo los nodos del json
    nodos_nuevos = values.get('direccion_nodos')

    # Compruebo si se han registrado nodos o no
    if nodos_nuevos is None:
        return "Error: No se ha proporcionado una lista de nodos", 400

    # Obtengo la dirección desde la que se hace el POST (direccion del nodo actual)
    nodo_actual = request.url_root.rstrip('/')

    # Obtengo todas las direcciones de los nodos en la red (incluyendo el nodo principal y el nodo actual)
    nodos_todos = list(nodos_red.union(set(nodos_nuevos)))
    nodos_todos.append(nodo_actual)

    # Creo una copia de la blockchain actual en formato JSON
    blockchain_json = [bloque.toDict() for bloque in blockchain.lista_bloques]

    # Envío la información de la red y la copia del blockchain a cada nodo de la lista
    for nodo in nodos_nuevos:
        # Excluyo al propio nodo
        nodos_sin_propio = [n for n in nodos_todos if n != nodo]
        data_to_send = {
            "nodos_direcciones": nodos_sin_propio,
            "blockchain": blockchain_json
        }
        response = requests.post(
            nodo + "/nodos/registro_simple",
            json=data_to_send,
            headers={'Content-Type': "application/json"}
        )
        # Si un nodo no actualiza correctamente la blockchain
        if response.status_code != 200:
            return f"Error al enviar el blockchain al nodo {nodo}", 500

    # Muestro el registro de los nodos
    response = {
        'mensaje': 'Se han incluido nuevos nodos en la red',
        'nodos_totales': list(nodos_red.union(set(nodos_nuevos)))
    }
    for nodo in nodos_nuevos:
        nodos_red.add(nodo)

    return jsonify(response), 201
```

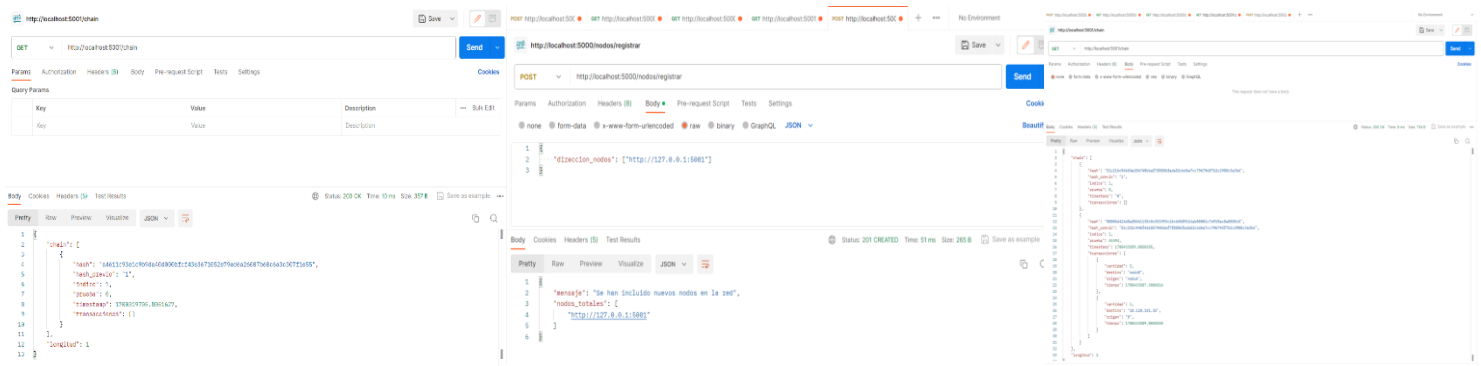
registrar_nodo_actualiza_blockchain()

Objetivo: Actualiza la cadena de bloques del nodo actual con la información recibida de otros nodos.

Funcionamiento: Recibe una solicitud POST con datos JSON que incluyen las direcciones de los nodos y la información de la cadena de bloques. Con estos datos del json, reconstruye la blockchain que contiene, la cual pertenecía al nodo que hizo la llamada de registro_simple. Una vez reconstruida la cadena y visto que los datos de la blockchain están bien construidos, es decir, pasa exitosamente la función integra_bloque, devuelve esta cadena. De esta manera, tras reconstruir la cadena de bloques, la actualiza con los nuevos bloques recibidos. Si da error en la función integra_bloque, devuelve error al integrar el bloque en la blockchain, y restaura la blockchain que tenía el nodo antes de intentar reemplazarla por la nueva.

```python
@app.route('/nodos/registro_simple', methods=['POST'])
def registrar_nodo_actualiza_blockchain():
    # Variables Globales
    global blockchain

    # Obtengo el json enviado con el POST
    read_json = request.get_json()

    # Obtengo el las direciones de los nodos y la blockchain del json
    direcion_nodos = read_json.get("nodos_direcciones")
    blockchain_json = read_json.get("blockchain")

    # Compruebo si se ha recibido bien la información de los nodos y la blockchain
    if direcion_nodos is None or blockchain_json is None:
        return "Datos insuficientes para actualizar el blockchain", 400

    # Actualizo la lista de nodos con la recibida
    nodos_red.update(direcion_nodos)

    # Guardo la blockchain actual del nodo, por si el registro sale mal y no se puede sincronizar
    # con el nodo principal
    lista_actual_blockchain = blockchain.lista_bloques

    # Construyo el blockchain a partir del JSON recibido, ignorando el primer bloque
    blockchain_leida = []
    for bloque_json in blockchain_json[1:]:
        nuevo_bloque = BlockChain.Bloque(
            indice=bloque_json['indice'],
            transacciones=bloque_json['transacciones'],
            timestamp=bloque_json['timestamp'],
            hash_previo=bloque_json['hash_previo'],
        )
        nuevo_bloque.prueba=bloque_json['prueba']
        nuevo_bloque.hash=bloque_json['hash']
        blockchain_leida.append(nuevo_bloque)

    # Reinicio la BlockChain del nodo
    blockchain.lista_bloques = [blockchain.lista_bloques[0]]

    # Verifico que los bloques de la blockchain pasan las pruebas de integra_bloque
    for bloque in blockchain_leida:
        # Si no superan las pruebas de integra bloque, devuelve error, y el nodo vuelve
        # a tener la blockchain que tenía antes de intentar registrarse
        if not blockchain.integra_bloque(bloque, bloque.hash):
            blockchain.lista_bloques = lista_actual_blockchain
            return "Error al integrar el bloque en el blockchain", 500

    # Devuelvo la blockchain actualizada
    response = {
        'chain': [bloque.toDict() for bloque in blockchain.lista_bloques if bloque.hash is not None],
        'longitud': len(blockchain.lista_bloques)
    }

    return jsonify(response), 200
```

A continuación, se puede ver un ejemplo de como si varía la cadena de un nodo después de registrarlo en la red:



resuelve_conflictos()

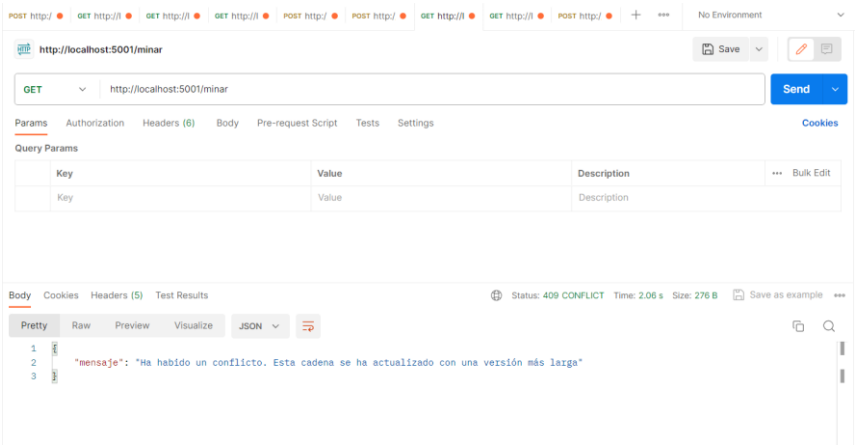
Objetivo: Esta función se encarga de resolver conflictos entre las cadenas de bloques de diferentes nodos en la red.

Funcionamiento: Itera sobre todos los nodos de la red (excluyendo el propio nodo), solicitando la cadena de bloques a cada nodo y compara la longitud de su cadena con la del nodo actual. Si encuentra una cadena más larga que la actual, actualiza la cadena local con la más larga, reconstruyendo los bloques de la blockchain más larga que ha encontrado, y actualizando la blockchain del nodo actual. Devuelve True si se actualizó la cadena local con una más larga, indicando que hubo un conflicto y se resolvió, de lo contrario, devuelve False.

```python
def resuelve_conflictos():
    # Variables Globales
    global blockchain

    # Obtengo los datos del nodo y blockchain actual
    longitud_actual = len(blockchain.lista_bloques)
    cadena_más_larga = None
    nodo_actual = request.url_root.rstrip('/')
    # Compruebo para cada nodo de la red
    for nodo in nodos_red:
        # No incluyo al propio nodo
        if nodo != nodo_actual:
            # Obtengo la cadena de cada nodo
            response = requests.get(f"{nodo}/chain")
            if response.status_code == 200:
                # Obtengo la longitud de la cadena y la blockchain del nodo
                longitud_nodo = response.json().get('longitud')
                cadena_nodo = response.json().get('chain')

                # Si se ha encontrado una cadena más larga
                if longitud_nodo > longitud_actual:
                    longitud_actual = longitud_nodo
                    cadena_más_larga = cadena_nodo

    # Si he encontrado una cadena más larga, la remplazo y devuelvo True
    if cadena_más_larga:
        # Guardo la blockchain actual del nodo, por si el registro sale mal y no se puede sincronizar
        # con el nodo principal
        lista_actual_blockchain = blockchain.lista_bloques

        # Construyo el blockchain a partir del JSON recibido, ignorando el primer bloque
        blockchain_leida = []
        for bloque_json in cadena_más_larga[1:]:
            nuevo_bloque = BlockChain.Bloque(
                indice=bloque_json['índice'],
                transacciones=bloque_json['transacciones'],
                timestamp=bloque_json['timestamp'],
                hash_previo=bloque_json['hash_previo'],
            )
            nuevo_bloque.prueba=bloque_json['prueba']
            nuevo_bloque.hash=bloque_json['hash']
            blockchain_leida.append(nuevo_bloque)

        # Reinicio la BlockChain del nodo
        blockchain.lista_bloques = [blockchain.lista_bloques[0]]

        # Verifico que los bloques de la blockchain pasan las pruebas de integra_bloque
        for bloque in blockchain_leida:
            # Si no superan las pruebas de integra bloque, devuelve error, y el nodo vuelve
            # a tener la blockchain que tenía antes de intentar registrarse
            if not blockchain.integra_bloque(bloque, bloque.hash):
                blockchain.lista_bloques = lista_actual_blockchain
                return "Error al integrar el bloque en el blockchain", 500

        return True

    # Si no ha habido conflicto la comparar las cadenas
    else:
        return False
```

La función resuelve_conflictos() es esencial en la lógica de la red de la cadena de bloques, ya que asegura que todos los nodos tengan una igual y actualizada de la cadena principal, evitando bifurcaciones y asegurando la integridad de la red descentralizada.

A continuación, se puede ver un ejemplo de un conflicto entre nodos al tener distinta longitud las blockchains de los nodos:



ping()

Objetivo: Realiza una llamada PING a otros nodos y espera respuestas PONG.

Funcionamiento: Envía mensajes PING a otros nodos de la red y espera sus respuestas PONG. En este mensaje incluye la dirección del nodo actual, el mensaje de PING, y el momento en el que se envía el mensaje. Recopila las respuestas recibidas y muestra los mensajes PONG con información sobre el retardo, si las respuestas se han llevado a cabo de manera exitosa.

```python
# Método para hacer una llamada PING a los nodos y comprobar que recibe las respuestas PONG
@app.route('/ping', methods=['GET'])
def ping():
    # Obtengo la dirección del nodo actual
    nodo_actual = request.url_root.rstrip('/')

    # Creo el mensaje que voy a enviar a los nodos de la red
    mensaje_ping = {
        "host_origen": nodo_actual,
        "mensaje": "PING",
        "timestamp": datetime.datetime.now().isoformat()
    }

    # Lista para almacenar las respuestas de los nodos
    respuestas = []

    # Envío el mensaje de PING a cada nodo en la red
    for nodo in nodos_red:
        # No incluyo al propio nodo
        if nodo != nodo_actual:
            # Obtengo la respuesta PONG de los nodos a los que envío el PING
            response = requests.post(f"{nodo}/pong", json=mensaje_ping)

            #Si me han enviado bien la respuesta, la añado a la lista de respuestas
            if response.status_code == 200:
                respuesta_nodo = response.json()
                respuestas.append(respuesta_nodo)

    # Mensaje final una vez he recibido las respuestas PONG  de los nodos
    respuesta_final = f"#PING de {nodo_actual}. Respuestas: "
    if respuestas:
        # Por cada respuesta, muestro el mensaje PONG uq em ha enviado
        for respuesta in respuestas:
            respuesta_final += f"PONG {respuesta['host']} Retardo: {respuesta['delay']}"
        respuesta_final += "#Todos los nodos responden"

    # Si no se he recibido respuestas de los nodos
    else:
        respuesta_final += "No hay respuestas de los nodos"

    return jsonify({"respuesta_final": respuesta_final}), 200
```

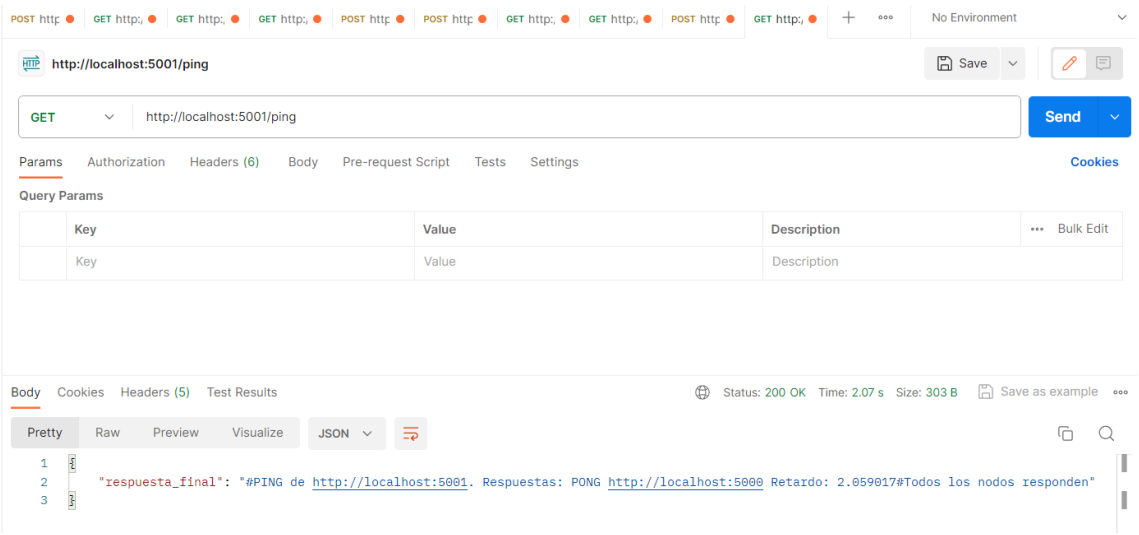
pong()

Objetivo: Responde a una solicitud PING de otro nodo con un mensaje PONG.

Funcionamiento: Recibe una solicitud POST con datos JSON que representan un mensaje PING y responde al nodo que realizó la solicitud con un mensaje PONG, indicando el retardo entre el envío y la recepción del mensaje.

```python
# Método que devuelve el mensaje PONG cuando recibe una llamada PING de un nodo
@app.route('/pong', methods=['POST'])
def pong():
    # Recibo el mensaje PING del nodo que ha hecho la llamada
    mensaje_ping = request.get_json()

    # Obtengo los datos del json del mensaje PING
    host_origen = mensaje_ping.get("host_origen")
    mensaje = mensaje_ping.get("mensaje")
    timestamp = mensaje_ping.get("timestamp")

    # Obtengo la dirección del nodo actual
    nodo_actual = request.url_root.rstrip('/')

    # Resto los timestamps para ver el retardo
    retardo = datetime.datetime.now() - datetime.datetime.fromisoformat(timestamp)

    # Creo el mensaje PONG que voy a enviar al que hizo la llamada PING
    respuesta_pong = {
        "host_origen": host_origen,
        "host": nodo_actual,
        "mensaje": mensaje,
        "delay": retardo.total_seconds()
    }

    return jsonify(respuesta_pong), 200
```

A continuación, se puede observar un ejemplo de una llamada PING desde un nodo en el puerto 5001:

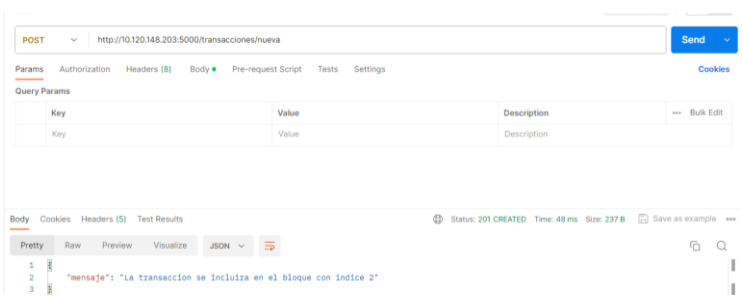


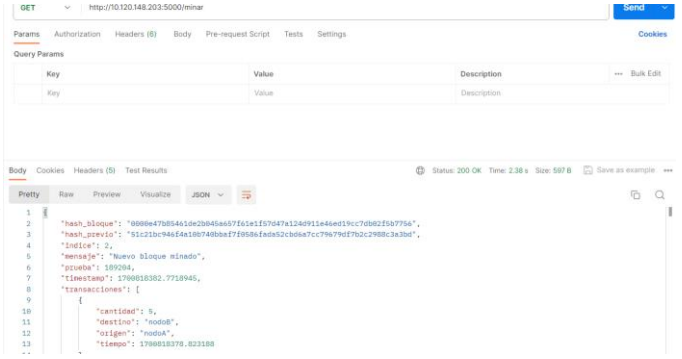
En resumen, estas funciones están diseñadas para gestionar diferentes aspectos de una red de cadena de bloques distribuida, incluyendo transacciones, minería de bloques, sincronización entre nodos y mantenimiento de la integridad de la cadena de bloques en un entorno descentralizado.

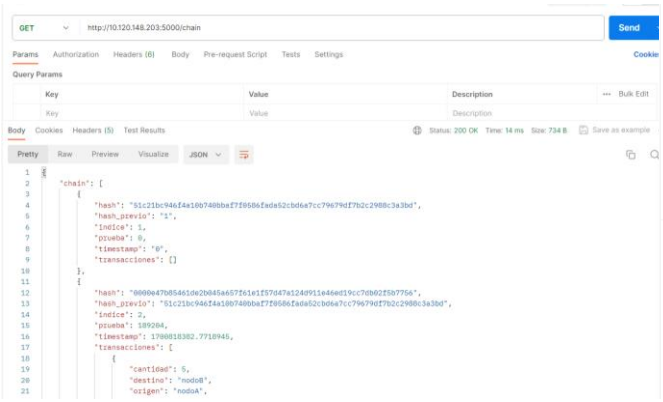
Ejecución en una Máquina Virtual (Ubuntu)

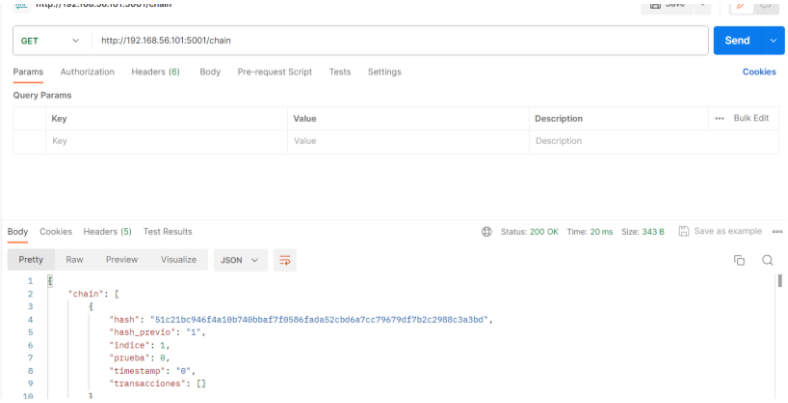
A continuación, se puede ver un ejemplo de la ejecución de la blockchain desde una máquina virtual con la distribución Ubuntu corriendo, donde se inicia una instancia en el puerto 5001 desde la máquina virtual, y otra desde el ordenador en el puerto 5000:

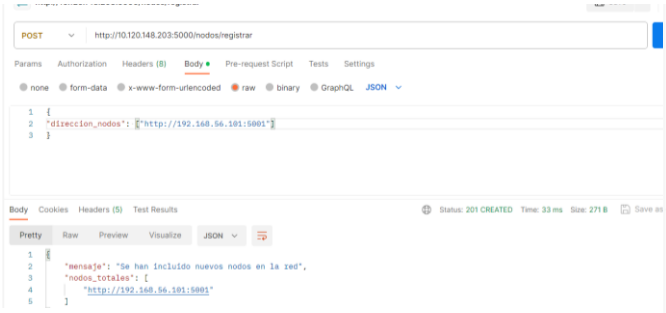
```
ulisesdiez@ulisesdiez-VirtualBox:~/Desktop/BlockChain$ cd Ulises_Diez_Santaolalla_Ignacio_Felices_Ver
a_Blockchain/
ulisesdiez@ulisesdiez-VirtualBox:~/Desktop/BlockChain/Ulises_Diez_Santaolalla_Ignacio_Felices_Vera_Bl
ockchain$ python3 Blockchain_app.py -p 5001
 * Serving Flask app 'Blockchain_app'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSG
I server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5001
 * Running on http://10.0.2.15:5001
Press CTRL+C to quit
127.0.0.1 - - [20/Nov/2023 18:24:08] "GET /chain HTTP/1.1" 200 -
127.0.0.1 - - [20/Nov/2023 18:24:39] "POST /nodos/registro_simple HTTP/1.1" 200 -
127.0.0.1 - - [20/Nov/2023 18:26:28] "GET /chain HTTP/1.1" 200 -
```

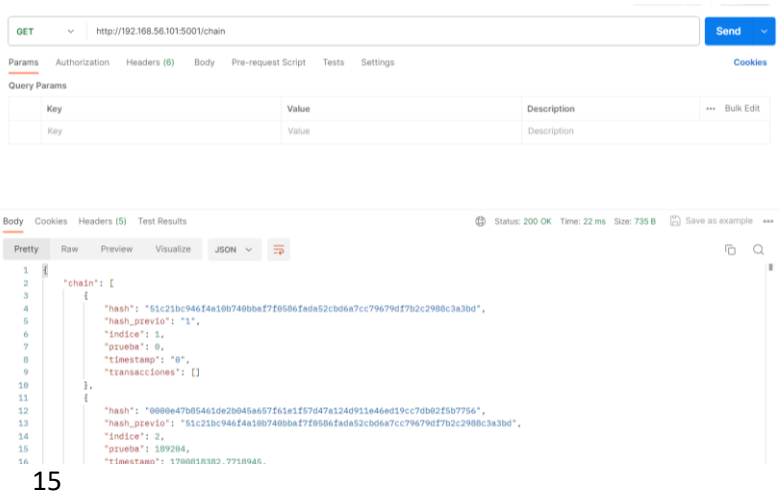












Desarrollo del módulo “pruebas_requests.py”

Este módulo se desarrolló para comprobar el funcionamiento de las distintas funciones que se crearon a lo largo del programa. Con este programa se hacen llamadas a los diferentes métodos del archivo “BlockChain_app.py”, la cual a su vez utiliza elementos del módulo “BlockChain.py”, por lo que se analiza el funcionamiento global y correcto de todos los métodos.

En primer lugar, se importaron las librerías necesarias para el desarrollo del código:

- import requests
- import json
- import time

Objetivo: Realizar una serie de operaciones en cada nodo de la red, como enviar transacciones, minar bloques, obtener información de la cadena, registrar nuevos nodos, realizar PINGS a los nodos y obtener las respuestas PONG.

Descripción de Acciones:

Configuración de Datos Iniciales:

- cabecera: Configura la cabecera común para las solicitudes HTTP.
- transaccion_nueva: Datos simulados para una nueva transacción. Se crean varias transacciones, para que sean diferentes las transacciones que se añaden a lo largo de las pruebas a cada nodo.
- urls: Lista de URLs de los nodos en la red.

Iteración a través de los Nodos:

Para cada URL en la lista urls:

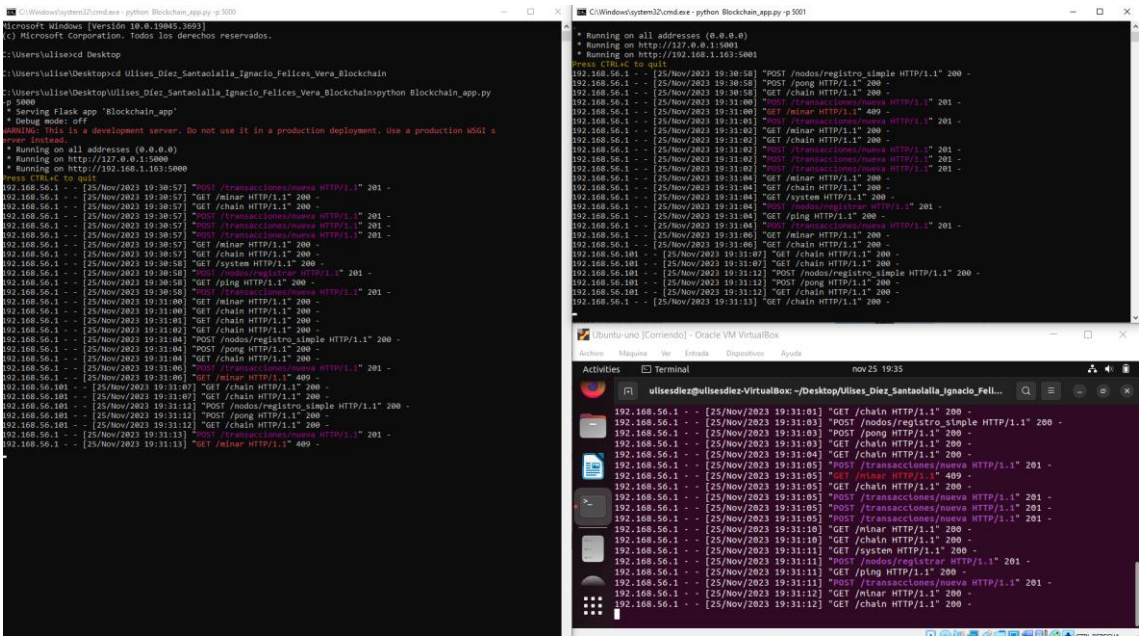
- Envío de Nueva Transacción (POST):
  - Envía una solicitud de nueva transacción al nodo.
  - Imprime la respuesta obtenida.
- Minado de Bloque (GET):
  - Realiza una solicitud para minar un bloque en el nodo.
  - Imprime la respuesta obtenida.
- Obtención de la Cadena Completa (GET):
  - Obtiene la cadena de bloques completa del nodo.
  - Imprime la respuesta obtenida.
- Envío de Nuevas Transacciones (POST):
  - Envía varias solicitudes de nuevas transacciones al nodo.
  - Imprime la respuesta obtenida.
- Minado de Bloque (GET):
  - Realiza una solicitud para minar un bloque en el nodo.
  - Imprime la respuesta obtenida.
- Obtención de la Cadena Completa (GET):
  - Obtiene la cadena de bloques completa del nodo.
  - Imprime la respuesta obtenida.
- Obtención de Detalles del Nodo (GET):
  - Solicita y obtiene detalles sobre el nodo actual.
  - Imprime la respuesta obtenida.
- Registro de Nuevos Nodos (POST):
  - Registra nuevos nodos en la red.
  - Imprime la respuesta obtenida.

- Realización de Ping a Nodos (GET):
    - Envía una solicitud de PING a los nodos en la red.
    - Imprime las respuestas PONG recibidas de los nodos.
- Prueba de conflictos de BlockChain con los nuevos nodos añadidos:
    - Añade una nueva transacción al nodo principal y mina un bloque.
    - Añade una nueva transacción a un nodo de la red y mina un bloque.
    - Salta un conflicto al detectar que los tamaños de las blockchains de los nodos de la red son de diferente tamaño.
- Espera un segundo antes de pasar al siguiente nodo.

```python
# Cabecera JSON común a todas las solicitudes
cabecera = {'Content-type': 'application/json', 'Accept': 'text/plain'}

# Datos de transacción para prueba
transaccion_nueva = {'origen': 'nodoA', 'destino': 'nodoB', 'cantidad': 10}
# Datos para realizar varias transacciones, con diferentes cantidades
transaccion_nueva_1 = {'origen': 'nodoA', 'destino': 'nodoB', 'cantidad': 20}
transaccion_nueva_2 = {'origen': 'nodoA', 'destino': 'nodoB', 'cantidad': 15}
transaccion_nueva_3 = {'origen': 'nodoA', 'destino': 'nodoB', 'cantidad': 35}
# Datos de una nueva transacción, utilizada para probar los conflictos de la blockcahin
# una vez se han registrado los nodos
transaccion_nueva_4 = {'origen': 'nodoA', 'destino': 'nodoB', 'cantidad': 54}
transaccion_nueva_5 = {'origen': 'nodoA', 'destino': 'nodoB', 'cantidad': 46}

# URL de los nodos en la red   'http://192.168.56.101:5002',
urls = [
    'http://192.168.56.1:5000',
    'http://192.168.56.1:5001',
    'http://192.168.56.101:5002'
]

# Realizo la prueba con cada uno de los nodos
for url in urls:
    print(f"\nNodo: {url}")

    # Envío una nueva transacción al nodo
    r = requests.post(f'{url}/transacciones/nueva', data=json.dumps(transaccion_nueva), headers=cabecera)
    print("Respuesta de nueva transacción:", r.text)

    # Mino un bloque en el nodo
    r = requests.get(f'{url}/minar')
    print("Respuesta de minar bloque:", r.text)

    # Obtengo la cadena completa del nodo
    r = requests.get(f'{url}/chain')
    print("Respuesta de obtener cadena completa:", r.text)

    # Envío varias transacciones al nodo
    r = requests.post(f'{url}/transacciones/nueva', data=json.dumps(transaccion_nueva_1), headers=cabecera)
    print("Respuesta de nueva transacción:", r.text)
    r = requests.post(f'{url}/transacciones/nueva', data=json.dumps(transaccion_nueva_2), headers=cabecera)
    print("Respuesta de nueva transacción:", r.text)
    r = requests.post(f'{url}/transacciones/nueva', data=json.dumps(transaccion_nueva_3), headers=cabecera)
    print("Respuesta de nueva transacción:", r.text)

    # Mino un bloque en el nodo
    r = requests.get(f'{url}/minar')
    print("Respuesta de minar bloque:", r.text)
```

```python
# Obtengo la cadena completa del nodo
r = requests.get(f'{url}/chain')
print("Respuesta de obtener cadena completa:", r.text)

# Obtengo los detalles del nodo
response = requests.get(f'{url}/system')
print("Respuesta de obtener detalles del nodo:", response.text)

# Registro nuevos nodos en la red
nodos = []
nuevos_nodos = False
for otras_url in urls:
    if otras_url != url:
        nodos.append(otras_url)
        nuevos_nodos = True
if nuevos_nodos:
    data = {'direccion_nodos': nodos}
    response = requests.post(f'{url}/nodos/registrar', json=data)
    print("Respuesta de registrar nodos:", response.text)

# Hago una llamada PING y obtengo las respuestas PONG
response = requests.get(f'{url}/ping')
print("Respuesta de ping a nodos:", response.text)

# Se comprueba que haya conflictos cuando se añade nueva transacción y se mina un bloque, ya que al comparar
# las blockchains de los nodos de la red, tendrían distinta longitud
# Envío una nueva transacción al nodo
r = requests.post(f'{url}/transacciones/nueva', data=json.dumps(transaccion_nueva_4), headers=cabecera)
print("Respuesta de nueva transacción:", r.text)
# Mino un bloque en el nodo
r = requests.get(f'{url}/minar')
print("Respuesta de minar bloque:", r.text)
# Añado una transacción y mino un bloque en otro nodo para ver que salta conflicto
r = requests.post(f'{nodos[0]}/transacciones/nueva', data=json.dumps(transaccion_nueva_5), headers=cabecera)
print("Respuesta de nueva transacción:", r.text)
# Mino un bloque en el nodo
r = requests.get(f'{nodos[0]}/minar')
print("Respuesta de minar bloque:", r.text)

# Espero un tiempo antes de hacer la siguiente prueba
print(f"---------------- Pruebas en el nodo {url} terminadas ----------------")
time.sleep(1)
```

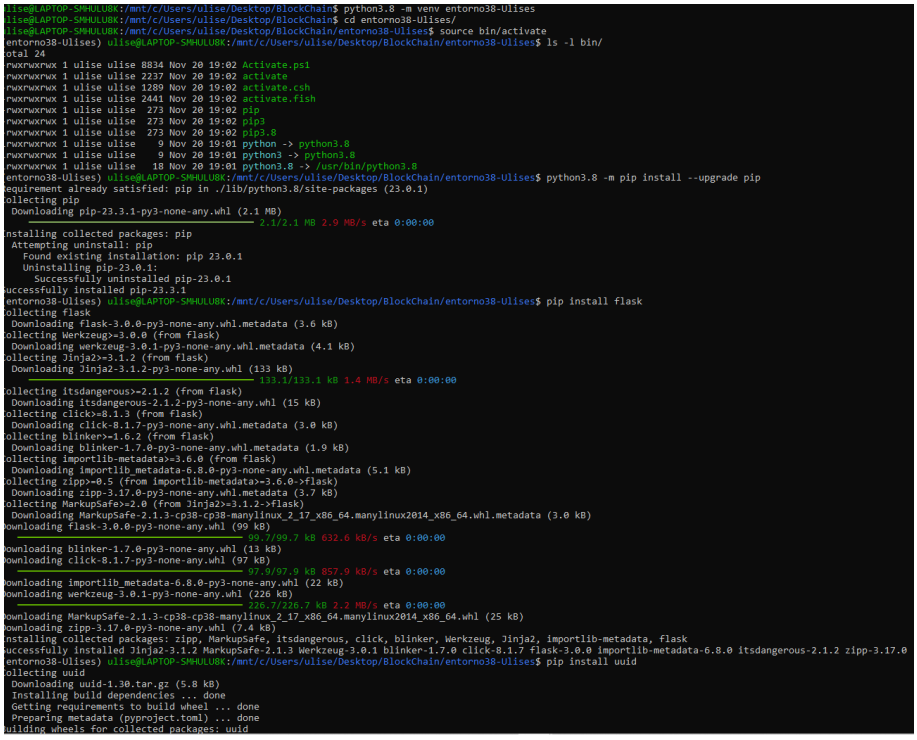
A continuación, se puede observar un ejemplo de la ejecución de este programa. Para poner a prueba todas las funciones, y en diferentes sistemas, se probó con tres nodos. Dos se inicializaron en el ordenador host, en los puertos 5000 y 5001, mientras que la otra instancia se inicializó en la máquina virtual VirtualBox con una distribución Ubuntu corriendo, en el puerto 5002.

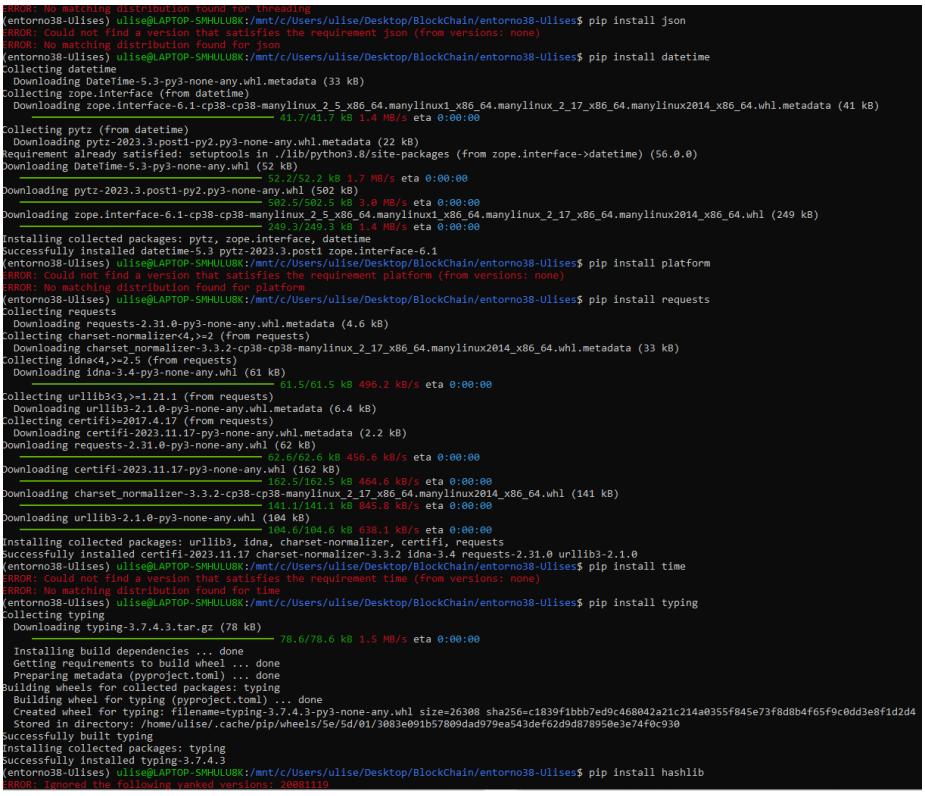


A continuación se puede observar el resultado de este archivo “pruebas_request.py”:

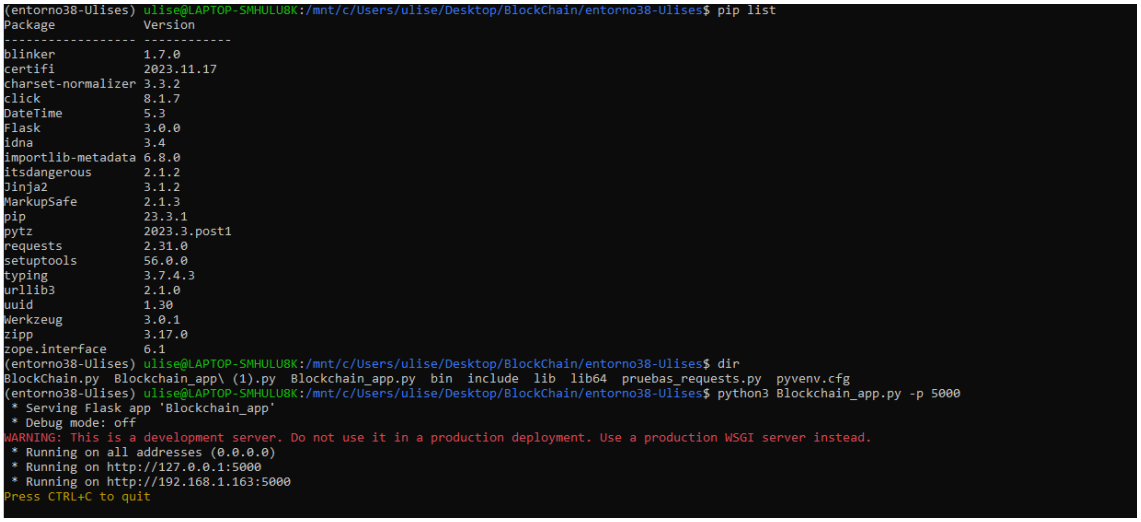
Desarrollo del archivo “requirements.txt”

Para la obtención de un fichero de requirements.txt que especifique claramente las librerías que se han instalado y sus versiones, no lo ejecuté directamente sobre la terminal ya que aparecerían otras librerías innecesarias para este programa BlockChain que fueron instaladas en el ordenador para otros objetivo. Para obtener únicamente las librerías necesarias para la ejecución de este programa, generé un nuevo entorno virtual en el ordenador e instalé las librerías que se necesitaban importar para el funcionamiento del programa, como se puede ver a continuación:





Al no funcionar correctamente el comando pip freeze, al igual que hice en la práctica 3, ejecuté pip list y copié su contenido en el documento de texto, obteniendo de esta manera el archivo “requirements.txt”.



Como se puede ver en la imagen, con estas librerías instaladas, en este nuevo entorno virtual ya se puede ejecutar de manera correcta el programa, por lo que estaría completa la instalación y lista para su utilización la BlockChain.