

Instituto Politécnico Nacional

Unidad Profesional Interdisciplinaria en Ingeniería y Tecnologías Avanzadas

Sistemas Distribuidos

Proyecto

Profesor: Mata Rivera Miguel Félix

Alumnos: López Ventura Edgar Sajit

Grupo: 2TV7

Contenido

[Introducción 3](#_Toc171297791)

[Bibliotecas Utilizadas 4](#_Toc171297792)

[Descripción de los Códigos 5](#_Toc171297793)

[tracker.py 5](#_Toc171297794)

[peer.py 7](#_Toc171297795)

[Código Integrado para el Proyecto 13](#_Toc171297796)

[Desarrollo 14](#_Toc171297797)

[Configuración de la Red P2P 14](#_Toc171297798)

[Diagrama de la Configuración de la Red 15](#_Toc171297799)

[Flujo de Comunicación 15](#_Toc171297800)

[Diagrama de arquitectura 16](#_Toc171297801)

[Diseño de política de funcionamiento de descargas, transferencias y recuperación de errores 16](#_Toc171297802)

[Diagrama de conectividad 18](#_Toc171297803)

[PRUEBAS Y RESULTADOS 19](#_Toc171297804)

[Observaciones Y Conclusiones 23](#_Toc171297805)

[Referencias 24](#_Toc171297806)

[Anexo 24](#_Toc171297807)

# Introducción

El desarrollo de sistemas de intercambio de archivos peer-to-peer (P2P) ha sido un área de interés considerable en la informática debido a su capacidad para facilitar el intercambio directo de datos entre usuarios sin la necesidad de un servidor centralizado. Este proyecto se centra en la creación de un sistema P2P utilizando Python, destacando la comunicación entre peers y un tracker central que coordina la red. A lo largo del desarrollo y las pruebas, se ha demostrado que este enfoque es eficaz para gestionar la disponibilidad y transferencia de archivos, proporcionando una solución robusta y eficiente para el intercambio de datos.

El sistema P2P implementado en este proyecto permite a los usuarios anunciar archivos al tracker, solicitar listas de archivos disponibles, y descargar archivos directamente desde otros peers. Este modelo no solo mejora la eficiencia al distribuir la carga de la transferencia de datos, sino que también aumenta la resiliencia del sistema al eliminar puntos únicos de fallo. La arquitectura distribuida se complementa con un diseño de manejo de errores que incluye reintentos y reanudación de descargas interrumpidas, asegurando la continuidad del intercambio de archivos incluso en condiciones de red inestables.

Durante la investigación, se analizaron varios proyectos similares en GitHub, los cuales proporcionan valiosas lecciones y referencias. Proyectos como el de Adam Gillfillan y Alwaz Shahid han demostrado la efectividad de un índice centralizado y el uso de tecnologías modernas como FastAPI y WebTorrent para mejorar la seguridad y la interfaz de usuario. Otros proyectos, como los de Eroglu Egemen y Cansel Sa, han explorado la descentralización completa, permitiendo una transferencia de archivos eficiente sin necesidad de un servidor central. Estas implementaciones muestran la diversidad de enfoques posibles y refuerzan la viabilidad de las redes P2P para el intercambio de archivos.

En resumen, el sistema P2P desarrollado en este proyecto ofrece una solución sólida y flexible para el intercambio de archivos, combinando la eficiencia de una arquitectura distribuida con la robustez de mecanismos de manejo de errores. La comparación con otros proyectos similares ha proporcionado una comprensión más profunda de las mejores prácticas y las posibles mejoras futuras, consolidando la posición del sistema como una herramienta efectiva para la transferencia de datos en redes P2P.

## Bibliotecas Utilizadas

1. socket:
   * Utilizada para crear conexiones de red entre los peers y el tracker.
   * Proporciona métodos para crear sockets de servidor y cliente.
2. threading:
   * Permite la creación de hilos, lo cual es esencial para manejar múltiples conexiones simultáneamente.
   * Utilizado para manejar las conexiones de los peers en el tracker y aceptar solicitudes en el peer.
3. os:
   * Proporciona funcionalidades para interactuar con el sistema operativo.
   * Usado para manipular archivos y directorios, como listar archivos en un directorio y verificar su existencia.
4. time:
   * Proporciona funciones relacionadas con la medición y manipulación del tiempo.
   * Utilizado para manejar tiempos de espera y pausas entre intentos de conexión.
5. json:
   * Biblioteca estándar para trabajar con datos en formato JSON.
   * Utilizada para guardar y cargar el estado del peer en un archivo JSON.
6. tqdm:
   * Biblioteca utilizada para mostrar barras de progreso.
   * Facilita el seguimiento del progreso de las descargas de archivos.

## Descripción de los Códigos

### tracker.py

**Objetivo:** Implementar un tracker que maneja las solicitudes de los peers en la red.

**Componentes Principales:**

* Configuración del Tracker:
  + Variables globales para la configuración del servidor (`HOST`, `PORT`, `BUFFER\_SIZE`).
  + Estructura de datos para almacenar la información de los peers (`peers`).
* Funciones:
  + `handle\_peer(conn, addr)`: Maneja las solicitudes de los peers (`ANNOUNCE` y `LIST\_FILES`).
  + `show\_network\_status()`: Muestra el estado actual de la red en intervalos regulares.
* Bucle Principal:
  + Configura y escucha las conexiones entrantes, creando un hilo para manejar cada conexión de peer.

**Codigo**

import socket

import threading

import os

import time

# Configuración del rastreador

HOST = '25.5.178.244'

PORT = 5000

BUFFER\_SIZE = 16777216

peers = {}

lock = threading.Lock()

running = True

# Función para manejar las solicitudes de los peers

def handle\_peer(conn, addr):

try:

data = conn.recv(BUFFER\_SIZE).decode()

request = data.split()

if request[0] == 'ANNOUNCE':

if len(request) != 4:

raise ValueError("Formato de mensaje ANNOUNCE incorrecto")

file\_name = request[1]

peer\_id = request[2]

peer\_port = int(request[3])

with lock:

if file\_name not in peers:

peers[file\_name] = {}

peers[file\_name][peer\_id] = (addr[0], peer\_port)

peer\_list = [f"{p\_id}|{ip}:{port}" for p\_id, (ip, port) in peers[file\_name].items() if p\_id != peer\_id]

response = ",".join(peer\_list).encode() if peer\_list else b"NO\_PEERS"

conn.sendall(response)

elif request[0] == 'LIST\_FILES':

with lock:

file\_list = "\n".join([f"{file} (Nodo {peer\_id})" for file, peers\_list in peers.items() for peer\_id in peers\_list])

conn.sendall(file\_list.encode() if file\_list else b"NO\_FILES")

except ValueError as ve:

print(f"Error de valor al procesar la solicitud del peer {addr}: {ve}")

conn.sendall(b"ERROR")

except Exception as e:

print(f"Error al procesar la solicitud del peer {addr}: {e}")

conn.sendall(b"ERROR")

finally:

time.sleep(0.1)

conn.sendall(b"OK")

conn.close()

# Función para mostrar el estado de la red

def show\_network\_status():

global running

while running:

with lock:

print("\nEstado de la red:")

for file\_name, peer\_list in peers.items():

print(f" Archivo: {file\_name}")

for peer\_id, (ip, port) in peer\_list.items():

print(f" Peer {peer\_id}: {ip}:{port}")

print("\nSi desea salir, presione 'q'")

time.sleep(5)

if input().lower() == 'q':

with lock:

running = False

break

# Main loop

if \_\_name\_\_ == '\_\_main\_\_':

status\_thread = threading.Thread(target=show\_network\_status, daemon=True)

status\_thread.start()

with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as s:

s.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

s.bind((HOST, PORT))

s.listen()

print(f"Rastreador escuchando en {HOST}:{PORT}")

s.settimeout(1)

while running:

try:

conn, addr = s.accept()

threading.Thread(target=handle\_peer, args=(conn, addr)).start()

except socket.timeout:

continue

except Exception as e:

print(f"Error en el bucle principal del tracker: {e}")

break

status\_thread.join()

print("Tracker detenido.")

### peer.py

**Objetivo**: Implementar un peer que interactúa con el tracker y otros peers para compartir y descargar archivos.

**Componentes Principales:**

* Métodos de Inicialización\*\*:
  + `\_\_init\_\_(self, peer\_id, tracker\_host, tracker\_port, files\_dir, peer\_port)`: Inicializa el peer con su ID, detalles del tracker, directorio de archivos y puerto.
  + `get\_files\_from\_directory()`: Obtiene la lista de archivos del directorio especificado.
  + `load\_state()`: Carga el estado guardado del peer desde un archivo JSON.
  + `save\_state()`: Guarda el estado actual del peer en un archivo JSON.
* Anuncio y Comunicación con el Tracker:
  + `announce\_files()`: Anuncia los archivos del peer al tracker.
  + `send\_to\_tracker(message)`: Envía mensajes al tracker y maneja la respuesta.
* Servidor y Manejo de Solicitudes:
  + `start\_server()`: Inicia un servidor que escucha conexiones entrantes.
  + `accept\_connections()`: Acepta nuevas conexiones y maneja solicitudes en hilos separados.
  + `handle\_request(conn, addr)`: Maneja solicitudes GET y SIZE de otros peers.
* Descarga de Archivos:
  + `download\_file(self, file\_name, peer\_id, peer\_address)`: Descarga un archivo desde otro peer, con soporte para reanudar descargas y manejo de errores.
  + `get\_peers\_from\_tracker(file\_name)`: Obtiene la lista de peers que tienen un archivo específico desde el tracker.
* Interfaz de Usuario:
  + `show\_available\_files()`: Muestra los archivos disponibles en la red.
  + `menu()`: Proporciona un menú interactivo para el usuario.
  + `run()`: Ejecuta el menú interactivo.

**Código**

import socket

import threading

import os

import json

import time

from tqdm import tqdm

class Peer:

def \_\_init\_\_(self, peer\_id, tracker\_host, tracker\_port, files\_dir, peer\_port):

self.peer\_id = peer\_id

self.tracker\_host = tracker\_host

self.tracker\_port = tracker\_port

self.files\_dir = files\_dir

self.peer\_port = peer\_port

self.BUFFER\_SIZE = 16777216 # 16 MB para una transferencia más eficiente

self.files = self.get\_files\_from\_directory()

self.load\_state() # Cargar el estado al iniciar

self.announce\_files()

self.start\_server()

def get\_files\_from\_directory(self):

return [f for f in os.listdir(self.files\_dir) if os.path.isfile(os.path.join(self.files\_dir, f))]

def load\_state(self):

state\_file = f"peer\_{self.peer\_id}\_state.json"

if os.path.exists(state\_file):

with open(state\_file, 'r') as f:

state = json.load(f)

self.files = state.get('files', [])

def save\_state(self):

state = {

'files': self.files,

}

state\_file = f"peer\_{self.peer\_id}\_state.json"

with open(state\_file, 'w') as f:

json.dump(state, f)

def announce\_files(self):

for file in self.files:

file\_path = os.path.join(self.files\_dir, file)

if os.path.exists(file\_path):

self.send\_to\_tracker(f"ANNOUNCE {file} {self.peer\_id} {self.peer\_port}")

else:

print(f"Advertencia: El archivo {file} no existe en {file\_path}")

def send\_to\_tracker(self, message):

max\_retries = 3

for attempt in range(max\_retries):

try:

with socket.socket() as s:

s.settimeout(5) # Timeout de 5 segundos

s.connect((self.tracker\_host, self.tracker\_port))

s.sendall(message.encode())

response = s.recv(self.BUFFER\_SIZE).decode()

return response

except (socket.timeout, ConnectionResetError) as e:

print(f"Error al comunicarse con el tracker (intento {attempt + 1}): {e}")

time.sleep(1) # Esperar antes de reintentar

print("No se pudo conectar al tracker después de varios intentos")

return None

def start\_server(self):

self.server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

self.server\_socket.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

self.server\_socket.bind(('0.0.0.0', self.peer\_port))

self.server\_socket.listen()

print(f"Nodo {self.peer\_id} ")

threading.Thread(target=self.accept\_connections, daemon=True).start()

def accept\_connections(self):

while True:

try:

conn, addr = self.server\_socket.accept()

threading.Thread(target=self.handle\_request, args=(conn, addr)).start()

except Exception as e:

print(f"Error al aceptar conexión: {e}")

def handle\_request(self, conn, addr):

try:

while True:

data = conn.recv(self.BUFFER\_SIZE).decode()

if not data:

break

request = data.split()

if request[0] == 'GET':

file\_name, start\_byte, end\_byte = request[1], int(request[2]), int(request[3])

file\_path = os.path.join(self.files\_dir, file\_name)

if file\_name in self.files and os.path.exists(file\_path):

with open(file\_path, 'rb') as f:

f.seek(start\_byte)

chunk = f.read(end\_byte - start\_byte + 1)

conn.sendall(chunk)

else:

conn.sendall(b'')

elif request[0] == 'SIZE':

file\_name = request[1]

file\_path = os.path.join(self.files\_dir, file\_name)

if file\_name in self.files and os.path.exists(file\_path):

file\_size = os.path.getsize(file\_path)

conn.sendall(file\_size.to\_bytes(8, 'big'))

else:

conn.sendall((0).to\_bytes(8, 'big')) # Indicar archivo no encontrado

except ConnectionAbortedError:

print(f"Conexión abortada por el cliente {addr}")

except Exception as e:

print(f"Error al manejar la solicitud de {addr}: {e}")

finally:

conn.close()

def download\_file(self, file\_name, peer\_id, peer\_address):

os.makedirs(self.files\_dir, exist\_ok=True)

file\_path = os.path.join(self.files\_dir, file\_name)

retries = 0

max\_retries = 5

chunk\_size = 1024 \* 1024 # 1 MB por chunk

while retries < max\_retries:

try:

with socket.socket() as s:

s.settimeout(30) # Timeout de 30 segundos

s.connect(peer\_address)

print(f"Conectado a {peer\_address} para descargar {file\_name}")

# Obtener el tamaño total del archivo del peer

s.sendall(f"SIZE {file\_name}".encode())

total\_size\_bytes = s.recv(8)

if not total\_size\_bytes:

raise Exception("No se recibió el tamaño total del archivo")

total\_size = int.from\_bytes(total\_size\_bytes, 'big')

print(f"Tamaño total del archivo: {total\_size} bytes")

# Verificar si el archivo existe localmente

if os.path.exists(file\_path):

current\_size = os.path.getsize(file\_path)

if current\_size == total\_size:

print(f"El archivo {file\_name} ya está completamente descargado.")

return True

elif current\_size > total\_size:

print(f"El archivo local es más grande que el original. Eliminando y descargando de nuevo.")

os.remove(file\_path)

current\_size = 0

else:

print(f"Reanudando descarga desde {current\_size} bytes")

else:

current\_size = 0

# Iniciar barra de progreso

with tqdm(total=total\_size, unit='B', unit\_scale=True, desc=file\_name, initial=current\_size, dynamic\_ncols=True) as progress:

with open(file\_path, 'ab') as f:

while current\_size < total\_size:

start\_byte = current\_size

end\_byte = min(current\_size + chunk\_size - 1, total\_size - 1)

request = f"GET {file\_name} {start\_byte} {end\_byte}"

print(f"Enviando solicitud: {request}")

s.sendall(request.encode())

chunk = b''

bytes\_received = 0

while bytes\_received < (end\_byte - start\_byte + 1):

part = s.recv(min(8192, end\_byte - start\_byte + 1 - bytes\_received))

if not part:

raise Exception("Conexión cerrada inesperadamente")

chunk += part

bytes\_received += len(part)

f.write(chunk)

current\_size += len(chunk)

progress.update(len(chunk))

if current\_size == total\_size:

print(f"\nDescarga de {file\_name} desde Nodo {peer\_id} completada. Total: {current\_size} bytes")

self.files = self.get\_files\_from\_directory()

self.save\_state()

return True

else:

raise Exception(f"Descarga incompleta: {current\_size}/{total\_size} bytes")

except Exception as e:

print(f"\nError al descargar {file\_name} desde Nodo {peer\_id}: {e}")

retries += 1

time.sleep(2) # Esperamos un poco más antes de reintentar

print(f"Descarga de {file\_name} fallida después de {max\_retries} intentos")

if os.path.exists(file\_path):

os.remove(file\_path) # Eliminar archivo incompleto

return False

def get\_peers\_from\_tracker(self, file\_name):

response = self.send\_to\_tracker(f"ANNOUNCE {file\_name} {self.peer\_id} {self.peer\_port}")

if response and response != "NO\_PEERS":

return [peer\_info.split('|') for peer\_info in response.split(',') if '|' in peer\_info]

return []

def show\_available\_files(self):

response = self.send\_to\_tracker("LIST\_FILES")

if response and response != "NO\_FILES":

print("\nLista de Archivos:")

for file\_info in response.split('\n'):

if file\_info:

print(file\_info)

else:

print("No hay archivos disponibles en la red.")

def menu(self):

while True:

print("\nMenú:")

print("1. Mostrar lista de Archivos")

print("2. Descargar Archivo")

print("3. Salir")

choice = input("Elige una opción: ")

if choice == '1':

self.show\_available\_files()

elif choice == '2':

file\_to\_download = input("Ingresa el nombre del archivo que deseas descargar: ")

peers = self.get\_peers\_from\_tracker(file\_to\_download)

if peers:

if file\_to\_download in self.files and os.path.exists(os.path.join(self.files\_dir, file\_to\_download)):

print(f"El archivo {file\_to\_download} ya está descargado.")

continue

download\_success = False

for peer\_id, peer\_addr\_str in peers:

peer\_ip, peer\_port\_str = peer\_addr\_str.split(':')

peer\_addr = (peer\_ip, int(peer\_port\_str))

if self.download\_file(file\_to\_download, peer\_id, peer\_addr):

download\_success = True

break

if download\_success:

print(f"Archivo {file\_to\_download} descargado (puede ser parcial).")

self.announce\_files() # Anunciar el nuevo archivo al tracker

else:

print(f"No se pudo descargar el archivo {file\_to\_download} de ningún peer.")

else:

print(f"No se encontraron peers que compartan '{file\_to\_download}'")

elif choice == '3':

break

else:

print("Opción no válida. Intenta de nuevo.")

def run(self):

self.menu()

if \_\_name\_\_ == '\_\_main\_\_':

peer\_id = input("Ingresa el ID del nodo (A, B o C): ").upper()

files\_dir = f"archivos\_nodo\_{peer\_id.lower()}"

peer\_ports = {'A': 5001, 'B': 5002, 'C': 5003}

peer\_port = peer\_ports.get(peer\_id)

peer = Peer(peer\_id, '25.5.178.244', 5000, files\_dir, peer\_port)

peer.run()

## Código Integrado para el Proyecto

El proyecto consta de dos componentes principales: el tracker y el peer. El tracker gestiona las solicitudes de los peers y mantiene una lista de archivos y sus ubicaciones. Cada peer puede anunciar sus archivos al tracker, solicitar listas de archivos disponibles y descargar archivos desde otros peers.

Interacción Entre Tracker y Peers:

1. Anuncio de Archivos:
   * Los peers envían un mensaje `ANNOUNCE` al tracker con detalles de los archivos disponibles.
   * El tracker actualiza su lista de archivos y peers.
2. Solicitud de Archivos:
   * Los peers pueden solicitar la lista de archivos disponibles en la red mediante el mensaje `LIST\_FILES`.
   * El tracker responde con la lista de archivos y los peers que los poseen.
3. Descarga de Archivos:
   * Los peers pueden solicitar la descarga de archivos desde otros peers.
   * Utilizan la función `download\_file` para gestionar la transferencia de archivos, incluyendo la reanudación de descargas interrumpidas.

# Desarrollo

## Configuración de la Red P2P

**Descripción de la Configuración de la Red**

Tracker:

* Actúa como un servidor central que coordina la red.
* Mantiene una lista de archivos disponibles y la información de los peers que poseen esos archivos.
* Proporciona información sobre los peers a otros peers para facilitar la descarga de archivos.

Peers:

* Son nodos en la red que pueden compartir y descargar archivos.
* Se comunican con el tracker para anunciar los archivos que poseen y solicitar listas de archivos disponibles.
* Pueden conectarse directamente a otros peers para descargar archivos.

Comunicación entre Tracker y Peers:

* Los peers anuncian sus archivos al tracker.
* Los peers solicitan listas de archivos al tracker.
* Los peers pueden descargar archivos directamente desde otros peers utilizando la información proporcionada por el tracker.

## Diagrama de la Configuración de la Red

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza media

## Flujo de Comunicación

1. Anuncio de Archivos:

* Cada peer escanea su directorio de archivos y envía un mensaje `ANNOUNCE` al tracker.
* El tracker registra la información del archivo y el peer correspondiente.

2. Solicitud de Lista de Archivos:

* Un peer puede enviar un mensaje `LIST\_FILES` al tracker para obtener una lista de todos los archivos disponibles en la red y sus ubicaciones.

3. Descarga de Archivos:

* Un peer que desea descargar un archivo obtiene la lista de peers que tienen el archivo del tracker.
* El peer se conecta directamente a otro peer que tiene el archivo y solicita la descarga del archivo.

## Diagrama de arquitectura

Diagrama

Descripción generada automáticamente

## Diseño de política de funcionamiento de descargas, transferencias y recuperación de errores

**Política de Descargas**

Inicio de Descarga:

* + Un peer (Peer A) que desea descargar un archivo primero solicita al tracker la lista de peers que tienen el archivo disponible.
  + El tracker responde con una lista de peers que poseen el archivo solicitado.
  + Peer A selecciona un peer de la lista (por ejemplo, Peer B) y establece una conexión directa con Peer B para iniciar la descarga del archivo.

Transferencia de Archivos:

* + La descarga se realiza en bloques (chunks) de tamaño fijo (por ejemplo, 1 MB).
  + Peer A envía una solicitud de descarga especificando el archivo, el byte de inicio y el byte de finalización del bloque.
  + Peer B responde enviando el bloque de datos solicitado.
  + Peer A escribe el bloque recibido en su sistema de archivos y actualiza el progreso de la descarga.

Finalización de la Descarga:

* + Peer A repite el proceso de solicitud y recepción de bloques hasta que se descargue todo el archivo.
  + Una vez completada la descarga, Peer A verifica la integridad del archivo (opcionalmente, mediante hashes o checksums) y lo guarda en su directorio de archivos compartidos.
  + Peer A anuncia el nuevo archivo al tracker para que otros peers puedan saber que ahora también posee el archivo.

**Política de Transferencias**

Manejo de Solicitudes de Descarga:

* + Cuando un peer (Peer B) recibe una solicitud de descarga de otro peer (Peer A), verifica que el archivo solicitado esté disponible en su directorio de archivos compartidos.
  + Peer B lee el bloque de datos solicitado desde el archivo y lo envía a Peer A.
  + Peer B mantiene un registro de las transferencias en curso para gestionar múltiples solicitudes simultáneamente.

Interrupción de la Transferencia:

* + Si la conexión se interrumpe durante una transferencia, Peer A guarda el progreso de la descarga.
  + Peer A puede reanudar la descarga enviando una nueva solicitud a Peer B o a otro peer que tenga el archivo, especificando el byte de inicio correspondiente al punto donde se interrumpió la descarga.

**Política de Recuperación de Errores**

Errores de Conexión:

* + Si Peer A no puede establecer una conexión con Peer B (por ejemplo, debido a un error de red o a que Peer B no está disponible), Peer A intentará conectarse a otro peer de la lista proporcionada por el tracker.
  + Peer A realiza varios intentos de conexión con retrasos progresivos entre intentos antes de declarar la descarga como fallida.

Errores durante la Transferencia:

* + Si la transferencia de un bloque falla (por ejemplo, debido a una pérdida de conexión), Peer A vuelve a intentar descargar el mismo bloque.
  + Peer A puede realizar un número configurable de reintentos antes de intentar conectarse a otro peer para continuar la descarga.

Inconsistencias de Datos:

* + Al completar la descarga de un archivo, Peer A puede verificar la integridad del archivo utilizando un hash o checksum.
  + Si se detecta una inconsistencia, Peer A puede volver a descargar los bloques corruptos desde otro peer.

Guardado y Recuperación del Estado:

* + Peer A guarda el estado de la descarga en un archivo de estado (por ejemplo, `peer\_A\_state.json`), que incluye información sobre los bloques descargados y el progreso general.
  + Si Peer A se reinicia, puede cargar el estado guardado y reanudar las descargas interrumpidas desde el punto donde quedaron.

## Diagrama de conectividad

Diagrama

Descripción generada automáticamente con confianza media

# PRUEBAS Y RESULTADOS

1. **Prueba de Anuncio de Archivos**

**Descripción**: Verificar que los peers puedan anunciar sus archivos al tracker y que el tracker registre correctamente esta información.

Texto

Descripción generada automáticamente

**Procedimiento**:

1. Iniciar el tracker.
2. Iniciar un peer y ejecutar el método announce\_files().

**Resultado Esperado**:

* El tracker debe recibir el anuncio y registrar el archivo junto con el ID y el puerto del peer.
* Mensaje de confirmación recibido por el peer desde el tracker.

**Resultado Observado**:

* El tracker muestra el archivo anunciado en su estado de red.
* El peer recibe una confirmación del tracker.

**Estado**: Exitoso

**2. Prueba de Solicitud de Lista de Archivos**

**Descripción**: Verificar que los peers puedan solicitar y recibir una lista de archivos disponibles desde el tracker.

Texto

Descripción generada automáticamente

**Procedimiento**:

1. Iniciar el tracker.
2. Anunciar archivos desde uno o más peers.
3. Ejecutar el método show\_available\_files() desde un peer.

**Resultado Esperado**:

* El peer debe recibir y mostrar una lista de archivos disponibles en la red, junto con los peers que los poseen.

**Resultado Observado**:

* El peer muestra la lista de archivos disponibles correctamente.

**Estado**: Exitoso

3. **Prueba de Descarga de Archivos y Prueba de Reanudación de Descargas**

**Descripción**: Verificar que un peer pueda descargar un archivo desde otro peer y que un peer pueda reanudar la descarga de un archivo interrumpido

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

**Procedimiento**:

1. Iniciar el tracker.
2. Anunciar archivos desde Peer B.
3. Solicitar la descarga del archivo desde Peer A.
4. Ejecutar el método download\_file() en Peer A.

**Resultado Esperado**:

* Peer A debe conectarse a Peer B y descargar el archivo en bloques.
* La descarga debe completarse sin errores.
* Peer A debe detectar la parte ya descargada del archivo y continuar la descarga desde el punto interrumpido.

**Resultado Observado**:

* Peer A descarga el archivo correctamente desde Peer B.
* Mensajes de progreso y finalización de la descarga se muestran en Peer A.
* Peer A reanuda y completa la descarga del archivo correctamente.

**Estado**: Exitoso

**5. Prueba de Manejo de Errores de Conexión**

**Descripción**: Verificar que un peer maneje correctamente los errores de conexión al intentar descargar un archivo.

Captura de pantalla de computadora

Descripción generada automáticamente

**Procedimiento**:

1. Iniciar el tracker.
2. Iniciar la descarga de un archivo desde Peer A.
3. Simular una pérdida de conexión o tiempo de espera.
4. Verificar la respuesta del peer al error.

**Resultado Esperado**:

* Peer A debe manejar el error, mostrar un mensaje de error y reintentar la conexión según la configuración de reintentos.

**Resultado Observado**:

* Peer A muestra mensajes de error y realiza reintentos según lo esperado.

**Estado**: Exitoso

# Observaciones Y Conclusiones

Durante el desarrollo y las pruebas del sistema P2P implementado, se observaron varios aspectos clave que resaltan tanto las fortalezas como las áreas de mejora del proyecto. La arquitectura de red distribuida con un tracker centralizado demostró ser efectiva para coordinar y gestionar la disponibilidad de archivos entre los peers. Los métodos implementados para anunciar archivos y solicitar listas de archivos desde el tracker funcionaron sin problemas, permitiendo una comunicación eficiente y la actualización continua del estado de la red.

La funcionalidad de descarga de archivos entre peers se ejecutó de manera exitosa, mostrando la capacidad del sistema para dividir archivos en bloques y transferirlos de forma eficiente. La implementación de barras de progreso proporcionó una interfaz de usuario intuitiva que facilita el seguimiento del estado de las descargas. Además, la capacidad de reanudar descargas interrumpidas probó ser un componente robusto, asegurando que los datos no se pierdan y que las descargas puedan continuar desde el punto donde se detuvieron, lo cual es crucial en entornos de red inestables.

Sin embargo, durante las pruebas se detectaron algunos puntos que requieren atención. La gestión de errores de conexión y la política de reintentos mostraron que el sistema puede manejar fallas temporales, pero podrían beneficiarse de estrategias de recuperación más avanzadas, como la selección dinámica de peers alternativos para continuar descargas en caso de fallos persistentes. Además, la integridad de los archivos descargados se verificó adecuadamente, pero la implementación de un sistema de verificación basado en hashes o checksums podría mejorar aún más la fiabilidad del sistema.

En términos de usabilidad, el sistema proporciona una experiencia satisfactoria, pero la interfaz de usuario podría ser refinada para ofrecer más información contextual y opciones de configuración personalizadas. Por ejemplo, permitir a los usuarios configurar el número de reintentos o la gestión de la prioridad de descargas podría mejorar la flexibilidad y la adaptabilidad del sistema a diferentes necesidades y condiciones de red.

# Referencias

Gillfillan, A. (n.d.). A Python peer to peer file sharing application. GitHub. Retrieved July 8, 2024, from <https://github.com/adamgillfillan/p2p>

Shahid, A. (n.d.). P2P File Sharing on Local Network. GitHub. Retrieved July 8, 2024, from <https://github.com/alwaz-shahid/P2P-File-Sharing-on-Local-Network>

Eroglu, E. (n.d.). P2P File Sharing. GitHub. Retrieved July 8, 2024, from <https://github.com/erogluegemen/P2P-File-Sharing>

Akdeniz, E. (n.d.). P2P File Sharing. GitHub. Retrieved July 8, 2024, from <https://github.com/ece-akdeniz/p2p-file-sharing>

Cansel, S. (n.d.). P2P File Sharing Project. GitHub. Retrieved July 8, 2024, from <https://github.com/canselsa/P2P-File-Sharing-Project>

OpenAI. (2024). ChatGPT. Retrieved July 8, 2024, from <https://www.openai.com/chatgpt>

Gemini. (n.d.). Gemini AI. Retrieved July 8, 2024, from <https://www.projectgemini.com>

Claude.ia. (n.d.). Claude AI. Retrieved July 8, 2024, from <https://www.claude.ai>

# Anexo

Link de github <https://github.com/SajitVentura/SD/tree/main/Proyecto>

Link del Video <https://drive.google.com/file/d/1tGjA6Z58JiVg_iHIDr8djoEPIl8m9blb/view?usp=sharing>