# Proyecto. Blockchain

Disponible: 4 de noviembre

Entrega: 5 de diciembre

# 1 Introducción: 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 resposabilidad 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 (p.e cuando se cae el servidor de Twitter o Whassap).

Por otro lado, en una red descentralizada, toda la carga de trabajo del procesamiento de los datos 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 ofrece mas 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.

# 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 blockckain). 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.

# 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 (e.g. 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).

¿Qué información vamos a almacenar en los bloques? 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.

# 4 Proyecto Blockchain

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

## 4.1 Entrega

El proyecto blockchain se realizará **por parejas**. Se admitirá un único grupo de 3 personas. Cada pareja enviará un archivo .zip siguiendo estas restricciones:

- El nombre del fichero seguirá el siguiente formato:

```
<nombreAlumno1>_<PrimerApellido1>_<SegundoApellido1>_<
    nombreAlumno2>_<PrimerApellido2>_<SegundoApellido2>
    _Blockchain.zip
```

  Por ejemplo:

```
Pablo_DeLasHeras_Fernandez_Martina_Perez_Gonzalez_Blockchain
    .zip
```

- El fichero zip debe contener todos los ficheros Python para hacer funcionar la aplicación de Blockchain. Se debe incluir también un informe en formato pdf, comentando cómo se ha ido abordando el proyecto adjuntando pantallazos de la funcionalidad de cada uno de los apartados. **En dicho informe y en todos los ficheros Python deben aparecer los nombres completos de los estudiantes**. Además, se debe entregar un fichero de requirements.txt donde se debe especificar **claramente** las librerías que se han instalado y sus versiones, para que los profesores puedan simular un entorno con todos esos detalles.

- **IMPORTANTE:** recordad que a la hora de programar, no sólo importa la **funcionalidad**. También importa el **estilo de código** y la **documentación**. Por esta razón, todo código entregado debe seguir un estilo correcto de programación y todas las funciones/clases deben estar correctamente documentadas. El código debe funcionar correctamente sin que haya excepciones no controladas o fallos que terminen la ejecución del programa de forma no controlada.

## 4.2 Introducción al proyecto

El proyecto se irá haciendo de forma incremental. Primeramente, desarrollaremos la parte de back-end, es decir, la lógica interna de la aplicación, donde se programará el código necesario para crear los bloques, la cadena de bloques (Blockchain), transacciones monetarias asociadas a cada bloque, etc. Dado que cada bloque tendrá un hash asociado (serie de caracteres que identifican al bloque), también se definirá el mecanismo necesario para calcular dichos hashes y poder construir cadenas de bloques correctas. De la misma manera, se diseñará un mecanismo para evitar que otros nodos de la red suplanten nuestra cadena de bloques. Posteriormente, se hará 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 desplegará en todos los nodos de nuestra red y podremos establecer una red sincronizada de nodos manteniendo toda la información asociada a los bloques y a las distintas transacciones.

## 4.3 Transacciones y bloques

La forma en la que vamos a almacenar los datos en nuestro proyecto de Blockchain va a ser empleando en formato JSON (JavaScript Object Notation). JSON no deja de ser 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 va a utilizar en el proyecto). Por lo tanto una transacción se vería 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"
}
```

Donde todos los campos de la transacción quedan contenidos dentro de las llaves {}. Las transacciones se van a agrupar en bloques. Por lo tanto, un bloque podrá componerse de una o más transacciones. Los bloques que contienen las transacciones se irán generando y añadiendo al blockchain. Dado que puede haber múltiples bloques, cada uno debería tener un identificador único. Por lo tanto, deberíamos crear una clase para almacenar los bloques. Primero, crearemos un fichero llamado "Blockchain.py" que será el empleado para la creación de las clases que representen a los bloques y el blockchain. La clase de bloque quedaría determinada de esta forma:

```python
class Bloque:
    def __init__(self, indice: int, transacciones: List[Dict], timestamp: float,
                                                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)
```

El campo de "prueba" lo detallaremos más adelante. El hash_previo indica la clave criptográfica del anterior bloque. Recordemos que los bloques deben estar encadenados uno detrás de otro y la forma de garantizar la integridad de dichos bloques es precisamente mediante dicho hash (además de los valores necesarios para crear el bloque, será necesario añadir otro atributo para almacenar el hash del bloque inicializado a None). Para poder calcular el hash de un bloque, vamos a hacer uso del siguiente método dentro de la clase de Bloque:

```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()
```

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 256 caracteres (hash) codificando toda la información del bloque. Es decir, dicha función, coge un bloque y lo transforma a una cadena de 256 caracteres.

## 4.4 Blockchain

Una vez definida la clase de Bloque, debemos definir ahora la clase de Blockchain. Como hemos comentado antes, el blockchain debe se 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). Puedes tomar como punto de partida la siguiente definición de la clase de Blockchain:

ambas en listas enlazadas? o listas de vectores??

```
class Blockchain(object):
    def __init__(self):
        self.dificultad =4
        # Codigo a completar (inicializacion de las listas de transacciones y de
                                                bloques)
```

Además, se pide crear otro método que se llame primer_bloque que lo que haga sea 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. Debido a que este primer bloque está solo para mostrar el inicio de la cadena, el campo del tiempo puede quedarse a 0. Continuaremos definiendo la clase de Blockchain con dos métodos nuevos, el de crear un nuevo bloque y el de crear una nueva transacción:

```
    def nuevo_bloque(self, hash_previo: str) ->Bloque:
        """
        Crea un nuevo bloque a partir de las transacciones que no estan
                                                confirmadas
        :param prueba: el valor de prueba a insertar en el bloque
        :param hash_previo: el hash del bloque anterior de la cadena
        :return: el nuevo gloque
        """
        #[...] Codigo a completar

    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
    """

        #[...] Codigo a completar
```

A la hora de crear un nuevo bloque, debemos introducir el conjunto de transacciones no confirmadas de la red en dicho bloque, indicando como tiempo la hora actual con time.time(). A la hora de crear una nueva transacción, debemos guardar en un diccionario los datos del origen, el destino, la cantidad, y el timestamp de dicha transacción. Una vez creada, se añadirá a la lista de transacciones no confirmadas. Siguiendo el formato JSON de la transacción, un ejemplo de bloque que podríamos tener sería el siguiente:

```
bloque = {
       'hash_bloque': "00000
           bc1bd8f93e180766d38eab25edd8cd2ccca815..."
    'hash_previo': "b02b56c51c2d0a4471d089775568ce693d0b3e0e537
    ..."
    'indice': 2,
    'timestamp': 1660661379.2007194,
     'prueba': 1000,
    'transacciones': [
        {
            'origen': "Pablo",
            'destino': "Javier",
            'cantidad ': 5,
        }
    ],
}
```

## 4.5 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 maliciosa modificase algún bloque de la cadena y recalculase todos los hashes de los bloques posteriores para manipular la cadena. Para ello vamos a añadir una prueba de trabajo que haga que en lugar de calcular cualquier hash para el bloque, calcule un hash que comience con al menos tantos ceros (0s) como el valor indicado de dificultad de blockchain (en este caso, 4). El prototipo del método de prueba_trabajo de la clase de Blockchain sería el siguiente:

```
    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)
        """
        #[Codificar el resto del metodo]
```

Este método de prueba trabajo debe inicializar el campo de prueba del bloque recibido por argumento a 0 y calcular el hash del bloque hasta que encuentre un hash que comience con tantos 0s como el campo de dificultad del blockchain. Cada vez que devuelva un hash que no cumpla la condición, incrementará el valor de prueba del bloque en 1 y volverá 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, necesitaremos otros dos métodos relacionados con la prueba de trabajo. El primero se llamará prueba_valida y servirá para comprobar que el valor de prueba es correcto. El segundo lo llamaremos integra_bloque, que introducirá el bloque en la cadena del blockchain si es un bloque correcto. En la documentación de ambos métodos se explica qué acciones deben realizar.

```
    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:
        """
        # [Codificar el resto del metodo]
```

```
    def integra_bloque(self, bloque_nuevo: Bloque, hash_prueba: str) ->bool:

        """
        Metodo para integran correctamente un bloque a la cadena de bloques.
        Debe comprobar que la prueba de hash 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 a integrar, 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)
        """
        # [Codificar el resto del metodo]
```

# 5 Aplicación web básica

Para esta parte de la práctica es necesario instalar los siguientes paquetes de Python que nos van a permitir hacer una pequeña aplicación web:

- Flask
- requests

Además, se va a pedir que se instale la aplicación de Postman en el sistema operativo host, que nos va a permitir realizar tests a nuestra aplicación web empleando el formato JSON.

Para crear la aplicación web, cread un fichero llamado Blockchain_app.py. Dicho fichero va contener todo el código necesario para el despliegue de nuestra aplicación. Vayamos paso a paso. Primeramente, añadid el siguiente código al fichero Blockchain_app.py:

```
import BlockChain
from uuid import uuid4

import socket
from flask import Flask, jsonify, request
from argparse import ArgumentParser

# Instancia del nodo
app =Flask(__name__)



# Instanciacion de la aplicacion
blockchain =BlockChain.Blockchain()

# Para saber mi ip
mi_ip =socket.gethostbyname(socket.gethostname())


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

    app.run(host='0.0.0.0', port=puerto)
```

Vamos a añadir 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.
- Además de esos tres métodos, se va a pedir que el proceso principal, antes de lanzar la aplicación, cree un hilo para que cada 60 segundos, dicho hilo haga una copia de seguridad del blockchain actual (toda la cadena de bloques que tenemos hecha). Dicha copia de seguridad se hará escribiendo el siguiente contenido en un fichero llamado "respaldo-nodo<ip_nodo>-<puerto_nodo>.json", y el contenido debe ser el siguiente json (se muestra una versión simplificada):

```
{
 "chain": [
  ...
 ],
 "longitud": ...
 "date": ...
}
```

  Donde "chain" hace referencia a la cadena de bloques (con sus respectivas transacciones), longitud hace referencia a la longitud de la cadena de bloques y date hace referencia a la fecha actual con el siguiente formato: dd/mm/aaaa hh:mm:ss. **Recordad que cuando se haga una copia de seguridad, se debe proteger la cadena de blockchain, para que ningún otro hilo pueda manipularla mediante ningún otro servicio. Es decir, siempre que se trabaje con la cadena de bloques, se debe impedir que ningún otro hilo acceda a ella.**

- Un método para obtener los detalles del nodo actual. En este caso, estamos interesados en poder consultar el sistema operativo del nodo, su versión y el tipo de procesador. Para ello, deberéis mirar cómo obtener esos detalles en Python. Por ejemplo, el servicio puede alojarse en la url /system de forma que cuando se haga un GET, se obtenga algo similar a esto:

```
{
    "maquina": "AMD64",
    "nombre_sistema": "Windows",
    "version": "10.0.19043"
}
```

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

```
@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 aqui
    index =...

    response ={'mensaje': f'La transaccion se incluira en el bloque con indice {
                                                index}'}
    return jsonify(response), 201


@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': ...# longitud de la cadena
    }
    return jsonify(response), 200
```

Si os fijáis 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.

En el caso del método de blockchain_completa, lo que hace es transformar cada bloque de la cadena en formato diccionario (método toDict) 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.

Tomando como ejemplo el código de los métodos anteriores, se pide completar el siguiente método de minar un bloque (será un método de GET, dado que no enviamos nada). Cuando se proceda 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 tendrá como origen el carácter

0, el destino será 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():
    # 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 minear el bloque, recibimos
                                                recompensa
        previous_hash =blockchain.last_block.hash

        # Recibimos un pago por minar el bloque. Creamos una nueva transaccion
                                                con:
        # Dejamos como origen el 0
        # Destino nuestra ip
        # Cantidad = 1
        # [Completar el siguiente codigo]


    return jsonify(response), 200
```

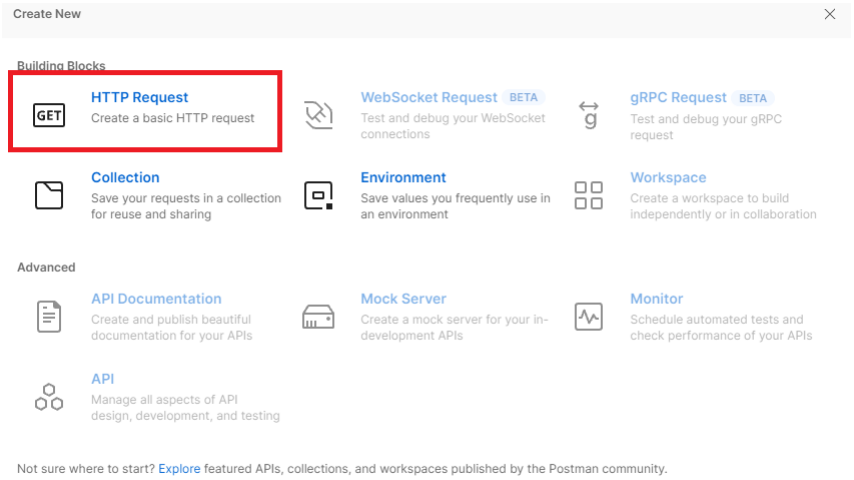
El código que faltaría por completar a la hora de minar primeramente debería crear un nuevo bloque con el hash_previo, proceder a realizar la prueba de trabajo y finalmente integrar el bloque en el blockchain con la prueba de trabajo calculada. Una vez generado, debe crear un mensaje de respuesta (response, como en los ejemplos anteriores) que contenga un mensaje diciendo "Nuevo bloque minado", el indice del nuevo bloque, las transacciones de dicho bloque, la variable de prueba de dicho bloque, el hash_previo, el hash del nuevo bloque y el timestamp en el que se ha generado el bloque.

## 5.1 Pruebas

Una vez que tengamos la primera parte del backend en en fichero de "Blockchain.py" y la parte de la aplicación en "Blockchain_app.py", ejecutaremos esta última, que debería quedarse esperando hasta que recibamos conexiones en el puerto 5000. Es decir, la salida debería ser algo parecido a esto:

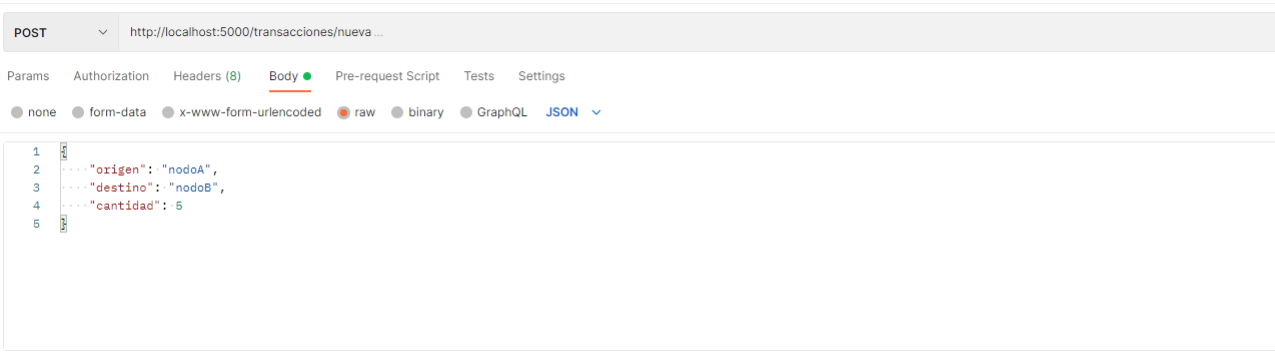
```
 * Serving Flask app 'Blockchain_app'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a
    production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://192.168.1.55:5000
Press CTRL+C to quit
```

Ahora, abrimos Postman y procedemos a crear una nueva petición HTTP (HTTP request) desde nuestro ordenador host, como se muestra en la siguiente imagen:

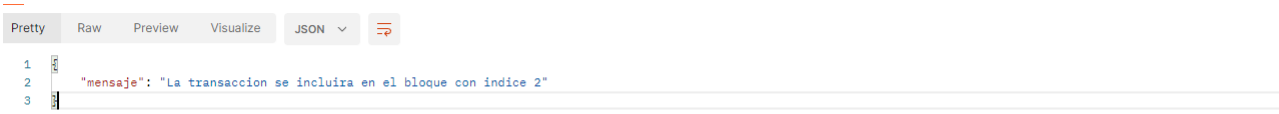



### 5.1.1 Nueva transacción

La primera acción que deberíamos realizar sería 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, como se muestra en la siguiente imagen:

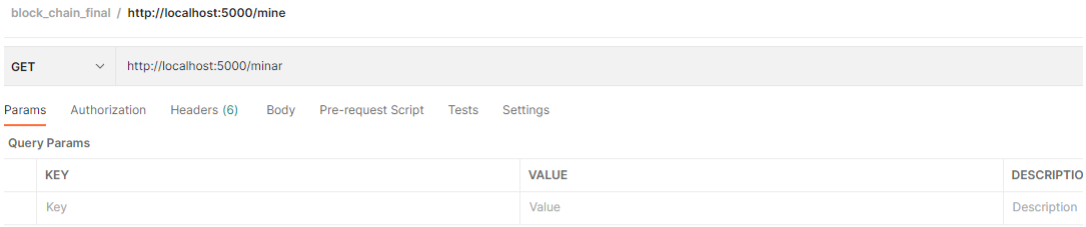


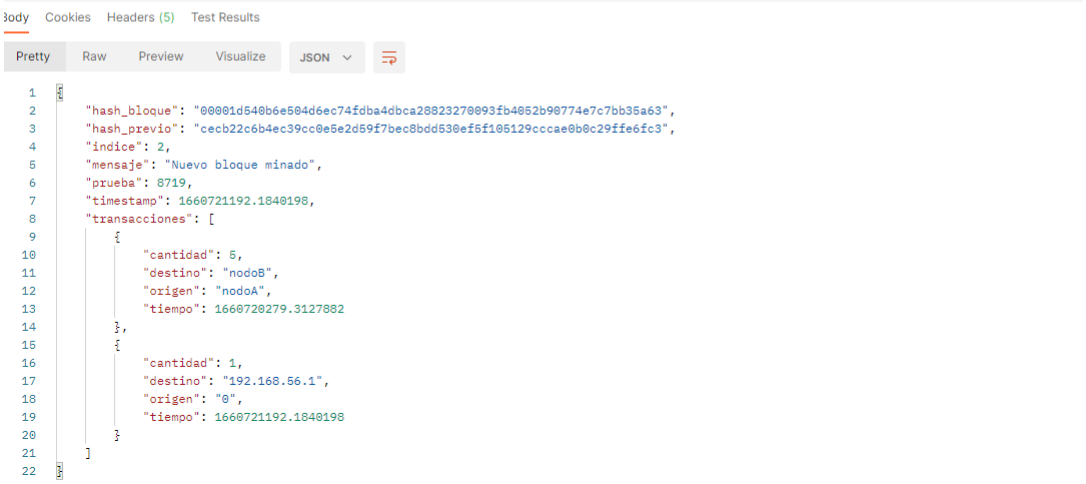

Si todo ha salido bien, obtendríamos el siguiente mensaje:



### 5.1.2 Minar un bloque

En este caso, vamos a solicitar al nodo minar un bloque. Para ello, debemos crear 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), obtendríamos lo siguiente (puede tardar un poco hasta que se complete la prueba de trabajo):

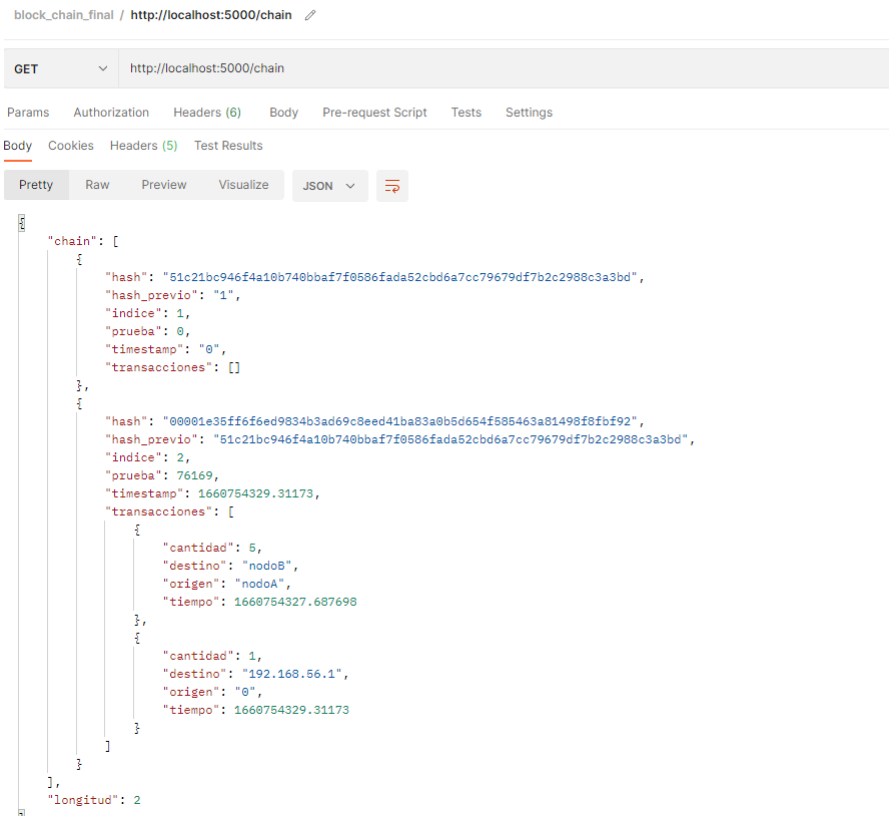



Fijaros que el hash del bloque empieza con 4 ceros como indicamos en nuestra prueba de trabajo y que haya 2 transacciones, no una, debido a que nuestro nodo debe recibir un "pago" por minar el bloque.

### 5.1.3 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, obtendríamos lo siguiente:




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.

# 6 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) los nodos se identificarán mediante la siguiente cadena de caracteres: "http://<ip>:<puerto>". Inicialmente, este conjunto estará vacío.

Además, crearemos dos funciones adicionales en nuestra aplicación. El primero, que llamaremos registrar_nodos_completo (accesible desde /nodos/registrar),

recibirá mediante el método de POST una lista de URLS siguiendo este formato: "http://<ip>:<puerto>" y lo que hará será almacenar los nodos recibidos en nodos_red y enviará 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. Para notificar a los nodos que quieren unirse a la red, emplearemos el segundo método que llamaremos "registrar_nodo_actualiza_blockchain". Así pues, los dos métodos serían algo parecido a esto:

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

    all_correct =True
    #[Codigo a desarrollar]
    # Fin codigo a desarrollar
    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


@app.route('/nodos/registro_simple', methods=['POST'])
def registrar_nodo_actualiza_blockchain():
    # Obtenemos la variable global de blockchain

    global blockchain

    read_json =request.get_json()
    nodes_addreses =read_json.get("nodos_direcciones")

    # [...] Codigo a desarrollar


    #[...] fin del codigo a desarrollar
    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
```

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

- Cuando queramos registrar un nodo en la aplicación deplegada, 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 ejecutaría el código de "registrar_nodos_completo", que debe por cada uno de los nodos de la lista recibida, crear un JSON para ser enviado a cada uno de los nodos de la lista. Dicho JSON debe contener 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 de 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 puede emplear este código:

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

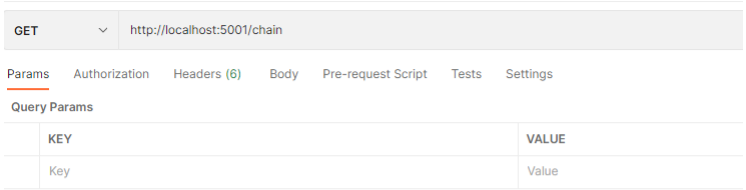
  Donde "data" sería un diccionario conteniendo dos claves y nodo hace referencia al nodo al que queremos enviar la copia de la blockchain. La primera clave, llamada "nodos_direcciones", contendría la lista de nodos a almacenar y la segunda clave, denominada "blockchain", contendría una copia del blockchain del nodo actual pasado a JSON. Para ello, se recomienda crear un método que permita crear una blockchain a partir de los campos recibidos por el JSON. Se pueden obtener cada uno de los campos accediendo al JSON como si fuera un diccionario y luego comprobar que el hash del JSON del bloque coincida con el hash del bloque que se ha recibido.

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

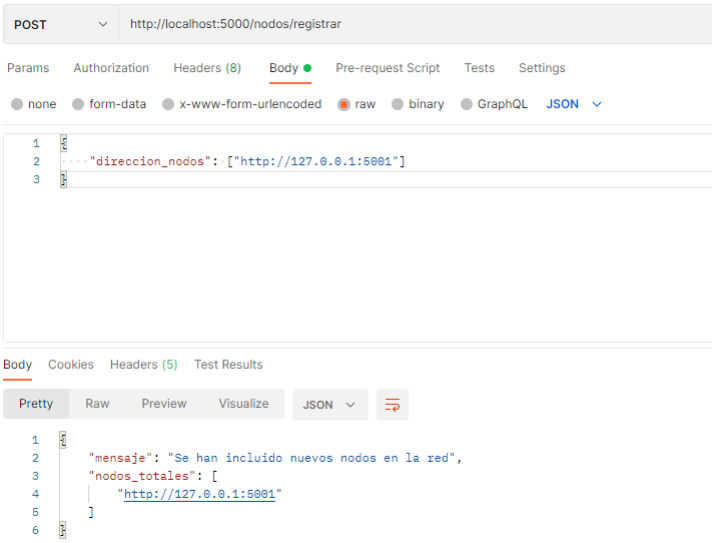
con "/nodos/registro_simple". Si todos los nodos reciben la copia de la blockchain de manera correcta, entonces se devuelve un mensaje diciendo los nodos que se han añadido a la red.

- 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, deberá 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 (para ello, puede emplearse el método de blockchain de integra_bloque).

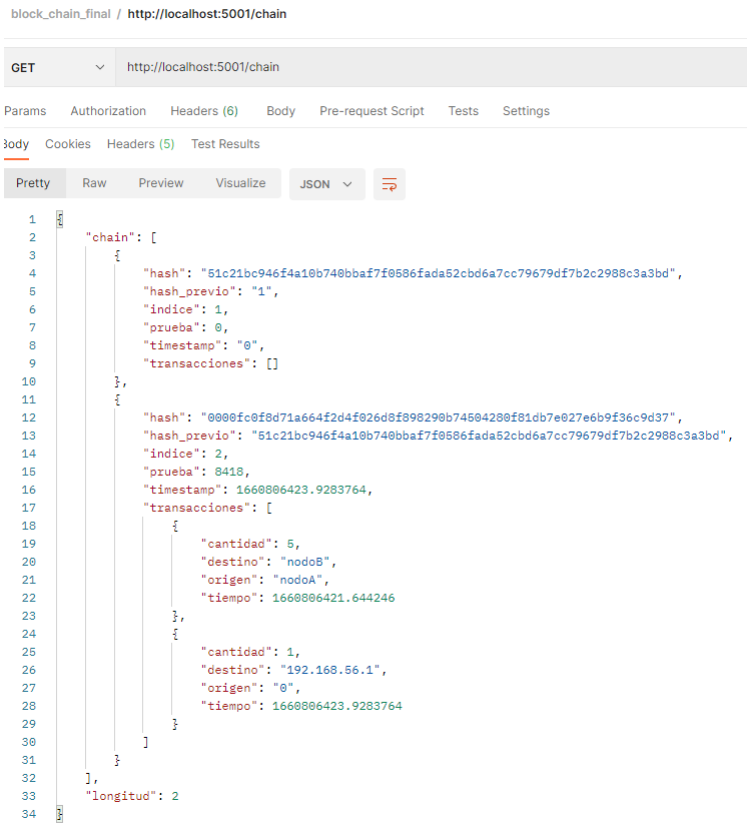
Una vez que esté todo hecho, podemos comprobar el funcionamiento de la aplicación de la siguiente forma. Primeramente, ejecutamos la aplicación en el puerto 5000, y lanzamos otra instancia en el puerto 5001. En la aplicación del puerto 5000, añadimos alguna transacción y minamos un bloque. Una vez hemos 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:




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:




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




## 6.1 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. Tomaremos una solución sencilla. Crearemos un algoritmo en Blockchain_app.py que comprobará 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 haremos será cambiar nuestra cadena actual por la más larga de la red. La función que crearemos se llamará "resuelve_conflictos" y será algo parecido a esto:

```
def resuelve_conflictos():
    """
    Mecanismo para establecer el consenso y resolver los conflictos
    """
    global blockchain

    longitud_actual =len(blockchain.chain)
```

```
    # [Codigo a completar]
```

El mecanismo de resolver los conflictos debe por cada nodo de la red, obtener la cadena de dicho nodo, empleándose para ello el siguiente código:

```
response =requests.get(str(nodo) +'/chain')
```

De esta forma, response contendrá un JSON con el que podremos acceder tanto a la cadena como a su longitud. Después, comprobaremos si la longitud de dicha cadena es mayor que la longitud de la cadena actual (almacenada en la variable de longitud_actual). Si es más larga, dicha cadena debe ser una candidata para ser empleada en la nueva red. Si al final hemos encontrado una cadena más larga, debemos devolver en el algoritmo de resolución de conflictos True, ya que ha habido un conflicto. En el caso de que nuestra cadena sea la más larga, devolveremos False y no habrá que hacer nada.

Todavía queda un problema que resolver, y es cuándo ejecutar el mecanismo de resolución de conflictos. Esto debe hacerse cada vez que minamos un bloque, por lo que habrá que modificar dicho método. Por lo tanto, antes de integrar el bloque en el minado, debemos llamar al método para resolver los conflictos. De esta forma, si nuestro nodo puede minar bien el bloque y no hay conflictos, lo integrará en la cadena y si por el contrario, se intentar minar un bloque de una cadena que no esté sincronizada, el método de minar debe devolver la siguiente respuesta:

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

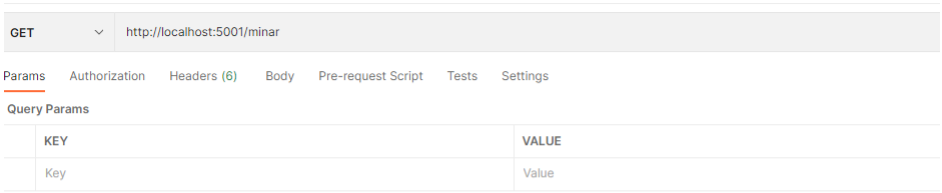
Obligando al nodo a volver a volver a capturar transacciones si quiere crear él un nuevo bloque.

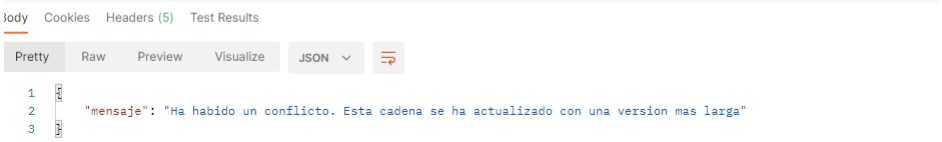
## 6.2 Pruebas

Para comprobar que el funcionamiento es correcto, una vez que hemos ligado el nodo 5001 al nodo 5000, añadiremos una transacción y minearemos un bloque en el nodo 5000, como se muestra en la imagen:

```
{
    "chain": [
        {
            "hash": "51c21bc946f4a10b740bbaf7f0586fada52cbd6a7cc79679df7b2c2988c3a3bd",
            "hash_previo": "1",
            "indice": 1,
            "prueba": 0,
            "timestamp": "0",
            "transacciones": []
        },
        {
            "hash": "0000c58d17f986528737e5246710852c4d25a22a99de0d5e5f3659f122a7517f",
            "hash_previo": "51c21bc946f4a10b740bbaf7f0586fada52cbd6a7cc79679df7b2c2988c3a3bd",
            "indice": 2,
            "prueba": 90078,
            "timestamp": 1660822174.4466178,
            "transacciones": [
                {
                    "cantidad": 5,
                    "destino": "nodoB",
                    "origen": "nodoA",
                    "tiempo": 1660822172.1389124
                },
                {
                    "cantidad": 1,
                    "destino": "192.168.56.1",
                    "origen": "0",
                    "tiempo": 1660822174.4466178
                }
            ]
        },
        {
            "hash": "0000466457e43099e4f3e58af8278acc3910ded3518df24d4dddd19eee0f057b",
            "hash_previo": "0000c58d17f986528737e5246710852c4d25a22a99de0d5e5f3659f122a7517f",
            "indice": 3,
            "prueba": 20849,
            "timestamp": 1660823451.5563853,
            "transacciones": [
                {
                    "cantidad": 5,
                    "destino": "nodoB",
                    "origen": "nodoA",
                    "tiempo": 1660822366.689997
                },
                {
                    "cantidad": 1,
                    "destino": "192.168.56.1",
                    "origen": "0",
                    "tiempo": 1660823451.5563853
                }
            ]
        }
    ],
    "longitud": 3
}
```


Una vez hecho eso, en el nodo 5001 añadiremos una transaccion y mineare-mos un bloque (debería saltar el error mencionado):




## 6.3 Pruebas en la máquina virtual

Se pide realizar las modificaciones necesarias en la ejecución de la aplicación para que en lugar de ejecutarse todo en el entorno windows o mac, que haya un nodo en el entorno windows/mac (host) y al menos otro nodo en una de las máquinas virtuales de Ubuntu (guest). Será necesario aportar pantallazos en el informe para justificar el correcto funcionamiento de la aplicación con **al menos** un nodo host y un nodo guest en la máquina virtual con un ubuntu.

## 6.4 Requests

Se pide, además de los archivos necesarios para el proyecto, crear un fichero Python llamado requests.py que contenga código para enviar peticiones a nuestra api blockchain haciendo uso de la librería de requests. Se puede tomar como ejemplo este código:

```
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)

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)
```

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. Por esa razón, es crítico que se prueben todos los métodos de la aplicación para probar que esta funciona de manera correcta. Se debe documentar el código indicando las pruebas que se están realizando.