UNIVERSIDAD NACIONAL DE INGENIERÍA

FACULTAD DE INGENIERIA ELÉCTRICA Y ELECTRÓNICA


Proyecto de Software

# "WoodSafe: Sistema de Autenticación y Registro de Documentos Digitales"

Versión 1.0

Fecha

01 de Marzo del 2025

Participantes

Huamán de la Cruz Eduardo Jhoseph (20231318K)

Huaytan Laurel Bryan Kiomi (20231252J)

Oscco Silva Johan Benjamin (20231410D)

Curso

Programación Orientada a Objetos (BMA15)

Periodo académico

24-3

Profesor

Tello Canchapoma Yuri Oscar

# Índice

# Introducción

En un mundo en constante digitalización, la salvaguarda de la información digital se ha transformado en una inquietud esencial para personas, empresas y gobiernos. Frente al incremento de riesgos cibernéticos y la creciente cantidad de datos guardados en internet, resulta esencial desarrollar soluciones innovadoras que garanticen la protección y la privacidad de dichos datos.

En estas circunstancias, la tecnología blockchain ha emergido como una opción alentadora para abordar estos desafíos. Debido a su carácter descentralizado, inalterable y claro, blockchain instaura un nuevo modelo en la administración y salvaguarda de la información digital. Su operación se fundamenta en una red peer-to-peer (P2P), en la que varios nodos verifican y guardan transacciones sin requerir la intervención de una autoridad central. Esta organización distribuida no solo reduce la susceptibilidad a ataques, sino que también asegura la accesibilidad de la información.

Un elemento crucial de blockchain es el uso de funciones hash criptográficas, las cuales convierten los datos en códigos singulares e irrevocables. Cada bloque de la cadena posee un hash que se conecta con el bloque previo, garantizando de esta manera la integridad y previniendo alteraciones no permitidas. Cualquier esfuerzo por modificar los datos causaría una variación en el hash, alertando a la red acerca de posibles intentos de infiltración. Esta propiedad transforma a blockchain en un instrumento sumamente fiable para la salvaguarda de datos digitales.

El propósito de este proyecto es investigar el empleo de blockchain como un instrumento crucial para proteger la información digital. Su puesta en marcha puede brindar múltiples ventajas, tales como un incremento en la seguridad, una disminución del peligro de fraude y un acatamiento de las regulaciones más eficaz. Mediante esta investigación, se analizarán las aplicaciones presentes de blockchain en la salvaguarda de datos y se sugerirá un modelo que utilice sus beneficios para incrementar la seguridad y la confianza en la administración de datos digitales.

## Antecedentes

Hoy en día, proteger los datos tanto personales como empresariales es algo que nos preocupa a todos. A medida que cada vez más información se almacena en línea, los riesgos de filtraciones, ciberataques y accesos no autorizados también han aumentado. Recientes investigaciones muestran que estas brechas de seguridad han causado pérdidas millonarias y han puesto en riesgo la privacidad de millones de personas en el mundo. "Se explora la importancia de la protección y seguridad de los datos personales en el campo de la educación mediante las promesas emergentes de los interesados en usar la tecnología blockchain."(Amo et al., 2020)

Uno de los grandes retos en la protección de datos es cómo se centraliza la información. Los sistemas convencionales suelen depender de servidores centralizados, lo que los hace más susceptibles a ataques y fallos.

> La custodia es distribuida, en tal sentido, nadie tiene el control completo de la red, pues diferentes usuarios almacenan distintos nodos que contienen copias actualizada de información; por lo que es tolerable a la falla en algún nodo, pues si alguna parte de blockchain falla, toda la red puede continuar trabajando con la última versión disponible de la información, además que se garantiza que la información nunca se pierda, al encontrarse contenida en varios nodos, por lo que siempre estará segura.(Luna, s.f.)

Blockchain actúa como un registro distribuido que guarda datos de manera segura utilizando técnicas criptográficas avanzadas. Cada vez que se realiza una transacción o se modifica algún dato, múltiples nodos en la red verifican esa acción, eliminando así la posibilidad de alteraciones maliciosas. Además, esta tecnología permite usar contratos inteligentes, que pueden añadir otra capa de seguridad y eficientizar los procesos de protección de datos digitales.

En los últimos años, diversas industrias han empezado a integrar blockchain para reforzar la seguridad de su información. Sectores como la banca, la salud y el comercio electrónico están utilizando esta tecnología para asegurar la integridad y confidencialidad de los datos. Tanto empresas como gobiernos han impulsado iniciativas enfocadas en la protección de la privacidad a través de blockchain.

WoodSafe tiene como objetivo aprovechar las ventajas de blockchain para desarrollar un sistema de protección de datos digitales que sea más robusto, confiable y accesible. Al incorporar esta tecnología, se busca reducir el riesgo de vulnerabilidades y aumentar la confianza en la gestión de información sensible. Implementar blockchain en la protección de datos es un paso clave hacia un futuro más seguro y transparente en el mundo digital.

## Objetivos

Objetivo principal: El objetivo de este proyecto es desarrollar un sistema blockchain mediante el método P2P(peer to pee) y funciones  hash, lo cuál permitirá verificar su autenticidad de forma rápida, segura y eficiente, con aplicaciones prácticas en ámbitos como la educación, empresas, el gobierno y profesionales independientes, promoviendo un impacto positivo al reducir costos, eliminar intermediarios y fortaleces la confianza en los documentos digitales.

Objetivos específicos:

- Implementar un sistema de hash para verificar la autenticidad de documentos.
- Crear un registro inmutable de las modificaciones de documentos.
- Establecer un mecanismo descentralizado de verificación.
- Garantizar la integridad de los datos almacenados.

Diagrama UML

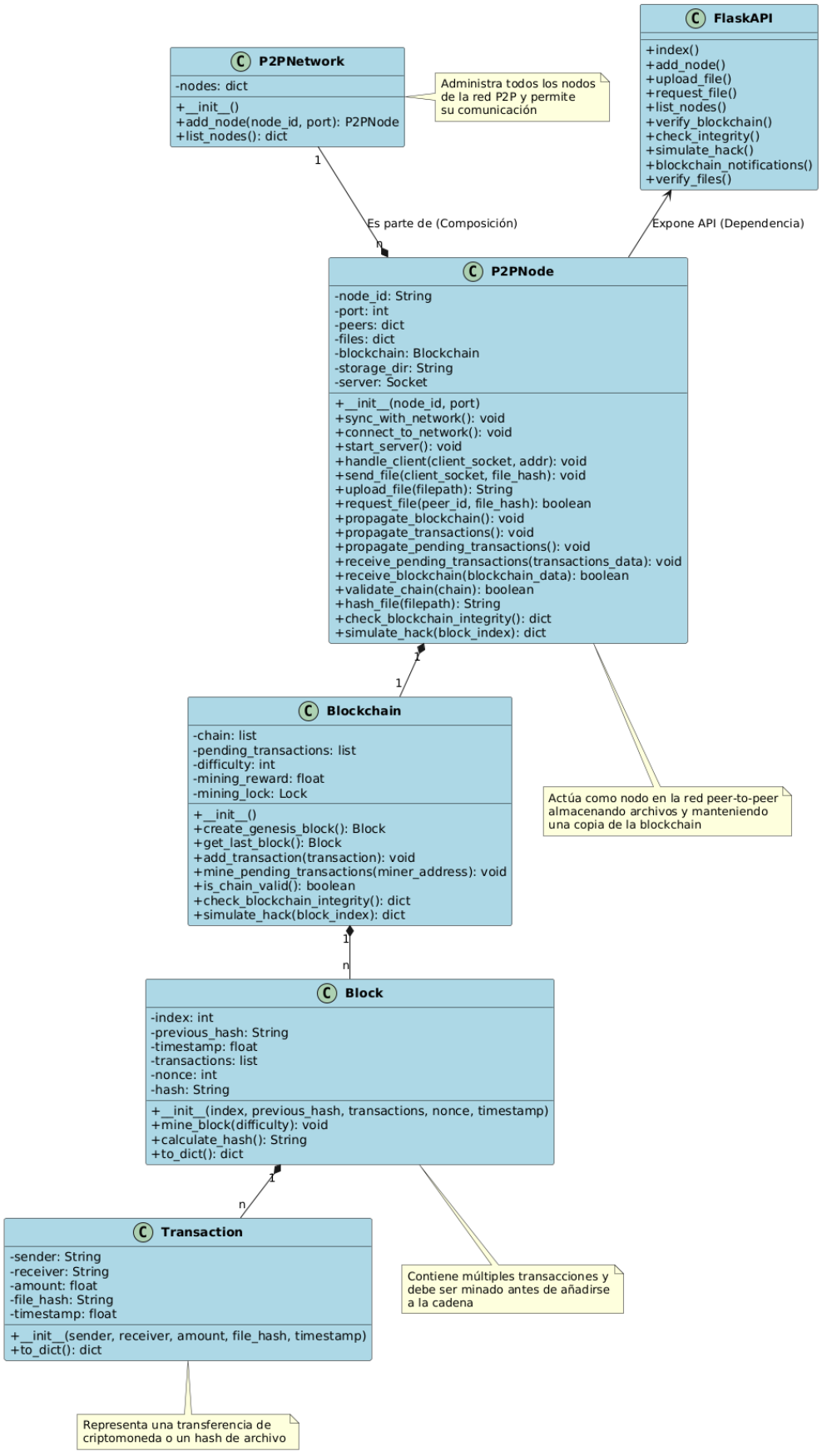
El Lenguaje Unificado de Modelado (UML, por sus siglas en inglés) es un lenguaje estandarizado destinado al desarrollo, donde modelar significa realizar un diseño previo de una aplicación. Es, esencialmente, diseñar la aplicación antes de desarrollarla y llevarla a cabo. Su objetivo es mostrar el diseño de una aplicación y compararlo con los requisitos establecidos antes de que el grupo de desarrollo comience a programar, donde vemos la agregación, composición, herencia y dependencia ***(ver figura 2).*** Por lo tanto, en este proyecto aplicaremos el UML para poder representar y dar una idea de nuestro diseño de sistema de Blockchain P2P para la transferencia de archivos.

El sistema este compuesto por cinco clases, los cuales en esta clase tiene atributos y métodos que interactúan entre sí ***(ver figura 1)***:

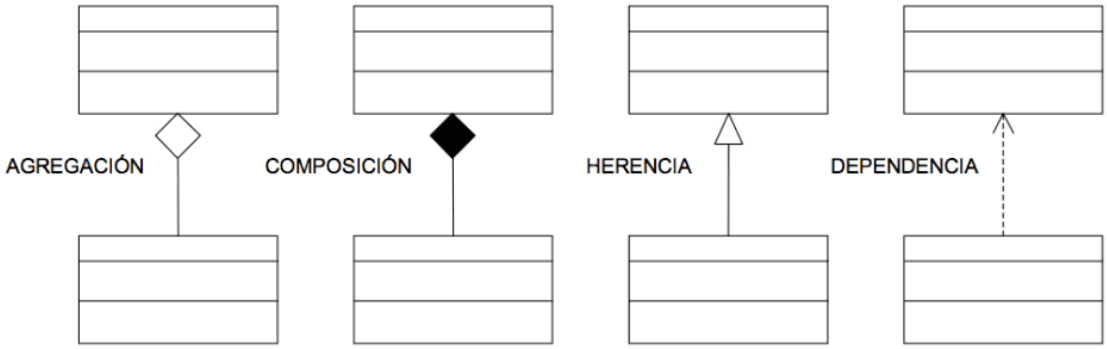
1. **Transaction**: Representa las operaciones realizadas en la red.
2. **Block**: Agrupa múltiples transacciones y forma parte de la cadena.
3. **Blockchain**: Estructura que contiene la cadena completa de bloques.
4. **P2PNode**: Representa cada nodo en la red peer-to-peer.
5. **P2PNetwork**: Gestiona la red completa de nodos.
6. **FlaskAPI**: Proporciona una interfaz web para interactuar con el sistema.

1. **Relación entre Blockchain y Block**:
    - Tipo: Asociación de composición (*Blockchain* contiene múltiples *Block*).
    - Descripción: Cada instancia de Blockchain mantiene una lista de bloques. Esta es una relación fuerte, ya que la clase de los bloques depende directamente de la existencia de la cadena de bloques.
2. **Relación entre Block y Transaction**:
    - Tipo: Asociación de composición (*Block* contiene múltiples *Transaction*).
    - Descripción: Cada bloque contiene una lista de transacciones. Las transacciones no tienen si no hay un bloque, lo que define esta relación de composición.
3. **Relación entre P2PNode y Blockchain**:
    - Tipo: Asociación de composición (*P2PNode* contiene exactamente una instancia de *Blockchain*).
    - Descripción: Cada nodo P2P mantiene su propia copia de la cadena de bloques, reforzando la idea de descentralización del sistema.
4. **Relación entre P2PNetwork y P2PNode**:
    - Tipo: Asociación de composición (*P2PNetwork* contiene múltiples *P2PNode*).
    - Descripción: La red P2P administra y contiene múltiples nodos, indicando una relación en la que los nodos son parte integral de la red.
5. **Relación entre FlaskAPI y P2PNode**:
    - Tipo: Dependencia (*FlaskAPI* depende de *P2PNode* para exponer servicios API).
    - Descripción: La clase FlaskAPI utiliza las funcionalidades de P2PNode para proporcionar una interfaz web a través de la cual se pueden realizar acciones como subir archivos, solicitar archivos y verificar la integridad de la blockchain.

**Diagrama de Clases - Sistema de Blockchain P2P para Transferencia de Archivos**




***(FIGURA 1)***




***(FIGURA 2)***

## DESCRIPCIÓN DE LAS CLASES:

**1.- Transaction :** Representa una transferencia de valor o información entre dos entidades en la red.

**Atributos principales:**

- ***sender:*** Identificador del Emisor
- ***receiver:*** Identificador del Receptor
- ***amount:*** Cantidad de valor transferido
- ***file_hash:*** Hash del archivo transferido (opcional)
- ***timestamp:*** Marca de tiempo de la transacción

**Métodos principales:**

- ***__init__(sender, receiver, amount, file_hash):*** Constructor
- ***to_dict():*** Convierte la transacción a un diccionario para su serialización

**Propósito**: Las transacciones son los componentes esenciales de la blockchain, ya que registran todas las transferencias de archivos o las recompensas obtenidas por el minado en el sistema.

**2.- Block:** Agrupa varias transacciones y se integra en la cadena de bloques.

**Atributos principales:**

- ***index:*** Posición del bloque en la cadena
- ***previous_hash:*** Hash del bloque anterior
- ***timestamp:*** Momento de creación del bloque
- ***transactions:*** Lista de transacciones incluidas

- ***nonce:*** Número utilizado en el proceso de minado
- ***hash:*** Hash que identifica al bloque

**Métodos principales:**

- ***__init__(index, previous_hash, transactions, nonce):*** Constructor
- ***mine_block(difficulty):*** Realiza el proceso de prueba de trabajo
- ***calculate_hash():*** Calcula el hash del bloque
- ***to_dict():*** Convierte el bloque a un diccionario

**Propósito**: Los bloques son contenedores de transacciones que, una vez minados, se añaden a la cadena de forma permanente, creando un registro inmutable.

### 3.- Blockchain: Estructura de datos que contiene la cadena completa de bloques.

**Atributos principales:**

- ***chain:*** Lista ordenada de bloques
- ***pending_transactions:*** Transacciones pendientes de incluir en un bloque
- ***difficulty:*** Dificultad del algoritmo de prueba de trabajo
- ***mining_reward:*** Recompensa por minar un bloque
- ***mining_lock:*** Bloqueo para evitar minería simultánea.

**Métodos principales:**

- ***__init__():*** Constructor
- ***create_genesis_block():*** Crea el bloque inicial
- ***get_last_block():*** Obtiene el último bloque de la cadena
- ***add_transaction(transaction):*** Añade una transacción pendiente
- ***mine_pending_transactions(miner_address):*** Mina las transacciones pendientes
- ***is_chain_valid():*** Verifica la integridad de la cadena

**Propósito**: La blockchain ofrece un registro que no se puede modificar y que está descentralizado, donde se registran todas las transacciones realizadas en la red.

### 4.- P2PNode: Representa un nodo individual en la red peer-to-peer.

**Atributos principales:**

- ***node_id:*** Identificador único del nodo
- ***port:*** Puerto en el que escucha el nodo
- ***peers:*** Diccionario de nodos conectados
- ***files:*** Diccionario de archivos almacenados
- ***blockchain:*** Instancia de Blockchain
- ***storage_dir:*** Directorio de almacenamiento
- ***server:*** Socket del servidor

**Métodos principales:**

- ***__init__(node_id, port):*** Constructor
- ***sync_with_network:*** Sincroniza el nodo con la red para obtener la blockchain más actualizada.
- ***connect_to_network():*** Conecta el nodo a la red
- ***start_server():*** Inicia el servidor de escucha
- ***handle_client(client_socket, addr):*** Maneja conexiones entrantes
- ***send_file(client_socket, file_hash):*** Envía un archivo a otro nodo
- ***upload_file(filepath):*** Sube un archivo al nodo
- ***request_file(peer_id, file_hash):*** Solicita un archivo a otro nodo
- ***propagate_blockchain():*** Distribuye la blockchain a otros nodos
- ***propagate_transactions():*** Propaga transacciones pendientes a otros nodos.
- ***propagate_pending_transactions():*** Propaga solo las transacciones que aún no han sido incluidas en un bloque.
- ***Receive_pending_transactions(transactions_data):*** Agrega transacciones recibidas a las pendientes.
- ***receive_blockchain(blockchain_data):*** Recibe actualizaciones de la blockchain
- ***validate_chain(chain):*** Valida una cadena de bloques
- ***hash_file(filepath):*** Calcula el hash de un archivo
- ***check_blockchain_integrity():*** Verifica la integridad de la blockchain
- ***simulate_hack(block_index):*** Simula un intento de alteración

**Propósito**: Los nodos P2P son los participantes activos de la red, que tienen la capacidad de almacenar archivos, transferirlos y mantener una copia de la blockchain.

### 5.- P2PNetwork : Gestiona la red completa de nodos P2P.

**Atributos principales:**

- ***nodes:*** Diccionario de nodos en la red

**Métodos principales:**

- ***__init__():*** Constructor
- ***add_node(node_id, port):*** Añade un nodo a la red
- ***list_nodes():*** Lista todos los nodos activos

**Propósito**: La red P2P coordina todos los nodos, facilitando su registro y gestión.

### 6.- FlaskAPI: Proporciona una interfaz web para interactuar con el sistema.

**Métodos principales:**

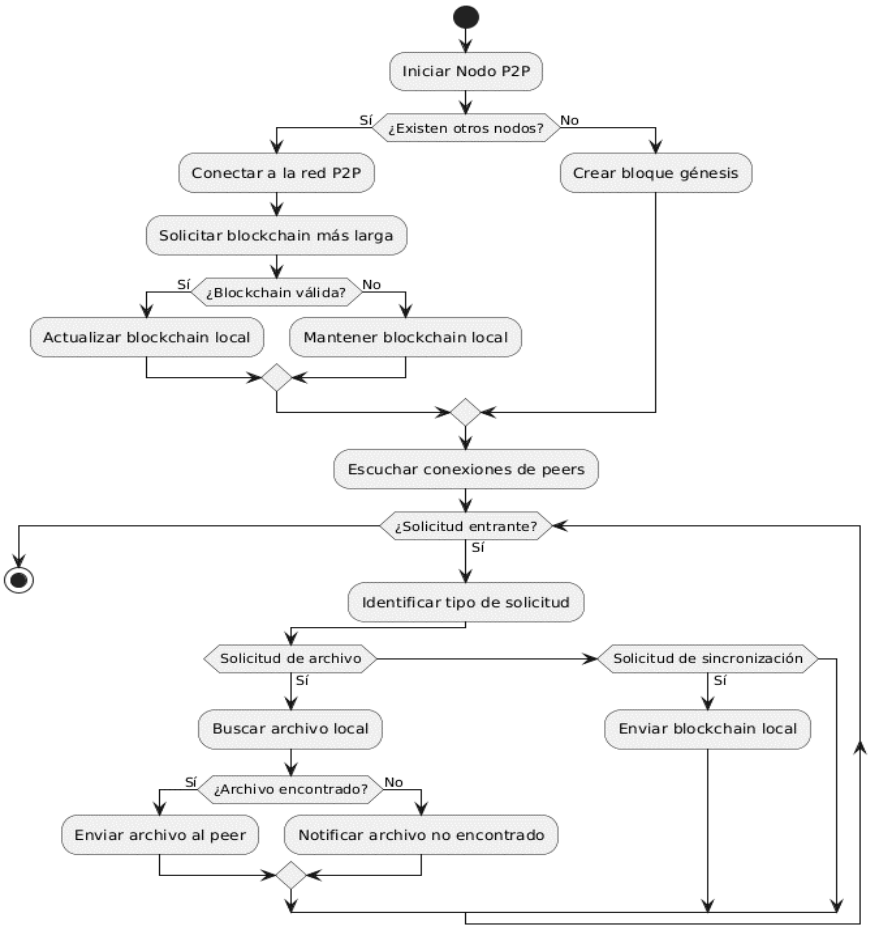
- ***index():*** Página principal
- ***add_node():*** Añade un nuevo nodo
- ***upload_file():*** Sube un archivo a un nodo
- ***request_file():*** Solicita un archivo entre nodos
- ***list_nodes():*** Lista los nodos disponibles

- ***verify_blockchain():*** Verifica la validez de la blockchain
- ***check_integrity():*** Comprueba la integridad de la blockchain
- ***simulate_hack():*** Simula un intento de modificación
- ***blockchain_notifications():*** Obtiene notificaciones de seguridad
- ***verify_files():***Verifica la integridad de los archivos almacenados en los nodos.

**Propósito**: La API proporciona una capa de abstracción para interactuar con el sistema a través de una interfaz web.

## DIAGRAMA DE ACTIVIDADES:

Representa el flujo de trabajo, en donde las acciones, decisiones y secuencia de ejecución. Son útiles para ilustrar procesos complejos porque hacen posible identificar el comportamiento del sistema y sus interacciones con eficacia. En nuestro sistema blockchain crear un diagrama de actividades simplifica la comprensión del flujo de las operaciones del nodo P2P (peer to peer) e ilustra los puntos donde el sistema decide, como la validación de la cadena de bloques o la búsqueda de archivos, y ordena el procedimiento que sigue el nodo desde el inicio de su funcionamiento hasta el comportamiento de gestión de solicitudes y sirve como esquema que guía la estructuración del código y la implementación, asegurando que se incluye cualquier situación que pueda producirse***(ver figura 3).***



***(FIGURA 3)***

## FLUJO DE OPERACIÓN

**1. Inicio del Nodo P2P**

El nodo comienza su ejecución con el proceso de inicialización. En este punto, el nodo verifica si existen otros nodos en la red.

- **¿Existen otros nodos?**
  - **Sí:** El nodo se conecta a la red P2P y solicita la blockchain más larga disponible para asegurar la coherencia de la información.
    - **¿Blockchain válida?**
      - **Sí:** La blockchain local se actualiza con la versión más larga obtenida.
      - **No:** El nodo mantiene su blockchain local, ya que no recibió una versión válida.
  - **No:** El nodo crea el bloque génesis, inicializando una nueva blockchain.

**2. Escucha de Conexiones de Peers**

Una vez conectado a la red (o tras crear el bloque génesis), el nodo comienza a escuchar conexiones entrantes de otros peers.

**3. Gestión de Solicitudes Entrantes**

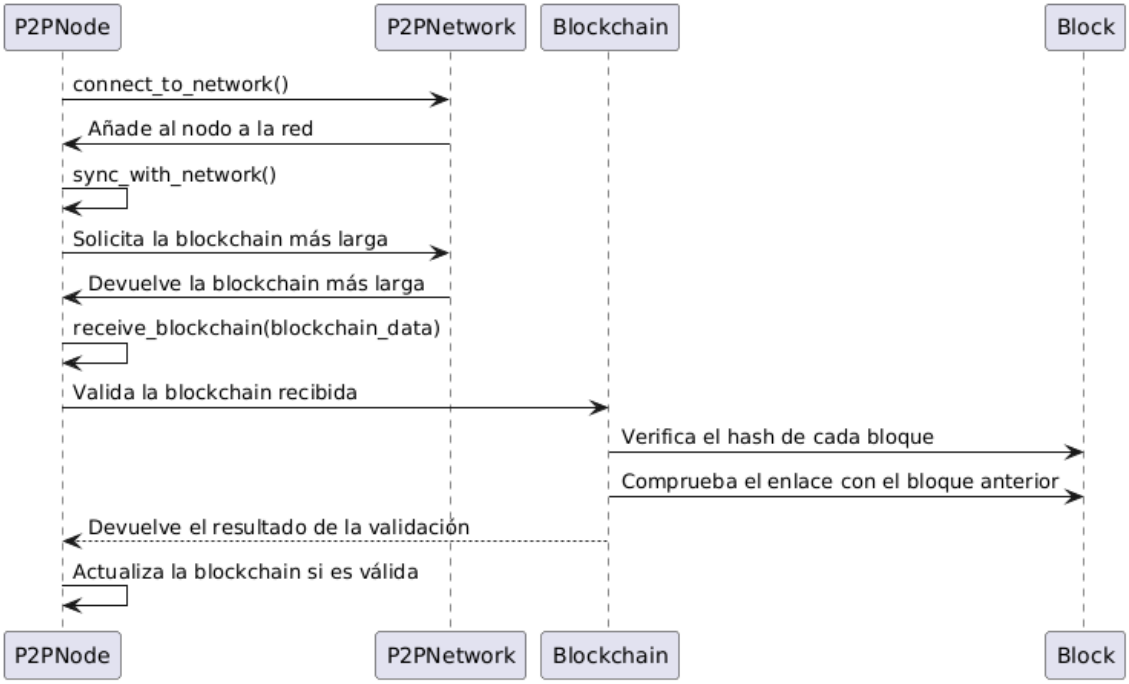
Cuando se recibe una solicitud, el nodo identifica el tipo de solicitud y ejecuta la acción correspondiente.

- **Solicitud de archivo:**
  - El nodo busca el archivo solicitado en su almacenamiento local.
    - **¿Archivo encontrado?**
      - **Sí:** El archivo es enviado al peer solicitante.
      - **No:** Se notifica al peer que el archivo no fue encontrado.
- **Solicitud de sincronización:**
  - El nodo envía su versión local de la blockchain al peer solicitante para mantener la coherencia de la red.

## DIAGRAMA DE SECUENCIA:

Es un diagrama que ilustra cómo interactúan los diferentes componentes de un sistema en un orden específico. En este caso, el diagrama de secuencia muestra la interacción entre el nodo P2P, la red P2P, la blockchain y los bloques durante el proceso de conexión y sincronización***.***
***(VER FIGURA 4)***

- **Conexión a la red**: El nodo P2P llama al método connect_to_network(), y la red P2P añade el nodo a la red.
- **Sincronización:** El nodo ejecuta sync_with_network(), solicitando la blockchain más larga disponible.
- **Recepción y validación:** El nodo recibe la blockchain, la valida comprobando el hash de cada bloque y verificando el enlace con el bloque anterior.
- **Actualización:** Si la blockchain recibida es válida, el nodo actualiza su versión local.




***(FIGURA 4)***

Realiza la siguiente secuencia:

- **P2PNode -> P2PNetwork: connect_to_network()**: El nodo intenta conectarse a la red P2P.
- **P2PNetwork -> P2PNode: Añade al nodo a la red**: La red lo acepta y lo añade a la lista de nodos activos.
- **P2PNode -> P2PNode: sync_with_network()**: El nodo inicia el proceso de sincronización de su blockchain.
- **P2PNode -> P2PNetwork: Solicita la blockchain más larga**: Pide la copia más actualizada y extensa de la blockchain.
- **P2PNetwork -> P2PNode: Devuelve la blockchain más larga**: La red le responde con la mejor versión de la blockchain.
- **P2PNode -> P2PNode: receive_blockchain(blockchain_data)**: El nodo recibe esos datos y los prepara para validación.
- **P2PNode -> Blockchain: Valida la blockchain recibida**: Pide a su módulo de blockchain que verifique la validez de la cadena.
- **Blockchain -> Block: Verifica el hash de cada bloque**: Cada bloque es revisado para asegurarse de que el hash es correcto.

- **Blockchain -> Block: Comprueba el enlace con el bloque anterior**: También se revisa que cada bloque esté correctamente encadenado con el anterior.
- **Blockchain --> P2PNode: Devuelve el resultado de la validación**: El resultado de la validación (válida o no válida) se devuelve al nodo.
- **P2PNode -> P2PNode: Actualiza la blockchain si es válida**: Si la blockchain es válida, el nodo actualiza su copia local.

Clases y Diseño del Código

El proyecto por presentar,"WoodSafe", emplea diversas bibliotecas y herramientas de Python para su correcto funcionamiento al querer cumplir con su objetivo de resguardar los datos digitales que se encomienden. Cada biblioteca tendrá una función específica, que a continuación se explicará detalladamente para identificarlos.

- Hashlib: Ayuda en la elaboración de de hashes criptográficos, utilizados en la verificación de datos y su seguridad.
- Json: Permite trabajar con datos en formato JSON, convirtiéndolos entre diccionarios de Python y cadenas JSON.
- Time: Nos da funciones relacionadas con la medición y manipulación del tiempo, lo que permite trabajar con marcas de tiempo, pausas en la ejecución del programa y conversiones entre diferentes formatos de tiempo.
- Threading: Nos facilita la ejecución de varias tareas de manera simultánea en un mismo programa.
- Socket: Proporciona funcionalidad para la comunicación en red mediante sockets, permitiendo la conexión entre dispositivos o procesos.
- Os: Nos permite interactuar con el sistema operativo, como leer variables de entorno, trabajar con archivos y ejecutar comandos del sistema.
- Flask: Framework para desarrollar aplicaciones web con Python.
    - Render_template: Permite renderizar archivos HTML en Flask.
    - Request: Maneja datos enviados por el cliente en una solicitud HTTP.
    - Jsonify: Convierte datos de Python a JSON para ser enviados como respuestas HTTP.

```python
import hashlib
import json
import time
import threading
import socket
import os
from flask import Flask, render_template, request, jsonify
```

Para este proyecto, usaremos el paradigma de Programación Orientada a Objetos, que gracias a su naturaleza de clases, nos ayudará a poder programar de manera ordenada y con las ventajas de este paradigma.

1. Clase “Transaction”: En esta clase, se va a representar a lo que estamos llamando una “transacción”, se crea una transacción con *sender(*remitente de la transacción*), receiver(*el receptor de la transacción*), amount(*valor que se transfiere*), file_hash(*servirá como prueba de integridad, en caso se modifique algo*) y timestamp*. Luego se almacenará la información en forma de diccionario usando *to_dict(),* luego, si la transacción conlleva consigo un archivo, se usará su hash para verificar su seguridad.

```python
# Clase para representar una transacción
class Transaction:
    def __init__(self, sender, receiver, amount, file_hash=None, timestamp=None):  # Cambio aquí
        self.sender = sender
        self.receiver = receiver
        self.amount = amount
        self.file_hash = file_hash
        self.timestamp = timestamp if timestamp is not None else time.time()  # Nueva lógica

```

2. Clase “Block”:

```python
# Clase para representar un bloque
class Block:
    def __init__(self, index, previous_hash, transactions, nonce=0, timestamp=None):  # Cambio aquí
        self.index = index
        self.previous_hash = previous_hash
        self.timestamp = timestamp if timestamp is not None else time.time()  # Nueva línea
        self.transactions = transactions
        self.nonce = nonce
        self.hash = self.calculate_hash()

    def mine_block(self, difficulty):
        target = '0' * difficulty
        while self.hash[:difficulty] != target:
            self.nonce += 1
            self.hash = self.calculate_hash()
        print(f"Bloque minado: {self.hash}")

    def calculate_hash(self):
        block_string = json.dumps({
            "index": self.index,
            "previous_hash": self.previous_hash,
            "timestamp": self.timestamp,
            "transactions": [tx.to_dict() for tx in self.transactions],
            "nonce": self.nonce
        }, sort_keys=True).encode()
        return hashlib.sha256(block_string).hexdigest()

    def to_dict(self):
        return {
            "index": self.index,
            "previous_hash": self.previous_hash,
            "timestamp": self.timestamp,
            "transactions": [tx.to_dict() for tx in self.transactions],
            "nonce": self.nonce,
            "hash": self.hash
        }
```

En esta clase, vamos a representar lo que es un bloque, para próximo aplicarse al Blockchain. Que también irá ligado a un hash criptográfico.
Lo que contiene en cada bloque es:

- Index: Posición del bloque en la cadena.
- Previous_hash: El hash del bloque anterior, para asegurar la conexión de estos.

- Transactions: Lista de transacciones registradas en el bloque.
- Nonce: Número que se ajusta durante la minería para cumplir con la dificultad.
- Timestamp: Fecha y hora de la creación del bloque.
- Hash: Identificador único de cada bloque.

a) Método *mine_block():* Para que se pueda añadir un bloque a la cadena, debe cumplir con el algoritmo ya planteado.
Se debe ajustar el *nonce* hasta encontrar un hash que comience con cierta cantidad de `0`. Cuanto mayor la dificultad (difficulty), más ceros requiere el hash y más tiempo lleva encontrarlo.

b) Método *to_dict()*: Este método convierte el bloque en un diccionario, útil para almacenarlo o enviarlo por la red.

3. Clase “Blockchain”:

```python
# Clase para representar la blockchain
class Blockchain:
    def __init__(self):
        self.chain = [self.create_genesis_block()]
        self.pending_transactions = []
        self.difficulty = 4
        self.mining_reward = 10
        self.mining_lock = threading.Lock()

    def create_genesis_block(self):
        return Block(0, "0", [])

    def get_last_block(self):
        return self.chain[-1]

    def add_transaction(self, transaction):
        if transaction not in self.pending_transactions:
            self.pending_transactions.append(transaction)

    def mine_pending_transactions(self, miner_address):
        with self.mining_lock:
            if not self.pending_transactions:
                return

            block = Block(len(self.chain), self.get_last_block().hash, self.pending_transactions.copy())
            block.mine_block(self.difficulty)
            self.chain.append(block)
            self.pending_transactions = [Transaction("SYSTEM", miner_address, self.mining_reward)]

    def is_chain_valid(self):
        for i in range(1, len(self.chain)):
            current_block = self.chain[i]
            previous_block = self.chain[i - 1]

            if current_block.hash != current_block.calculate_hash():
                return False

            if current_block.previous_hash != previous_block.hash:
                return False

        return True
```

Esta clase representa una cadena de bloques, que será el sistema descentralizado que permitirá almacenar transacciones de manera segura.

a) Constructor *_init_():* Iniciar Blockchain.
   - Chain: Lista que almacenará los bloques de la blockchain.
   - Pending_transactions: Transacciones que aún no se han incluido en un bloque.
   - Difficulty: Va a determinar cuantos ceros iniciales debe tener el hash del bloque minado.

- Mining_reward: Recompensa por minar un bloque (en este caso, 10 unidades).
- Mining_lock: Previene problemas cuando múltiples mineros intentan minar simultáneamente.

b) Método "*create_genesis_block()":* Método para crear el bloque génesis, que será, el primer bloque de la cadena y no tendrá transacciones ni un bloque anterior.

c) Método "*get_last_block*()": Obtención del último bloque.

d) Método "*add_transaction()*" : Agregar transacciones.
Recibe una transacción y la añade a la lista de transacciones pendientes, lo que nos ayudará a evitar duplicados antes de incluirse en algún bloque.

e) Método "*mine_pending_transactions()":* Minar un bloque.
El proceso de minería verifica si hay transacciones pendientes, crea un nuevo bloque con un índice basado en el tamaño de la blockchain, el hash del bloque anterior y una copia de las transacciones pendientes, *ejecuta mine_block(difficulty)* ajustando el *nonce* hasta encontrar un hash válido, añade el bloque minado a la cadena y otorga una recompensa de 10 unidades al minero mediante una nueva transacción del sistema.

f) Método "*is_chain_valid()*" : Validación de la Blockchain.
El método *is_chain_valid()* verifica la integridad de la blockchain asegurando que cada bloque tenga un hash correcto y que esté correctamente enlazado con el bloque anterior. Si algún hash ha sido alterado o el enlace entre bloques es inválido, la cadena se considera comprometida.

4. Clase "P2PNode": La clase P2PNode representa un nodo en una red peer-to-peer (P2P)**,** permitiendo la comunicación entre nodos, almacenamiento de archivos y sincronización de la blockchain.

```python
# Clase para representar un nodo P2P
class P2PNode:
    nodes = {}

    def __init__(self, node_id, port):
        self.node_id = node_id
        self.port = port
        self.peers = {}
        self.files = {}
        self.blockchain = Blockchain()
        self.storage_dir = f"node_{node_id}_files"
        os.makedirs(self.storage_dir, exist_ok=True)

        self.server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.server.bind(("127.0.0.1", self.port))
        self.server.listen(5)
        P2PNode.nodes[node_id] = self
        self.connect_to_network()
        self.sync_with_network()
        print(f"Nodo {node_id} creado en puerto {port}")
        print(f"Directorio de almacenamiento: {self.storage_dir}")

        threading.Thread(target=self.start_server, daemon=True).start()

    def sync_with_network(self):
        """Sincroniza con la blockchain más larga al iniciar"""
        if len(P2PNode.nodes) > 1:
            longest_chain = []
            for node in P2PNode.nodes.values():
                if len(node.blockchain.chain) > len(longest_chain):
                    longest_chain = node.blockchain.chain
            if longest_chain:
                self.receive_blockchain([block.to_dict() for block in longest_chain])

    def connect_to_network(self):
        for node_id, node in P2PNode.nodes.items():
            if node_id != self.node_id:
                self.peers[node_id] = node.port
                node.peers[self.node_id] = self.port
        print(f"🔗 Nodo {self.node_id} conectado a {len(self.peers)} peers")
```

a) Atributos principales:
   - Nodes = Diccionario estático que mantiene un registro de todos los nodos en la red.
   - port: Puerto donde el nodo escucha conexiones.
   - peers: Diccionario que almacena los nodos conectados.
   - files: Diccionario con los archivos almacenados en el nodo.
   - blockchain: Instancia de la blockchain del nodo.
   - storage_dir: Directorio donde el nodo guarda archivos.

b) Configuración del servidos P2P: Configura un servidor TCP en 127.0.0.1 y el puerto asignado, permitiendo reutilizar direcciones ocupadas con SO_REUSEADDR y aceptando hasta 5 conexiones simultáneas con listen(5).

   Resumen del bloque: Cada nodo tiene su propia blockchain, se conecta a otros nodos en la red, sincroniza su cadena con la más larga para mantener la coherencia, actúa como servidor TCP para la comunicación entre pares y almacena archivos localmente junto con una lista de transacciones.

```python
def start_server(self):
    print(f"[Nodo {self.node_id}] Servidor iniciado en puerto {self.port}")
    while True:
        client_socket, addr = self.server.accept()
        threading.Thread(target=self.handle_client, args=(client_socket, addr), daemon=True).start()

def handle_client(self, client_socket, addr):
    print(f"[Nodo {self.node_id}] Conectado con {addr}")
    try:
        data = client_socket.recv(4096).decode()
        if not data:
            return

        command, *args = data.split("::")
        if command == "REQUEST_FILE":
            requested_hash = args[0]
            if requested_hash in self.files:
                filepath = self.files[requested_hash]
                if os.path.exists(filepath):
                    self.send_file(client_socket, requested_hash)
                else:
                    client_socket.sendall("FILE_NOT_FOUND".encode())
            else:
                client_socket.sendall("FILE_NOT_FOUND".encode())
    except Exception as e:
        print(f"[Nodo {self.node_id}] Error en handle_client: {e}")
    finally:
        client_socket.close()

def send_file(self, client_socket, file_hash):
    filepath = self.files[file_hash]
    filename = os.path.basename(filepath)
    filesize = os.path.getsize(filepath)

    header = f"FILE::{filename}::{filesize}::{file_hash}"
    client_socket.sendall(header.encode())

    confirmation = client_socket.recv(1024).decode()
    if confirmation != "READY":
        return

    with open(filepath, "rb") as f:
        while chunk := f.read(4096):
            client_socket.sendall(chunk)
```

(Bloque 2)

*c)* Método “*start_server()*” : El servidor TCP se inicia en el puerto asignado, aceptando conexiones entrantes y gestionando cada cliente en un hilo separado. Cuando un cliente se conecta, el nodo recibe su solicitud y, si el comando es "*REQUEST_FILE*", busca el archivo solicitado verificando su existencia en *self.files*. Si el archivo está disponible, se envía su información

(nombre, tamaño y hash) al cliente y se espera la confirmación "READY" antes de proceder con la transferencia. Si el archivo no se encuentra, el nodo responde con "*FILE_NOT_FOUND*". Finalmente, cualquier error es capturado y la conexión se cierra correctamente.

*d)* Método "*handle_client(client_socket, addr*)" : Maneja la comunicación con cada cliente conectado, recibe los datos con g, analiza el mensaje y, si el comando es "REQUEST_FILE", obtiene el hash del archivo solicitado, verifica si está en self.files y si existe en el sistema. Si el archivo está disponible, lo envía con send_file(), de lo contrario, responde con "FILE_NOT_FOUND". Finalmente, captura errores y cierra la conexión en *finally*.

*e)* Método "*send_file(client_socket, file_hash)":* Prepara el archivo para su envío, obtiene su nombre y tamaño con *os.path.basename()* y *os.path.getsize(),* envía un encabezado con la información del archivo (nombre, tamaño y hash) y espera la confirmación del cliente ("*READ*Y") antes de proceder con la transferencia.

```
def propagate_transactions(self):
    chain_data = [block.to_dict() for block in self.blockchain.chain]
    for peer_id in self.peers:
        if peer_id in P2PNode.nodes:
            peer_node = P2PNode.nodes[peer_id]
            try:
                peer_node.receive_blockchain(chain_data)
            except Exception as e:
                print(f"Error propagando a {peer_id}: {str(e)}")

def propagate_blockchain(self):
    print(f"🔄 Propagando blockchain desde {self.node_id}")
    chain_data = [block.to_dict() for block in self.blockchain.chain]

    for peer_id in self.peers:
        if peer_id in P2PNode.nodes:
            peer_node = P2PNode.nodes[peer_id]
            print(f"📤 Enviando blockchain a {peer_id}")
            try:
                peer_node.receive_blockchain(chain_data)
            except Exception as e:
                print(f"Error propagando a {peer_id}: {str(e)}")

def upload_file(self, filepath):
    if not os.path.exists(filepath):
        return None

    file_hash = self.hash_file(filepath)
    if file_hash in self.files:
```

(Bloque 3)

f) Método "*propagate_transactions()":* Convierte la blockchain a un formato serializable, almacenando los bloques como diccionarios *en chain_data*, recorre la lista de nodos conectados (peers) y verifica si existen en P2PNode.nodes, luego llama *a receive_blockchain()* en cada nodo para enviar la blockchain actualizada y maneja errores para evitar que fallos en un nodo afecten la propagación.

g) Método "*propagate_blockchain()":* Muestra un mensaje indicando que se está propagando la blockchain, la convierte en una lista de diccionarios (*chain_data*) para compartirla entre nodos, recorre la lista de *peers* y les envía

la blockchain para asegurar que todos mantengan la misma versión, e incluye manejo de errores para evitar fallos en la propagación.

h) Método "upload_file(filepath)": Verifica si el archivo existe antes de continuar, calcula su hash con *self.hash_file(filepath)* para identificarlo de forma única y, si ya está en s*elf.files,* devuelve su hash en lugar de volver a subirlo.

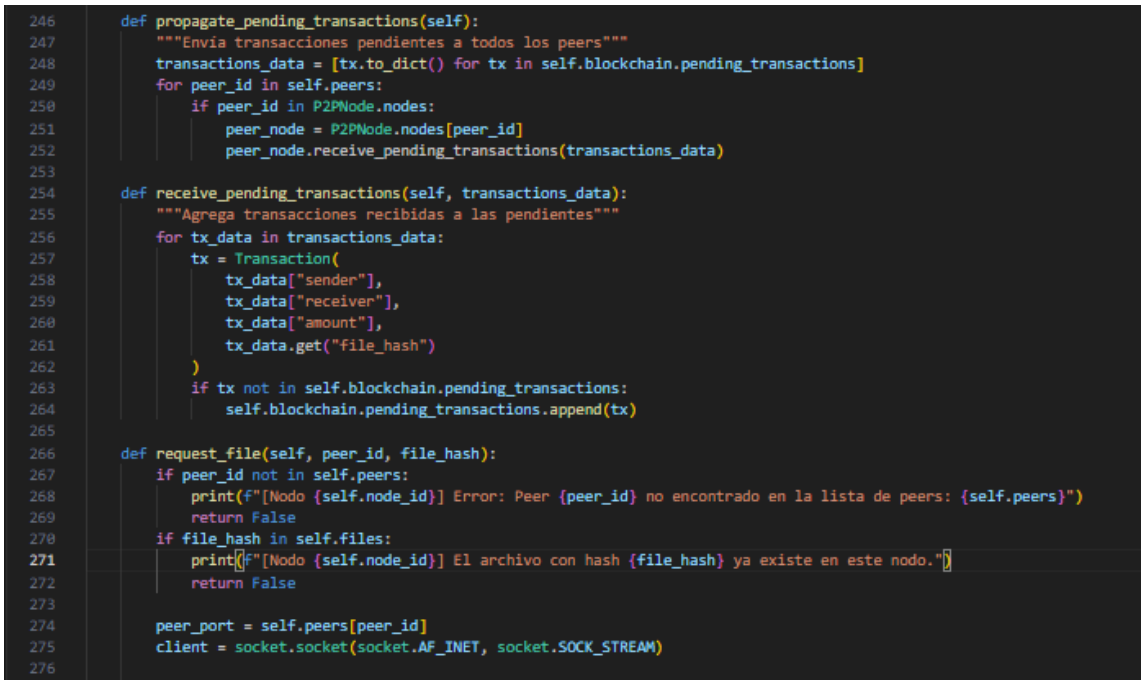
```
        # Copiar archivo al nodo
        dest_path = os.path.join(self.storage_dir, os.path.basename(filepath))
        with open(filepath, "rb") as src_file, open(dest_path, "wb") as dest_file:
            dest_file.write(src_file.read())

        self.files[file_hash] = dest_path

        # Crear transacción y minar bloque
        transaction = Transaction(self.node_id, "NETWORK", 1, file_hash)
        self.blockchain.pending_transactions = [
            tx for tx in self.blockchain.pending_transactions
            if tx.sender != "SYSTEM"
        ]
        self.blockchain.add_transaction(transaction)
        self.blockchain.mine_pending_transactions(self.node_id)
        time.sleep(1)  # Esperar 1 segundo para sincronizar
        # Propagar blockchain y transacciones pendientes
        self.propagate_blockchain()
        self.propagate_pending_transactions()  # Nuevo método para propagar transacciones

        return file_hash
```

(Bloque 4)

i) Copiar archivo al nodo: Genera la ruta de destino (*dest_path*) dentro del almacenamiento del nodo (*self.storage_dir*), abre el archivo original (*src_file*) en modo lectura binaria ("rb") y crea una copia en *dest_path* en modo escritura binaria ("wb"), luego guarda la referencia del archivo en self.files, asociando su hash con la ubicación en disco.

j) Crear transacción y minar bloque: Crea una transacción donde el nodo (*self.node_id*) envía 1 unidad a "NETWORK", vinculándola con el hash del archivo, elimina transacciones del sistema (*"SYSTEM*") de la lista de transacciones pendientes para evitar duplicados, añade la transacción a la lista de pendientes y mina un bloque para registrarla en la blockchain, luego espera 1 segundo (*time.sleep(1*)) para permitir la sincronización con otros nodos antes de la propagación.

k) Propagar blockchain y transacciones: Llama *a propagate_blockchain*() para compartir la blockchain actualizada con otros nodos y a *propagate_pending_transactions()* para distribuir transacciones aún no confirmadas en la red.



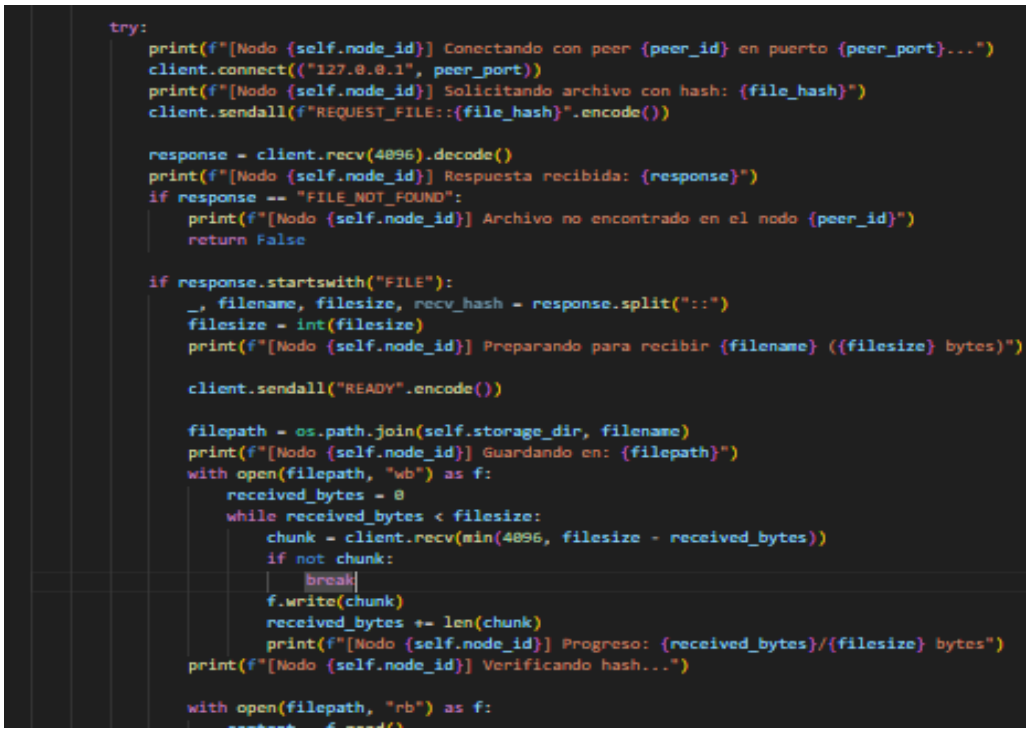
```
246     def propagate_pending_transactions(self):
247         """Envia transacciones pendientes a todos los peers"""
248         transactions_data = [tx.to_dict() for tx in self.blockchain.pending_transactions]
249         for peer_id in self.peers:
250             if peer_id in P2PNode.nodes:
251                 peer_node = P2PNode.nodes[peer_id]
252                 peer_node.receive_pending_transactions(transactions_data)
253 
254     def receive_pending_transactions(self, transactions_data):
255         """Agrega transacciones recibidas a las pendientes"""
256         for tx_data in transactions_data:
257             tx = Transaction(
258                 tx_data["sender"],
259                 tx_data["receiver"],
260                 tx_data["amount"],
261                 tx_data.get("file_hash")
262             )
263             if tx not in self.blockchain.pending_transactions:
264                 self.blockchain.pending_transactions.append(tx)
265 
266     def request_file(self, peer_id, file_hash):
267         if peer_id not in self.peers:
268             print(f"[Nodo {self.node_id}] Error: Peer {peer_id} no encontrado en la lista de peers: {self.peers}")
269             return False
270         if file_hash in self.files:
271             print(f"[Nodo {self.node_id}] El archivo con hash {file_hash} ya existe en este nodo.")
272             return False
273 
274         peer_port = self.peers[peer_id]
275         client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
276 
```

(Bloque 5)

l) Ropagación de transacciones pendientes: Convierte las transacciones pendientes en formato serializable (*transactions_data)* para enviarlas a otros nodos, recorre la lista de *peers* (nodos conectados) y verifica si existen en P2PNode.nodes para evitar errores, y llama *a receive_pending_transactions()* en cada nodo para que reciban y almacenen las transacciones pendientes.

m) Recepción de transacciones pendientes: Recorre la lista de transacciones recibidas (*transactions_data*) y las convierte en objetos *Transaction,* extrae los datos de la transacción (remitente, receptor, cantidad y hash del archivo si existe) y las añade a la lista de transacciones pendientes solo si no están duplicadas.

n) Solicitud de archivos a otros nodos: Verifica que el nodo solicitado *(peer_id*) esté en la lista de *peers (self.peers*) antes de intentar la conexión, cancela la solicitud si el archivo ya existe en self.files para evitar descargas innecesarias y obtiene el puerto del nodo (*peer_port*), creando un socket TCP (*client*) para establecer la comunicación.



```
try:
    print(f"[Nodo {self.node_id}] Conectando con peer {peer_id} en puerto {peer_port}...")
    client.connect(("127.0.0.1", peer_port))
    print(f"[Nodo {self.node_id}] Solicitando archivo con hash: {file_hash}")
    client.sendall(f"REQUEST_FILE::{file_hash}".encode())

    response = client.recv(4096).decode()
    print(f"[Nodo {self.node_id}] Respuesta recibida: {response}")
    if response == "FILE_NOT_FOUND":
        print(f"[Nodo {self.node_id}] Archivo no encontrado en el nodo {peer_id}")
        return False

    if response.startswith("FILE"):
        _, filename, filesize, recv_hash = response.split("::")
        filesize = int(filesize)
        print(f"[Nodo {self.node_id}] Preparando para recibir {filename} ({filesize} bytes)")

        client.sendall("READY".encode())

        filepath = os.path.join(self.storage_dir, filename)
        print(f"[Nodo {self.node_id}] Guardando en: {filepath}")
        with open(filepath, "wb") as f:
            received_bytes = 0
            while received_bytes < filesize:
                chunk = client.recv(min(4096, filesize - received_bytes))
                if not chunk:
                    break
                f.write(chunk)
                received_bytes += len(chunk)
                print(f"[Nodo {self.node_id}] Progreso: {received_bytes}/{filesize} bytes")
        print(f"[Nodo {self.node_id}] Verificando hash...")

        with open(filepath, "rb") as f:
```

(Bloque 6)

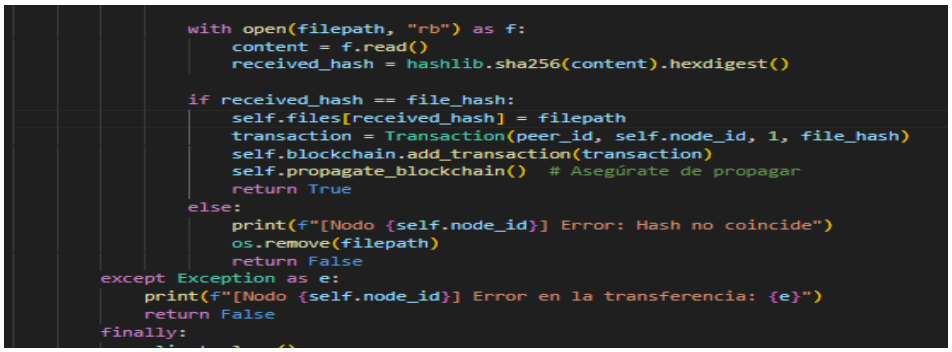
o) Establece la conexión con el nodo proveedor: Intenta conectar con el *peer_id* en *el peer_port* usando un socket TCP.

p) Recibe la respuesta del nodo proveedor: Si la respuesta es "*FILE_NOT_FOUND*", indica que el archivo no está disponible en ese nodo y finaliza la solicitud. Si la respuesta comienza con "FILE", extrae los detalles del archivo: *filename* (nombre del archivo), *filesize* (tamaño en bytes) y *recv_hash* (hash para verificar su integridad).

q) Prepara la recepción del archivo: Envía "*READY*" al nodo proveedor para confirmar que está listo para recibir. Crea un archivo en la ruta de almacenamiento (*self.storage_dir*) con el nombre recibido.

r) Recibe el archivo en bloques de datos: Lee los datos en bloques (chunk) de hasta 4096 bytes, escribe los bloques en el archivo local hasta completar el tamaño esperado y muestra el progreso de la transferencia en la consola.

s) Verifica la integridad del archivo: Una vez que la transferencia se completa, se prepara para comprobar si el hash del archivo recibido coincide con recv_hash, asegurando que no hubo corrupción o alteración en la transferencia.



```
                with open(filepath, "rb") as f:
                    content = f.read()
                    received_hash = hashlib.sha256(content).hexdigest()

                if received_hash == file_hash:
                    self.files[received_hash] = filepath
                    transaction = Transaction(peer_id, self.node_id, 1, file_hash)
                    self.blockchain.add_transaction(transaction)
                    self.propagate_blockchain()  # Asegúrate de propagar
                    return True
                else:
                    print(f"[Nodo {self.node_id}] Error: Hash no coincide")
                    os.remove(filepath)
                    return False
            except Exception as e:
                print(f"[Nodo {self.node_id}] Error en la transferencia: {e}")
                return False
            finally:
```

(Bloque 7)

Abre el archivo recibido en modo lectura binaria (rb) y calcula su hash SHA-256 para verificar su integridad. Si el hash recibido coincide con el esperado (*file_hash*), almacena la referencia del archivo en *self.files* y crea una transacción donde el nodo proveedor transfiere 1 unidad al nodo receptor, asociando la transacción con el archivo. Luego, propaga la *blockchain* para asegurar que la transacción sea reconocida por la red. Si el hash no coincide, muestra un mensaje de error, elimina el archivo corrupto y finaliza la operación. En caso de una excepción, captura el error y devuelve False. Finalmente, cierra la conexión con el nodo proveedor.

```python
def receive_blockchain(self, blockchain_data):
    print(f"📥 Nodo {self.node_id} recibiendo blockchain")
    try:
        received_chain = []
        for block_dict in blockchain_data:
            transactions = [
                Transaction(
                    tx["sender"],
                    tx["receiver"],
                    tx["amount"],
                    tx.get("file_hash"),
                    tx["timestamp"]  # ¡Este es el cambio clave!
                ) for tx in block_dict["transactions"]
            ]
            block = Block(
                block_dict["index"],
                block_dict["previous_hash"],
                transactions,
                block_dict["nonce"],
                block_dict["timestamp"]  # ¡Nuevo parámetro!
            )
            block.hash = block_dict["hash"]
            received_chain.append(block)

        if self.validate_chain(received_chain):
            if len(received_chain) > len(self.blockchain.chain) or (
                len(received_chain) == len(self.blockchain.chain) and
                received_chain[-1].timestamp > self.blockchain.chain[-1].timestamp
            ):
                print(f"✅ Blockchain actualizada en {self.node_id}")
                self.blockchain.chain = received_chain
                # Sincronizar transacciones pendientes
                self.blockchain.pending_transactions = [
                    tx for tx in self.blockchain.pending_transactions
                    if not any(tx.file_hash == block_tx.file_hash for block in received_chain for block_tx in block.transactions)
                ]
                return True
            else:
                print("ℹ️ La cadena recibida no es más larga o actual")
        else:
            print("❌ Cadena recibida no válida")

    except Exception as e:
        print(f"Error al recibir blockchain: {str(e)}")
    return False
```

(Bloque 8)

t) Método “receive_blockchain()”:
   - Recibe y procesa la blockchain: Muestra un mensaje indicando que el nodo está recibiendo la *blockchain* y convierte los datos recibidos (*blockchain_data*) en objetos Block y *Transaction*, asegurando que cada transacción incluya su marca de tiempo.
   - Valida la nueva cadena: Llama a *validate_chain(received_chain)* para verificar que la cadena es válida antes de aceptarla y solo la reemplaza

si la recibida es más larga o, en caso de empate, si la última transacción tiene un *timestamp* más reciente.

- Sincroniza transacciones pendientes: Filtra las transacciones pendientes, eliminando aquellas que ya están incluidas en la nueva blockchain.
- Manejo de errores: Si la cadena recibida no es válida, muestra un mensaje de error y la rechaza, además captura cualquier excepción para evitar que un fallo afecte la ejecución del nodo.

```
    def validate_chain(self, chain):
        for i in range(1, len(chain)):
            current_block = chain[i]
            previous_block = chain[i - 1]

            if current_block.hash != current_block.calculate_hash():
                return False

            if current_block.previous_hash != previous_block.hash:
                return False

        return True

    @staticmethod
    def hash_file(filepath):
        hasher = hashlib.sha256()
        with open(filepath, 'rb') as f:
            while chunk := f.read(4096):
                hasher.update(chunk)
        return hasher.hexdigest()

    def check_blockchain_integrity(self):
        # Verificar cada bloque individualmente
        for i, block in enumerate(self.blockchain.chain):
            calculated_hash = block.calculate_hash()
            if calculated_hash != block.hash:
                return {
                    "valid": False,
                    "tampered_block": i,
                    "stored_hash": block.hash,
                    "calculated_hash": calculated_hash
                }
        for i in range(1, len(self.blockchain.chain)):
            if self.blockchain.chain[i].previous_hash != self.blockchain.chain[i-1].hash:
                return {
                    "valid": False,
                    "tampered_block": i,
                    "issue": "previous_hash_mismatch"
                }

        return {"valid": True}
```

(Bloque 9)

u) Método "*validate_chain()":*

- Recorre la cadena desde el segundo bloque en adelante para verificar su validez.
- Verifica la integridad del bloque actual comprobando que su hash almacenado coincide con el calculado (*current_block.hash != current_block.calculate_hash()*). Si no coincide, la cadena es inválida y devuelve *False.*
- Verifica la relación entre los bloques comprobando que el *previous_hash* del bloque actual coincide con el hash del bloque anterior (*current_block.previous_hash != previous_block.hash)*. Si no coincide, la cadena es inválida y devuelve *False*.
- Si todos los bloques son válidos, devuelve True, indicando que la cadena está íntegra.

v) *hash_file(filepath):*
   - Crea un objeto sha256 para calcular el hash del archivo.
   - Lee el archivo en bloques de 4096 bytes para procesarlo eficientemente sin consumir demasiada memoria.
   - Actualiza el hash con cada bloque leído hasta completar el archivo.
   - Devuelve el hash hexadecimal del archivo, útil para verificar su integridad y evitar duplicados.

w) *check_blockchain_integrity(self):* Recorre todos los bloques de la blockchain, verificando que su hash almacenado coincida con el calculado. Si hay una discrepancia, retorna un diccionario con detalles del problema (*tampered_block, stored_hash, calculated_hash*). Verifica la relación entre los bloques, asegurando que cada *previous_hash* coincida con el hash del bloque anterior. Si hay un error, retorna un diccionario indicando el bloque afectado y el problema (*previous_hash_mismatch).* Si no se detectan alteraciones, devuelve *{"valid": True}*, indicando que la *blockchain* está intacta. Este conjunto de métodos garantiza la seguridad y coherencia de *la blockchain* en el nodo.

```python
def simulate_hack(self, block_index):
    if block_index >= len(self.blockchain.chain):
        return {"success": False, "message": "Índice de bloque fuera de rango"}

    block = self.blockchain.chain[block_index]

    # Verificar si hay transacciones en el bloque
    if not block.transactions:
        return {"success": False, "message": "No hay transacciones en este bloque para hackear"}

    # Verificar si alguna transacción tiene un hash de archivo
    file_transactions = [tx for tx in block.transactions if tx.file_hash]
    if not file_transactions:
        return {"success": False, "message": "No hay archivos en este bloque para hackear"}

    # Verificar si el nodo tiene el archivo localmente
    for tx in file_transactions:
        file_hash = tx.file_hash
        if file_hash and file_hash not in self.files:
            return {
                "success": False,
                "message": f"Este nodo no tiene el archivo con hash {file_hash} localmente. No puede hackear bloques de archivos que no posee."
            }

    # Guardar hash original para referencia
    original_hash = block.hash

    # Modificar la primera transacción que tenga un archivo
    target_tx = next((tx for tx in block.transactions if tx.file_hash), None)
    if target_tx:
        original_receiver = target_tx.receiver
        target_tx.receiver = "HACKER"

        # Actualizar timestamp pero NO ACTUALIZAR el hash almacenado
        block.timestamp = time.time()

        # Calcular nuevo hash pero solo para mostrar
        new_calculated_hash = block.calculate_hash()

        return {
            "success": True,
            "message": f"Hackeo simulado en bloque {block_index}",
            "original_hash": original_hash,
            "new_calculated_hash": new_calculated_hash,
            "original_receiver": original_receiver,
            "new_receiver": "HACKER"
        }

    return {"success": False, "message": "No se pudo simular el hackeo"}
```

(Bloque 10)

x) Método "*simulate_hack()":* Los pasos para lograr este método son.

- Verificar que el índice del bloque sea válido: Si *block_index* es mayor o igual a la longitud de la *blockchain*, el bloque no existe y se retorna un mensaje de error.
- Verificar si el bloque contiene transacciones: Si el bloque no tiene transacciones, se retorna un mensaje indicando que no hay nada que modificar.
- Buscar transacciones que contengan archivos: Se filtran transacciones dentro del bloque que tengan *un file_hash* (es decir, que estén relacionadas con archivos). Si no hay transacciones con archivos, se devuelve un mensaje indicando que no se pueden modificar archivos en ese bloque.
- Verificar que el nodo tenga una copia del archivo localmente: Si el nodo no tiene almacenado el archivo correspondiente a alguna transacción, el hackeo no se puede realizar porque el nodo no posee el archivo afectado.
- Modificar una transacción dentro del bloque: Se selecciona la primera transacción con un *file_hash* para alterarla. Se cambia el destinatario (*receiver*) a "HACKER", simulando que alguien ha manipulado la transacción. Se actualiza la marca de tiempo (*timestamp*) del bloque para reflejar la modificación. **Nota:** No se actualiza el hash del bloque, lo que hará que la *blockchain* quede inconsistente.
- Retornar los detalles del hackeo: Se devuelve un diccionario con: el hash original del bloque (*original_hash*), el nuevo hash calculado *(new_calculated_hash*), que es diferente al almacenado, el receptor original (*original_receiver*) y el nuevo receptor (*HACKER*). Esto permite visualizar cómo se ha modificado la información sin recalcular el hash del bloque, creando una vulnerabilidad en la cadena.
- Si no se encuentra una transacción modificable, retorna un mensaje indicando que no se pudo realizar el hackeo.
  ¿Porqué ponemos a prueba este método? Este método demuestra cómo un ataque a la *blockchain* puede ser detectado. Al modificar una transacción sin recalcular el hash del bloque, la función *check_blockchain_integrity()* detectará la inconsistencia cuando se valide la cadena, lo que confirma que la *blockchain* sigue siendo segura contra manipulaciones.

5. Clase “P2PNetwork”:

```python
# Clase para gestionar la red P2P
class P2PNetwork:
    def __init__(self):
        self.nodes = {}

    def add_node(self, node_id, port):
        try:
            node = P2PNode(node_id, port)
            self.nodes[node_id] = node
            print(f"Nodo {node_id} añadido a la red en puerto {port}")
            return node
        except Exception as e:
            print(f"Error añadiendo nodo: {e}")
            return None

    def list_nodes(self):
        result = {}
        for node_id, node in self.nodes.items():
            result[node_id] = {
                "puerto": node.port,
                "peers": node.peers,
                "archivos": {hash: os.path.basename(path) for hash, path in node.files.items()}
            }
        return result
```

La clase P2PNetwork representa una red de nodos en un sistema *peer-to-peer* (P2P), gestionando la conexión y administración de los nodos participantes. Al inicializarla, se crea un diccionario vacío *self.nodes* que almacenará todos los nodos de la red, donde cada clave es el *node_id* y el valor correspondiente es el objeto P2PNode.

El método *add_node(self, node_id, port)* permite agregar un nuevo nodo a la red. Intenta crear un objeto P2PNode con el identificador y puerto proporcionados, lo almacena *en self.nodes* y muestra un mensaje confirmando su adición. En caso de error, captura la excepción y devuelve *None* para indicar que el nodo no pudo ser agregado.

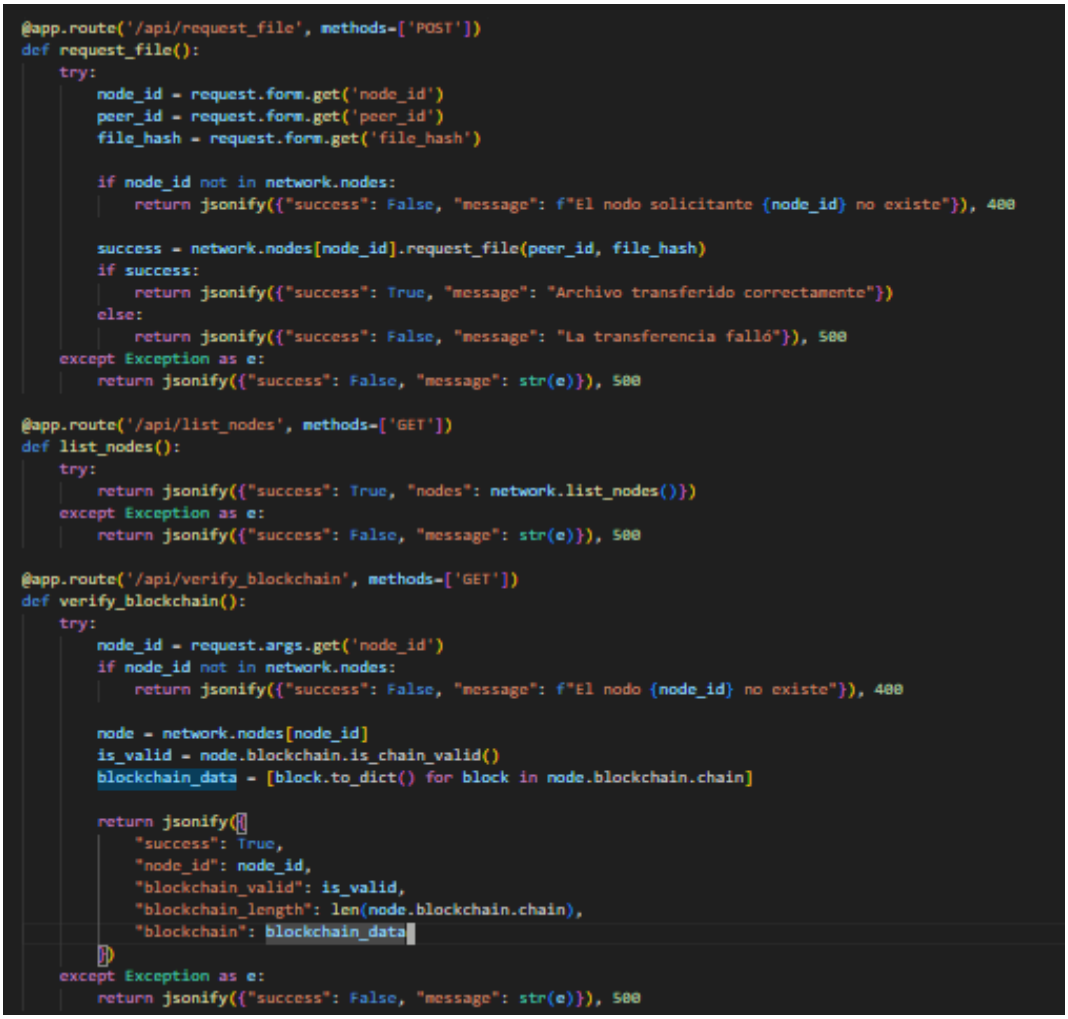
Por otro lado, el método *list_nodes(self)* recorre todos los nodos registrados y construye un diccionario *result* con información relevante. Para cada nodo, guarda el puerto en el que está escuchando ("puerto"), la lista de nodos con los que está conectado *("peers*") y un diccionario de archivos almacenados ("*archivos"), donde se mapea el hash de cada archivo a su nombre extraído con os.path.basename(path).* Finalmente, list_nodes() devuelve este diccionario con el estado actual de la red.

6. Iniciar aplicación Flask:

```python
# Inicialización de la aplicación Flask
app = Flask(__name__)
network = P2PNetwork()

@app.route('/')
def index():
    return render_template('index.html')

@app.route('/api/add_node', methods=['POST'])
def add_node():
    try:
        node_id = request.form.get('node_id')
        port = int(request.form.get('port'))

        if node_id in network.nodes:
            return jsonify({"success": False, "message": f"El nodo {node_id} ya existe"}), 400

        for existing_node in network.nodes.values():
            if existing_node.port == port:
                return jsonify({"success": False, "message": f"El puerto {port} ya está en uso"}), 400

        node = network.add_node(node_id, port)
        if node:
            return jsonify({"success": True, "message": f"Nodo {node_id} creado en puerto {port}"})
        else:
            return jsonify({"success": False, "message": "Error al crear el nodo"}), 500
    except Exception as e:
        return jsonify({"success": False, "message": str(e)}), 500

@app.route('/api/upload_file', methods=['POST'])
def upload_file():
    try:
        node_id = request.form.get('node_id')

        if node_id not in network.nodes:
            return jsonify({"success": False, "message": f"El nodo {node_id} no existe"}), 400

        if 'file' not in request.files:
            return jsonify({"success": False, "message": "No se envió ningún archivo"}), 400

        file = request.files['file']
        if file.filename == '':
            return jsonify({"success": False, "message": "Nombre de archivo vacío"}), 400

        temp_path = os.path.join('temp', file.filename)
        os.makedirs('temp', exist_ok=True)
        file.save(temp_path)

        file_hash = network.nodes[node_id].upload_file(temp_path)
        os.remove(temp_path)

        if file_hash:
            return jsonify({"success": True, "message": "Archivo subido correctamente", "hash": file_hash})
        else:
            return jsonify({"success": False, "message": "Error al subir el archivo"}), 500
    except Exception as e:
        return jsonify({"success": False, "message": str(e)}), 500
```

Esta sección del código define una aplicación *Flask* que actúa como una interfaz para gestionar una red de nodos en un sistema P2P. Se inicializa la aplicación con *app* = *Flask*(*__name__*) y se crea una instancia de P2PNetwork para gestionar los nodos.

- Ruta / (Página principal): *@app.route('/')* define la ruta raíz que simplemente renderiza una plantilla *HTML (index.html).* Esta página puede servir como una interfaz para interactuar con la red.
- Ruta */api/add_node* (Añadir un nodo): Este *endpoint* maneja solicitudes POST para agregar un nuevo nodo a la red, extrae el *node_id* y *port* del formulario enviado, verifica si el nodo ya existe en *network.nodes* y si el puerto está en uso, si alguna de estas condiciones se cumple, devuelve un mensaje de error con el código 400, llama al método *add_node()* de P2PNetwork, que intenta crear el nodo y lo almacena en la red, y devuelve una respuesta JSON indicando si el nodo se creó exitosamente o si ocurrió un error.
- Ruta */api/upload_file* (Subir un archivo a un nodo): Permite a un nodo subir un archivo a la red, obtiene el *node_id* del formulario y verifica si el nodo existe en la red, comprueba que se haya enviado un archivo y que el nombre del archivo no esté vacío, guarda temporalmente el archivo en la carpeta *temp/*, asegurándose de que la carpeta exista, llama al método *upload_file()* del nodo correspondiente para almacenar el archivo en la red y obtener su hash, elimina el archivo temporal una vez que se ha procesado y devuelve un JSON indicando si el archivo se subió con éxito y proporciona el hash generado.



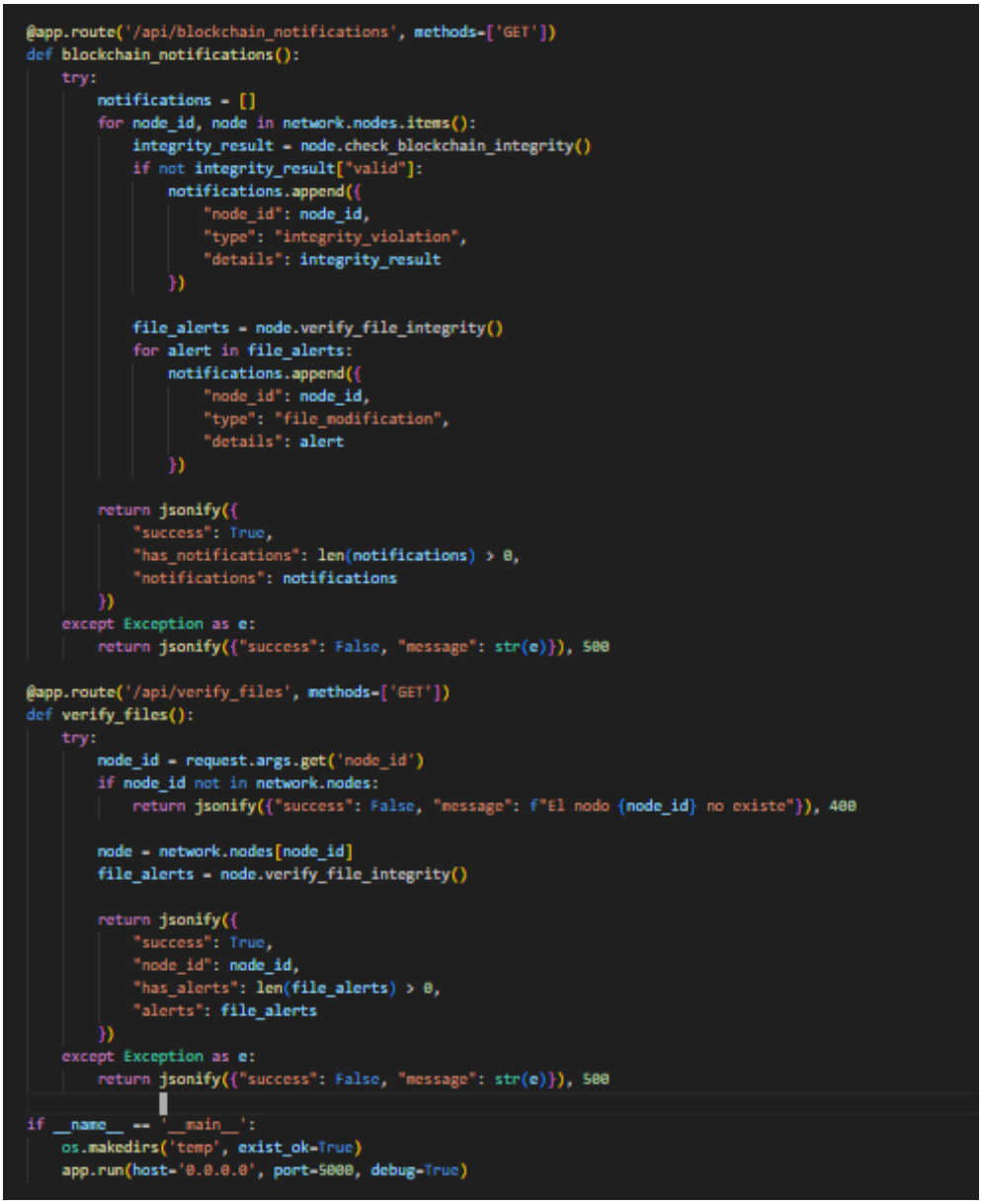
```python
@app.route('/api/request_file', methods=['POST'])
def request_file():
    try:
        node_id = request.form.get('node_id')
        peer_id = request.form.get('peer_id')
        file_hash = request.form.get('file_hash')

        if node_id not in network.nodes:
            return jsonify({"success": False, "message": f"El nodo solicitante {node_id} no existe"}), 400

        success = network.nodes[node_id].request_file(peer_id, file_hash)
        if success:
            return jsonify({"success": True, "message": "Archivo transferido correctamente"})
        else:
            return jsonify({"success": False, "message": "La transferencia falló"}), 500
    except Exception as e:
        return jsonify({"success": False, "message": str(e)}), 500

@app.route('/api/list_nodes', methods=['GET'])
def list_nodes():
    try:
        return jsonify({"success": True, "nodes": network.list_nodes()})
    except Exception as e:
        return jsonify({"success": False, "message": str(e)}), 500

@app.route('/api/verify_blockchain', methods=['GET'])
def verify_blockchain():
    try:
        node_id = request.args.get('node_id')
        if node_id not in network.nodes:
            return jsonify({"success": False, "message": f"El nodo {node_id} no existe"}), 400

        node = network.nodes[node_id]
        is_valid = node.blockchain.is_chain_valid()
        blockchain_data = [block.to_dict() for block in node.blockchain.chain]

        return jsonify({
            "success": True,
            "node_id": node_id,
            "blockchain_valid": is_valid,
            "blockchain_length": len(node.blockchain.chain),
            "blockchain": blockchain_data
        })
    except Exception as e:
        return jsonify({"success": False, "message": str(e)}), 500
```

Siguiendo con la ruta de la aplicación web:

- Ruta /api/*request_file* (Solicitar un archivo): Extrae los parámetros *node_id*, *peer_id* y *file_hash* del formulario enviado, verifica si el nodo solicitante existe en la red y, si no, devuelve un error 400, luego llama al método *request_file()* del nodo solicitante para solicitar el archivo desde ***peer_id*** y finalmente devuelve un JSON con *success*: True si la transferencia fue exitosa o *success:* False si falló la transferencia (500).
- Ruta */api/list_nodes* (Listar nodos): Llama a *network.list_nodes()* para obtener un diccionario con información de los nodos, devuelve un JSON con la lista de nodos con *success*: True y, si ocurre un error, devuelve *success:* False con el mensaje de error (500).
- Ruta /api*/verify_blockchain* (Verificar la *blockchain*): Obtiene el *node_id* desde los parámetros de la solicitud, verifica si el nodo existe; si no, devuelve un error 400, llama a *is_chain_valid()* del nodo para verificar la integridad de su *blockchain*, convierte la *blockchain* a formato JSON para enviarla en la respuesta y devuelve un JSON con *blockchain_valid:* True si la *blockchain* es válida o False si está alterada, *blockchain_length* con la cantidad de bloques en la cadena *y blockchain* con los datos de la *blockchain* en formato JSON.

```python
@app.route('/api/check_integrity', methods=['GET'])
def check_integrity():
    try:
        node_id = request.args.get('node_id')
        if node_id not in network.nodes:
            return jsonify({"success": False, "message": f"El nodo {node_id} no existe"}), 400

        node = network.nodes[node_id]
        integrity_result = node.check_blockchain_integrity()

        return jsonify({
            "success": True,
            "node_id": node_id,
            "integrity": integrity_result
        })
    except Exception as e:
        return jsonify({"success": False, "message": str(e)}), 500

@app.route('/api/simulate_hack', methods=['POST'])
def simulate_hack():
    try:
        node_id = request.form.get('node_id')
        block_index = int(request.form.get('block_index', 1))

        if node_id not in network.nodes:
            return jsonify({"success": False, "message": f"El nodo {node_id} no existe"}), 400

        node = network.nodes[node_id]
        hack_result = node.simulate_hack(block_index)

        return jsonify({
            "success": True,
            "node_id": node_id,
            "hack_result": hack_result
        })
    except Exception as e:
        return jsonify({"success": False, "message": str(e)}), 500
```

- Ruta /api/*check_integrity* (Verificar la integridad de la *blockchain):* Extrae *node_id* de los parámetros de la solicitud, verifica si el nodo existe en la red; si no, devuelve un error 400. Llama al método *check_blockchain_integrity()* del nodo, el cual analiza si los bloques han sido manipulados. Devuelve un JSON con: *success:* True si la operación fue exitosa, *node_id:* El identificador del nodo, *integrity*: Resultado de la verificación, que puede incluir detalles sobre cualquier alteración en la b*lockchai*n. Si ocurre un error, devuelve un JSON con *success*: False y el mensaje de error (500).
- Ruta /api/*simulate_hack* (Simular un ataque a la *blockchain*): Extrae *node_id* y *block_index* del formulario enviado en la solicitud, verifica si el nodo existe en la red; si no, devuelve un error 400. Llama al método *simulate_hack(block_index*), que modifica una transacción en el bloque especificado. Devuelve un JSON con: *success*: True si la simulación se realizó con éxito, *node_id*: El identificador del nodo afectado, *hack_result*: Resultado de la simulación, incluyendo detalles de la alteración, como el cambio en el destinatario de una transacción. Si ocurre un error, devuelve *success:* False con el mensaje de error (500).



```python
@app.route('/api/blockchain_notifications', methods=['GET'])
def blockchain_notifications():
    try:
        notifications = []
        for node_id, node in network.nodes.items():
            integrity_result = node.check_blockchain_integrity()
            if not integrity_result["valid"]:
                notifications.append({
                    "node_id": node_id,
                    "type": "integrity_violation",
                    "details": integrity_result
                })

            file_alerts = node.verify_file_integrity()
            for alert in file_alerts:
                notifications.append({
                    "node_id": node_id,
                    "type": "file_modification",
                    "details": alert
                })

        return jsonify({
            "success": True,
            "has_notifications": len(notifications) > 0,
            "notifications": notifications
        })
    except Exception as e:
        return jsonify({"success": False, "message": str(e)}), 500

@app.route('/api/verify_files', methods=['GET'])
def verify_files():
    try:
        node_id = request.args.get('node_id')
        if node_id not in network.nodes:
            return jsonify({"success": False, "message": f"El nodo {node_id} no existe"}), 400

        node = network.nodes[node_id]
        file_alerts = node.verify_file_integrity()

        return jsonify({
            "success": True,
            "node_id": node_id,
            "has_alerts": len(file_alerts) > 0,
            "alerts": file_alerts
        })
    except Exception as e:
        return jsonify({"success": False, "message": str(e)}), 500

if __name__ == '__main__':
    os.makedirs('temp', exist_ok=True)
    app.run(host='0.0.0.0', port=5000, debug=True)
```

- Ruta /api/*blockchain_notifications* (Obtener notificaciones de la *blockchain*): Crea una lista vacía *notifications* para almacenar las alertas detectadas. Itera sobre todos los nodos en la red. Llama a *check_blockchain_integrity(*) en cada nodo y, si detecta una alteración en la *blockchain*, agrega una notificación de tipo "*integrity_violation*". Luego, llama *a verify_file_integrity()* en cada nodo y, si detecta modificaciones en archivos, agrega una notificación de tipo "*file_modification*". Finalmente, devuelve un JSON con *success:* True, *has_notifications*, que indica si se detectaron alertas, y *notifications,* que contiene la lista con los detalles de cada problema detectado.
- Ruta /api/*verify_files* (Verificación de archivos en un nodo): Extrae *node_id* de los parámetros de la solicitud. Verifica si el nodo existe en la red; si no, devuelve un error 400. Llama al método *verify_file_integrity*() del nodo y, si encuentra alteraciones en archivos, las almacena en *file_alerts*. Devuelve un JSON con *success:* True, *node_id,* que es el identificador del nodo analizado, *has_alerts,* que indica si se encontraron alteraciones en los archivos, *y alerts*, que contiene la lista con los detalles de las modificaciones. Si ocurre un error, devuelve *success:* False y un mensaje de error con código 500.
- Y como parte final del script: Asegura que el directorio *temp*/ existe con *os.makedirs('temp', exist_ok*=True). Inicia la aplicación *Flask* con *app.run*(), permitiendo acceso desde cualquier dirección IP (host='0.0.0.0') en el puerto 5000 con *debug*=True activado para mostrar errores en la consola.

# Conclusiones

Al tener un interfaz cómodo a la vista, y sencillo de entender, no es necesario ser expertos en el tema para poder usar WoodSafe. Lo cual ayuda mucho a las personas que quieren empezar a guardar sus datos de manera digital, tanto como a las empresas y negocios que también necesitan de este servicio.

Un gran acierto es llegar a buscar la descentralización, ya que no depende de un servidor local que si se llega a “caer”, perderíamos el uso de la pagina, por eso también es un punto a favor de WoodSafe, que aparte de ser descentralizado, también es super seguro por el uso de Hash y del sistema Blockchain.

El uso del hashing en WoodSafe garantiza que cada documento cuente con un identificador único e inmutable, así como un DNI, permitiendo verificar su autenticidad y detectar cualquier alteración de manera inmediata. En lo cual ofrece transparencia, seguridad y confianza, asegurando que la información almacenada se mantenga protegida contra intentos de fraude.

WoodSafe es un sistema creado para asegurar la integridad y protección en documentos digitales utilizando blockchain y algoritmos de hashing avanzados. Lo cual cualquier intento de modificación no autorizada se detecta y registrada, lo cual nos permite identificar fácilmente posibles ataques.

La descentralización como eje de seguridad y disponibilidad WoodSafe aprovecha la naturaleza P2P de blockchain para eliminar vulnerabilidades de sistemas centralizados, garantizando que los documentos permanezcan accesibles incluso ante fallos parciales de la red. Esta arquitectura distribuida, combinada con la replicación automática de datos entre nodos, asegura resistencia ante ataques y continuidad operativa sin puntos únicos de fallo.

Al integrar una interfaz web intuitiva con procesos blockchain complejos, WoodSafe democratiza el acceso a tecnologías criptográficas, permitiendo que usuarios sin expertise técnico gestionen documentos con seguridad empresarial. La inclusión de simulaciones de ataques educativas refuerza la transparencia del sistema, validando su eficacia contra manipulaciones en escenarios reales.

Referencias:

Amo, D., Alier, M., García-Peñalvo, F., Fonseca, D., & Casañ, M. J. (2020). Privacidad, seguridad y legalidad en soluciones educativas basadas en Blockchain: Una Revisión Sistemática de la Literatura. *RIED Revista Iberoamericana de Educación a Distancia*, *23*(2), 213. https://doi.org/10.5944/ried.23.2.26388

Luna, M. (s/f). *Uso de la tecnología blockchain en la gestión documental*. Archivensoluciones.com. Recuperado el 3 de marzo de 2025, de https://archivensoluciones.com/uso-de-la-tecnologia-blockchain-en-la-gestion-documental/