

22

Table of Contents

[Overview 3](#_Toc196767070)

[Implemented Features 3](#_Toc196767071)

[Challenges Faced 3](#_Toc196767072)

[Phase 3 Plan 4](#_Toc196767073)

[Overview: 4](#_Toc196767074)

[1. Upgrade to Argon2 Password Hashing 4](#_Toc196767075)

[2. Session Persistence 4](#_Toc196767076)

[Implementation Plan 5](#_Toc196767077)

[1. File Encryption on Upload 5](#_Toc196767078)

[**Approach**: 5](#_Toc196767079)

[2. File Decryption on Download 5](#_Toc196767080)

[3. File Integrity Verification 5](#_Toc196767081)

[4. Key Management 6](#_Toc196767082)

[Network Architecture Overview 6](#_Toc196767083)

[2. Peer Behavior 7](#_Toc196767084)

[3. Discovery Mechanism: Rendezvous Server 7](#_Toc196767085)

[4. Communication and Operations 7](#_Toc196767086)

[5. Authentication and Session Management 8](#_Toc196767087)

[System Architecture 8](#_Toc196767088)

[Authentication Design Document 9](#_Toc196767089)

[1. User Credentials 9](#_Toc196767090)

[2. Password Hashing Algorithm 9](#_Toc196767091)

[3. Session Management 9](#_Toc196767092)

[Password handling: 10](#_Toc196767093)

[System Architecture 11](#_Toc196767094)

[COMPONENT DETAILS: 12](#_Toc196767095)

[AUTHENTICATION FLOW: 12](#_Toc196767096)

[FILE TRANSFER FLOW (Authenticated): 12](#_Toc196767097)

[User Manual 13](#_Toc196767098)

[Step 1: 13](#_Toc196767099)

[Step 2: 15](#_Toc196767100)

[Connect and Register peers 15](#_Toc196767101)

[Step 3: 16](#_Toc196767102)

[Show List of files: 16](#_Toc196767103)

[Download file 17](#_Toc196767104)

[Upload file: 18](#_Toc196767105)

[If not logged: 21](#_Toc196767106)

[File rendezvous\_server.py: 21](#_Toc196767107)

[File Peer.py: 22](#_Toc196767108)

P2P implemented + Phase 2

# Overview

In Phase 2, we enhanced the P2P file sharing system by adding secure user authentication mechanisms.  
Users must register and log in to perform file sharing actions, ensuring controlled access to shared files.

# Implemented Features

* **User Registration and Login**:  
  Users can create accounts by choosing a username and password. Authentication is required before accessing any file-sharing operation.
* **Password Hashing**:  
  Passwords are not stored in plain text. We use **SHA-256** hashing with a randomly generated salt. User credentials are securely stored in a local JSON file (user\_data/users.json).
* **Session Management**:  
  Upon successful login, users receive a **session token** (32-character hex string).  
  Sessions are stored **in-memory** (active\_sessions dictionary) and have a configurable timeout (default **3 minutes**).  
  Expired sessions are automatically invalidated.
* **Authentication-Integrated File Sharing**:  
  All file operations (LIST\_FILES, DOWNLOAD, UPLOAD) require a valid session token, enforcing that only authenticated users can interact with peers.

# Challenges Faced

* **Session Token Management**:  
  Implementing secure, per-peer session tokens and ensuring session renewal during activity required careful synchronization between client and server.
* **Handling Peer-to-Peer Dynamic Connections**:  
  Each peer maintains independent authentication per connected peer, which added complexity compared to a traditional centralized server model.
* **Partial Data Transfers**:  
  During file downloads, ensuring that file transfers were terminated correctly (<EOF> marker) without causing connection resets was challenging.

# **Phase 3 Plan**

## Overview:

* **File Encryption**
  + Implement AES-256 for file encryption
  + Key derivation from user credentials
* **Integrity Verification**
  + SHA-256 checksums for uploaded files
  + Automatic validation on download
* **Enhanced Security**
  + Upgrade to Argon2 password hashing

### 1. **Upgrade to Argon2 Password Hashing**

**Current Limitation**:  
The system uses SHA-256 with salting, which is cryptographically secure but vulnerable to brute-force attacks if the password database is compromised. Modern systems use memory-hard algorithms like Argon2 to resist GPU/ASIC cracking.

**Recommendation**:  
Replace SHA-256 with **Argon2id** (the current industry standard for password hashing). Key advantages:

* **Memory-hard computation**: Requires significant RAM, making large-scale attacks impractical.
* **Time and parallelism tuning**: Can be adjusted to balance security and performance.
* **Built-in salt handling**: Simplifies implementation while maintaining security.

### 2. **Session Persistence**

**Current Limitation**:  
Sessions are stored in memory, so they vanish if the peer server restarts, forcing users to reauthenticate.

**Recommendation**:  
Persist sessions to a **lightweight database** (e.g., SQLite) with:

* **Encrypted session tokens**: Store only hashed tokens to prevent misuse if the database is breached.
* **Automatic cleanup**: Regularly purge expired sessions.

## **Implementation Plan**

### **File Encryption on Upload**

### **Approach**:

* Use **AES-256-GCM** (symmetric encryption) for files:
  + Combines encryption and authentication (detects tampering).
  + Operates in "streaming mode" to handle large files efficiently.
* Generate a **unique symmetric key per file** (not shared globally).

**Workflow**:

1. User selects a file to upload.
2. System:
   * Generates a random AES key (secrets.token\_bytes(32)).
   * Encrypts the file in chunks (to avoid memory overload).
   * Stores the encrypted file in shared\_files/.

### **File Decryption on Download**

**Approach**:

* Only users with the correct key can decrypt files.
* Decryption happens after file transfer to avoid exposing plaintext.

**Workflow**:

1. User requests a file download.
2. System:
   * Transfers the encrypted file (as binary chunks).
   * Provides the AES key to the recipient (via secure channel).
   * Recipient decrypts the file locally using the key.

### **File Integrity Verification**

**Approach**:

* Use **SHA-256 hashes** to verify file integrity:
  + Generate a hash before encryption (stores with file metadata).
  + Recompute hash after decryption and compare.

**Workflow**:

1. On upload:
   * Compute SHA-256(file\_content) → store as file\_hash.
2. On download:
   * Recompute SHA-256(decrypted\_content).
   * Abort if hashes mismatch (tampering detected).

### **Key Management**

**Simplified Approach**:

* **Option A (Single Key)**:
  + Use one pre-shared AES key for all files (easiest but insecure).
  + **Risk**: Compromised key exposes all files.
* **Option B (Per-File Keys)**:
  + Generate a unique AES key per file.
  + Share keys securely:
    - Encrypt the AES key with the recipient’s public key (if PKI is available).
    - **Fallback**: Use **password-derived keys** (e.g., hash the recipient’s password + salt).
* **Option C (Diffie-Hellman Lite)**:
  + Peers agree on a shared secret during session setup.
  + Derive file keys from this secret.
  + **Trade-off**: More complex but eliminates key storage.

|  |  |  |
| --- | --- | --- |
| Upload Flow | Transfer Flow | Download Flow |
| * Hash file (SHA-256) * 2. Encrypt (AES-256) | * Send encrypted chunks over TCP * Send metadata:   + File hash   + - Key (secured) | * Receive encrypted chunks * Decrypt locally with provided key * 3. Verify hash |

## **Network Architecture Overview**

The system consists of two major components:

| **Component** | **Role** |
| --- | --- |
| Rendezvous Server (rendezvous\_server.py) | Simple centralized server used ONLY for helping peers discover each other. |
| Peer Node (peer.py) | Each peer is both a server (serving files) and a client (requesting files). |

## 2. **Peer Behavior**

Every peer.py node does **two things simultaneously**:

* **Server Mode**:
  + Listens for incoming connections from other peers.
  + Serves file requests (list, download, upload) **only to authenticated users**.
* **Client Mode**:
  + Connects to other peers to request files or upload files.
  + Handles login/registration to each peer separately.

## 3. **Discovery Mechanism: Rendezvous Server**

* When a peer starts:
  1. It connects to the **Rendezvous Server** (rendezvous\_server.py).
  2. Sends a REGISTER <IP> <PORT> command to announce itself.
  3. The server adds it to the known peer list.
  4. The server sends back a list of **other currently active peers** (except the newly joined one).
* **Peers can refresh the peer list dynamically** during runtime by connecting again to the rendezvous server.

## 4. **Communication and Operations**

After discovery, a peer can:

| **Operation** | **How It Works** |
| --- | --- |
| Login/Register | A peer must login/register to another peer to get a session token. |
| List Files | Authenticated peer sends session\_token LIST\_FILES and receives file names. |
| Download File | Authenticated peer sends session\_token DOWNLOAD filename and receives file content. |
| Upload File | Authenticated peer sends session\_token UPLOAD filename, sends file, and gets confirmation. |

## 5. **Authentication and Session Management**

* **Usernames and passwords** are created during registration and are hashed with **SHA-256** and a random salt.
* Upon successful login, a **unique session token** is issued.
* Session tokens are stored **in-memory** and expire after inactivity (3 minutes).

# System Architecture

A diagram of a computer network

Description automatically generated

# 

# Authentication Design Document

## 1. User Credentials

* Usernames and passwords are collected at registration.
* Passwords are **hashed using SHA-256** combined with a randomly generated **salt**.
* Stored in user\_data/users.json in the format:

{

"username1": {

"password\_hash": "hashed\_password\_here",

"salt": "random\_salt\_here"

},

"username2": { ... }

}

## 2. Password Hashing Algorithm

* **Algorithm Used**: SHA-256
* **Salt Generation**: secrets.token\_hex(16) (16 random hex characters)
* **Hashing Process**:

hashed\_password = SHA-256(password + salt)

def hash\_password(password, salt=None):

if not salt:

salt = secrets.token\_hex(16)

salted = password + salt

return hashlib.sha256(salted.encode()).hexdigest(), salt

* **Security Properties**:
  + Salt prevents rainbow table attacks
  + SHA-256 provides collision resistance
  + Future upgrade path to Argon2

## 3. Session Management

* On successful login, a **session token** (32 hex characters) is issued.
* Session data is stored in-memory inside:

active\_sessions = {

session\_token: {

"username": username,

"expiry": datetime.now() + timedelta(minutes=3)

}

}

* **Session Expiry**: Sessions expire after 30 minutes of inactivity. Renewal happens automatically on each valid operation.
* **Validation**:
  + Incoming commands must start with a valid session token.
  + If session is missing or expired, peer responds with authentication error.

# Password handling:

def hash\_password(password, salt=None):

    if salt is None:

        salt = secrets.token\_hex(16)

    hashed = hashlib.sha256((password + salt).encode()).hexdigest()

    return hashed, salt

def verify\_password(stored\_hash, salt, input\_password):

    input\_hash = hashlib.sha256((input\_password + salt).encode()).hexdigest()

    return input\_hash == stored\_hash

USER\_DATA\_FILE = "user\_data/users.json"

def load\_user\_data():

    if not os.path.exists(USER\_DATA\_FILE):

        os.makedirs(os.path.dirname(USER\_DATA\_FILE), exist\_ok=True)

        with open(USER\_DATA\_FILE, 'w') as f:

            json.dump({}, f)

        return {}

    with open(USER\_DATA\_FILE, 'r') as f:

        return json.load(f)

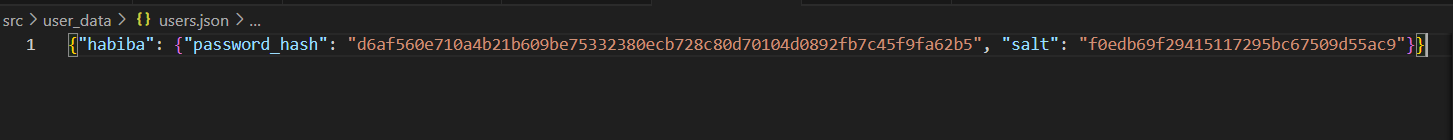
def save\_user\_data(data):

    with open(USER\_DATA\_FILE, 'w') as f:

        json.dump(data, f)

A screenshot of a computer

Description automatically generated



# System Architecture

A diagram of a server

Description automatically generated

## COMPONENT DETAILS:

1. Rendezvous Server:

- Centralized peer discovery

- Maintains list of active peers

- Returns peer list to registering nodes

- Simple TCP server (port 5000)

2. Peer Nodes (Each peer has both client and server components):

|  |  |
| --- | --- |
| Peer Server | Peer Client |
| * Listens on TCP port * Handles:   + Authentication   + File Operation * Session Management * User Data Storage | * Connects to other peers * Provides user menu * Initiates:   + Authentication   + File Transfers |

## AUTHENTICATION FLOW:

1. Peer registers with Rendezvous Server

2. Peer gets list of available peers

3. User selects peer and chooses:

A diagram of a computer program

Description automatically generated

### FILE TRANSFER FLOW (Authenticated):

1. User selects operation (upload/download)
2. Client verifies session token
3. **For download:**

A black text on a white background

Description automatically generated

1. **For upload:**

A black text on a white background

Description automatically generated

# User Manual

## Step 1:

* **Run rendezvous\_server.py**
* **Run 2 peer.py**

A screen shot of a computer

Description automatically generated

**Peer 1:**

A screenshot of a computer

Description automatically generated

**Peer 2:**

A screen shot of a computer

Description automatically generated

**Refresh Peer list in Peer 1 terminal:**

A screenshot of a computer

Description automatically generated

## Step 2:

### Connect and Register peers

**Peer 1:**

A screenshot of a computer

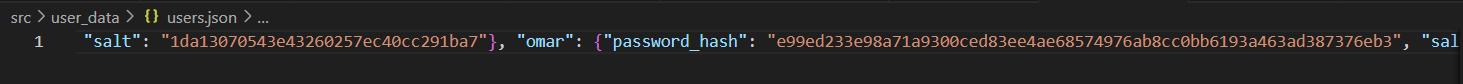
Description automatically generated

**Peer 2:**

A screenshot of a computer

Description automatically generated

**Users.json:**



## Step 3:

### Show List of files:

**It displays the list from shared\_files folder**

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

### Download file

**It downloads from the list of files in the shared\_files**

A screenshot of a computer

Description automatically generated

**It is saved in folder received\_files**

A screenshot of a computer

Description automatically generated

### Upload file:

**It opens the file explorer window to select file you want to share with other peers**

A screenshot of a computer

Description automatically generated

**Select the file and click ‘open’**

A screenshot of a computer

Description automatically generated

**After selecting, the session was opened for a while so it timed out and shows that authentication again is needed:**

A screenshot of a computer

Description automatically generated

**Continue after logging in again:**

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

## If not logged:

A screenshot of a computer program

Description automatically generated

# File rendezvous\_server.py:

# rendezvous\_server.py

import socket

import threading

HOST = '0.0.0.0'

PORT = 5000

peers = []

def handle\_client(conn, addr):

    try:

        data = conn.recv(1024).decode().strip()

        if data.startswith("REGISTER"):

            \_, peer\_ip, peer\_port = data.split()

            peer\_port = int(peer\_port)

            if (peer\_ip, peer\_port) not in peers:

                peers.append((peer\_ip, peer\_port))

                print(f"[+] Peer registered: {peer\_ip}:{peer\_port}")

            # Send back list of peers (excluding the newly registered peer)

            known\_peers = "\n".join(f"{ip} {port}" for ip, port in peers if (ip, port) != (peer\_ip, peer\_port))

            conn.sendall(known\_peers.encode())

    except Exception as e:

        print(f"[!] Error handling peer: {e}")

    finally:

        conn.close()

def start\_rendezvous\_server():

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

        s.bind((HOST, PORT))

        s.listen()

        print(f"[\*] Rendezvous server listening on {HOST}:{PORT}")

        while True:

            conn, addr = s.accept()

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

if \_\_name\_\_ == "\_\_main\_\_":

    start\_rendezvous\_server()

# File Peer.py:

# peer.py

import os

import socket

import threading

import hashlib

import secrets

import json

from datetime import datetime, timedelta

import tkinter as tk

from tkinter import filedialog

SHARED\_DIR = "shared\_files"

RECEIVED\_DIR = "received\_files"

USER\_DATA\_FILE = "user\_data/users.json"

SESSION\_TIMEOUT = 3  # minutes

RENDEZVOUS\_PORT = 5000

os.makedirs(SHARED\_DIR, exist\_ok=True)

os.makedirs(RECEIVED\_DIR, exist\_ok=True)

os.makedirs(os.path.dirname(USER\_DATA\_FILE), exist\_ok=True)

active\_sessions = {}

# --- User Authentication Helpers ---

def load\_user\_data():

    if not os.path.exists(USER\_DATA\_FILE):

        with open(USER\_DATA\_FILE, 'w') as f:

            json.dump({}, f)

    with open(USER\_DATA\_FILE, 'r') as f:

        return json.load(f)

def save\_user\_data(users):

    with open(USER\_DATA\_FILE, 'w') as f:

        json.dump(users, f)

def hash\_password(password, salt=None):

    if salt is None:

        salt = secrets.token\_hex(16)

    hashed = hashlib.sha256((password + salt).encode()).hexdigest()

    return hashed, salt

def verify\_password(stored\_hash, salt, input\_password):

    input\_hash = hashlib.sha256((input\_password + salt).encode()).hexdigest()

    return input\_hash == stored\_hash

# --- Peer Server: Handle incoming requests ---

def handle\_incoming\_peer(conn, addr):

    try:

        data = conn.recv(1024).decode().strip()

        if not data:

            return

        authenticated = False

        if ' ' in data and len(data.split()[0]) == 32:

            session\_token, command = data.split(maxsplit=1)

            if session\_token in active\_sessions and datetime.now() < active\_sessions[session\_token]["expiry"]:

                authenticated = True

                # Renew session

                active\_sessions[session\_token]["expiry"] = datetime.now() + timedelta(minutes=SESSION\_TIMEOUT)

        else:

            command = data

        users = load\_user\_data()

        if command.startswith("REGISTER"):

            \_, username, password = command.split(maxsplit=2)

            if username in users:

                conn.sendall(b"ERROR: Username already exists")

            else:

                password\_hash, salt = hash\_password(password)

                users[username] = {"password\_hash": password\_hash, "salt": salt}

                save\_user\_data(users)

                conn.sendall(b"OK: Registration successful")

        elif command.startswith("LOGIN"):

            \_, username, password = command.split(maxsplit=2)

            if username not in users:

                conn.sendall(b"ERROR: User not found")

            else:

                user\_data = users[username]

                if verify\_password(user\_data["password\_hash"], user\_data["salt"], password):

                    token = secrets.token\_hex(16)

                    active\_sessions[token] = {"username": username, "expiry": datetime.now() + timedelta(minutes=SESSION\_TIMEOUT)}

                    conn.sendall(f"OK: {token}".encode())

                else:

                    conn.sendall(b"ERROR: Invalid password")

        elif command == "LIST\_FILES":

            if not authenticated:

                conn.sendall(b"ERROR: Authentication required")

            else:

                files = os.listdir(SHARED\_DIR)

                conn.sendall("\n".join(files).encode())

        elif command.startswith("DOWNLOAD"):

            if not authenticated:

                conn.sendall(b"ERROR: Authentication required")

            else:

                requested\_file = command.split(maxsplit=1)[1]

                file\_path = os.path.join(SHARED\_DIR, requested\_file)

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

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

                        while True:

                            chunk = f.read(4096)

                            if not chunk:

                                break

                            conn.sendall(chunk)

                    conn.sendall(b"<EOF>")

                else:

                    conn.sendall(b"ERROR: File not found")

        elif command.startswith("UPLOAD"):

            if not authenticated:

                conn.sendall(b"ERROR: Authentication required")

            else:

                filename = command.split(maxsplit=1)[1]

                if not filename.isprintable() or '/' in filename:

                    conn.sendall(b"ERROR: Invalid filename")

                    return

                file\_path = os.path.join(SHARED\_DIR, filename)

                conn.sendall(b"READY")

                with open(file\_path, "wb") as f:

                    while True:

                        data = conn.recv(4096)

                        if data.endswith(b"<EOF>"):

                            f.write(data[:-5])

                            break

                        f.write(data)

                conn.sendall(b"OK: File uploaded successfully")

        else:

            conn.sendall(b"ERROR: Unknown command")

    except Exception as e:

        print(f"[!] Error: {e}")

    finally:

        conn.close()

def start\_peer\_server(port):

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

        s.bind(('0.0.0.0', port))

        s.listen()

        print(f"[\*] Peer server listening on port {port}...")

        while True:

            conn, addr = s.accept()

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

# --- Peer Client Menu ---

def peer\_client\_menu(peers):

    session\_token = None

    peer\_sessions = {}

    while True:

        print("\n=== Peer Client Menu ===")

        print("1. Connect to a peer and Login/Register")

        print("2. List available files")

        print("3. Download file")

        print("4. Upload file")

        print("5. Exit")

        print("6. Refresh peer list")

        choice = input("Select an option (1-5): ")

        if choice == "1":

            peer\_ip, peer\_port = select\_peer(peers)

            session\_token = authenticate\_with\_peer(peer\_ip, peer\_port)

            peer\_sessions[(peer\_ip, peer\_port)] = session\_token

        elif choice == "2":

            peer\_ip, peer\_port = select\_peer(peers)

            if peer\_ip is None:

                continue

            session\_token = peer\_sessions.get((peer\_ip, peer\_port))

            if not session\_token:

                print("[!] You must login to this peer first. Use option 1.")

                continue

            list\_files(peer\_ip, peer\_port, session\_token)

        elif choice == "3":

            peer\_ip, peer\_port = select\_peer(peers)

            if peer\_ip is None:

                continue

            session\_token = peer\_sessions.get((peer\_ip, peer\_port))

            if not session\_token:

                print("[!] You must login to this peer first. Use option 1.")

                continue

            download\_file(peer\_ip, peer\_port, session\_token)

        elif choice == "4":

            peer\_ip, peer\_port = select\_peer(peers)

            if peer\_ip is None:

                continue

            session\_token = peer\_sessions.get((peer\_ip, peer\_port))

            if not session\_token:

                print("[!] You must login to this peer first. Use option 1.")

                continue

            upload\_file(peer\_ip, peer\_port, session\_token)

        elif choice == "5":

            print("Goodbye!")

            break

        elif choice == "6":

            peers = refresh\_peer\_list(rendezvous\_ip, my\_port)

def select\_peer(peers):

    if not peers:

        print("[!] No peers available yet. Try again later.")

        return None, None

    print("\nAvailable Peers:")

    for idx, (ip, port) in enumerate(peers):

        print(f"{idx+1}. {ip}:{port}")

    choice = int(input("Select peer number: ")) - 1

    return peers[choice]

def authenticate\_with\_peer(ip, port):

    with socket.create\_connection((ip, port)) as s:

        print("\nAuthentication Menu:")

        print("1. Login")

        print("2. Register")

        auth\_choice = input("Select option: ")

        username = input("Username: ")

        password = input("Password: ")

        if auth\_choice == "1":

            s.sendall(f"LOGIN {username} {password}".encode())

        elif auth\_choice == "2":

            s.sendall(f"REGISTER {username} {password}".encode())

        else:

            print("Invalid choice")

            return None

        response = s.recv(1024).decode()

        if response.startswith("OK:"):

            print("[+] Authentication successful!")

            if " " in response:

                return response.split(": ")[1].strip()

        else:

            print(f"[!] Authentication failed: {response}")

            return None

def list\_files(ip, port, token):

    with socket.create\_connection((ip, port)) as s:

        s.sendall(f"{token} LIST\_FILES".encode())

        files = s.recv(4096).decode()

        print("\nAvailable files:")

        print(files)

def download\_file(ip, port, token):

    with socket.create\_connection((ip, port)) as s:

        s.sendall(f"{token} LIST\_FILES".encode())

        files = s.recv(4096).decode().split("\n")

        for idx, f in enumerate(files):

            print(f"{idx+1}. {f}")

        file\_choice = int(input("Select file number: ")) - 1

        filename = files[file\_choice]

        s.sendall(f"{token} DOWNLOAD {filename}".encode())

        with open(f"{RECEIVED\_DIR}/{filename}", "wb") as f:

            while True:

                try:

                    chunk = s.recv(4096)

                    if not chunk:

                        # Connection closed

                        break

                    if b"<EOF>" in chunk:

                        f.write(chunk.replace(b"<EOF>", b""))  # Remove marker

                        break

                    f.write(chunk)

                except Exception as e:

                    print(f"[!] Download error: {e}")

                    break

        print(f"[+] File '{filename}' downloaded successfully into '{RECEIVED\_DIR}/'")

def upload\_file(ip, port, token):

    root = tk.Tk()

    root.withdraw()

    root.attributes('-topmost', True)

    filepath = filedialog.askopenfilename(title="Select file to upload")

    root.destroy()

    if not filepath:

        print("Upload cancelled")

        return

    filename = os.path.basename(filepath)

    with socket.create\_connection((ip, port)) as s:

        s.sendall(f"{token} UPLOAD {filename}".encode())

        response = s.recv(1024)

        if response != b"READY":

            print(f"Server error: {response.decode()}")

            return

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

            while True:

                chunk = f.read(4096)

                if not chunk:

                    break

                s.sendall(chunk)

        s.sendall(b"<EOF>")

        final\_response = s.recv(1024).decode()

        print(final\_response)

def refresh\_peer\_list(rendezvous\_ip, my\_port):

    new\_peers = []

    try:

        with socket.create\_connection((rendezvous\_ip, 5000)) as s:

            my\_ip = socket.gethostbyname(socket.gethostname())

            s.sendall(f"REGISTER {my\_ip} {my\_port}".encode())

            data = s.recv(4096).decode()

            if data.strip():

                for line in data.split("\n"):

                    ip, port = line.split()

                    new\_peers.append((ip, int(port)))

    except Exception as e:

        print(f"[!] Could not refresh peers: {e}")

    print("[+] Peers refreshed:", new\_peers)

    return new\_peers

# --- Main ---

if \_\_name\_\_ == "\_\_main\_\_":

    my\_port = int(input("Enter your peer server port: "))

    rendezvous\_ip = input("Enter Rendezvous Server IP: ")

    threading.Thread(target=start\_peer\_server, args=(my\_port,), daemon=True).start()

    my\_ip = socket.gethostbyname(socket.gethostname())

    with socket.create\_connection((rendezvous\_ip, RENDEZVOUS\_PORT)) as s:

        s.sendall(f"REGISTER {my\_ip} {my\_port}".encode())

        data = s.recv(4096).decode()

    peers = []

    if data.strip():

        for line in data.split("\n"):

            ip, port = line.split()

            peers.append((ip, int(port)))

    print("[+] Peers discovered:", peers)

    peer\_client\_menu(peers)