PROYECTO BLOCKCHAIN

Jorge Ibinarriaga Robles

Introducción

Este trabajo consiste en una implementación básica de un BlockChain (cadena de bloques) a través de una aplicación descentralizada de un conjunto de nodos conectados a una red.

El programa tiene dos librerías principales:

- Implementación cadena **BlockChain**: se encarga de crear el objeto Blockchain, que es un bloque de transacciones, y el objeto **Transaccion**. Contiene todas las funcionalidades del BlockChain.
- **Aplicación descentralizada**: crea una red de nodos descentralizada, conectada mediante peticiones HTTP de tipo post y get. Crea un Blockchain que es compartida por el resto de nodos para minar.

Módulo Blockchain.py

Transacción

Las transacciones en el programa son almacenadas en una clase. Esto no fue una buena decisión ya que dió complicaciones en el resto del programa, lo mejor hubiese sido usar un diccionario. El método to_dict() transforma el objeto a diccionario

```python
class Transaccion:
    def __init__(self, origen, destino, cantidad, timestamp):
        self.origen = origen
        self.destino = destino
        self.cantidad = cantidad
        self.timestamp = timestamp
    def __str__(self):
        return f"""
        Origen: {self.origen}
        Destino: {self.destino}
        Cantidad: {self.cantidad}
        Timestamp: {self.timestamp}
        """
    def to_dict(self):
        return {
            "origen": self.origen,
            "destino": self.destino,
            "cantidad": self.cantidad,
            "timestamp": self.timestamp
        }

    @classmethod
```

Bloque

Cada bloque va a contener una lista de transacciones. El bloque además contiene métodos y atributos básicos para su incorporación en el blockchain, como calcular hash y hash_previo. El time_stramp y prueba es lo que hacen al bloque único y son cruciales para la integridad y seguridad en la cadena.

```python
class Bloque:
    def __init__(self, indice:int, transacciones: List,
                 hash_previo:str, prueba: int = 0, timestamp = time.time()):
        """
        Constructor de la clase 'bloque'.
        Parámetros:
        -Indice: ID único de nuestro bloque
        -Transacciones: lista con todas las transacciones.
        -Timestamp: momento en el que el bloque fue generado.
        -hash_previo: clave criptográfica del anterior bloque
        (bloques encadenados uno detrás del otro)
        El hash del bloque inicializado es None.
        -param_prueba: prueba de trabajo
        """
        self.indice = indice
        self.transacciones = transacciones
        self.timestamp = timestamp
        self.hash_previo = hash_previo
        self.prueba = prueba
        self.hash = None
```

Calcular hash, calcula un **hash único** dado un bloque. Es decir, un bloque va a tener asignado un hash único.

El hash se calcula a partir del bloque:

```python
    def calcular_hash(self):
        """
        Devuelve el hash de un bloque
        """
        diccionario = dict()
        diccionario["indice"] = self.indice
        diccionario["timestamp"] = self.timestamp
        diccionario["hash_previo"] = self.hash_previo
        diccionario["prueba"] = self.prueba
        diccionario["hash"] = self.hash
        diccionario["transacciones"] = []
        for transaccion in self.transacciones:
            if type(transaccion) == dict:
                diccionario["transacciones"].append(transaccion)
            else:
                diccionario["transacciones"].append(transaccion.to_dict())
        block_string = json.dumps(diccionario, sort_keys=True)
        return hashlib.sha256(block_string.encode()).hexdigest()
```

El método toDict() se usa para pasar el bloque a JSON, que será de utilidad en la aplicación. El método __str__ se usa para comprobación, no tiene utilidad.

```python
    def toDict(self):
        lista = []
        for transaccion in self.transacciones:
            if type(transaccion) == dict:
                lista.append(transaccion)
            else:
                lista.append(transaccion.__dict__)
        return {
            "indice" : self.indice,
            "transacciones" :lista,
            "timestamp" : self.timestamp,
            "hash_previo" : self.hash_previo,
            "prueba" : self.prueba,
            "hash" : self.hash
        }
    def __str__(self):
        return f"""
        Indice: {self.indice}
        Transacciones:
        {[(t.origen, t.destino, t.cantidad, t.timestamp) for t in self.transacciones]}
        timestamp: {self.timestamp}
        hash_previo: {self.hash_previo}
        prueba: {self.prueba}
        hash: {self.hash}
        """
```

Blockchain

Como se menciona anteriormente el objeto Blockchain es una cadena de bloques.

El Blockchain se inicializa como una cadena de bloques con una lista de transacciones no confirmadas que contiene las transacciones que se van a añadir en el siguiente bloque. El primer bloque de la cadena se le asigna por defecto un valor del hash previo 1 y una lista de transacciones vacía. El atributo de dificultad es de utilidad para la seguridad a la hora de minar el bloque, a mayor dificultad, más difícil es minar por lo que será más seguro.

```python
class  BlockChain():
    """
    Lista de bloques.
    """
    def __init__(self):
        self.dificultad = 4
        self.transacciones_no_confirmadas = []
        self.chain = []
        self.primer_bloque()
    def primer_bloque(self):
        bloque = Bloque(indice = 1, transacciones=[], hash_previo = "1")
        bloque.hash = bloque.calcular_hash()
        self.chain.append(bloque)
```

Las transacciones nuevas son añadidas en la lista de transacciones no confirmadas de la siguiente manera:

```python
def nueva_transaccion(self, origen:str, destino:str, cantidad: float) -> int:
    """
    Crea una nueva transaccion a partir de un origen, un destino y una
    cantidad y la incluye en las listas de transacciones.
    Devuelve el indice del bloque que va a almacenar la transacción
    """
    timestramp = time.time() #momento en el que se añade la transacción
    transaccion = Transaccion(origen, destino, cantidad, timestramp)
    self.transacciones_no_confirmadas.append(transaccion)
    return len(self.chain) + 1
```

El algoritmo de prueba valida sirve en ir aumentando la prueba de trabajo (cambia el hash, porque cambia el bloque) hasta que calcular hash encuentra un hash con tantos ceros como la dificultad. Esto se usa para aumentar la dificultad de minar, aumentando la seguridad de la red

```python
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.calcular_hash()
    while hash[:self.dificultad] != "0"*self.dificultad:  #el hash no empieza por 0000
        bloque.prueba += 1 #numero de veces que ha habido que calcular el hash para cumplir la restricción de ceros
        hash = bloque.calcular_hash() #recalcula el hash
    return hash
```

A la hora de integrar un bloque se comprobará que su hash empiece por tantos ceros como la dificultad (prueba trabajo) y que el hash atribuido ese bloque coincida con el hash al calcular dicho bloque, es decir, que el bloque no haya sido modificado.

Esto se hará con prueba valida:

```python
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:
    """
    return (hash_bloque[:self.dificultad] == "0" * self.dificultad) & (hash_bloque == bloque.calcular_hash())
```

Finalmente, para integrar un bloque comprobaremos la prueba valida con el hash_prueba (hash atribuido a ese bloque) y que el hash previo coincida efectivamente con el hash del último bloque de la cadena. Si esto se verifica, se añadirá el bloque a la cadena y se reseteará las transacciones no confirmadas de la Blockchain ya que estas ya han sido añadidas al bloque.

```python
def integrar_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.prueba_valida(bloque_nuevo, hash_prueba)) & (self.chain[-1].hash == bloque_nuevo.hash_previo):
        print(f"Se ha añadido el bloque {bloque_nuevo.indice} correctamente")
        bloque_nuevo.hash = hash_prueba
        self.chain.append(bloque_nuevo)
        self.transacciones_no_confirmadas = [] #resetea las transacciones al haber añadido el bloque
        return True
    else:
        return False
```

Módulo Blockchain_app.py

Este módulo se encargará del diseño de la red descentralizada. Contendrá dos hilos principales; el hilo principal __main__ que se encargará de ejecutar el entorno Flask y otro hilo que hará una copia de seguridad de forma periódica.

La red contendrá una blockchain común entre todos los nodos.

Main

El main se encargará de crear el hilo de la copia de seguridad y ejecutarlo (hilo padre).

Además, inicializa la aplicación Flask en el host “0.0.0.0” en el puerto especificado.

```python
if __name__ == '__main__':
    parser =ArgumentParser()
    parser.add_argument('-p', '--puerto', default = 5000
                        , type=int, help='puerto para escuchar')
    args =parser.parse_args()
    puerto =args.puerto
    #hilo copia de seguridad
    hilo = Thread(target= copia_seguridad, args= (puerto,))
    hilo.start()
    app.run(host="0.0.0.0", port=puerto)
    hilo.join()
```

Copia de seguridad

Este hilo se encargará de la copia de seguridad: creará un archivo json y enviará la blockchain almacenada en la red en formato json de forma periódica. Este hilo contiene una sección critica, ya que mientras se escribe la blockchain en el fichero, ningún otro hilo puede modificar la blockchain, por lo que la sección critica será protegida por un semáforo mutex que se liberará después de escribir el fichero.

```
#COPIA DE SEGURIDAD
def copia_seguridad(puerto):
    while True:
        print("Realizando copia de seguridad")
        #Sección Crítica
        mutex.acquire()
        with open(f"respaldo-nodo{mi_ip}-{puerto}.json", "w") as archivo_json:
            json.dump(copia_blockchain(), archivo_json, indent=2)
        #Fin sección crítica
        mutex.release()
        print("Realizada copia de seguridad")
        time.sleep(60)
```

Peticiones HTTP de la red

Todas las peticiones de la red serán acompañadas con decoradores para indicar la ruta y el método a utilizar.

Para añadir una transacción a la red se usará el método POST que recibirá un archivo JSON y proceserá dicho archivo para enviárselo a la blockchain de la red

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

La funcionalidad /chain se encargará de enviar un archivo JSON de la blockchain actual de la red:

```
@app.route('/chain', methods=['GET'])
def blockchain_completa():
    response = {
        # Solamente permitimos la cadena de aquellos bloques finales que tienen hash
        'chain': [b.toDict() for b in blockchain.chain if b.hash is not None],
        'longitud': len(blockchain.chain),
                    }
    print("Los nodos se han añadido a la red correctamente")
    return jsonify(response), 200
```

/sistema se usará para obtener detalles del nodo conectado a dicha red, como el nombre del sistema operativo, la versión o el tipo de procesador.

```python
@app.route('/sistema', methods=['GET'])
def get_system_details():
    system_details = {
        "maquina": platform.machine(),
        "nombre_sistema": platform.system(),
        "version": platform.version(),
        "procesador": platform.processor()
    }
    return jsonify(system_details)
```

La función minar, se encargará de añadir un bloque a la red (tarea computacionalmente difícil) aunque en este entorno, simplificado, no lleva más de unos segundos.

Esta función, primero comprobará que hay transacciones no confirmadas, por que sino no se podrán añadir a la red, luego añadirá una transacción, que será la recompensa obtenida por minar un bloque, posteriormente, procederá a crear el bloque y añadirlo al Blockchain de la red (en ese puerto).

Esto lo hará una vez hecha la prueba de trabajo, para buscar un hash que empiece por tantos ceros como la dificultad (esto es lo que hace minar complicado).

Después ejecutará la función resolver conflictos, que su funcionamiento será explicado después, pero devolverá un booleano que indicará si hay conflicto o no. Si hay conflicto, no se podrá integrar el bloque a la red y se devolverá un mensaje, y si no hay, se integrará el bloque a la red, ejecutándose prueba valida para comprobar que el hash del bloque es valido.

```python
@app.route('/minar', methods=['GET'])
def minar():
    # 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, recibimosrecompensa
        # Recibimos un pago por minar el bloque. Creamos una nueva transaccion con:
        # Dejamos como origen el 0
        # Destino nuestra ip
        # Cantidad = 1
        blockchain.nueva_transaccion("0", mi_ip, 1)
        previous_hash =blockchain.chain[-1].hash
        bloque = blockchain.nuevo_bloque(previous_hash)
        blockchain.prueba_trabajo(bloque)

        #RESOLVER CONFLICTOS
        conflicto = resuelve_conflictos()
        if conflicto:
            response = {
                'mensaje': "Ha habido un conflicto. Esta cadena se ha actualizado con una version mas larga"
            }
            #Obliga al nodo a capturar transacciones si quiere crear un nuevo bloque
        else:
            blockchain.integrar_bloque(bloque, bloque.calcular_hash())
            response = {
                'mensaje' : "Nuevo bloque minado",
                'indice' : bloque.indice,
                'transacciones' : [transaccion.__dict__ for transaccion in bloque.transacciones],
                'prueba' : bloque.prueba,
                'hash_previo' : bloque.hash_previo,
                'hash' : bloque.hash,
                'timestramp' : bloque.timestamp
            }

    return jsonify(response), 200
```

Conexión con otros nodos de la red

Para registrar otros nodos a la red, se enviará un archivo JSON con los nodos que se desean añadir indicando su ruta (dirección IP y puerto) que les identifica. Los nodos se procesarán y se añadirán a nodos_red, un set almacenado en la red como variable global que conteniene todos los nodos de la red. Para cada nodo que se desea añadir se enviará a dicho nodo una lista conteniendo el nodo de la red (principal) y todos los nodos a registrar, menos el mismo.

```python
#REGISTRAR NODOS (APLICACIÓN WEB DESCENTRALIZADA)
@app.route('/nodos/registrar', methods=['POST'])
def registrar_nodos_completo():
    values = request.get_json()
    global blockchain
    global nodos_red
    nodos_nuevos = values.get('direccion_nodos') #nodos_nuevos es una lista con las direcciones de cada nodo

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

    all_correct = True
    for nodo in nodos_nuevos:
        print(nodo)
        try:
            nodos_red.add(nodo)
            nodos_direcciones = [n for n in nodos_nuevos if n != nodo]
            nodos_direcciones.append(f"http://{mi_ip}:{puerto}")
            print(nodos_direcciones)
            data = {
                'nodos_direcciones': nodos_direcciones, #lista de las direcciones de los otros nodos
                'blockchain': blockchain.copia()
                }                                       #El dumps lo pasa a formato str
            response = requests.post(nodo +"/nodos/registro_simple", data=json.dumps(data) , headers ={'Content-Type':"application/json"})
        except requests.exceptions.RequestException as e:
            all_correct = False
            print(f"Error al notificar el nodo {nodo}: {e}")

    if all_correct:
        response ={
        'mensaje': 'Se han incluido nuevos nodos en la red',
        'nodos_totales': list(nodos_red)
        }
    else:
        response = {
        'mensaje': 'Error notificando el nodo estipulado',
        }
    return jsonify(response), 201
```

Cada nodo que se desea añadir recibirá por tanto una lista con todos los nodos de la red menos el mismo y la blockchain del nodo que ha hecho la petición para añadir los nodos (nodo principal).

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

    """
    Recibe una lista con los otros nodos de la red y una copia en formato JSON del Blockchain del nodo principal
    """
    # Obtenemos la variable global de blockchain
    global blockchain
    global nodos_red
    read_json = request.get_json()
    nodes_addreses =read_json.get("nodos_direcciones")
    print(nodes_addreses)
    for direccion in nodes_addreses:
        nodos_red.add(direccion)
    print(nodos_red)
    blockchain_recibida = read_json.get('blockchain')
    #Crear la cadena, blockchain_leida, a partir de la blockchain recibida en formato JSON
    blockchain_leida = crear_blockchain(blockchain_recibida)
    if blockchain_leida is None:
        return "El blockchain de la red esta currupto", 400
    else:
        blockchain = blockchain_leida
        return "La blockchain del nodo" + str(mi_ip) +":" +str(puerto) + "ha sido correctamente actualizada", 200
```

La blockchain, recibida en formato JSON será creada iterando sobre cada elemento del JSON. Contriyéndose bloque a bloque. Cabe destacar que como ya ha sido calculada la prueba de trabajo no hace falta volverla a calcular, por lo que el hash asociado no cambia, al tener ya el valor de prueba de trabajo.

```python
def crear_blockchain(blockchain_recibida):

    """
    Construye la blockchain recibida bloque a bloque a partir del JSON recibido.
    Además, comprueba por cada bloque del blockchain del JSON, debde crear el bloque y comprobar que el hash es valido
    """
    blockchain_leida = Blockchain.BlockChain()
    #Añade el primer bloque
    b = blockchain_recibida[0]
    blockchain_leida.chain[0].hash = b.get("hash")
    blockchain_leida.chain[0].timestamp = b.get("timestamp")

    #Añade los demás bloques
    for b in blockchain_recibida[1:]:
        bloque = Blockchain.Bloque(
            indice= b.get("indice"),
            transacciones = [Blockchain.Transaccion(**transaccion) for transaccion in b.get("transacciones")],
            hash_previo = b.get("hash_previo"),  prueba = b.get("prueba"), timestamp = b.get("timestamp"),
            )

        blockchain_leida.integrar_bloque(bloque, bloque.calcular_hash())

    return blockchain_leida
```

En esta aplicación, el objetivo de la red es que todos los nodos tengan la misma blockchain en la red. Para garantizar esto, si un nodo intenta minar un bloque y hay una blockchain existente en otro nodo más larga que la blockchain del nodo, entonces el nodo debe actualizar la blockchain con dicho nodo. De esta forma se garantiza que no se añadan bloques si ya hay una blockchain en otro nodo mas larga.

```python
def resuelve_conflictos():
    """
    Mecanismo para establecer el consenso y resolver los conflictos.
    Tiene que haber una única cadena entre todos los nodos de la red, por lo que si un todo tiene una cadena
    más larga, todos los nodos deben de ser actualizados con la cadena más larga
    """
    global blockchain
    global nodos_red
    longitud_actual = len(blockchain.chain)
    conflicto = False
    for nodo in nodos_red:
        response = requests.get(str(nodo)+'/chain')
        response = response.json()
        cadena_nodo = response.get("chain")
        longitud_cadena_nodo = response.get("longitud")
        if longitud_cadena_nodo > longitud_actual: #la cadena del nodo es mayor es actu la cadena actual
            blockchain.chain = [Blockchain.Bloque(**{k: v for k, v in b.items() if k != 'hash'}) for b in cadena_nodo]
            for i, bloque in enumerate(blockchain.chain):
                bloque.hash = cadena_nodo[i]["hash"]
            #cambiamos la cadena actual por la más larga de la red
            conflicto = True #ha habido un conflicto (hay una cadena más larga que la del nodo actual)
    return conflicto
```

Prótocolo ICMP

Este protocolo se encarga de comprobar la conexión de los nodos.

Consiste en enviar un mensaje a un mensaje (mensaje PING) a cada nodo de la red y recibir una respuesta de cada nodo (respuesta PONG), indicando su nodo, puerto y el mensaje enviado por PING.

La respuesta de todos estos nodos se concatena en el JSON que devuelve el método.

```python
@app.route('/ping', methods=['POST'])
def ping():
    global nodos_red
    global puerto
    global mi_ip
    #url del nodo
    """
    Verifica que el nodo esté conectado a la red.
    Envía mensaje PING al nodo
    """
    todos_responden = True
    respuesta_final = ""
    for nodo in nodos_red:
        data = {

            "IP Host" : mi_ip,
            "puerto" : puerto,
            "mensaje" : "PING"
        }
        t0 = time.time()
        response = requests.post(nodo +"/pong", data=json.dumps(data) , headers ={'Content-Type':"application/json"}) #envía el mensaje al nodo
        tf = time.time()
        if response.status_code == 200: #el nodo ha recibido el mensaje
            respuesta = response.json()
            respuesta_final  += f"#{respuesta.get('mensaje')}"
            respuesta_final += f"Respuesta: {respuesta.get('respuesta')}"
            retardo = tf-t0
            respuesta_final += f"Retardo: {retardo}"
        else:
            todos_responden = False
    if todos_responden:
        respuesta_final += "#Todos los nodos responden"
    return jsonify({"respuesta_final": respuesta_final })
```

```python
@app.route('/pong', methods=['POST'])
def pong():
    global mi_ip
    global puerto
    """
    El nodo responde al mensaje PING con un mensaje PONG
    """
    try:
        mensaje = request.get_json()
        response = {
            "IP nodo": mi_ip,
            "puerto" : puerto,
            "mensaje" : f"{mensaje.get('mensaje')} de {mensaje.get('IP Host')}: {mensaje.get('puerto')}",
            "respuesta": f"PONG{mi_ip}:{puerto}"

        }
        return jsonify(response), 200 #El nodo responde
    except requests.exceptions.RequestException as e:
        return jsonify({"Error" : e})
```

Es importante tener en cuenta que todos los métodos que involucran la comunicación entre otros nodos capturan como excepción “requests.exceptions.RequestException” que se lanza en caso de fallar la conexión de los nodos.

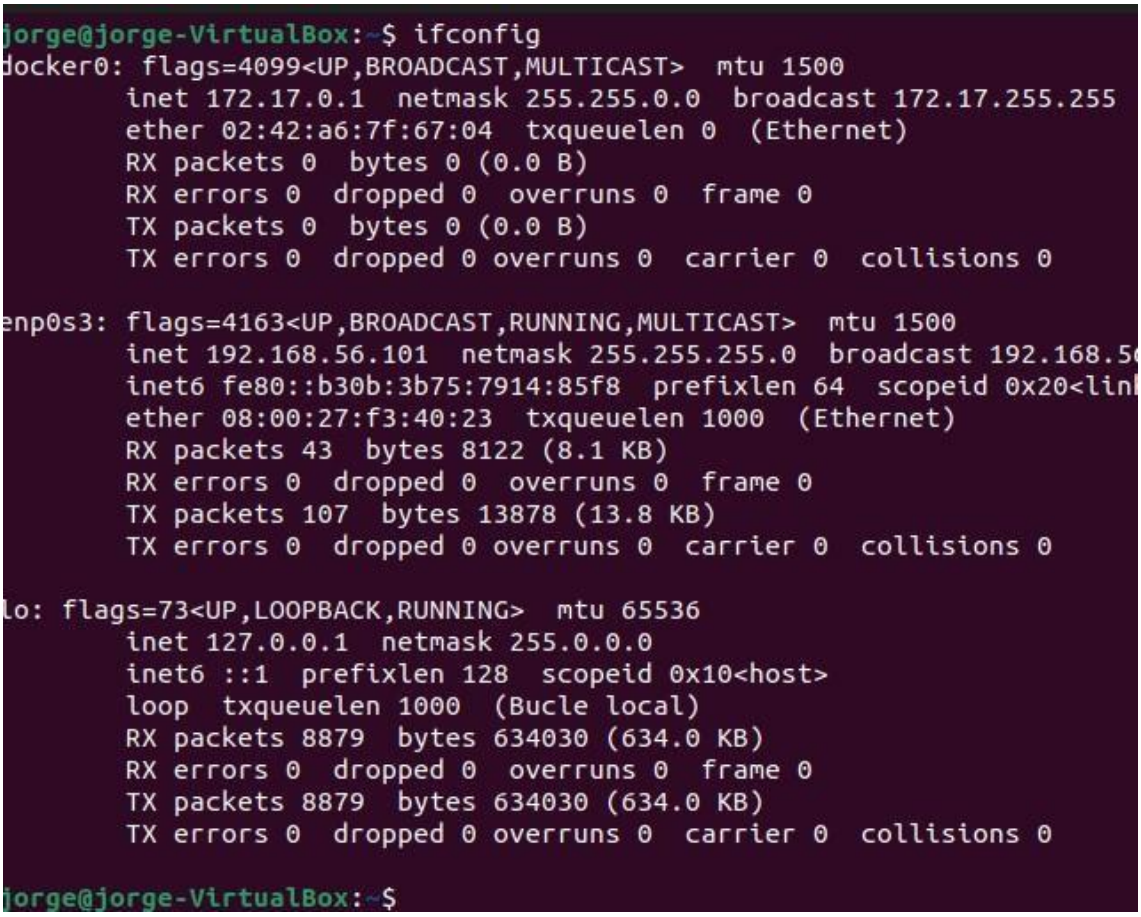
Módulo Request.py

Se encarga de realizar de forma sencilla y automatizada todas las pruebas en la aplicación.

```python
import requests
import json
# Cabecera JSON (comun a todas)
cabecera ={'Content-type': 'application/json', 'Accept': 'text/plain'}
# datos transaccion
transaccion_nueva ={'origen': 'nodoA', 'destino': 'nodoB', 'cantidad': 10}
r =requests.post('http://127.0.0.1:5000/transacciones/nueva', data =json.dumps(
transaccion_nueva), headers=cabecera)
print(r.text)
#MINAR
r =requests.get('http://127.0.0.1:5000/minar')
print(r.text)
r =requests.get('http://127.0.0.1:5000/chain')
print(r.text)
#AÑADIR NODOS
nodos_nuevos = {"direccion_nodos":["http://127.0.0.1:5001"]}
r = requests.post('http://127.0.0.1:5000/nodos/registrar', data = json.dumps(nodos_nuevos), headers=cabecera)
print(r.text)
#Comprobamos conexión (ICMP)
r = requests.post('http://127.0.0.1:5000/ping')
print(r.text)
#Detalles del sistema
r = requests.get('http://127.0.0.1:5001/sistema')
print(r.text)
#Comprobamos que tengan la misma cadena:
r =requests.get('http://127.0.0.1:5000/minar')
print(f"Cadena nodo 5000: {r.text}")
r =requests.get('http://127.0.0.1:5001/minar')
print(f"Cadena nodo 5001: {r.text}")
#Minamos nodo 5000
transaccion_nueva ={'origen': 'nodoC', 'destino': 'nodoD', 'cantidad': 2}
r =requests.post('http://127.0.0.1:5000/transacciones/nueva', data =json.dumps(
transaccion_nueva), headers=cabecera)
print(r.text)
r =requests.get('http://127.0.0.1:5000/minar')
print(r.text)
#Intentamos minar nodo 5001
transaccion_nueva ={'origen': 'nodoC', 'destino': 'nodoD', 'cantidad': 2}
r =requests.post('http://127.0.0.1:5001/transacciones/nueva', data =json.dumps(
transaccion_nueva), headers=cabecera)
print(r.text)
r =requests.get('http://127.0.0.1:5001/minar')
print(r.text)
```

Pruebas máquina virtual

Obtenemos la dirección IP de la máquina virtual:



Después compartimos el programa a la maquina virtual mediante la carpeta compartida y cambiamos la variable mi_ip por la IP de la maquina virtual.

Instalamos paquetes necesarios:

Después ejecutamos el programa en la máquina virtual y desde el host.