INFORME TECNICO – PROYECTO 1

P2P - COMUNICACION ENTRE PROCESOS MEDIANTE API REST Y RPC

DANIEL CARDONA GONZALEZ

JUAN JOSE TAMAYO OSPINA

MARIA LUCIA CADAVID MARTINEZ

ALVARO ENRIQUE OPSINA SANJUAN

ARQUITECTURA DE NUBE Y SISTEMAS DISTRIBUIDOS

UPB

2025

FACULTAD DE INGENIERÍA DE SISTEMAS E INFORMÁTICA

### Objetivo

#### 1.1 Objetivo general

Diseñar e implementar un prototipo de sistema P2P (peer-to-peer) para compartición de archivos distribuida y descentralizada donde cada nodo (peer) exponga microservicios que permitan: registro y consulta del índice de archivos, descubrimiento de recursos en la red y ejecución de transferencias simuladas (ECO/DUMMY), utilizando REST API y gRPC como middlewares de comunicación.

#### 1.2 objetivos específicos

1. Implementar la lógica de un peer con módulos servidor (PServidor) y cliente (PCliente) que puedan: publicar su índice de archivos, responder consultas de localización y ejecutar endpoints ECO/DUMMY de upload y download.
2. Definir y documentar una arquitectura P2P no estructurada basada en servidor(es) de directorio/localización (con posibilidad de variantes) y las interacciones entre peers usando REST y gRPC.
3. Garantizar concurrencia en los microservicios del PServidor para soportar múltiples solicitudes simultáneas.
4. Diseñar el formato del archivo de configuración y el procedimiento de bootstrap para arranque dinámico del peer.
5. Desplegar y probar la solución en entorno local y contenerizada (Docker), y documentar pruebas con al menos tres peers interactuando entre sí.
6. Entregar documentación técnica y un video que demuestren el diseño, la implementación y las pruebas.

### Marco teórico

A continuación, se presentan breves definiciones de conceptos claves para entender el proyecto, su arquitectura y sus diferentes componentes a la hora de su implementación, ejecución y funcionamiento:

#### **Redes P2P**

Las redes peer-to-peer (P2P) constituyen una arquitectura distribuida en la que todos los nodos, conocidos como peers, tienen las mismas funciones y derechos dentro del sistema. A diferencia de los modelos cliente-servidor tradicionales, en los que existe un servidor central encargado de gestionar los recursos, en una red P2P los nodos actúan simultáneamente como clientes y servidores, lo que permite que cada uno aporte y consuma información directamente de otros. Esta descentralización favorece la escalabilidad y la tolerancia a fallos, ya que la red puede seguir operando incluso si un número de peers se desconecta. Sin embargo, también presenta retos importantes en cuanto al descubrimiento de recursos, la coordinación de los nodos y la eficiencia en la propagación de consultas.

#### **Redes P2P no estructuradas y estructuradas**

Las redes P2P no estructuradas se caracterizan porque los peers establecen conexiones de manera aleatoria o arbitraria. Esto permite una gran flexibilidad y simplicidad en la implementación, aunque la búsqueda de recursos puede resultar costosa en términos de mensajes debido a que suele recurrirse a técnicas de inundación o directorios centralizados de localización. En contraste, las redes P2P estructuradas emplean mecanismos como las Tablas Hash Distribuidas (DHT), que asignan claves a nodos específicos mediante algoritmos de hashing. Este enfoque mejora la eficiencia en la búsqueda y recuperación de recursos, aunque requiere mayor coordinación para mantener la estructura de la red. En el proyecto se adoptará una red P2P no estructurada con directorios de localización, dado que su simplicidad facilita la construcción del prototipo.

#### **Microservicios**

El enfoque de microservicios es un paradigma de diseño de software que busca dividir un sistema complejo en servicios pequeños, independientes y especializados en una única tarea. Cada microservicio se ejecuta como un proceso autónomo, con su propia lógica y datos, y se comunica con los demás mediante interfaces bien definidas, generalmente a través de protocolos de red ligeros como HTTP o gRPC. Esta arquitectura ofrece ventajas como la escalabilidad independiente de cada componente, la facilidad de despliegue y la tolerancia a fallos localizados. En el contexto del sistema P2P, la adopción de microservicios permite separar funciones críticas como la consulta de archivos, la localización de recursos y la simulación de carga y descarga (ECO/DUMMY), favoreciendo un diseño modular y mantenible.

#### **REST y gRPC**

REST (Representational State Transfer) es un estilo arquitectónico que define principios para la construcción de servicios web a través de operaciones HTTP estándar como GET, POST, PUT y DELETE. Se caracteriza por su simplicidad, amplia adopción y facilidad de integración, ya que los mensajes suelen intercambiarse en formato JSON. Por otro lado, gRPC es un framework de comunicación de alto rendimiento que utiliza HTTP/2 como protocolo de transporte y Protocol Buffers (Protobuf) como lenguaje de serialización. gRPC permite llamadas a procedimientos remotos (RPC) de forma eficiente, soportando transmisión en streaming y comunicación bidireccional. En este proyecto, REST se utilizará principalmente para el descubrimiento de peers y la consulta de índices de archivos, mientras que gRPC se empleará en los servicios que requieran mayor eficiencia o en la simulación de transferencia de datos.

#### **Concurrencia en PServidor**

La concurrencia es un aspecto fundamental en sistemas distribuidos que buscan atender múltiples solicitudes de manera simultánea. En un servidor P2P, la concurrencia permite que diferentes peers puedan consultar recursos o solicitar servicios al mismo tiempo sin que se bloquee la ejecución de otros procesos. Esto puede lograrse mediante múltiples enfoques, como el uso de hilos, procesos, programación asíncrona o goroutines, dependiendo del lenguaje de implementación. Garantizar la concurrencia en los microservicios asegura la escalabilidad del sistema y una experiencia fluida para los usuarios, además de permitir pruebas de carga que validen la capacidad del sistema frente a varios clientes concurrentes.

#### **Archivo de configuración y bootstrap**

Cada peer requiere un archivo de configuración para iniciar de forma dinámica, lo que se conoce como bootstrap. Este archivo define parámetros esenciales como la dirección IP de escucha, el puerto asociado al middleware, el directorio de archivos a indexar y las direcciones de peers amigos (titular y suplente). Durante el proceso de arranque, el peer utiliza esta información para registrarse en la red, anunciar sus recursos y solicitar información de otros nodos disponibles. La existencia de un peer suplente permite tolerancia a fallos en caso de que el titular no esté disponible, garantizando la resiliencia del sistema.

#### **Índice de archivos**

El índice de archivos es una estructura de datos que describe los recursos disponibles en un peer. Cada entrada del índice contiene atributos como el nombre del archivo, su tamaño, fecha de última modificación y la URL o URI donde puede ser accedido dentro del nodo. Este índice no transfiere los archivos en sí, sino que sirve como catálogo consultable que permite localizar qué peer posee un recurso determinado. Gracias a este mecanismo, los peers pueden intercambiar consultas de manera rápida sin necesidad de transmitir contenido pesado.

#### **Servicios ECO/DUMMY**

Los servicios ECO/DUMMY corresponden a endpoints de prueba que simulan la transferencia de archivos en el sistema. Aunque no se envían archivos reales, estos servicios permiten validar que las llamadas entre peers funcionan de manera adecuada y que la lógica de carga y descarga puede ejecutarse. En términos prácticos, un servicio ECO podría devolver un mensaje de confirmación al peer solicitante, mientras que un servicio DUMMY podría generar datos ficticios que emulen una descarga. Este tipo de servicios son esenciales en prototipos, ya que permiten comprobar la comunicación y los flujos de interacción sin necesidad de manejar volúmenes de datos reales.

#### **Seguridad y consideraciones operativas**

La seguridad es un aspecto crítico en cualquier sistema distribuido. Aunque el presente proyecto es de carácter académico y no requiere mecanismos avanzados, resulta conveniente considerar prácticas básicas como el aislamiento del directorio compartido para evitar exponer archivos sensibles del sistema, el uso de tokens simples de autenticación para controlar el acceso a los microservicios y la implementación de timeouts para gestionar fallos en la comunicación. En un entorno real, estos mecanismos se complementarían con el uso de TLS/HTTPS, autenticación robusta y políticas de autorización más estrictas, lo cual garantizaría tanto la integridad como la confidencialidad de la información compartida.

#### **Despliegue en contenedores**

El uso de contenedores como Docker facilita la portabilidad, el aislamiento y la reproducibilidad del sistema. Cada peer puede ejecutarse dentro de un contenedor independiente que incluye todos sus microservicios, bibliotecas y dependencias. Esto permite desplegar fácilmente múltiples peers en una misma máquina mediante Docker Compose o en entornos distribuidos como AWS Academy. El despliegue en contenedores asegura que las pruebas puedan replicarse en distintos entornos sin alterar el comportamiento del sistema, lo que refuerza la confiabilidad del prototipo desarrollado.

### Descripción del servicio y problema abordado

El presente proyecto busca dar solución al problema del intercambio de archivos en entornos distribuidos sin depender de un servidor centralizado. En los sistemas tradicionales cliente-servidor, los usuarios dependen de un único punto de control para almacenar, gestionar y distribuir la información, lo que genera limitaciones de escalabilidad, riesgo de fallos críticos y concentración de recursos en una sola entidad. Estas limitaciones se hacen evidentes en escenarios donde múltiples usuarios desean compartir recursos entre sí de manera descentralizada, con tolerancia a fallos y flexibilidad para unirse o retirarse de la red sin afectar su funcionamiento global.

El servicio propuesto se implementa bajo un modelo de red P2P en el cual cada nodo actúa tanto como proveedor como consumidor de recursos. Cada peer mantiene un índice de los archivos disponibles en un directorio local configurable y lo pone a disposición de otros peers mediante microservicios expuestos a través de dos middlewares complementarios: REST y gRPC. El primero facilita las consultas de descubrimiento y localización de archivos, mientras que el segundo optimiza la interacción mediante llamadas a procedimientos remotos de mayor eficiencia. Este diseño permite que cualquier nodo de la red pueda solicitar información sobre los archivos compartidos y recibir referencias hacia el peer que posee el recurso deseado.

La arquitectura planteada no contempla en esta versión la transferencia real de archivos, sino que implementa servicios simulados de carga y descarga, denominados ECO y DUMMY. Estos servicios cumplen la función de validar la correcta comunicación entre nodos y demostrar la factibilidad del modelo de compartición distribuida. De esta manera, se resuelve el problema de interacción entre procesos distribuidos sin necesidad de transferir grandes volúmenes de datos, garantizando que el sistema sea más sencillo de probar, desplegar y documentar en un entorno académico.

El problema abordado se centra entonces en cómo diseñar un sistema distribuido que permita el descubrimiento de recursos y la interacción entre procesos autónomos, manteniendo la descentralización propia de las redes P2P. Este enfoque atiende tanto a necesidades técnicas como a aprendizajes académicos, pues fomenta la comprensión de conceptos fundamentales de arquitecturas de sistemas distribuidos, uso de microservicios, concurrencia, protocolos de comunicación y despliegue en entornos virtualizados. Al resolver este problema mediante el prototipo planteado, se ofrece una base sólida para escalar en versiones futuras hacia la transferencia real de archivos y la incorporación de mecanismos más avanzados de seguridad, persistencia y sincronización.

### Arquitectura del sistema y diagramas

#### 4.1 Arquitectura

##### **1. Componentes principales**

1. Peer (nodo):  
    Cada peer se compone de:
   1. PServidor: conjunto de microservicios que exponen funcionalidades al resto de la red.
   2. PCliente: módulo cliente encargado de consumir servicios de otros peers.
2. Archivo de configuración (Bootstrap):  
    Define parámetros de inicio del peer: IP, puerto(s), directorio de archivos compartidos, peer amigo titular y suplente.

##### **2. Microservicios del PServidor**

1. Servicio de Localización (REST):  
    Permite registrar al peer en la red e intercambiar información con un peer amigo (titular o suplente). Facilita la consulta de peers disponibles.
2. Servicio de Consulta de Índice (REST):  
    Expone el listado de archivos locales del peer, incluyendo su URI/URL de acceso.
3. Servicio de Descarga Simulada (gRPC – DUMMY Download):  
    Atiende solicitudes de descarga devolviendo un flujo de datos ficticio o un eco de confirmación.
4. Servicio de Carga Simulada (gRPC – DUMMY Upload):  
    Recibe solicitudes de carga de archivo y devuelve confirmaciones o datos simulados.
5. **Manejo de concurrencia en PServidor:**  
   El PServidor está diseñado para atender múltiples solicitudes de manera simultánea, garantizando la concurrencia en el acceso a los microservicios. Esto implica que varios peers pueden consultar índices, realizar operaciones de localización o invocar los servicios de carga y descarga de forma paralela sin bloquear el funcionamiento del servidor. Para lograrlo, cada microservicio utiliza mecanismos de programación concurrente propios del middleware seleccionado: hilos y procesos ligeros en entornos síncronos, o bien programación asíncrona y event-driven en entornos basados en REST y gRPC. De esta forma, se asegura la escalabilidad del sistema y la eficiencia en escenarios donde varios nodos interactúan al mismo tiempo con un mismo peer, lo cual refleja el comportamiento esperado en sistemas distribuidos reales.

##### **3. Módulo PCliente**

El PCliente es el componente que habilita a cada peer para interactuar con la red y consumir los servicios expuestos por otros peers. Sus principales responsabilidades son:

1. Consulta de índice remoto: permite solicitar a otro peer su listado de archivos disponibles a través de un endpoint REST.
2. Localización de recursos: consulta al peer amigo titular o suplente para identificar en qué nodo se encuentra un archivo específico.
3. Invocación de servicios DUMMY: establece comunicación vía gRPC con un peer remoto para simular operaciones de descarga o carga de un archivo.
4. Gestión de fallos: si un peer no responde, reintenta con el suplente o marca el recurso como no disponible, asegurando tolerancia a fallos en la red.

##### **4. Interacciones entre componentes**

El funcionamiento del sistema sigue un flujo definido:

1. Bootstrap: al iniciar, el peer lee su archivo de configuración y se conecta a su peer amigo titular mediante el servicio de localización. Si este no responde, se conecta al suplente.
2. Registro: el peer informa al servidor de directorio (peer amigo) sus datos de conexión y el índice de archivos compartidos.
3. Consulta: cuando un peer necesita un recurso, utiliza su módulo PCliente para enviar una petición REST al peer amigo, solicitando información sobre la ubicación del archivo.
4. Resolución: el peer amigo responde con el índice propio o con referencias hacia otros peers que disponen del recurso solicitado.
5. Descarga simulada: el PCliente del peer solicitante establece comunicación gRPC directa con el peer que tiene el archivo y ejecuta el servicio de Descarga DUMMY. Este responde con datos simulados que validan la correcta invocación.
6. Carga simulada: de manera similar, el peer solicitante puede enviar datos ficticios hacia otro peer mediante el servicio de Carga DUMMY para validar la operación de subida.
7. Concurrencia: todos los microservicios del PServidor permiten recibir múltiples solicitudes de manera simultánea, garantizando la interacción entre varios peers en paralelo.

#### 4.2 Diagramas

1. Diagrama de arquitectura de alto nivel

A diagram of a computer

AI-generated content may be incorrect.

1. Diagrama de flujo de procesos (secuencia búsqueda y descarga)

A black screen with green lines

AI-generated content may be incorrect.

1. Diagrama de flujo de bootstrap (actividad UML)

A diagram of a flowchart

AI-generated content may be incorrect.

### Especificación de protocolos y APIs

La arquitectura del sistema P2P implementa dos tipos de middleware de comunicación: REST API y gRPC. Cada uno cumple un rol específico dentro del flujo de interacción entre peers. REST se emplea para las operaciones de descubrimiento, registro y consulta de índices de archivos, debido a su simplicidad, legibilidad y compatibilidad universal. gRPC se utiliza en los servicios de simulación de transferencia (ECO/DUMMY), dado que proporciona comunicación binaria eficiente y soporte nativo para transmisión de datos en streaming. Esta combinación asegura un diseño modular y robusto, aprovechando las fortalezas de cada protocolo.

#### 5.1 REST API

Base URL: http://<IP>:<PUERTO\_REST>/api/v1/

Formato de datos: JSON

Endpoints principales:

1. Registro de peer
   * Método: POST /register
   * Entrada (JSON):

*{*

*"peer\_id": "peer1",*

*"ip": "192.168.1.10",*

*"puerto": 5000,*

*"archivos": [*

*{ "nombre": "documento.pdf", "url": "http://192.168.1.10:5000/files/documento.pdf" }*

*]*

*}*

* + Respuesta (JSON):

*{*

*"status": "ok",*

*"peers\_conocidos": [*

*{ "peer\_id": "peer2", "ip": "192.168.1.20", "puerto": 5000 }*

*]*

*}*

1. Consulta de peers registrados

* Método: GET /peers
* Respuesta (JSON):

*{*

*"peers": [*

*{ "peer\_id": "peer1", "ip": "192.168.1.10", "puerto": 5000 },*

*{ "peer\_id": "peer2", "ip": "192.168.1.20", "puerto": 5000 }*

*]*

*}*

1. Consulta del índice de archivos

* Método: GET /files
* Respuesta (JSON):

*{*

*"archivos": [*

*{ "nombre": "documento.pdf", "url": "http://192.168.1.10:5000/files/documento.pdf" },*

*{ "nombre": "imagen.jpg", "url": "http://192.168.1.10:5000/files/imagen.jpg" }*

*]*

*}*

1. Localización de archivo específico

* Método: GET /locate?nombre=<archivo>
* Respuesta (JSON):

*{*

*"resultado": "encontrado",*

*"peer": { "peer\_id": "peer3", "ip": "192.168.1.30", "puerto": 5000 },*

*"url": "http://192.168.1.30:5000/files/documento.pdf"*

*}*

1. Dummy Upload/Download (REST opcional para pruebas simples)

Download Dummy:

* Método: GET /dummy/download/<archivo>>
* Respuesta (JSON):

*{*

*"status": "ok",*

*"mensaje": "Descarga simulada de archivo documento.pdf"*

*}*

Upload Dummy:

* Método: POST /dummy/upload
* Payload (JSON):

*{*

*"nombre": "nuevo.txt",*

*"contenido": "Este es un contenido simulado"*

*}*

* Respuesta (JSON):

*{*

*"status": "ok",*

*"mensaje": "Carga simulada de archivo nuevo.txt"*

*}*

6. Upload/Download Real (REST extendido)

* Download Real:
  + Método: GET /download/<archivo>
  + Respuesta: stream binario (archivo real).
* Upload Real:
  + Método: POST /upload
  + Tipo de contenido: multipart/form-data
  + Campo esperado: file
  + Respuesta (JSON):

*{*

*"status": "ok",*

*"mensaje": "Archivo cargado correctamente",*

*"archivo": "nuevo.txt"*

*}*

#### 5.2 gRPC API

Archivo de definición (Protobuf):

*syntax = "proto3";*

*service FileTransfer {*

*// Descarga simulada*

*rpc DummyDownload (FileRequest) returns (stream FileChunk);*

*// Carga simulada*

*rpc DummyUpload (stream FileChunk) returns (UploadStatus);*

*}*

*// Solicitud de archivo*

*message FileRequest {*

*string filename = 1;*

*}*

*// Chunk de archivo simulado*

*message FileChunk {*

*bytes content = 1;*

*int32 chunk\_number = 2;*

*}*

*// Estado de carga simulada*

*message UploadStatus {*

*bool success = 1;*

*string message = 2;*

*}*

Descripción de métodos gRPC

1. DummyDownload: el cliente solicita un archivo ficticio y el servidor responde con un stream de FileChunk generados artificialmente.
2. DummyUpload: el cliente envía chunks simulados al servidor, que devuelve un UploadStatus confirmando la recepción.
3. Download (real): el cliente solicita un archivo existente en el directorio compartido del peer. El servidor lo divide en chunks binarios y los envía en streaming.
4. Upload (real): el cliente envía chunks binarios de un archivo real hacia el servidor, que los reconstruye y guarda en su directorio compartido.

### 6. Algoritmos de particionamiento y distribución

#### 6.1 Particionamiento de recursos

En este sistema, el particionamiento se basa en los archivos almacenados localmente en cada peer.

1. Cada peer mantiene un directorio compartido definido en su archivo de configuración (peer1.json en el ejemplo).
2. Durante el Bootstrap, el peer registra su índice de archivos y lo expone mediante el microservicio /files.
3. No se replica el contenido, sino que cada peer es dueño exclusivo de sus archivos, lo que genera una distribución natural por origen.

De esta manera, el espacio de recursos se particiona dinámicamente en función del directorio de cada nodo, sin necesidad de un algoritmo de hash global ni tablas distribuidas (característico de las redes no estructuradas).

#### 6.2 Estrategia de distribución en la red

La distribución de la información (índices de archivos) se realiza mediante:

1. Registro de peers → al iniciar, cada nodo contacta a su peer amigo titular o suplente (según disponibilidad) para anunciarse y obtener la lista de peers conocidos.
2. Propagación del índice → cada peer puede enriquecer su tabla interna con archivos de otros peers usando los endpoints /add\_peer y /add\_file, simulando un mecanismo de intercambio de metadatos distribuido.
3. Consulta distribuida → cuando un peer busca un archivo, consulta primero su tabla local (peer\_files) y, si no está, recurre a la lista de peers conocidos vía /locate.

Esto genera una distribución descentralizada, donde cada peer mantiene una vista parcial de la red. No existe un nodo central, y el sistema tolera fallas al contar con peer amigo suplente.

#### 6.3 Algoritmos empleados

1. Localización por exploración limitada (Flooding controlado):  
    Un peer consulta a su peer amigo. Si este no tiene el recurso, puede redirigir hacia otros peers de su lista. Este mecanismo evita inundar toda la red, pero garantiza cierto nivel de descubrimiento distribuido.
2. Particionamiento implícito por directorios:  
    No se aplica un particionamiento explícito mediante hashing, sino que cada nodo actúa como partición natural. Cada directorio compartido es, en sí, una unidad de particionamiento.
3. Distribución por registro incremental:  
    Cuando un nuevo archivo se agrega en un peer (/upload o /add\_file), este se incorpora a la tabla local y puede compartirse en la red al ser consultado, manteniendo la consistencia eventual de los índices.

Este esquema minimiza la sobrecarga de control, ya que no requiere estructuras globales como DHT (Distributed Hash Tables).

La descentralización mejora la tolerancia a fallas: si un peer desaparece, los demás siguen operando con la información disponible.

El particionamiento implícito por directorio local es suficiente para la versión inicial del sistema, en la que la transferencia es simulada (ECO/DUMMY).

A futuro, se puede extender con algoritmos más avanzados de distribución (ejemplo: Consistent Hashing o Chord) si se requiere balanceo de carga o búsqueda más eficiente.

### 7. Descripción del entorno de ejecución en Docker y AWS

#### 7.1 Infraestructura base en AWS

El sistema se despliega en una máquina virtual (EC2) de Amazon Web Services.

1. Sistema operativo: instancia basada en Linux (Amazon Linux).
2. Red: la VM está conectada a Internet mediante una IP pública y a una red privada interna de AWS (VPC).
3. Puertos expuestos: los servicios REST y gRPC definidos en cada peer (por defecto 5000 para REST y 5001 para gRPC) se exponen mediante reglas de seguridad (Security Groups).
4. Persistencia de datos: cada peer monta un directorio compartido local (./shared\_files\_peerX) donde se almacenan los archivos accesibles en la red P2P.

De esta forma, la máquina de AWS actúa como nodo anfitrión que simula toda la red de peers en contenedores Docker.

#### 7.2 Virtualización mediante Docker

Cada peer del sistema se ejecuta dentro de un contenedor Docker independiente, lo cual garantiza:

1. Aislamiento: cada contenedor corre su propia instancia del servicio P2P (REST + gRPC).
2. Reproducibilidad: la construcción del contenedor a partir del Dockerfile asegura que todos los peers tienen el mismo entorno de ejecución.
3. Escalabilidad: nuevos peers pueden añadirse fácilmente modificando docker-compose.yml e iniciando nuevos servicios.

Dockerfile:

El Dockerfile define la imagen base y dependencias del sistema:

1. Imagen base de Python 3 (runtime principal del peer).
2. Instalación de librerías necesarias (Flask para REST, grpcio para gRPC).
3. Copia del código fuente (main.py) y de los archivos de configuración (peerX.json).
4. Definición del punto de entrada (CMD) para lanzar el peer.

Docker Compose:

El archivo docker-compose.yml orquesta múltiples peers en una sola máquina:

1. Cada servicio representa un peer, con su propio archivo de configuración (peer1.json, peer2.json, etc.).
2. Se definen los puertos mapeados de host a contenedor (5000:5000, 5001:5001, etc.) para permitir comunicación tanto interna (entre contenedores) como externa (con usuarios o pruebas).
3. El volumen local mapea la carpeta de archivos compartidos al contenedor, asegurando persistencia.

#### 7.3 Integración Docker + AWS

El despliegue combina las capacidades de AWS y Docker:

1. AWS EC2 → provee el hardware virtualizado y la red accesible públicamente.
2. Docker → gestiona los peers como contenedores aislados y reproducibles.

Construcción de la imagen Docker:

*docker-compose up –build –force-recreate*

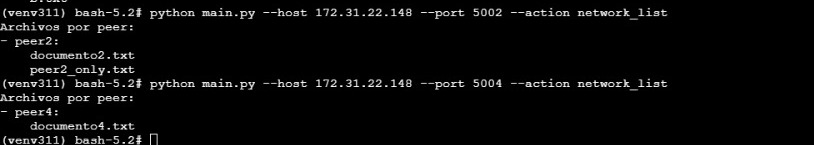
### 8. Nota en victorias tempranas

Para observar y seguir el proceso de desarrollo, implementación y resolución del trabajo de generaron una serie de milestones e issues en el repositorio de GitHub, allí se puede evidenciar el flujo de trabajo y la definición de una ruta de desarrollo clara para la resolución del trabajo.

### 9. Pruebas y análisis de resultados

1. Listar la red de peers (prueba de cliente)

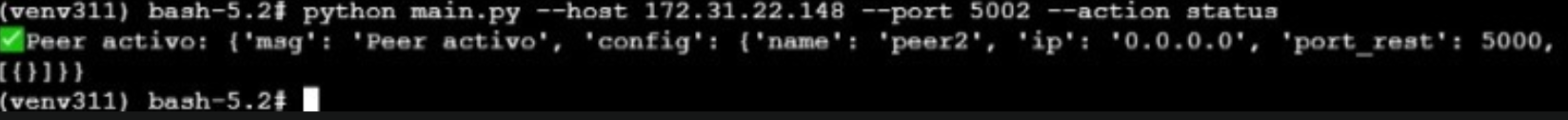
* Objetivo: Verificar que un peer puede consultar la lista de peers registrados en la red.

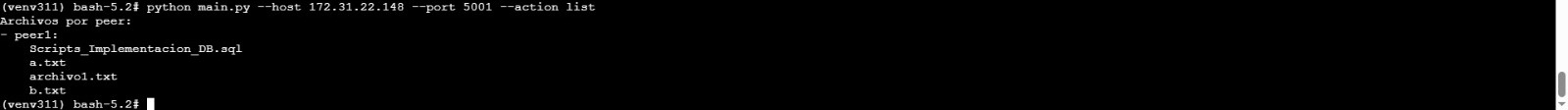


* Análisis: La prueba fue exitosa; confirma que el servicio de localización y registro funciona correctamente y que el intercambio de información entre peers se mantiene actualizado.

2. Ver estado del peer (prueba de cliente)

* Objetivo: Validar que cada peer exponga un endpoint de estado accesible

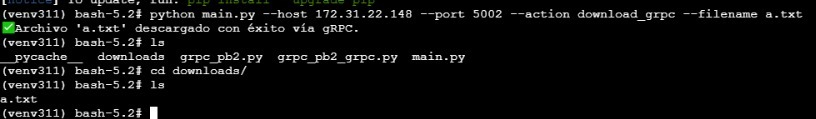




* Análisis: La prueba demuestra que los peers están activos y listos para recibir solicitudes, cumpliendo la función de monitoreo básico.

3. Localizar archivo y descargarlo vía gRPC (prueba de cliente)

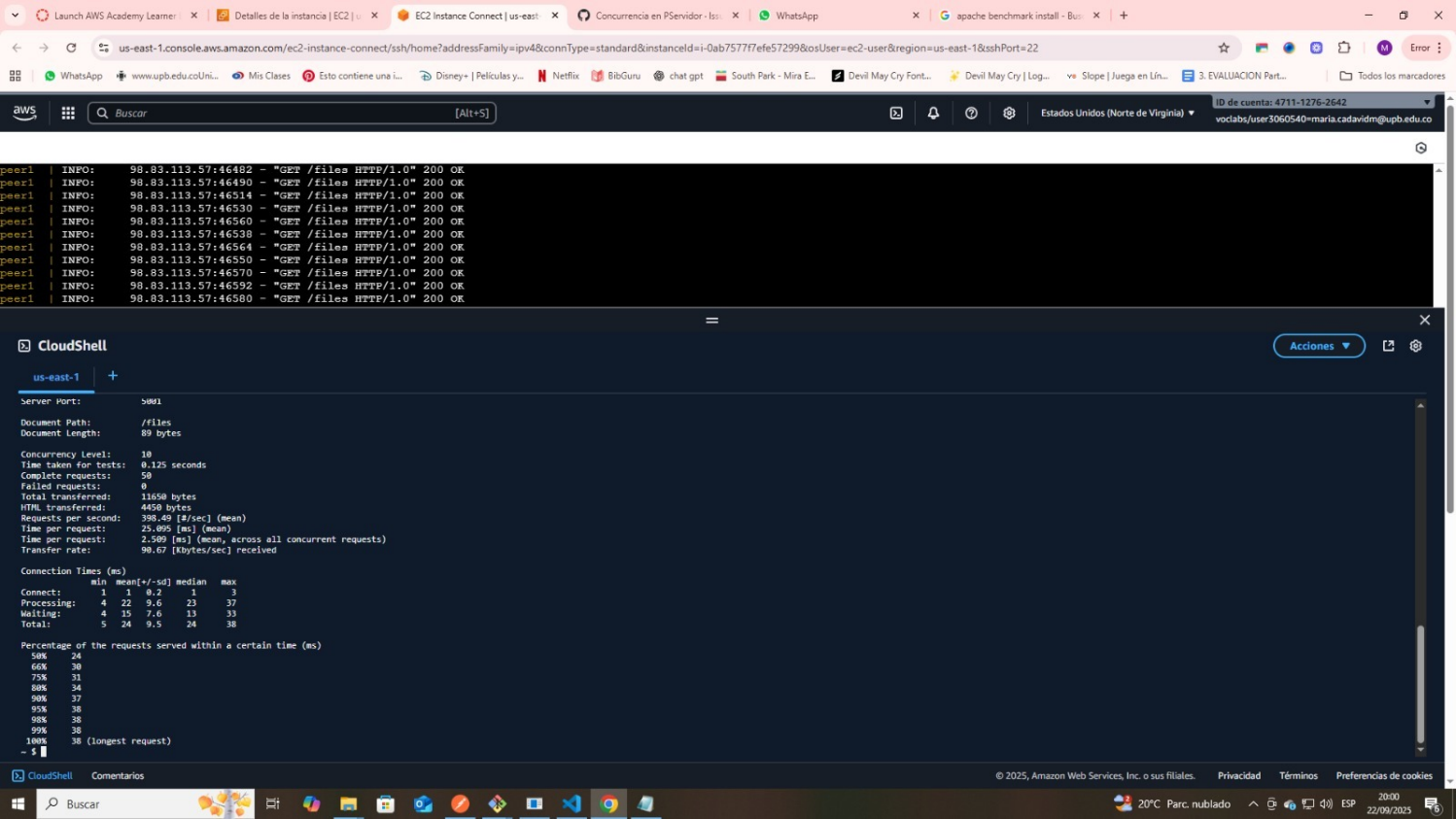
* Objetivo: Confirmar que un peer puede localizar un archivo en otro nodo y descargarlo usando gRPC.



* Análisis: La prueba fue satisfactoria. Además, se verificó que el puerto 5001 del peer 1 puede solicitar un archivo del peer 4, lo que valida la interoperabilidad y la comunicación directa entre peers.

4. Prueba de concurrencia

* Objetivo: Evaluar la capacidad del servidor para manejar múltiples solicitudes simultáneas.



* Análisis: El servidor respondió de manera estable, confirmando que el manejo de concurrencia en PServidor es efectivo.

5. Pruebas del servidor con IP pública (download, locate, root, files, add file, add peer)

* Objetivos:
  + Validar que el servicio gRPC de descarga pueda ser invocado desde fuera de la red local, a través de AWS.
  + Verificar la localización de archivos mediante la API REST desde un cliente externo.
  + Validar la accesibilidad básica del peer vía HTTP externo.
  + Consultar archivos compartidos de un peer vía Internet.
  + Comprobar la capacidad de agregar archivos a la tabla de un peer remoto.
  + Verificar la capacidad de registrar peers remotos en la red.

A screenshot of a computer

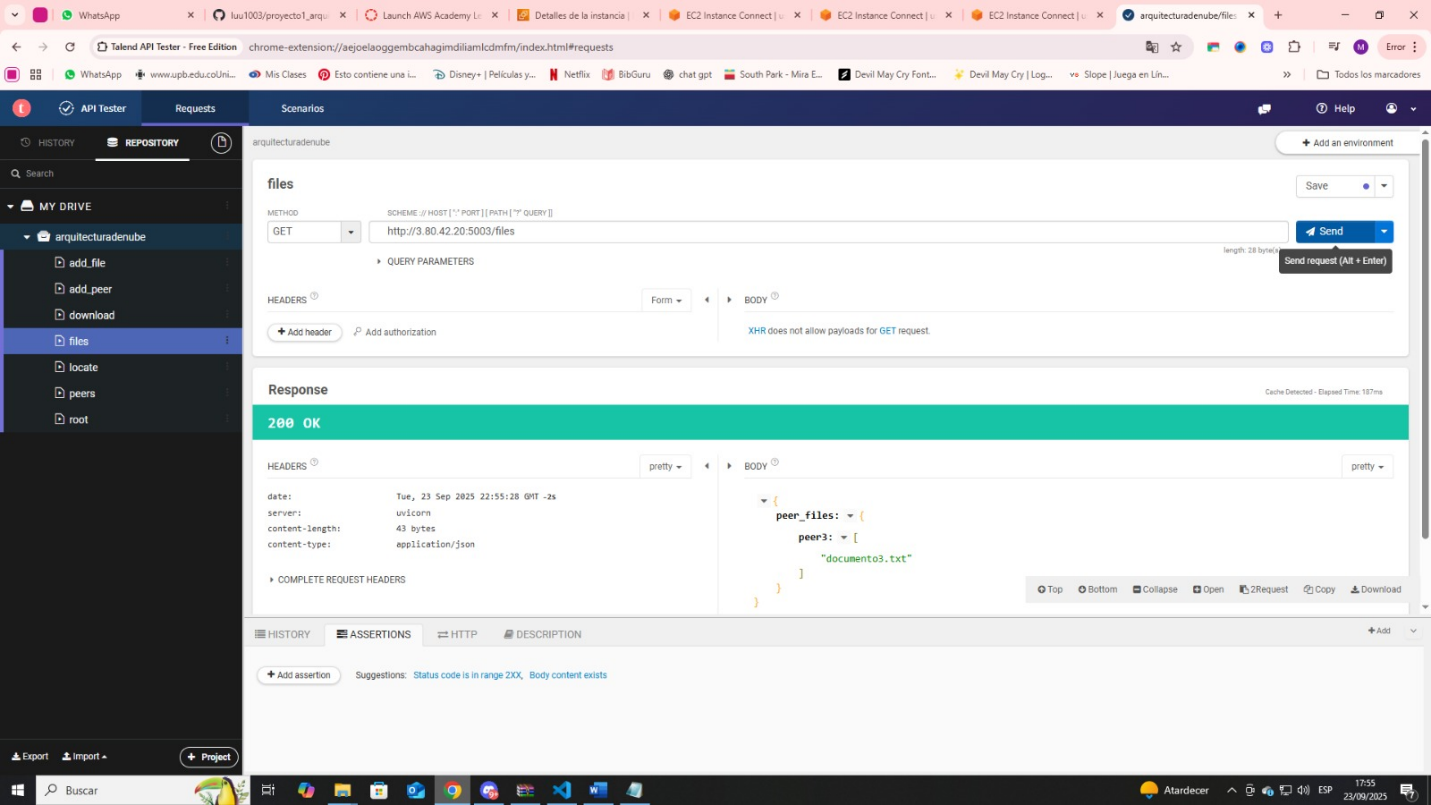
AI-generated content may be incorrect.

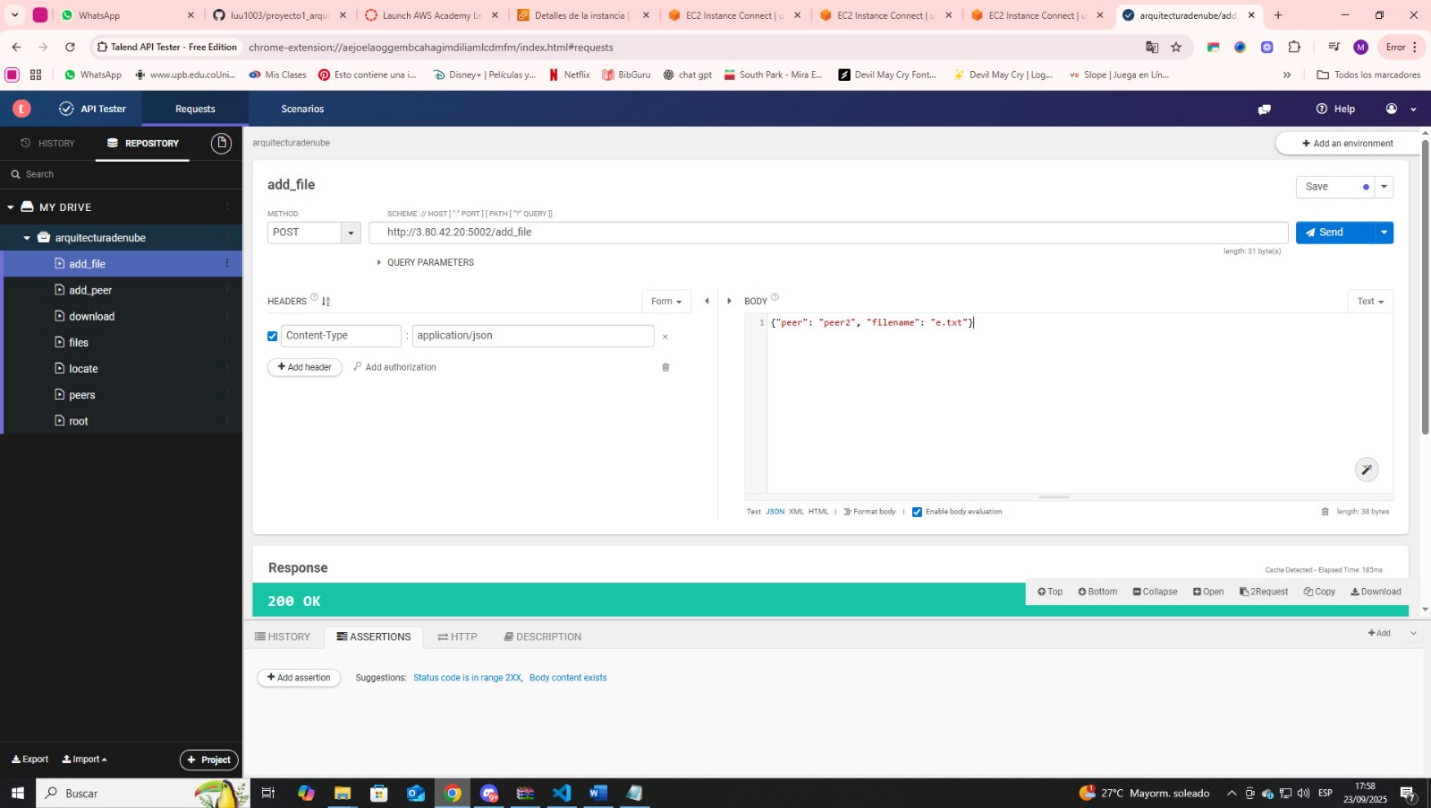
A screenshot of a computer

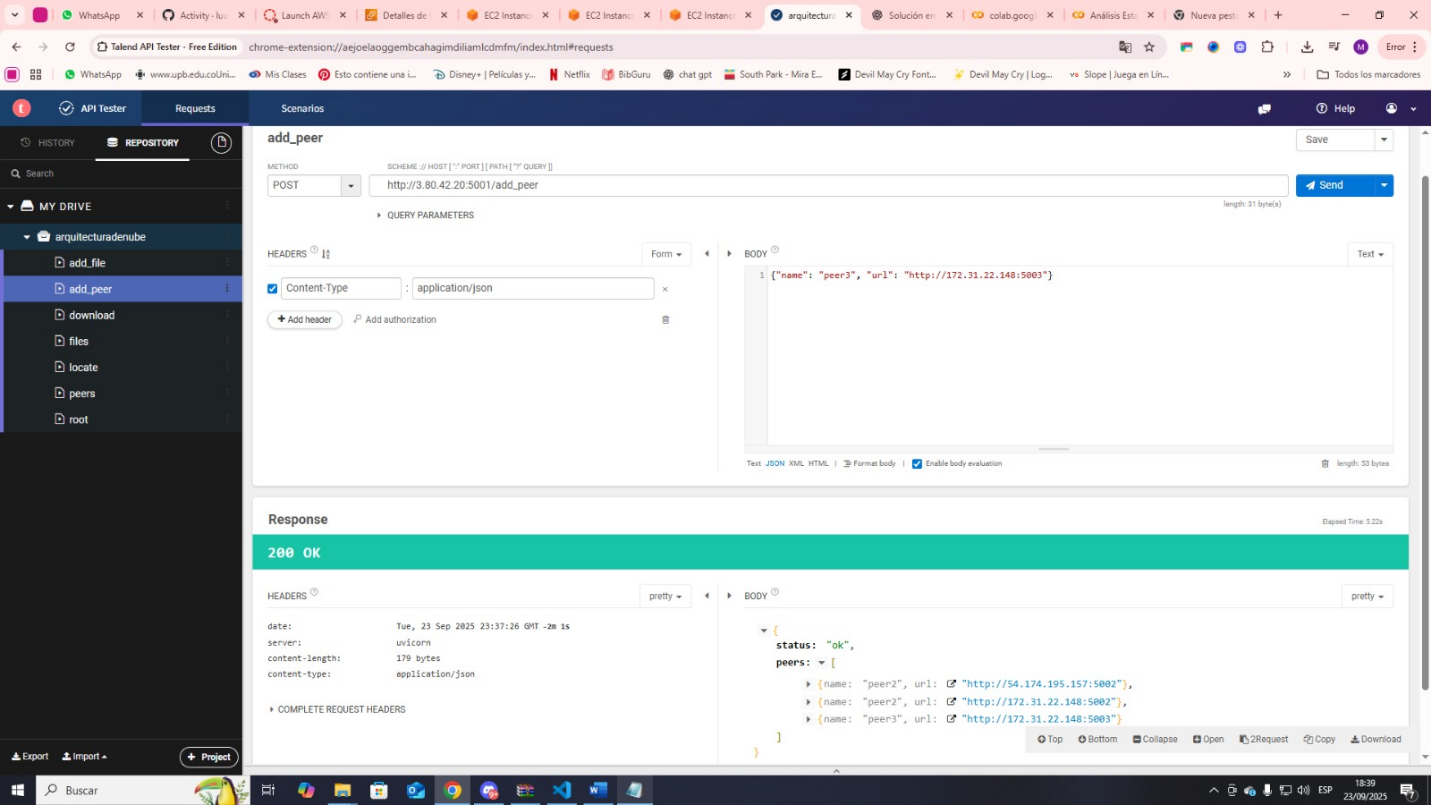
AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.



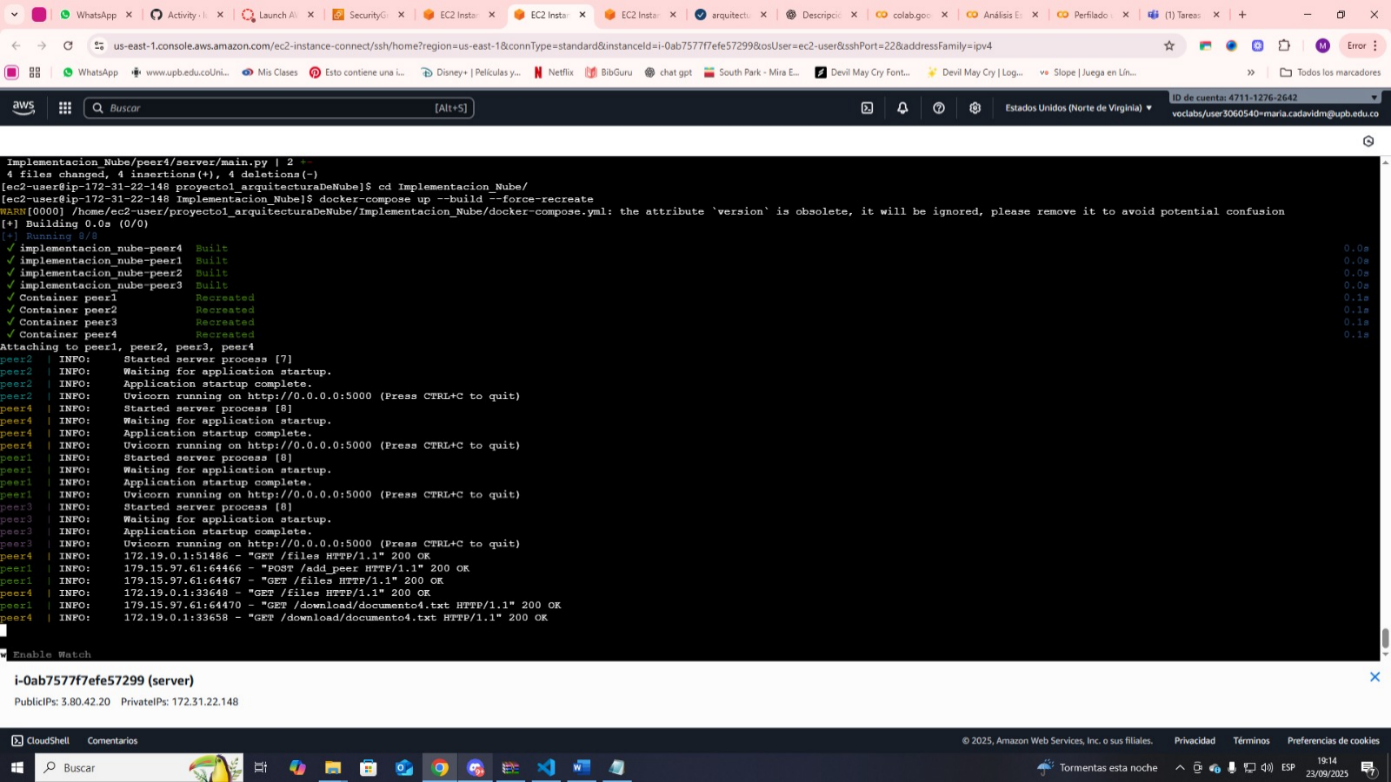




* Análisis:
  + Se confirma que la configuración de puertos abiertos en AWS (5001) es correcta y permite accesos externos.
  + Se confirma la funcionalidad distribuida en un entorno real de Internet.
  + El servicio respondió correctamente, confirmando la accesibilidad.
  + Correcto funcionamiento del microservicio de índice en entorno real.
  + validanda la actualización dinámica de índices.
  + evidencia la descentralización y la expansión dinámica de la red.

6. Evidencia de comunicación entre peers (peer1 ↔ peer4)

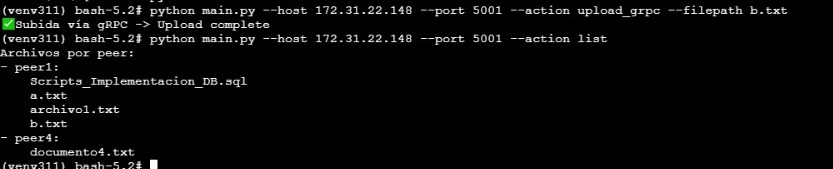
* Objetivo: Demostrar que los peers pueden interactuar de manera directa.
* Procedimiento: Peer 1 solicitó un archivo al Peer 4 vía gRPC usando el puerto 5001.

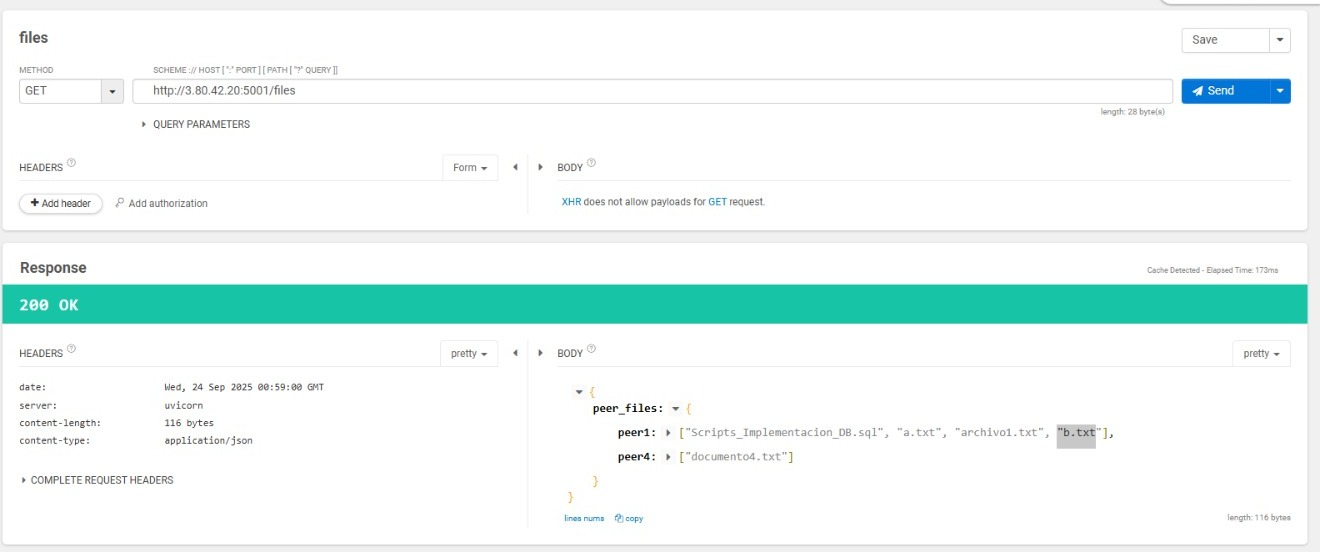


* Análisis: Se comprueba que los peers no solo conocen la existencia de otros, sino que también intercambian recursos directamente, cumpliendo con la esencia de una red P2P.

7. Prueba de subida de archivos y verificación

* Objetivo: Validar la funcionalidad de carga (upload) en un peer.





* Análisis: La prueba confirma que el sistema soporta la operación de subida y que la información se refleja correctamente en el índice de archivos.