

Table of contents

Contents

| | |
|--|----|
| Introduction | 5 |
| The research process | 5 |
| The Ingenuity of my project | 6 |
| Project capabilities..... | 7 |
| 1. User Authentication and Login: | 7 |
| 2. Peer-to-Peer Communication:..... | 7 |
| 3. File Management and Security:..... | 7 |
| 4. Dynamic IP Discovery: | 8 |
| 5. Secure Socket Communication: | 8 |
| 6. Thread Management:..... | 8 |
| 7. Error Handling:..... | 9 |
| Project Structure | 10 |
| Technologies Used..... | 12 |
| Programming language..... | 12 |
| Libraries and Frameworks..... | 12 |
| Communication | 12 |
| Areas of Interest..... | 12 |
| Central Algorithms | 14 |
| Formulation and Analysis of the Algorithmic Problem | 14 |
| Description of Existing Algorithms for Solving the Problem | 14 |
| Review of the Chosen Solution | 14 |
| Original Developments..... | 15 |
| Description of the Development Environment | 17 |
| Development Tools | 17 |
| Testing Tools | 17 |
| Abilities and responsibilities of each code..... | 18 |
| Login code: | 18 |
| Main code: | 18 |
| Client code:..... | 19 |
| Find_ip code: | 19 |

| | |
|---|----|
| Server code: | 19 |
| Dev_txt code:..... | 21 |
| Random_path code:..... | 21 |
| Recreate file code:..... | 21 |
| Description of the Communication Protocol | 22 |
| Verbal Description of the Communication Protocol | 22 |
| Details of All the Messages Flowing in the System | 22 |
| System screens..... | 24 |
| Screen Flow Diagram | 24 |
| Login UI | 25 |
| Main..... | 29 |
| send file | 31 |
| request file | 34 |
| illustration..... | 37 |
| Description of the Data Structures | 40 |
| Login database (.localappdata.db): | 40 |
| Paths Database (paths.db): | 40 |
| Sent Peers Database (sent_peers.db): | 40 |
| Stored files..... | 41 |
| Other data structures used..... | 41 |
| Overview of Weaknesses and Threats | 42 |
| Implementation of the Project..... | 43 |
| Part I - Overview of Modules/Departments | 43 |
| 1. Main Module | 43 |
| 2. Client Module..... | 44 |
| 3. Server Module..... | 45 |
| 4. Encryption Module | 45 |
| 5. dev_txt Module..... | 45 |
| 6. random_path Module | 46 |
| 7. recreate_file Module..... | 46 |
| 8. Common Module..... | 46 |
| Imported Modules/Classes:..... | 47 |
| Part II – unique code | 49 |

| | |
|--|----|
| Random file division and storage | 49 |
| Part III - Testing Documentation | 54 |
| Sending Very Long Text Files to Peers | 54 |
| Sending Text Files Containing All Special Characters | 54 |
| Handling Interrupted Connections | 54 |
| Stress Testing with Multiple Simultaneous Transfers | 55 |
| Error management | 55 |
| Use and directions | 58 |
| Reflection | 59 |
| Description of the Work Process | 59 |
| Learning Process | 59 |
| Tools for the Future | 59 |
| Insights from the Process | 60 |
| Hindsight and Improvements | 60 |
| Self-Study Questions for Students | 60 |
| Acknowledgments | 60 |
| Project Code | 61 |
| Login.py | 61 |
| Main.py | 67 |
| client.py | 74 |
| server.py | 76 |
| Common | 79 |
| Dev_txt | 82 |
| Recreate_file | 85 |
| Random_path | 86 |
| Bibliography: | 87 |

Introduction

My project is a secure peer-to-peer (P2P) file sharing system designed to hide sensitive files while ensuring their safety and integrity. Another use for this code is to act as a backup option for important files, as the file is split and saved in random paths which significantly lowers the chance of an accidental complete loss of the file. Each peer in the network acts as both a client and a server, capable of sending and receiving files from other peers. The system uses a combination of encryption for secure communication and a unique file splitting mechanism that divides files into random chunks, stored across various paths on users' systems. These chunks can be reassembled upon request. The code employs multi-threading to allow concurrent operations and uses a shared 'common' module for global variables and inter-thread communication. The overall design focuses on providing a secure and efficient method for distributing and retrieving sensitive files across multiple peers in the network.

This project is currently suited for home use as a backup tool or as a way to hide sensitive information. This project has greater potential to be used as a decentralized hyper secure data storage system by implementing adjustments which allow retrieving any of the files entered by any of the peers connected.

I chose this project after accidentally deleting a python code file. I was devastated but it made me come up with an idea to store highly important data in a system that distributes the data across multiple computers, so that the chance of the info being lost completely will be very small. I later decided a system like this will also be helpful to save sensitive files.

I expected to meet difficulty when it comes to handling the peer communication and message protocol, since I need to communicate large amounts of data between a very complex system.

The research process

Reviewing the current market

My system shares similarities with some peer2peer systems such as IPFS, BitTorrent and RAID configurations.

- Both my system and IPFS are designed for secure and decentralized file sharing, but they are made to be used in different cases and environments. IPFS is more suitable for global data distribution, while my system is ment for personal use and

focuses on secure and vigilant storage of sensitive files within a smaller network, emphasizing encryption and secrecy.

- My system and BitTorrent share the fundamental P2P structure and chunk-based file sharing mechanism. However, my system is specifically designed for secure handling of sensitive files, including encryption for both storage and transmission, and random storage paths to obscure data location. BitTorrent focuses on fast paced distribution of larger files with more basic encryption features, relying on community and metadata for peer discovery and file integrity checks.
- My peer-to-peer (P2P) system offers several advantages over traditional RAID methods for securing sensitive files. It enhances security through encryption and random chunking, making unauthorized access and data reconstruction by hackers difficult. The decentralized nature of the P2P network increases resilience, avoiding single points of failure such as a main server and allowing dynamic scalability as new peers join. Additionally, it is cost-effective and highly available to average users, requiring no specialized hardware, and allowing easy maintenance and expansion.

The Ingenuity of my project

My offers a new approach to data security and file backing. This project brings together several advanced concepts in peer-to-peer networking, encryption, and file management to create a secure, efficient, and resilient system for sharing sensitive files. The combination of these features, especially the focus on randomized chunk distribution, secure communication, and automated peer discovery, sets my project apart from many existing P2P systems.

Project capabilities

1. User Authentication and Login:

- In essence:

Checks user credentials and allows access to the system securely.

- Actions:

Verify username and password against stored credentials.

Encrypt and securely store user passwords.

- Necessary capabilities:

- User interface
- Registration of user input
- Integrity checks for user input
- Encryption of data
- Database storage

- Objects:

UI, Database, hash encryptions

2. Peer-to-Peer Communication:

- In essence:

Enables direct communication between peers over a network without relying on a central server.

- Actions:

Create server and client threads for each peer.

Handle incoming requests and initiate communication with other peers.

- Necessary capabilities:

- Socket communication
- Thread use
- Communication protocol
- Encryption of data

- Objects:

Threads, Database, Encryption class, socket

3. File Management and Security:

- In essence:

Manages file splitting, distribution, and reconstruction for enhanced security and maintaining data integrity.

- Actions:
 - Split files into random chunks for dispersion.
 - Distribute chunks across the network.
 - Reconstruct files from distributed chunks upon request.
- Necessary capabilities:
 - Database storage
 - Data encryption
- Objects:
 - Database, hash encryptions, OS

4. Dynamic IP Discovery:

- In essence:
 - discovers available peers on the network by scanning IP addresses with open ports.
- Actions:
 - Scan IP addresses with open ports.
 - Establish connections with available peers.
- Necessary capabilities:
 - Threads
 - Socket communication
- Objects:
 - Threads, socket

5. Secure Socket Communication:

- In essence:
 - Ensures secure communication between peers by encrypting data transmission over sockets.
- Actions:
 - Encrypt transmitted data.
 - Decrypt received data.
- Necessary capabilities:
 - Socket communication
 - Data encryption
- Objects:
 - Socket, Encryption class

6. Thread Management:

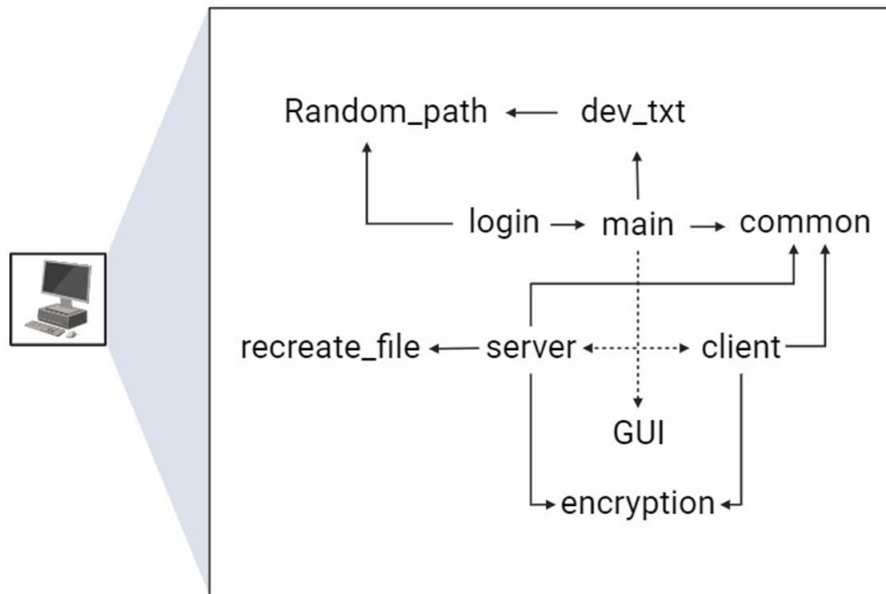
- In essence:
 - Coordinates the execution of multiple threads to handle concurrent tasks and optimize system performance.

- Actions:
 - Create and manage threads for various functionalities such as server-client communication and peer seeking.
- Necessary capabilities:
 - Threading module for creating and managing threads
- Objects:
 - Threads

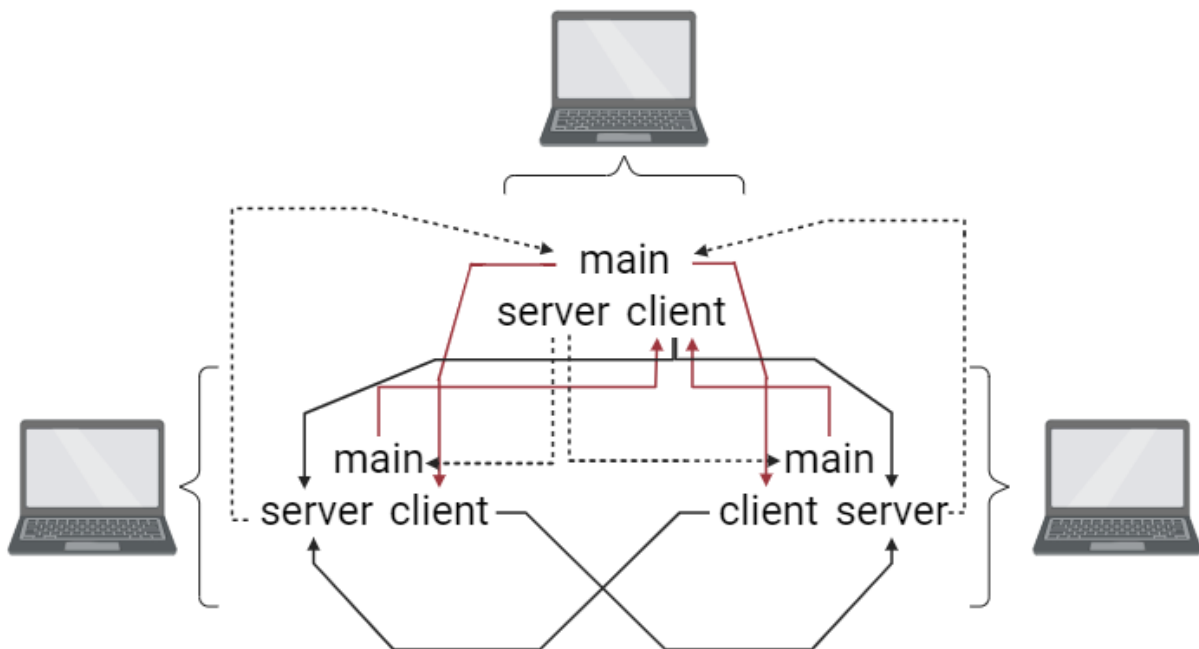
7. Error Handling:

- In essence:
 - Identifies and handles errors to maintain system stability and integrity.
- Actions:
 - Implement error handling mechanisms to catch and manage exceptions.
- Necessary capabilities:
 - Try and Exception in code
- Objects:
 - Exception handling mechanisms throughout the codebase.

Project Structure



- dotted line indicates connection via thread



Technologies Used

Programming language

- Python: Python is just the primary programming language I have adopted for this project. The language is simple and readable, and its standard library is vast. Again, due to the existence of so many different libraries and frameworks in Python, it is possible to do rapid development and prototyping.

Libraries and Frameworks

- SQLite: A lightweight, disk-based database library SQLite is used for the management of user login information, storage of file paths, and recording of the peer connections, all without any need for a separate process of the server.
- Pygame: A collection of modules for building graphical user interfaces. They are used in the system to handle the login interface with the user and other interactions.
- Hashlib: Provides a wide range of "secure hash and message digest algorithms" used to fingerprint user passwords.
- Socket: Allows network communication using low-level networking interfaces to establish client-server connections.
- Threading: Provides support for running multiple operations concurrently, thus managing many different client connections and background operations in general.
- Random: It generates random numbers and selects random elements and is used during the creation of random chunk sizes and random file storage paths.
- Re: it provides regular expression matching operations, which help a lot in string manipulations, like IP address parsing.

Communication

- Peer-to-Peer (P2P) Communication: It uses a decentralized model in which every peer acts as a client and server, hence has increased security and scalability by avoiding the central point of failure. Communication is done using the encrypted sockets and channels.
- AES Encryption: Ensures data integrity during transmission by using the Advanced Encryption Standard (AES). AES is a symmetric encryption algorithm widely recognized for its strength and efficiency. The project uses AES to encrypt and decrypt data exchanged between peers, protecting it from unauthorized access and tampering.

Areas of Interest

- Data Security: It helps protect sensitive files by encrypting them and storing them in a secure place through robust authentication mechanisms.
- Distributed Systems: Deals with efficient data management and distribution on many nodes in a P2P network.
- Concurrency: It makes use of threading to allow many operations to be performed simultaneously, which gives the system improved performance and responsiveness.

- Graphical User Interface (GUI): Enhances user experience with intuitive interfaces for activities like login and file management developed using Pygame.
- File Management: It applies advanced techniques to partition files, secure the storage of files, and reassemble data that guarantees data integrity and security.

The project will focus on integrating these technologies to achieve a high, reliable, and efficient system that ensures the handling and transferring of sensitive files within a distributed network setting.

Central Algorithms

Formulation and Analysis of the Algorithmic Problem

The primary algorithmic problem addressed in this project is the secure and efficient storage, retrieval, and transmission of sensitive files in a peer-to-peer (P2P) network. This involves splitting files into chunks, distributing these chunks across multiple peers, encrypting the data to ensure confidentiality, and reliably reconstructing the original file upon request.

Description of Existing Algorithms for Solving the Problem

Several existing algorithms and methods are relevant to this problem:

1. File Chunking:
 - Fixed-Size Chunking: This method splits a file into equal-sized chunks. It is simple but can lead to inefficiencies if the file size is not a multiple of the chunk size.
 - Content-Defined Chunking (CDC): This technique splits files based on the content, ensuring that similar files have similar chunks. It is more complex but handles variable data sizes better.
2. Peer-to-Peer Communication:
 - Centralized Directory: A central server keeps track of all file locations and peers. This is simple but creates a single point of failure.
 - Distributed Hash Table (DHT): Each peer maintains a portion of the hash table, allowing decentralized management of file locations. This increases resilience but adds complexity.
3. Encryption:
 - AES (Advanced Encryption Standard): A widely used symmetric encryption algorithm that ensures data confidentiality. It is efficient and secure.
 - RSA (Rivest-Shamir-Adleman): An asymmetric encryption algorithm used for secure key exchange. It is secure but computationally intensive.
4. Data Integrity:
 - MD5/SHA-1 Hashing: These algorithms generate a hash value for data integrity verification. They are fast but less secure compared to newer standards.
 - SHA-256: A member of the SHA-2 family, offering better security for hash functions.

Review of the Chosen Solution

Justifying the Choice and Rejecting the Alternative Solutions

1. File Chunking:
 - Chosen Solution: Random chunking with variable sizes
 - Justification: Random chunking increases security by making it harder to predict and reassemble the chunks without knowledge of the specific randomization used. Variable chunk sizes further obfuscate the file structure, enhancing security over fixed-size chunking.
 - Rejected Solution: Fixed-size chunking was rejected due to its predictability and inefficiency with files that do not evenly divide into chunks.
2. Peer-to-Peer Communication:
 - Chosen Solution: Peer-to-peer with a simple tracking mechanism using SQLite
 - Justification: Using a lightweight, embedded database like SQLite to track file chunks and peer locations provides a balance between simplicity and functionality. It avoids the complexity and overhead of a fully distributed hash table while eliminating the single point of failure inherent in a centralized directory.
 - Rejected Solution: A fully decentralized DHT was deemed too complex for the scope of this project, and a centralized directory was avoided due to its vulnerability to single-point failures.
3. Encryption:
 - Chosen Solution: AES for data encryption
 - Justification: AES is efficient, widely adopted, and provides a high level of security for symmetric encryption. Its performance is suitable for encrypting file chunks before transmission, ensuring data confidentiality.
 - Rejected Solution: RSA was rejected for data encryption due to its computational intensity, making it impractical for encrypting large amounts of data.
4. Data Integrity:
 - Chosen Solution: MD5 for hashing
 - Justification: MD5 is fast and sufficient for basic integrity checks within the scope of this project. While it has known vulnerabilities, its use here is limited to non-critical integrity verification.
 - Rejected Solution: SHA-256 was considered more secure but was rejected in favor of MD5 due to the latter's lower computational overhead, which is adequate for the project's needs.

Original Developments

The project incorporates original development in the integration of these algorithms to create a secure, efficient, and user-friendly P2P file storage and retrieval system. The following features highlight the originality:

- Randomized File Chunking: Files are split into randomly sized chunks, enhancing security and making reassembly by unauthorized parties more difficult.

- SQLite-Based Peer Tracking: A very effective method for tracking file chunks and their peer locations, avoiding the complexity of more advanced systems while maintaining quality.

By combining these algorithms and techniques, the project offers a secure, efficient solution for managing sensitive files in a distributed environment.

Description of the Development Environment

Development Tools

1. Integrated Development Environment (IDE)

- PyCharm: The primary IDE used for this project. PyCharm provides powerful coding assistance, debugging, and testing tools, making it an excellent choice for Python development.

Features: Intelligent code completion, on-the-fly error checking, quick-fixes, integrated debugging, and testing tools.

2. Version Control System

- Git: A distributed version control system used to track changes in the source code during software development.

Features: Branching, merging, and repository management.

- GitHub: A web-based platform for version control and collaborative development.

Features: Code hosting, issue tracking, pull requests, and project management tools.

Testing Tools

All testing was conducted using PyCharm's integrated testing tools. These tools provide a comprehensive environment for writing, executing tests and debugging.

Abilities and responsibilities of each code

Login code:

- Initiates Login: Triggers the login process to authenticate users.
- calls Main Code: Upon successful login, initiates the execution of the main project code.
- Database Storage: Stores the login database securely, with its path saved in "paths.db".
- Random Storage: provides random storage of the login database for enhanced security.
- User Interface: uses Pygame for creating a graphical user interface for login.

Allows users to login with pygame ui. Login method then calls "main" which is the main code for this project. The login database is randomly stored. The path to the login database is saved as ".localappdata.db" in another database called "paths.db".

Main code:

- Server and Client Initialization: Launches server and client classes as separate threads.
- Starts User Interface: opens the user interface to pass user input and display relevant information.
- Peer-to-Peer Communication: the main code acts as the central component for peer communication, handling sending requests and responses from peers' servers.
- Peer Attributes: Each instance of the main code represents a peer, comprising both server and client attributes.
- Communication Flow: conducts communications - the main code communicates with the client, the client communicates with a peer server, and the peer server communicates back to the main code.

This code wakes the server and client classes via threads. The main code calls the ui and processes the user input from the ui. The "main" code also handles the response from peer servers. basically the "main" code is the "peer" code, it has two main attributes which are its server and client. Each peer has both. the system works like this - Main communicates to Client, Client communicates to a peer Server, the peer Server communicates to back again to our Main.

Client code:

- Peer Discovery: Utilizes the `find_ip` class to discover available peers by scanning IP addresses with the port 1000 open.
- Thread Management: Implements two threads, one dedicated to continuously searching for peers and another waiting for messages to send from the "Main" function.
- Connection Establishment: Attempts to connect to discovered peers and adds them to the list of connected servers.
- Message Sending: Monitors for messages from the "Main" function and sends them to connected peers.

This code uses a class called `find_ip` which returns a list of ip addresses with the port 1000 currently open. The code uses two threads, one to continually look for available peers and one to wait for messages to send from the "Main" function.

Find_ip code:

- IP Address Scanning: Retrieves the first three parts of the user's IP address, which are common among devices on the same network.
- Port Connectivity: Attempts to connect to port 1000 on each IP address within the same network range.
- Thread-Based Efficiency: Utilizes 255 threads, each responsible for checking one IP address, to expedite the scanning process and enhance efficiency.

This is the `find_ip` class code. It works by getting the first three values of the user's ip address, which will also be the first three parts of the ip address of anyone on the same network. then the code tries to connect to the port 1000 in every ip address that starts with our address and ends in any number between 1 and 255. to do this the code opens 255 threads, each checking for one ip address. This makes the process very efficient.

Server code:

- Handling Requests:

- Processes two types of requests from peer clients: entering a file into the system and
- requesting a file from the system.
- Enter File into System:
 - Peer client sends a message with the format: "Enter file";{file path};{chunk number};{chunk information}.
 - Splits the file information into random pieces and stores them in random paths within a specified directory.
 - Records file paths and name in a database named paths.db.
- Request File from System:
 - Peer client sends a message requesting a file with the format: "Request file";{name}.
 - Searches paths.db for the requested file name.
 - Retrieves paths associated with the file name from the database.
 - Utilizes a class named recreate_file to assemble the file information from each path correctly.
- Database Management:
 - Uses paths.db to store both login database location and file paths.
- Communication with Peer Clients:
 - Processes messages from peer clients and sends back appropriate responses.
 - Communication involves encryption and decryption for secure data transfer.

The server code communicates with peer client classes and processes their requests. there are two request available to the user:

1. Enter file into the system - the peer client will send a message built like this : "Enter file";{file path};{the number of this chunk};{the information of the chunk} it's necessary to number the chunks we send so we may piece the information together when requested again. When a file is entered the server uses classes to cut the information into random pieces and store them in random paths in a given directory. The paths along with the name of the file are remembered in a database called paths.db, the same database we use to remember the location of the login database.
2. Request file from the system - the peer client will send a message asking for its file back, the message is built like this : "Request file";{name} . the server will open paths.db and search for the name given in the message. it will retrieve the paths which will be in a single string organized like this: {path};{path};{path}.... the code will use a class named recreate_file which will retrieve the information from each path correctly and send it back to the peer, this message will be processed by the peers 'main' code.

Dev_txt code:

- Data Chunking: Splits data into random chunks.
- Random Data Storage: Allows saving the data randomly throughout the computer.
- File-Based Chunking: Provides the capability to split data into chunks and store them as separate files.
- Flexible Chunk Size: Dynamically determines the chunk size based on the total length of the data.

the code splits data into random chunks, and allows saving the data randomly throughout the computer

Random_path code:

- Directory Traversal: Descends into subdirectories randomly to generate paths within the existing directory structure.

this code provides a random path in a given directory

Recreate file code:

- File Reconstruction: Assembles data chunks into a single output file.
- Sorting Files: Sorts the file list based on the numerical prefix.
- Data Retrieval: Retrieves information from each file path correctly to reconstruct the original file.

This code allows piecing back together a file from a list of file paths. the name of the files have to be numbered in the order of the information. For example, to piece back together a txt file containing the word "hi" the file will need the paths of the files containing the pieces of info.

- For example, two paths were generated for the file "1.user_data.txt", "2.config.txt". the file "1.user_data.txt" will contain the text "h" and the file "2.config.txt" will contain the text "i". this code will use the paths and provide the string "hi". (because there are only 2 letters, if 2 files were created each must contain exactly one character).

Description of the Communication Protocol

Verbal Description of the Communication Protocol

The communication protocol used in this project is designed to facilitate secure and efficient peer-to-peer (P2P) file transfer and management. The protocol ensures that messages are structured in a consistent manner, allowing peers to correctly interpret and process the data. Each message includes specific fields to identify the type of request, the associated data, and any necessary metadata.

- Structure/Number of Bytes:
 - Each message is a string that can widely vary in length depending on the type and content of the message, e.g. file names and file information. The fields within the message are separated by semicolons (;).
 - Example message structure:
Message Type;FilePath;ChunkNumber;ChunkData

Details of All the Messages Flowing in the System

1. Message: "Enter file"
 - Name of the Message: Enter file
 - Sent From/To: Sent from the client to the server
 - Structure of the Fields in the Message:
 - Message Type: Indicates the type of request ("Enter file").
 - File Path: The path of the file being uploaded.
 - Chunk Number: The specific chunk number of the file being sent.
 - Chunk Data: The data contained in this chunk of the file.
 - Example: Enter file;G:\file1\file2\file3\file4\text.txt;1;chunkdata
2. Message: "Request file"
 - Name of the Message: Request file
 - Sent From/To: Sent from the client to the server
 - Structure of the Fields in the Message:
 - Message Type: Indicates the type of request ("Request file").
 - File Name: The name of the file being requested.
 - Example: Request file;text.txt
3. Message: "found server"
 - Name of the Message: Found server
 - Sent From/To: Sent from the server to the client
 - Structure of the Fields in the Message:

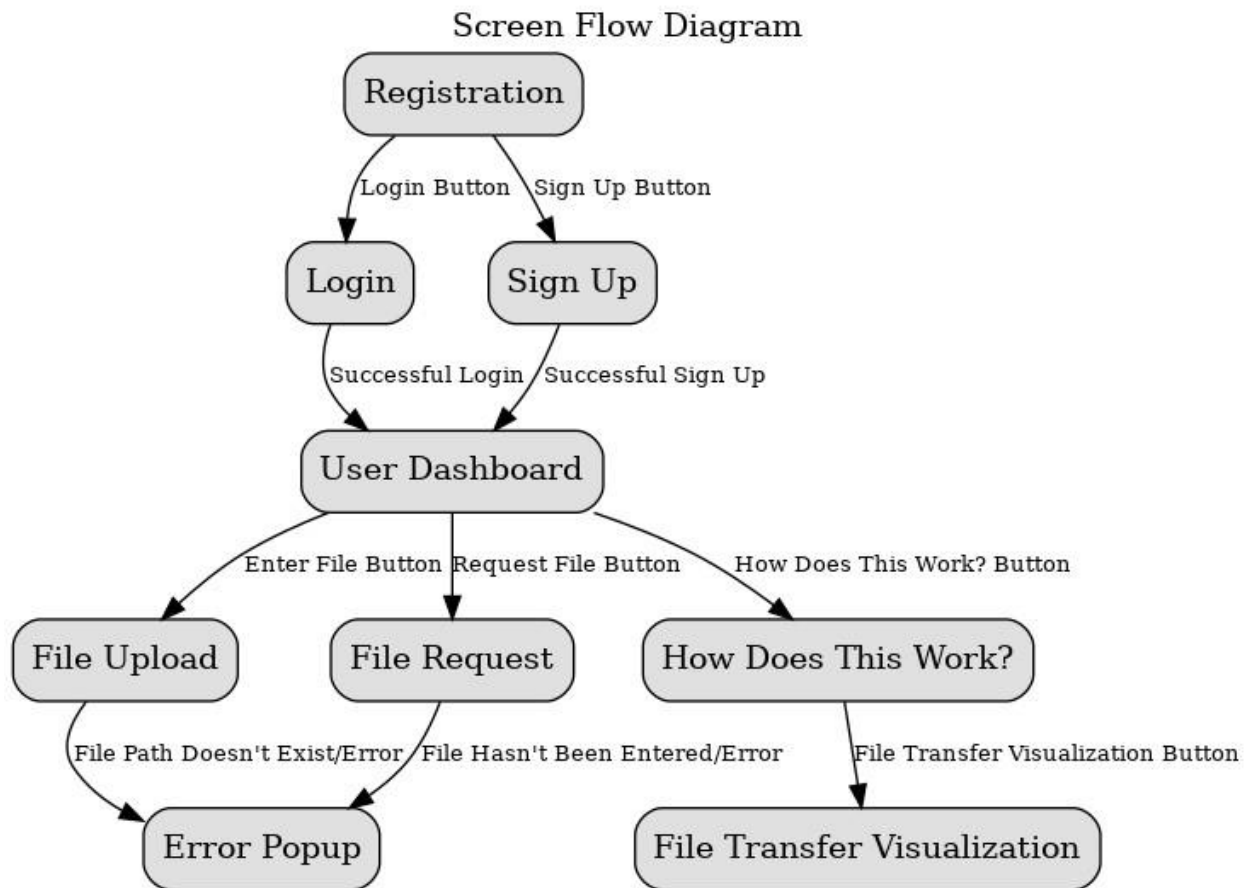
- Message Type: Indicates the type of message ("found server").
- 4. Message: "Request file response"
 - Name of the Message: Request file response
 - Sent From/To: Sent from the server to the client
 - Structure of the Fields in the Message:
 - Message Type: Indicates the type of message ("Request file response").
 - File Name: The name of the file being sent.
 - Chunk Number: The specific chunk number of the file being sent.
 - Chunk Data: The data contained in this chunk of the file.
 - Example: Request file response;text.txt;1;chunkdata

If file is not found server will respond like this:

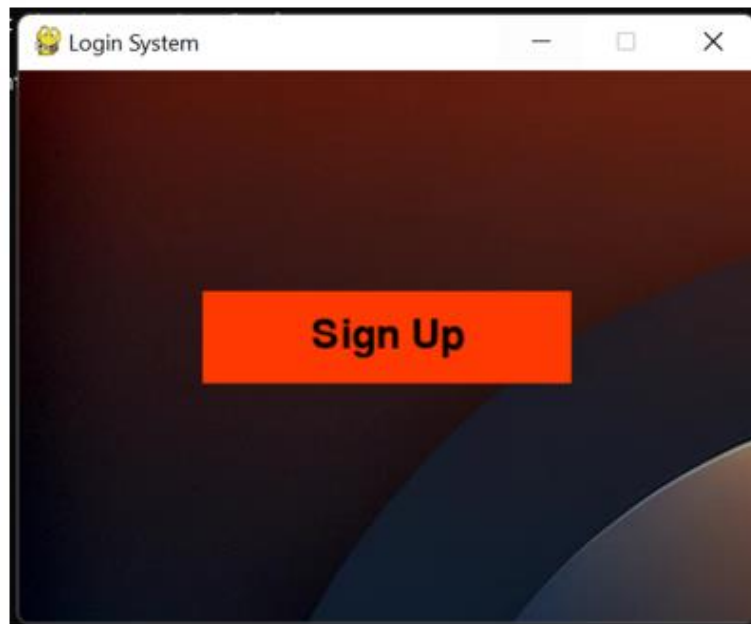
- Structure of the Fields in the Message:
 - Message Type: Indicates the type of message ("Request file response").
 - Error message: Alerts user that peer doesn't have the file information asked.
- Example: Request file response;File not found

System screens

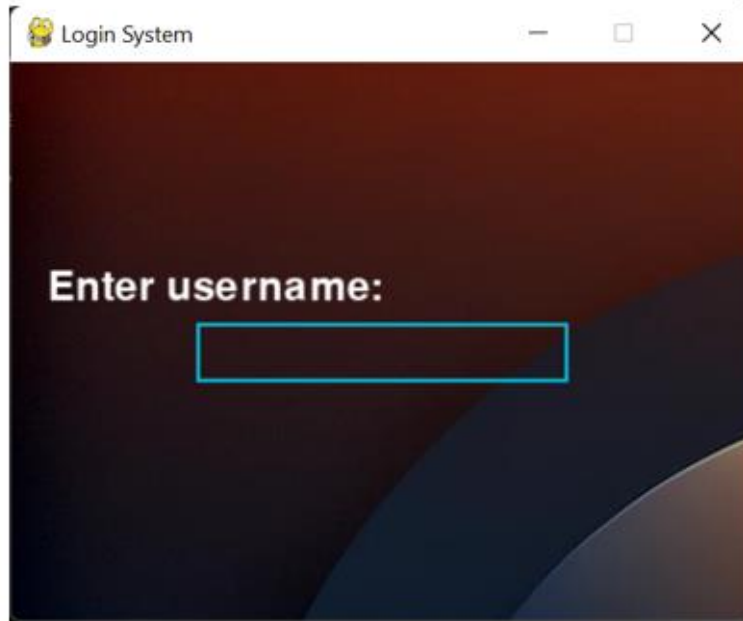
Screen Flow Diagram



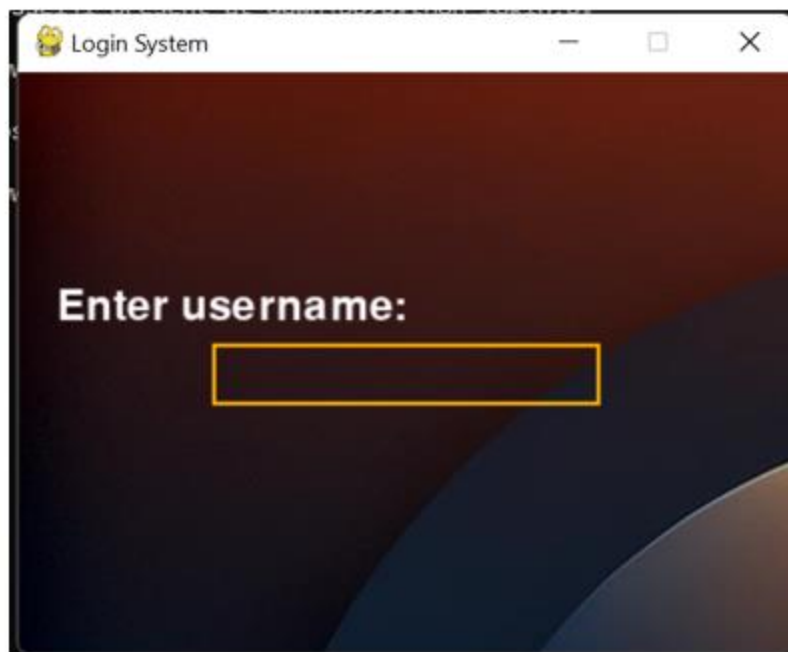
Login UI



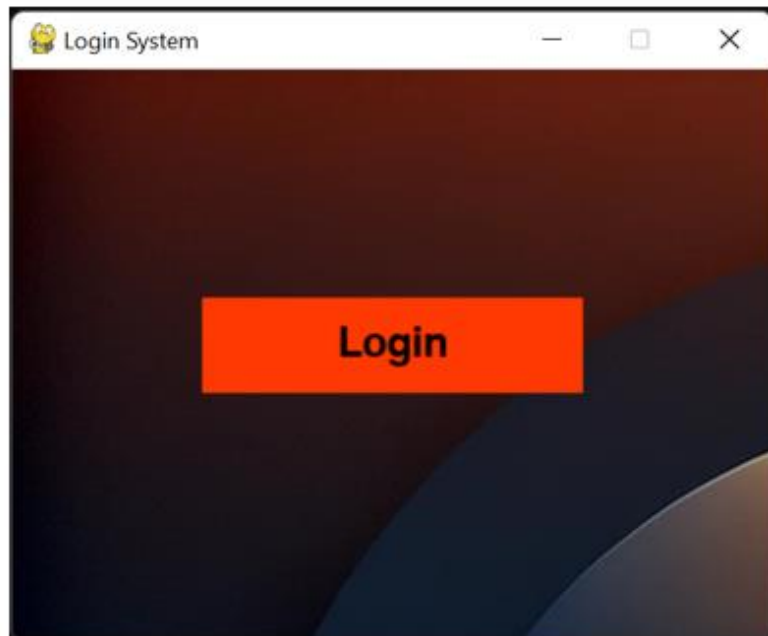
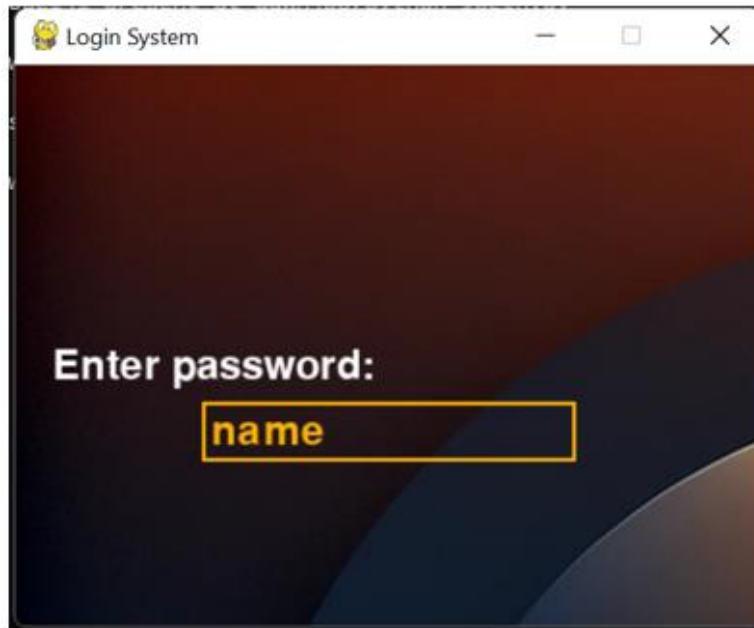
If paths.db is empty (no one has signed in yet)



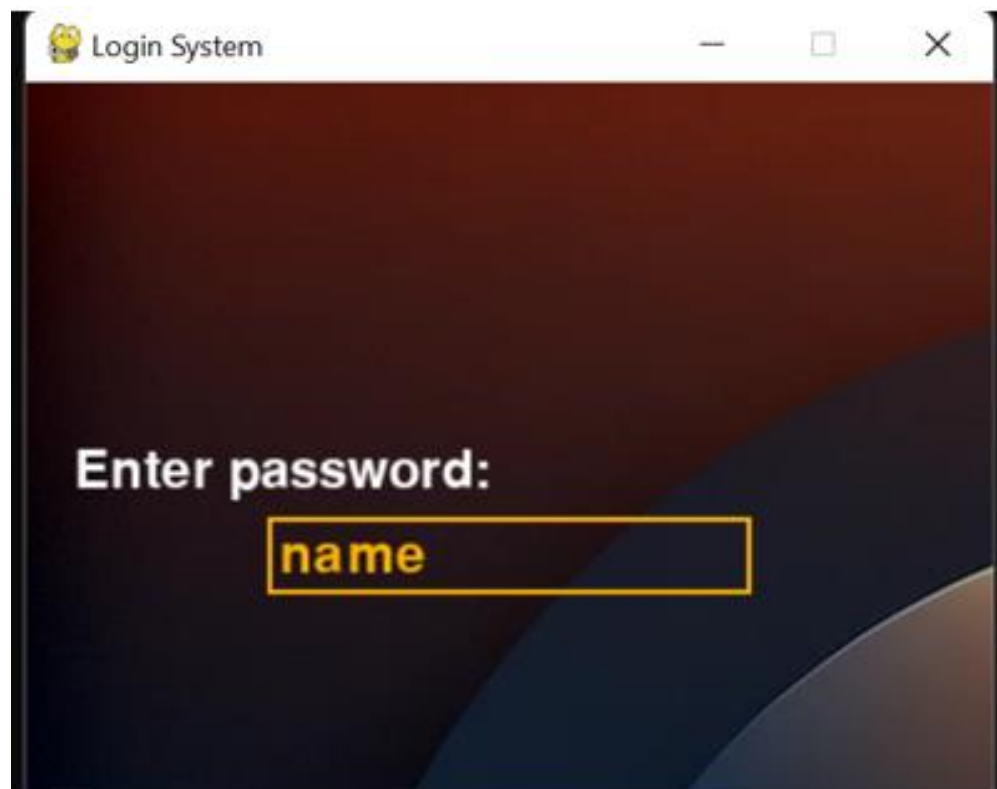
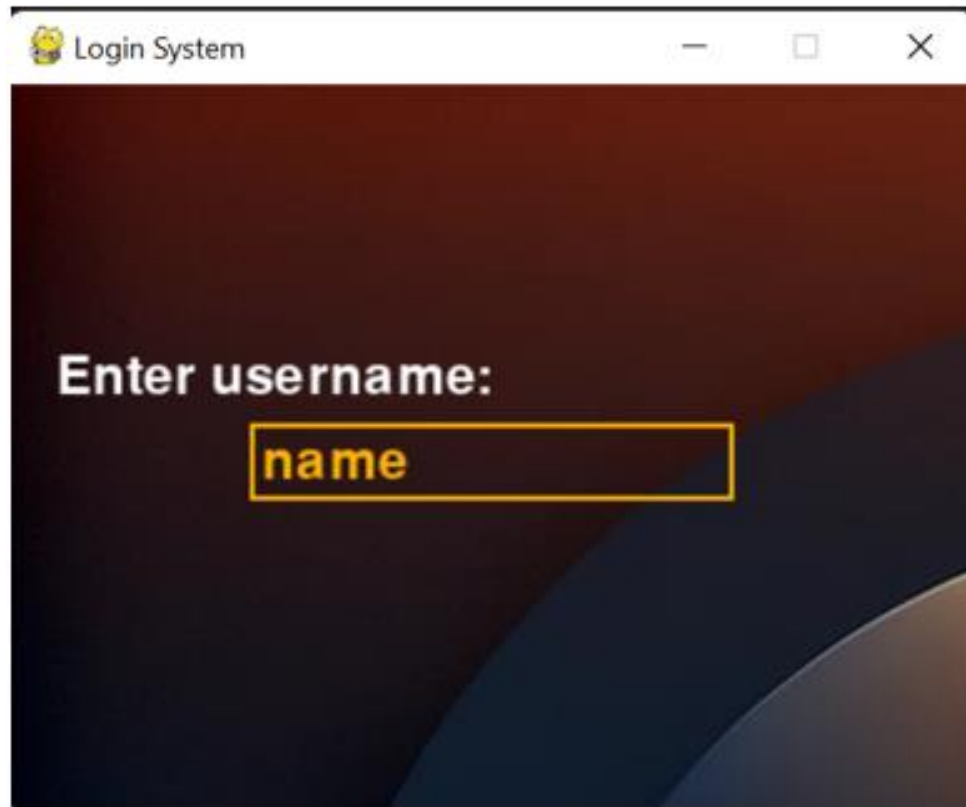
Text box to take user input. Needs to be clicked.



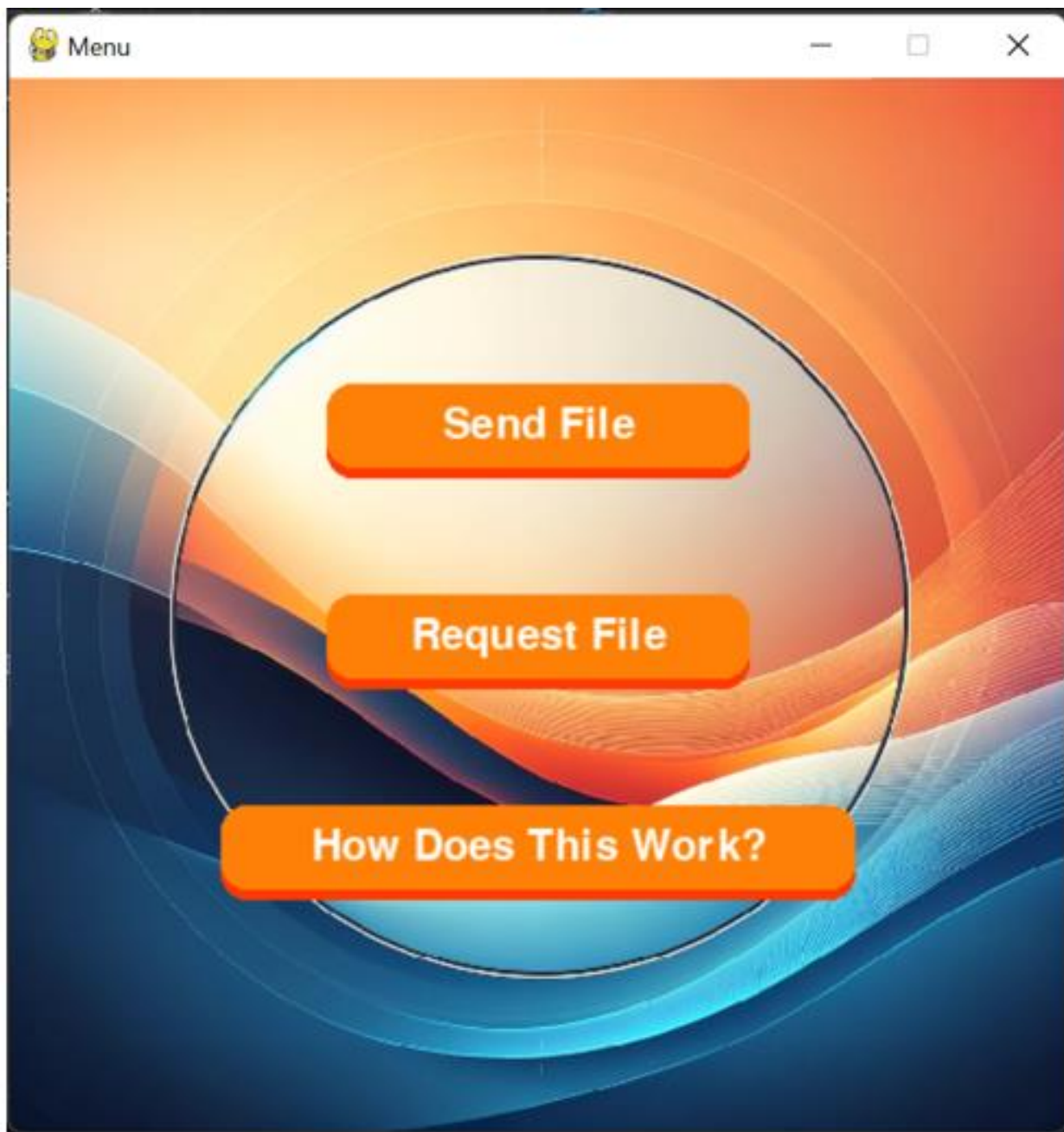
When clicked text box changes color and allows user to write.



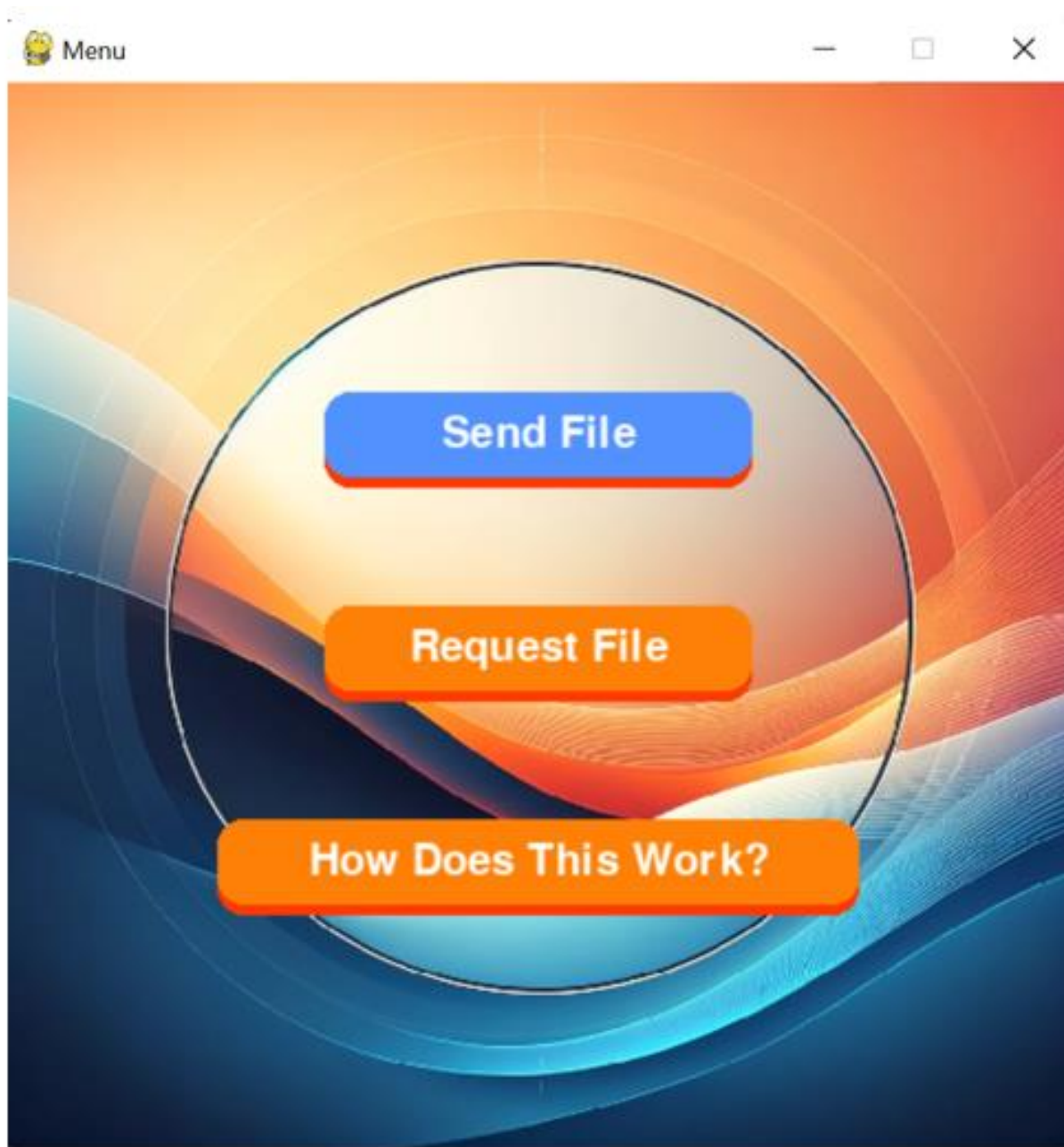
If 'paths.db' isn't empty, the system will allow the user to log in.



Main

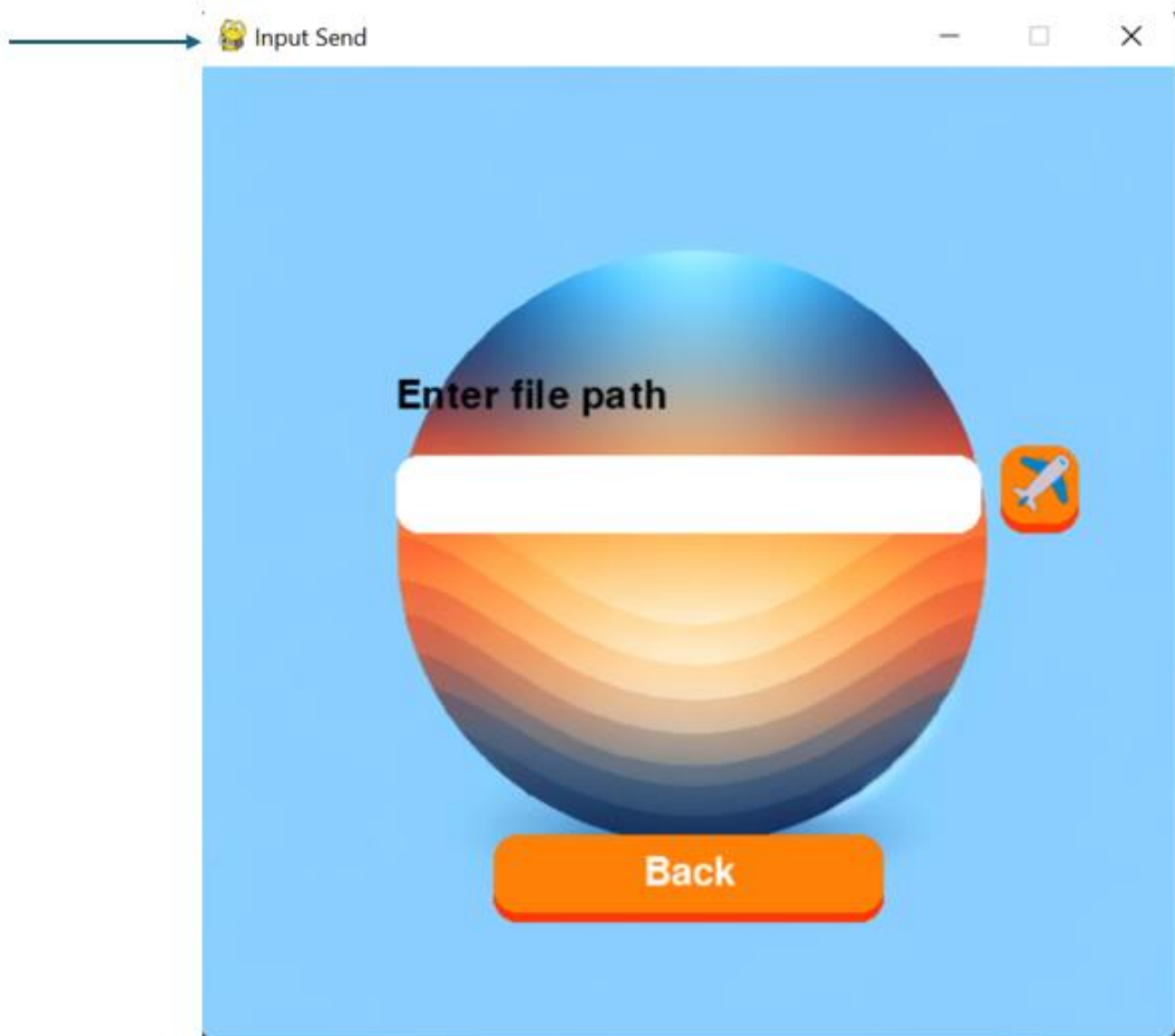


If login is completed, main window appears.



Interactive buttons will change color when mouse hovers over them. They will have a “click” animation when clicked by user.

send file



The image shows a web browser window with the title "Input Send". The main content area has a light blue background. In the center, there is a large, colorful sphere with a gradient from blue at the top to orange at the bottom. Overlaid on the sphere is the text "Enter file path" in bold black font. Below the text is a white rectangular text input field. To the right of the input field is a small orange square button with a white airplane icon. Below the sphere is a large orange rounded rectangular button with the word "Back" in white text. A green arrow points from the text "send file" to the "Input Send" window title.

Text box allows user input. To send information you must press the send (airplane emoji) button. Pressing enter will delete all info for conviniance. Page name will indicate if this is the “send” or “request” options.

Enter file path

c:hi

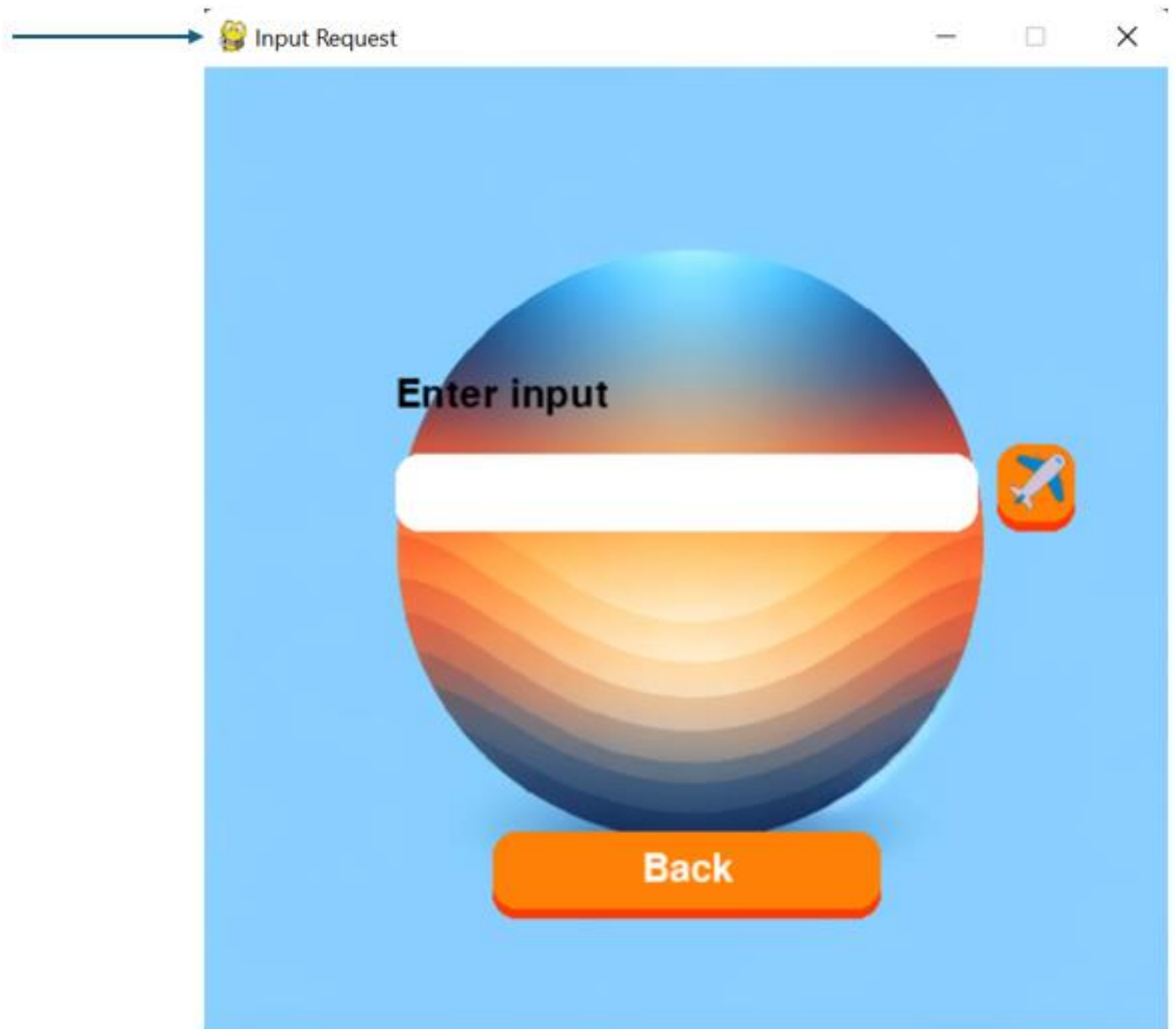


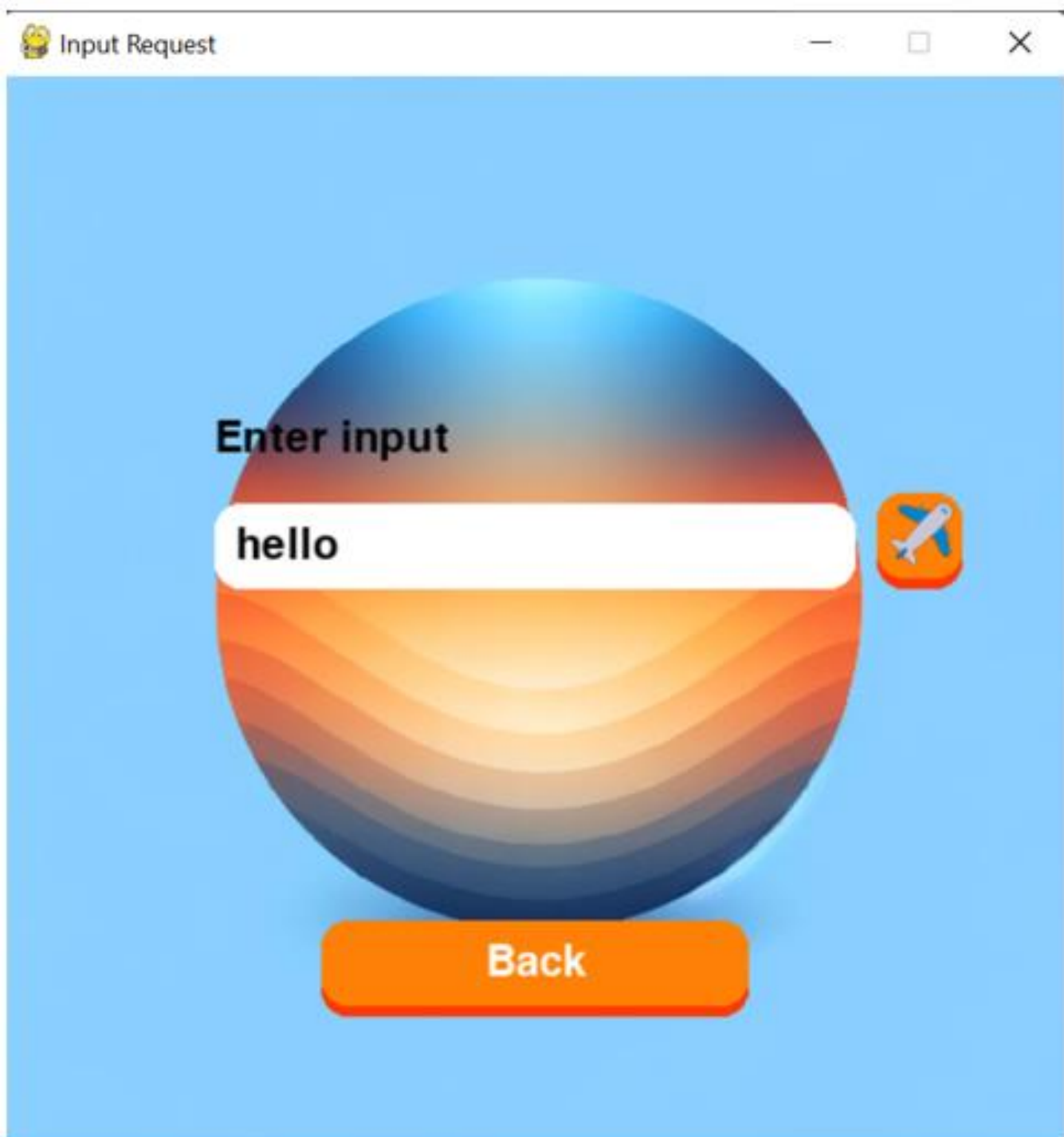
Back

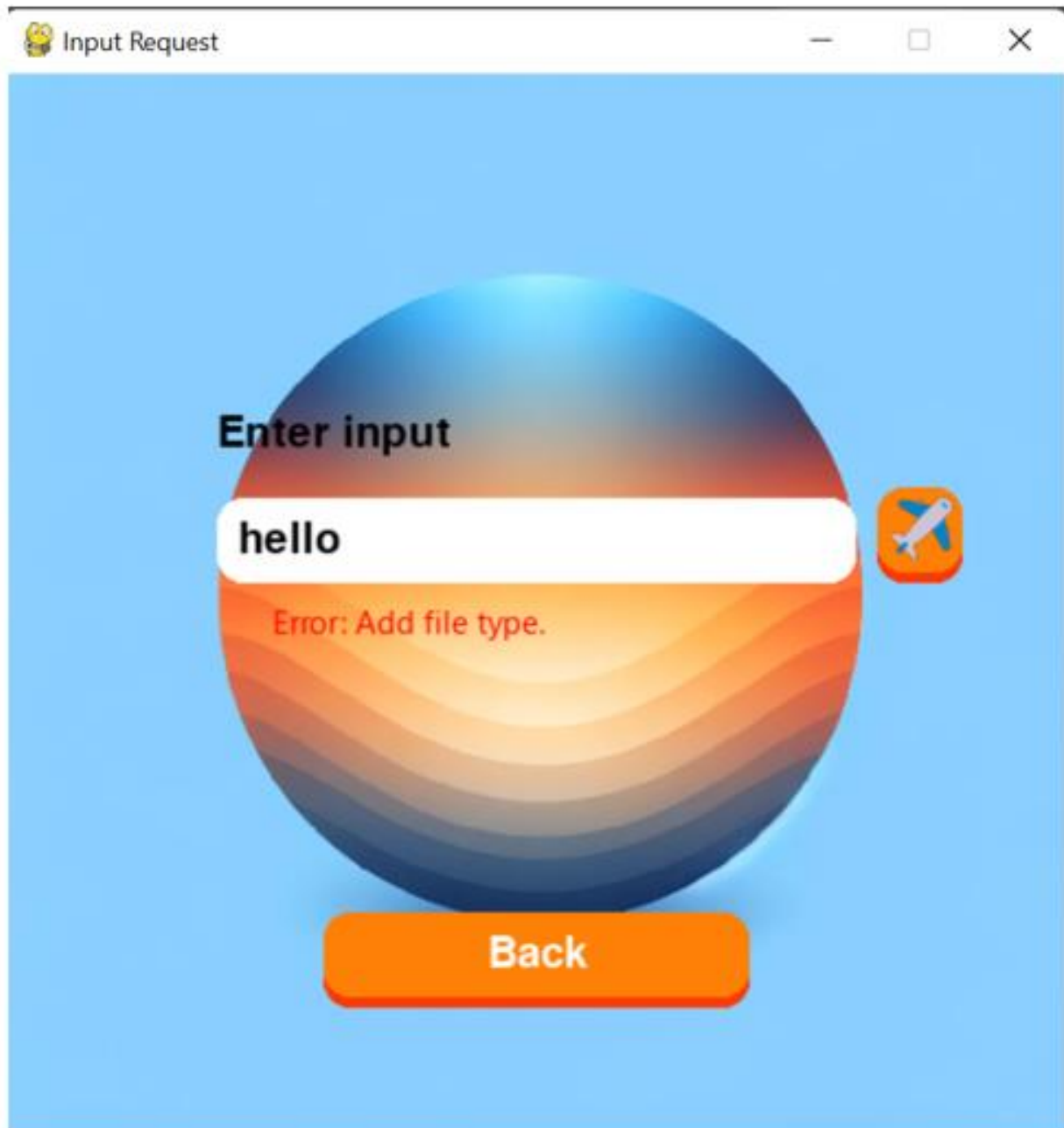


The ui will use OS to check if provided path exists. An error message will appear if it is not.

request file

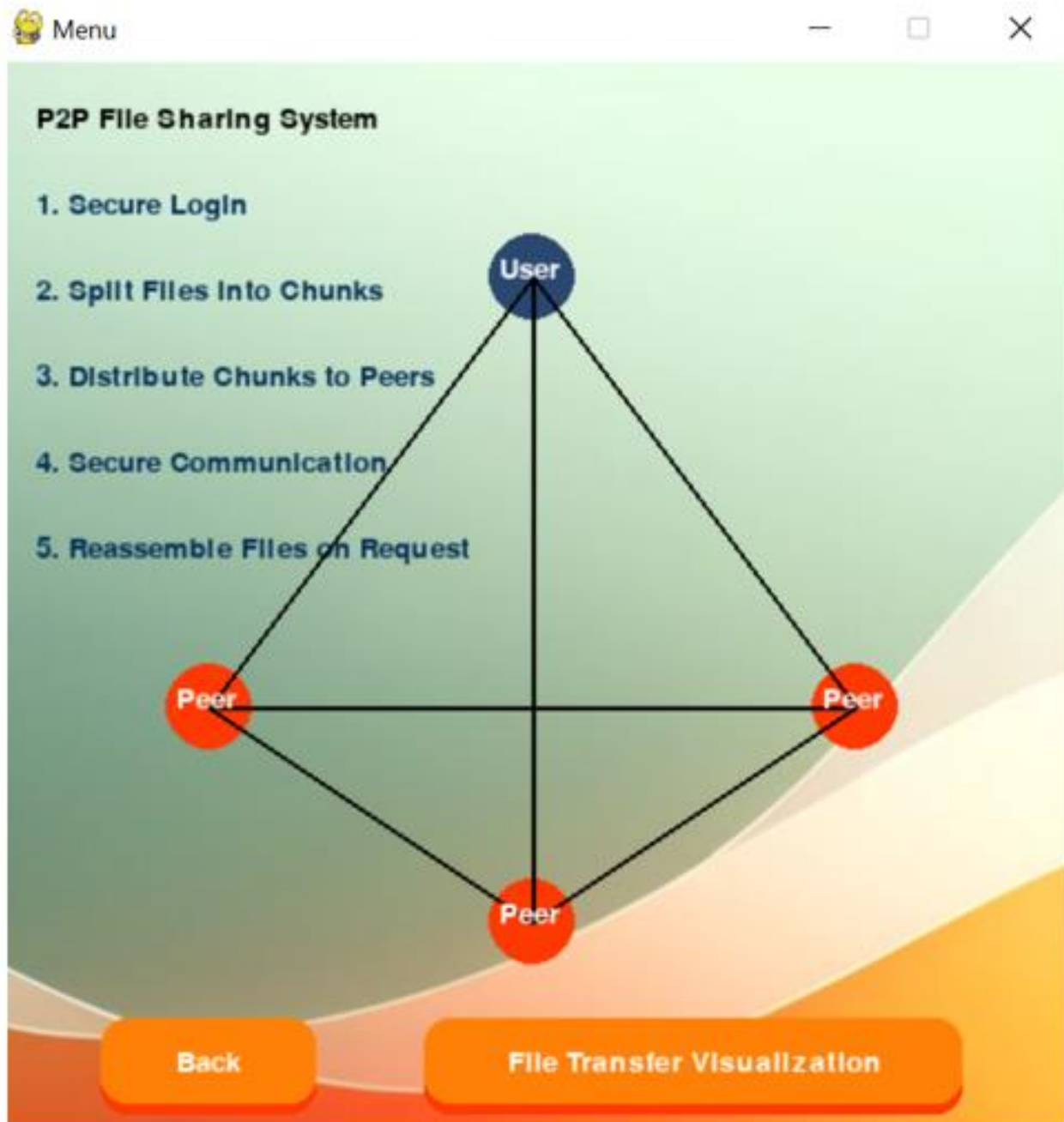






File name must also include the file type extension (e.g. .txt , .py)

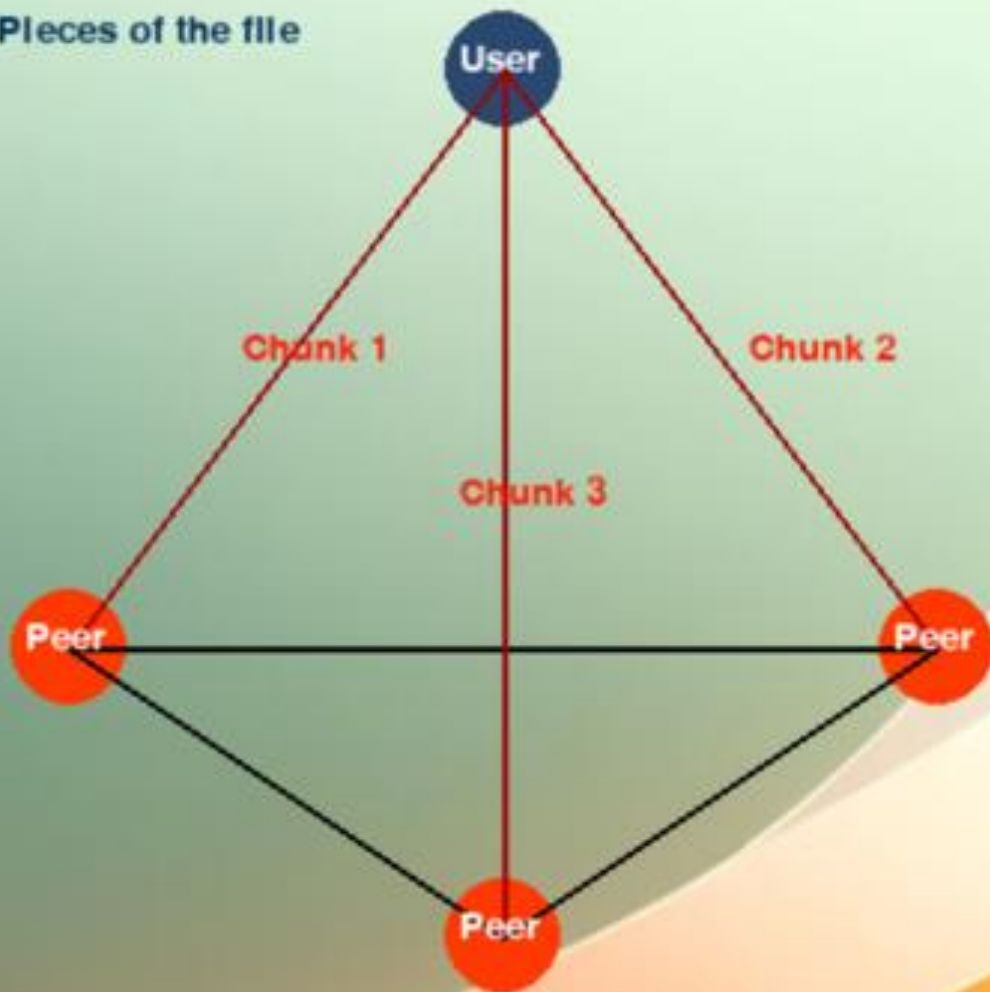
illustration



User has the option to view an illustration demonstrating how the system is built so they may be sure their data is secure.

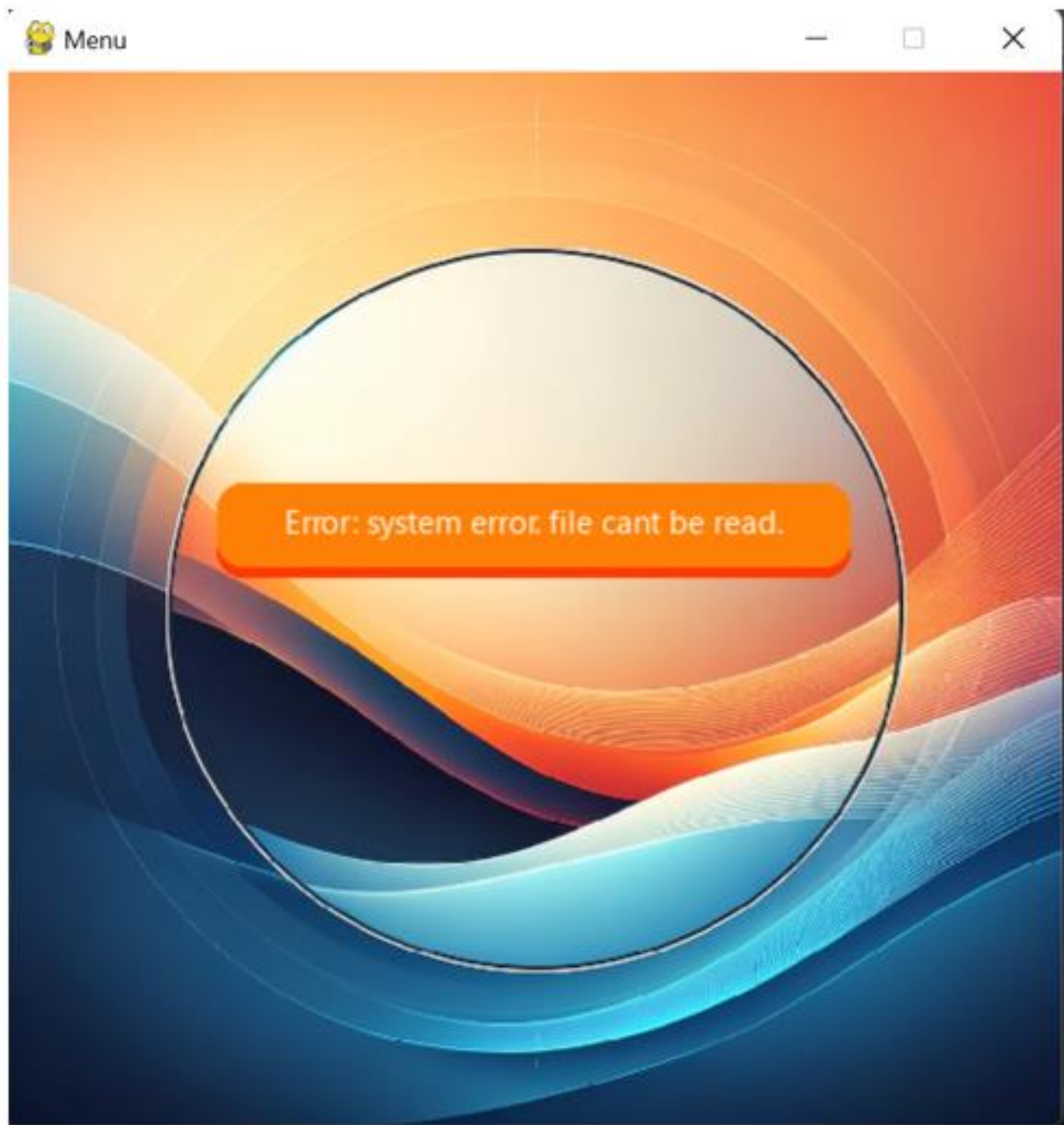
File Transfer Visualization

Chunks: Pieces of the file



Back

Error



If any issues arise from the main code, this message will pop-up.

Description of the Data Structures

Login database (.localappdata.db):

- Purpose: Stores user login information securely by using an inconspicuous name.
- Tables: user credentials
 - Fields:
 - username: TEXT, Primary Key
 - password: TEXT, Hashed password,

| username | password |
|-----------|--|
| Talia_225 | "5f4dcc3b5aa765d61d8327deb882cf99" (MD5 hash of "password") |

Paths Database (paths.db):

- Purpose: Stores the paths to the login database and chunked file pieces.
- Tables: paths
 - Fields:
 - name: TEXT, Primary Key
 - path: TEXT, Path to the database or file chunk

**first line will always hold the location of the login database.*

| name | path |
|--------------------|---|
| ".localappdata.db" | C:file1/file2 |
| Xtx.txt | C:file2/file4;C:file1\file6;C:file2\file3 |

Sent Peers Database (sent_peers.db):

- Purpose: Tracks which peers have received chunks of a particular file.
- Tables: Sent_peers

- Fields:
 - File: TEXT, e.g.
 - Peer: TEXT, IP address of the peer

| file | peer |
|-------------|-------------|
| example.txt | 192.168.1.5 |

Stored files

4. Chunked File Storage:

- Purpose: Stores pieces of files split into chunks.
- Structure:
 - Files are named with an index and a random name for obfuscation, e.g., "1.user_data.txt", "2.config.txt".
 - These files are stored in various directories to further enhance security.

Other data structures used

In this project, various data structures such as lists and queues are used to manage and handle data effectively. Lists are primarily utilized to store collections of objects, such as peers, sent files, and file paths. For example, the ``client_Peer`` class maintains a list of peers, ensuring easy access and management of connected clients. Similarly, the ``sent_file`` class utilizes a list to keep track of the peers to whom a file has been sent. Lists are essential for dynamic storage and retrieval of data, allowing the program to handle multiple entities efficiently.

Queues are used for handling message passing between different components of the system. For instance, the ``recv_q`` queue is used to receive and store messages from other peers, ensuring that incoming data is processed sequentially. Queues guarantee the synchronization of operations, preventing race conditions and ensuring good thread communication.

The choice of these data structures is crucial due to their inherent advantages and disadvantages. Lists provide flexibility in managing collections of data, Queues, on the other hand, ensure the orderly processing of messages. Overall, the combination of lists and queues optimizes data management and communication within the peer-to-peer system.

Overview of Weaknesses and Threats

The Traffic Layer

Protocol Security (Triple Handshake, TCP):

- Threat: Protocol-level attacks, such as SYN flooding, can disrupt communication.
- Solution: Use secure versions of protocols where possible, using P2P prevents the whole system from collapsing, since it is decartelized it would be much harder to achieve SYN flooding

Data breach:

- Threat: Weak or no encryption can lead to data breaches.
- Solution: Use strong encryption standards (AES) for encrypting sensitive data both in transit and at rest (database encryptions and hash).

Code Injection:

- Threat: Code injection vulnerabilities can allow attackers to execute arbitrary code on the system.
- Solution: Implement strict input validation to prevent injection attacks. avoid executing code that relies on user input.

Handling SQL Injection:

- Threat: SQL injection remains a critical threat if not handled properly.
 - Solution: Using parameterized queries is crucial to mitigate SQL injection threats.
- ➔ `cursor.execute("INSERT INTO users (username, password) VALUES (?, ?)", (username, hashed_password))`

Implementation of the Project

Part I - Overview of Modules/Departments

1. Main Module

Main_thread Class

- Name: main_thread
- Role: Handles responses from peer servers and processes user input.
- Operations:
 - run():
 - Function: Handles incoming messages, processes file requests, and manages file chunk assembly.
 - Parameters: None
 - Returns: None
 - is_seq(digits):
 - Function: Checks if a list of digits forms a consecutive sequence.
 - Parameters: digits
 - Returns: Boolean indicating if the sequence is consecutive
 - rearrange_text(jumbled_text):
 - Function: Reconstructs text from segments based on their order.
 - Parameters: jumbled_text
 - Returns: Reconstructed text
 - count_peer(file_name):
 - Function: Counts the number of peers associated with a file in the database.
 - Parameters: file_name
 - Returns: Number of peers

run Class

- Name: run
- Role: Initializes and manages the server, client, and GUI components. Orchestrates the main functionality of the system.
- Operations:

- record_sent_peers():
 - Function: Records which peers have received parts of a file and stores this information in the database 'sent_peers'.
 - Parameters: sent_file
 - Returns: None
- does_file_exist_in_db():
 - Function: Checks if a file with the given name exists in the database 'sent_peers'.
 - Parameters: file_name
 - Returns: Boolean indicating whether the file exists
- main():
 - Function: Initializes the server, client, and GUI components, starts threads for the server and client, and runs the main loop.
 - Parameters: None
 - Returns: None
- send_server():
 - Function: Handles sending requests and files to connected peers. Manages the queue of requests and ensures data is sent securely.
 - Parameters: None
 - Returns: None

2. Client Module

- Name: client
- Role: Manages peer connections, handles file enter/request requests, and interacts with the server module.
- Operations:
 - connect_to_server():
 - Function: Scans for available peer servers, connects to them, and manages the list of connected peers.
 - Parameters: None
 - Returns: None
 - client_send():
 - Function: Sends encrypted messages from the peer's send queue to the connected server.
 - Parameters: None
 - Returns: None
 - run():
 - Function: Initializes and starts threads for connecting to servers, sending messages, and receiving messages.
 - Parameters: None
 - Returns: None

3. Server Module

- Name: server
- Role: Listens for incoming connections, processes client requests, and manages file storage/retrieval.
- Operations:
 - main_server():
 - Function: Initializes the server, binds it to a port, listens for incoming connections, and starts a new thread for each client connection.
 - Parameters: None
 - Returns: None
 - handle_client():
 - Function: Processes client requests, decrypts received messages, handles file uploads and requests, and sends encrypted responses.
 - Parameters: client_socket, encryption
 - Returns: None
 - send_file():
 - Function: Splits the file into chunks, stores the paths in the database, and ensures data integrity.
 - Parameters: data, file_name, paths_conn, paths_cursor
 - Returns: None

4. Encryption Module

- Name: secure_socket
- Role: Provides encryption and decryption functionalities to ensure secure communication.
- Operations:
 - encrypt():
 - Function: Encrypts the provided data using a predefined encryption key.
 - Parameters: data
 - Returns: Encrypted data
 - decrypt():
 - Function: Decrypts the provided data using a predefined encryption key.
 - Parameters: data
 - Returns: Decrypted data

5. dev_txt Module

- Name: chunk_splitter
- Role: Splits files into random chunks and generates random paths for storage.
- Operations:

- `split_into_chunks()`:
 - Function: Splits the provided text into random chunks and returns a list of paths where the chunks are stored.
 - Parameters: text
 - Returns: List of paths
- `split_data()`:
 - Function: Splits the provided text into the specified number of chunks.
 - Parameters: txt, chunk_count
 - Returns: List of chunks

6. random_path Module

- Name: file_manager
- Role: Generates random paths within a given directory.
- Operations:
 - `generate_file_name()`:
 - Function: Generates a random string of letters and digits as a file name.
 - Parameters: None
 - Returns: String representing the file name
 - `generate_random_path()`:
 - Function: Generates a random path within the directory structure.
 - Parameters: None
 - Returns: String representing the random path

7. recreate_file Module

- Name: file_assembler
- Role: Reassembles file chunks into the original file.
- Operations:
 - `assemble_files_into_memory(new_path)`:
 - Function: Assembles file chunks into a single file at the specified path.
 - Parameters: new_path
 - Returns: None
 - `assemble_files_txt()`:
 - Function: Assembles file chunks into a single string.
 - Parameters: None
 - Returns: Assembled string

8. Common Module

- Name: common
- Role: Provides shared variables and classes used across multiple components.

- Operations:
 - Queue():
 - Function: Initializes a queue for inter-thread communication.
 - Parameters: None
 - Returns: Queue object
 - client_Peer:
 - Function: Initializes a peer client with an IP address and a send queue.
 - Parameters: ip
 - Returns: None
 - sent_file:
 - Function: Initializes a sent file with the provided file name and a list of peers to which it was sent.
 - Parameters: file_name
 - Returns: None
 - Globals:
 - Function: Initializes global variables such as port number and list of sent files.
 - Parameters: None
 - Returns: None

Imported Modules/Classes:

sqlite3

- Purpose: Provides SQLite database management, used in login, mainThread, run. Manages user login information, stores file paths, and records peer connections.

socket

- Purpose: Provides socket communication, used in Client, Server, find_ip. handles network connections between peers, enabling communication for file transfers.

threading

- Purpose: Provides threading for concurrent operations. used in mainThread, run, Client, Server, find_ip. Allows concurrent execution of tasks, such as handling multiple client connections and scanning for available IPs.

OS

- Purpose: Provides functions for interacting with the operating system. Used in login, mainThread, run, file_manager, file_assembler. Manages file paths, directories, and performs operations like file deletion.

time

- Purpose: Provides time-related functions. Used in Client, Server, mainThread, run. Introduces delays, manages timing for operations like scanning and communication intervals.

sys

- Purpose: Provides access to some variables used or maintained by the interpreter. Used in login, handles system-specific parameters and functions, such as exiting the application.

hashlib

- Purpose: Provides a common interface to many secure hash and message digest algorithms, used in login. Hashes user passwords for secure storage.

pygame

- Purpose: Provides functionality for creating graphical user interfaces. Used in login, gui. Manages the login interface and user interactions through GUI elements.

random

- Purpose: Provides functions for generating random numbers and selecting random elements. Used in chunk_splitter, file_manager, main. generates random chunk sizes for file splitting and random paths for file storage.

re

- Purpose: Provides regular expression matching operations. Used in find_ip, extracts IP address components for scanning network ranges.

Part II – unique code

Random file division and storage

One of the standout features of this project is the approach to file division and storage. The system employs a unique method to split files into random-sized chunks and distribute these chunks across a variety of locations within the system's directory structure.

Random File Division

1. Unpredictable number of chunks:
 - The 'chunk_splitter' class randomly determines the number of chunks it will divide the data to. The number of chunks must be greater than 2 and smaller than the size of the file. By avoiding uniform chunk numbers, it makes it harder to reassemble the file by a hostile force.
2. Unpredictable chunk sizes:
 - The 'chunk_splitter' class randomly determines the size of each chunk. By avoiding uniform chunk sizes, it becomes much harder for an attacker to predict the structure of the original file. This randomness makes it more difficult to recognize chunks as false files.

This method helps in balancing the load across different storage paths, preventing any single path from becoming a bottleneck.

Relevant code

```
➔ class chunk_splitter:
    def __init__(self):
        self.path = random_path.file_manager()
        self.chunk_count = 0

    def split_into_chunks_file(self, input_file):
        # read input
        with open(input_file, 'r') as f:
            text = f.read()

        # guess chunk size
        total_length = len(text)
        avg_chunk_size = total_length // self.chunk_count
```



```

# split into chunks
chunks = []
start = 0
for i in range(self.chunk_count - 1):
    chunk_size = random.randint(avg_chunk_size - avg_chunk_size // 2,
    avg_chunk_size + avg_chunk_size // 2)
    chunks.append(text[start:start + chunk_size])
    start += chunk_size

# last chunk
chunks.append(text[start:])

# write to files
for i, chunk in enumerate(chunks):
    new_path = f"{self.path.generate_random_path()}.txt"
    with open(os.path.join(new_path), 'w') as f:
        f.write(chunk)
    print(f"new path save is : {new_path}").

```

Randomized Storage Paths

1. Enhanced Security through Obfuscation:

- The file_manager class generates random paths for storing each chunk. These paths are created by traversing randomly through subdirectories within a specified root directory.
- This random path generation adds an extra layer of security, as it obscures the locations of the file chunks, making it exceedingly difficult for unauthorized users to locate and piece together the original file.
- Each chunk being stored in a different location ensures that even if one part of the system is compromised, the attacker would need access to all other parts to reconstruct the file.
- Files are saved under pseudonyms that make them inconspicuous to the average user.

Relevant code

```

➔ def generate_random_path(self):
    current_dir = os.path.join(self.root_dir, "Users") # Start from the Users
    directory
    for _ in range(random.randint(1, num_dirs)):
        sub_dirs = [d for d in os.listdir(current_dir) if
        os.path.isdir(os.path.join(current_dir, d))]
        if not sub_dirs:
            break
        current_dir = os.path.join(current_dir, random.choice(sub_dirs))
    return rf"{current_dir}\\{self.generate_file_name()}"

```

Decentralized Storage

- By distributing chunks across multiple paths and potentially multiple devices, the system mitigates the risk of data loss due to hardware failure or targeted attacks.

Relevant code is the entire P2P. It includes the 'main', 'client' and 'server' codes.

Reassembly Process

- The `file_assembler` class meticulously reassembles the file by reading the chunks from their respective paths in the correct order.
- This precise reassembly process ensures that the original file is reconstructed without any loss of data or corruption.

Relevant code

```
➔ def assemble_files_txt(self):
    txt = ""
    sorted_files = self.sort_files(self.file_paths)
    for file_name in sorted_files:
        file_path = os.path.join(self.root_dir, file_name)
        with open(file_path, 'r') as f_in:
            for line in f_in:
                txt += line
    return txt
```

- Reassembly in the main code utilizes communications protocols to correctly piece together the original file and resaving it.

Relevant code

```
➔ elif parts[0] == "Request file response": # Request file;file name;num;data
    if parts[1] == "File not found":
        print("peers didnt find file")
        break
    file_path = r'D:\tml\output_combined.txt'
    total_chunks = self.count_peer(parts[1])
    print(str(total_chunks) + " is total num of chunks")
    data_write = ";".join([parts[2], parts[3]])
    with open(file_path, 'a') as f_out:
        with open(file_path, 'r') as file:
            data_read = file.read()
            if data_read == "":
                f_out.write(data_write)
                print("File is empty")
            else:
                f_out.write('; ' + data_write)
```

```

with open(file_path, 'r') as file:
    data_read = file.read()
    numbers = []
    print(f"{data_read} is the data read")
    data_list = data_read.split(';')
    print(str(data_list) + " is data list")
    for i in range(len(data_list)):
        if i % 2 == 0:
            print(str(list[i]) + " is appended into nums")
            numbers.append(data_list[i])
    print(str(numbers) + " is num list")
    print(str(self.is_seq(numbers)) + " that nums are seq")
    if self.is_seq(numbers):
        if len(numbers) == total_chunks:
            info = self.rearrange_text(data_read)
            print(info + " is info being printed")
            with open(file_path, 'w') as file:
                file.write("")
            with open(file_path, 'a') as file:
                file.write(info)

```

```

➔ def is_seq(self, digits):
    if not digits:
        return False

```

```

    digits_set = set(digits)
    n = len(digits)

```

```

    for i in range(1, n + 1):
        if str(i) not in digits_set:
            return False

```

```

    return True

```

```

➔ def rearrange_text(self, jumbled_text):
    # Split the jumbled text into segments
    segments = jumbled_text.split(';')

    # Initialize a dictionary to store segments based on their order
    ordered_segments = {}

    # Iterate over the segments and store them in the dictionary based
    on their order
    for i in range(0, len(segments), 2):
        order = int(segments[i])
        segment = segments[i + 1]

```

```

        ordered_segments[order] = segment

    # Reconstruct the text based on the ordered segments
    ordered_text = ".join([ordered_segments[i] for i in range(1,
len(ordered_segments) + 1)])

    return ordered_text

→ def count_peer(self, file_name): # peers will be peer;peer;peer;peer
    db_file = os.path.join(os.path.dirname(os.path.abspath(__file__)),
'sent_peers.db')
    conn = sqlite3.connect(db_file)
    cursor = conn.cursor()
    cursor.execute("CREATE TABLE IF NOT EXISTS SentPeers (File
TEXT, Peer TEXT)")

    cursor.execute("SELECT Peer FROM SentPeers WHERE File=?",
(file_name,))
    result = cursor.fetchall() # Fetch all rows

    peers = str([row[0] for row in result]).split(';') # Extract peer
information from the result

    conn.commit()
    conn.close()

    return len(peers)

```

Part III - Testing Documentation

Sending Very Long Text Files to Peers

- Purpose of the Test: The goal of this test was to determine if the system could handle and successfully transmit very long text files to peers without any data loss or corruption.
- What Was Actually Done: A text file containing a very large amount of text (over 10,000 bytes) was created. The file was then sent to a peer using the P2P file sharing system. The peer received the file and verified its integrity by comparing it with the original file.
- Results of the Test: The system successfully transmitted the large text file without any data loss or corruption. The file received by the peer was identical to the original file.
- Problems Discovered and Solutions: No problems were discovered during this test. The system handled the large file transmission efficiently.

Sending Text Files Containing All Special Characters

- Purpose of the Test: The purpose of this test was to check if the system could correctly handle and transmit text files containing all possible special characters.
- What Was Actually Done: A text file containing every special character available on the keyboard (e.g., !@#\$%^&*()_+{|}:<>?) was created. This file was then sent to a peer. The peer received the file and checked its contents to ensure that all special characters were present and correct.
- Results of the Test: The system successfully transmitted the text file containing special characters, except for the ';' character. Since it is used in the peer communication as a partition, a file having this character causes major issues. The user needs to be made aware of that and avoid use of the ';' char.

Handling Interrupted Connections

- Purpose of the Test: To determine how the system handles interruptions in the connection during file transmission and whether it can resume or recover the transmission.
- What Was Actually Done: During the transmission of a large file, the network connection was intentionally interrupted. After a short period, the

connection was restored. The system's ability to resume the transmission was then evaluated.

- Results of the Test: The system couldn't resume the file transmission from the point of interruption once the connection was restored. There was no data loss, since the users code had yet to delete its copy of the file.
- Problems Discovered and Solutions: a better protocol should be implemented – the users main code should await peer server confirmation that it saved the data preventing accidental loss.

Stress Testing with Multiple Simultaneous Transfers

- Purpose of the Test: To evaluate the system's performance and stability when multiple files are being transmitted simultaneously to different peers.
- What Was Actually Done: Multiple peers were set up to simultaneously send and receive files of various sizes. The system's performance, including transmission speed and stability, was monitored.
- Results of the Test: The system maintained stable performance and handled multiple simultaneous transfers effectively. There were no crashes or very significant slowdowns.
- Problems Discovered and Solutions: During the initial tests, the system experienced slight slowdowns when handling simultaneous transfers. This problem is understandable and doesn't significantly hinder the user experience.

Error management

Error management is crucial for making sure the system is reliable. Throughout the code, various error handling techniques are employed to address different scenarios. Here's an overview of the error management strategies used in each class:

1. Login Class:

Error handling during database operations: The login class handles potential errors that may occur during database operations such as connection errors, query failures, or data retrieval issues. It employs try-except blocks to catch exceptions and provides appropriate error messages to the user.

Input validation: Before processing user input, the login class validates the input data to ensure it meets certain criteria (e.g., username and password format). If invalid data is detected, it prompts the user to provide correct input.

2. Main Class:

Error handling during thread initialization: When creating threads for server and client operations, the main class handles potential errors that may occur during thread creation. It catches exceptions raised during thread initialization and logs or displays error messages accordingly.

Exception handling during UI operations: If errors occur during UI operations (e.g., user input processing), the main class catches exceptions and prevents them from crashing the program. It provides feedback to the user about the error and gracefully handles the situation.

3. Client Class:

Connection error handling: The client class handles potential errors that may occur during connection attempts to peer servers. It employs try-except blocks to catch socket-related exceptions such as connection timeouts or refused connections. Depending on the error type, it may retry the connection or notify the user about the connection failure.

Exception handling during message processing: When receiving and processing messages from the server, the client class incorporates error handling mechanisms to deal with unexpected message formats or transmission errors. It verifies message integrity and ensures that the system remains stable even in the presence of malformed or corrupted messages.

4. Server Class:

Error handling during client interactions: The server class implements robust error handling mechanisms to manage client connections and message processing. It catches exceptions raised during client interactions, such as socket errors or message parsing failures, and handles them gracefully without disrupting the server's functionality.

Database operation error management: During database operations, the server class employs error handling techniques to address potential issues such as database file corruption, SQL query errors, or database connection failures. It logs relevant information about the error and takes appropriate actions to maintain system stability.

5. Other Classes (e.g., Find_IP, Dev_TXT, Random_Path, Recreate_File):

Similar to the main classes, error management in these classes involves handling specific exceptions that may occur during their respective operations. For instance, the Find_IP class handles socket-related errors during IP scanning, while the Dev_TXT class manages file I/O errors during data splitting operations.

Use and directions

1. Download python 3.7
2. Unzip the file "tml".
3. Open command in the "tml" file on file explorer, type the following:
 - >python -m venv venv
 - >venv\Scripts\activate
 - >pip install -r requirements.txt
4. Enter the directry you want the system to opperate in in common.py in this section of the code:

```
class Globals:
    # srcName=""
    # dstName=""
    port = 1000
    sent_file_list: List[sent_file] = []
    dir = 'g:' ←
```

5. Run the function "login.py" on cmd

For full range of use, open at least two other computers on the same network.

Project Code

Login.py

```
import pygame
import sqlite3
import hashlib
import sys
import os

# Import the file_manager class from common.py
from random_path import file_manager
from main import run
# Initialize Pygame
pygame.init()

# Set up the screen
screen_width = 400
screen_height = 300
screen = pygame.display.set_mode((screen_width, screen_height))
pygame.display.set_caption("Login System")

# Colors
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
GRAY = (200, 200, 200)
RED = (255, 0, 0)

# Fonts
font = pygame.font.Font(None, 32)

# Database file path for paths.db
script_dir = os.path.dirname(os.path.abspath(__file__))
paths_db_file = os.path.join(script_dir, 'paths.db')
```

```

# Connect to paths.db database
paths_conn = sqlite3.connect(paths_db_file)
paths_cursor = paths_conn.cursor()

# Create table if not exists
paths_cursor.execute("CREATE TABLE IF NOT EXISTS paths
    (name TEXT PRIMARY KEY NOT NULL,
    path TEXT NOT NULL)")

# Function to insert path into paths.db
def insert_path(name, path):
    paths_cursor.execute("INSERT INTO paths (name, path) VALUES (?, ?)", (name,
path))
    paths_conn.commit()

# Function to check if path exists in paths.db
def path_exists(name):
    paths_cursor.execute("SELECT * FROM paths WHERE name=?", (name,))
    if paths_cursor.fetchone():
        return True
    return False

# Function to display message
def message_to_screen(msg, color, y_displace=0):
    text_surface = font.render(msg, True, color)
    text_rect = text_surface.get_rect(center=(screen_width/2, screen_height/2 +
y_displace))
    screen.blit(text_surface, text_rect)

# Function to create button
def create_button(msg, x, y, width, height, inactive_color, active_color, action=None):
    mouse = pygame.mouse.get_pos()
    click = pygame.mouse.get_pressed()

    if x + width > mouse[0] > x and y + height > mouse[1] > y:
        pygame.draw.rect(screen, active_color, (x, y, width, height))
        if click[0] == 1 and action is not None:
            action()
    else:

```

```

pygame.draw.rect(screen, inactive_color, (x, y, width, height))

text_surface = font.render(msg, True, BLACK)
text_rect = text_surface.get_rect(center=(x + width / 2, y + height / 2))
screen.blit(text_surface, text_rect)

# Function to display text input box
def text_input_box(msg, y_displace=0):
    input_box = pygame.Rect(100, 140 + y_displace, 200, 32)
    color_active = pygame.Color('lightskyblue3')
    color_inactive = pygame.Color('dodgerblue2')
    color = color_inactive
    active = False
    text = ""
    while True:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                sys.exit()
            if event.type == pygame.MOUSEBUTTONDOWN:
                if input_box.collidepoint(event.pos):
                    active = not active
                else:
                    active = False
                color = color_active if active else color_inactive
            if event.type == pygame.KEYDOWN:
                if active:
                    if event.key == pygame.K_RETURN:
                        return text
                    elif event.key == pygame.K_BACKSPACE:
                        text = text[:-1]
                    else:
                        text += event.unicode
        screen.fill((30, 30, 30))
        pygame.draw.rect(screen, color, input_box, 2)
        text_surface = font.render(msg, True, WHITE)
        screen.blit(text_surface, (input_box.x - 80, input_box.y - 30))
        text_surface = font.render(text, True, color)
        screen.blit(text_surface, (input_box.x + 5, input_box.y + 5))
    pygame.display.flip()

```

```

# Function to check login
def check_login(username, password, cursor):
    hashed_input_password = hashlib.md5(password.encode()).hexdigest()
    cursor.execute("SELECT * FROM users WHERE username=?", (username,))
    user_data = cursor.fetchone()
    if user_data:
        hashed_db_password = user_data[1]
        if hashed_input_password == hashed_db_password:
            return True
    return False

# Function to create new user
def create_user(username, password, cursor, conn):
    hashed_password = hashlib.md5(password.encode()).hexdigest()
    try:
        cursor.execute("INSERT INTO users (username, password) VALUES (?, ?)",
            (username, hashed_password))
        conn.commit()
    except sqlite3.IntegrityError:
        # Username already exists
        return False
    return True

# Main function
def main():
    # Database file path for users.db
    db_name = '.localappdata.db'

    # Check if path exists in paths.db
    if path_exists(db_name):
        paths_cursor.execute("SELECT path FROM paths WHERE name=?", (db_name,))
        db_path = paths_cursor.fetchone()[0] # db_name is the secretive name of the login
        data base. path to it is stored on the first line of paths_data_base
    else:
        # Generate a random path
        fm = file_manager()
        db_path = fm.generate_random_path()
        insert_path(db_name, db_path)

```

```

# Connect to users.db database
db_conn = sqlite3.connect(db_path)
db_cursor = db_conn.cursor()

# Create table if not exists
db_cursor.execute("""CREATE TABLE IF NOT EXISTS users
    (username TEXT PRIMARY KEY NOT NULL,
    password TEXT NOT NULL)""")

while True:
    screen.fill(BLACK)
    message_to_screen("Login System", WHITE, -50)
    create_button("Login", 100, 120, 200, 50, GRAY, WHITE, lambda:
login(db_cursor))
    create_button("Sign Up", 100, 200, 200, 50, GRAY, WHITE, lambda:
sign_up(db_cursor, db_conn))
    pygame.display.update()

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

# Function for login
def login(cursor):
    screen.fill(BLACK)
    message_to_screen("Login", WHITE, -50)
    message_to_screen("Username:", WHITE, -20)
    username = text_input_box("Enter username:", 0)
    message_to_screen("Password:", WHITE, 20)
    password = text_input_box("Enter password:", 40)

    if check_login(username, password, cursor):
        message_to_screen("Login successful!", WHITE, 80)
        call_main = run()
        call_main.main()
    else:
        message_to_screen("Login failed. Please try again.", RED, 80)
        pygame.display.update()
        pygame.time.wait(2000) # Pause for 2 seconds before returning to main menu

```

```

# Function for sign up
def sign_up(cursor, conn):
    screen.fill(BLACK)
    message_to_screen("Sign Up", WHITE, -50)
    message_to_screen("Username:", WHITE, -20)
    username = text_input_box("Enter username:", 0)
    message_to_screen("Password:", WHITE, 20)
    password = text_input_box("Enter password:", 40)

    if create_user(username, password, cursor, conn):
        message_to_screen("User created successfully!", WHITE, 80)
    else:
        message_to_screen("Username already exists. Please choose a different
username.", RED, 80)
    pygame.display.update()
    pygame.time.wait(2000) # Pause for 2 seconds before returning to main menu

if __name__ == "__main__":
    main()

```

Main.py

```
import os
import sqlite3

import server
from client import Client
from time import sleep
from threading import Thread
from common import *
import gui as ui
from dev_txt import chunk_splitter

class mainThread(Thread):
    def __init__(self):
        Thread.__init__(self)
        print("mainThread start")
        self.more_than_2 = True

    def run(self):
        while True: # handles answer
            if not recv_q.empty():
                msg = recv_q.get()
                print(msg + " is message")
                parts = msg.split(';')
                if parts[0] == "found server":
                    a = Client() # create another client
                    a.start()
                elif parts[0] == "Request file": # Request file;file name;num;data
                    file_path = r'D:\tml\output_combined.txt'
                    total_chunks = self.count_peer(parts[1])
                    print(str(total_chunks) + " is total num of chunks")
                    data_write = ";".join([parts[2], parts[3]])
                    with open(file_path, 'a') as f_out:
                        with open(file_path, 'r') as file:
                            data_read = file.read()
                            if data_read == "":
                                f_out.write(data_write)
                                print("File is empty")
                            else:
                                f_out.write('; ' + data_write)
                    with open(file_path, 'r') as file:
```



```

        data_read = file.read()
        numbers = []
        print(f"{data_read} is the data read")
        data_list = data_read.split(';')
        print(str(data_list) + " is data list")
        for i in range(len(data_list)):
            if i % 2 == 0:
                print(str(list[i]) + " is appended into nums")
                numbers.append(data_list[i])
        print(str(numbers) + " is num list")
        print(str(self.is_seq(numbers)) + " that nums are seq")
        if self.is_seq(numbers):
            if len(numbers) == total_chunks:
                info = self.rearrange_text(data_read)
                print(info + " is info being printed")
                with open(file_path, 'w') as file:
                    file.write("")
                with open(file_path, 'a') as file:
                    file.write(info)

    elif parts[0] == "Enter file":
        try:
            print(f"{parts[1]}")
            # os.remove(parts[1])
            print(f"File {os.path.basename(parts[1])} has been deleted successfully.")
        except OSError as e:
            print(f"Error deleting the file: {e}")
        else:
            print("3331 from server ", msg)
            sleep(0.02) # sleep 20 ms - let the cpu breath

def is_seq(self, digits):
    if not digits:
        return False

    digits_set = set(digits)
    n = len(digits)

    for i in range(1, n + 1):
        if str(i) not in digits_set:
            return False

    return True

```

```

def rearrange_text(self, jumbled_text):
    # Split the jumbled text into segments
    segments = jumbled_text.split(';')

    # Initialize a dictionary to store segments based on their order
    ordered_segments = {}

    # Iterate over the segments and store them in the dictionary based on their order
    for i in range(0, len(segments), 2):
        order = int(segments[i])
        segment = segments[i + 1]
        ordered_segments[order] = segment

    # Reconstruct the text based on the ordered segments
    ordered_text = ".join([ordered_segments[i] for i in range(1, len(ordered_segments) + 1)])

    return ordered_text

def count_peer(self, file_name): # peers will be peer;peer;peer;peer
    db_file = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'sent_peers.db')
    conn = sqlite3.connect(db_file)
    cursor = conn.cursor()
    cursor.execute("CREATE TABLE IF NOT EXISTS SentPeers (File TEXT, Peer TEXT)")

    cursor.execute("SELECT Peer FROM SentPeers WHERE File=?", (file_name,))
    result = cursor.fetchall() # Fetch all rows

    peers = str([row[0] for row in result]).split(';') # Extract peer information from the result

    conn.commit()
    conn.close()

    return len(peers)

def has_consecutive_sequence(self, numbers):
    # Sort the list of numbers
    sorted_numbers = sorted(numbers)

    # Check if the sorted list forms a consecutive sequence
    for i in range(1, len(sorted_numbers)):
        if int(sorted_numbers[i]) != int(sorted_numbers[i - 1]) + 1:
            return False

    # Check if all digits from 1 to the largest number are present

```

```

return sorted_numbers == list(range(1, sorted_numbers[-1] + 1))

def append_data(self, info_list, new_data):
    # Split the info list into pairs (number, data)
    pairs = info_list.split(';')
    if len(pairs) % 2 != 0:
        # Handle cases where the length of pairs is not even
        print("Warning: Uneven number of elements in info_list.")
        return self.pair_sort([info_list, new_data])

    else:
        self.more_than_2 = True

    # Create pairs of elements from the split list
    pairs = [(pairs[i], pairs[i + 1]) for i in range(0, len(pairs), 2)]

    # Append the new data with its corresponding number
    pairs.append(new_data)

    # Sort the pairs based on the numeric values
    sorted_pairs = sorted(pairs, key=lambda x: int(x[0]) if x[0].isdigit() else float('inf'))

    # Reconstruct the sorted string
    sorted_info = ';'.join([pair[0] + ';' + pair[1] for pair in sorted_pairs])

    return sorted_info

def pair_sort(self, pairs):
    self.more_than_2 = False
    # Split each pair into numeric and data components
    split_pairs = [pair.split(';') for pair in pairs]

    # Sort the pairs based on the numeric values
    sorted_pairs = sorted(split_pairs, key=lambda x: int(x[0]) if x[0].isdigit() else float('inf'))

    # Reconstruct the sorted pairs into a list
    try:
        sorted_list = [f"{pair[0]};{pair[1]}" for pair in sorted_pairs]
    except IndexError:
        return pairs
    return sorted_list

```

class run:

```

def __init__(self):
    self.gui = ui.Main_menu()
    self.server = server.server()
    self.client = Client()

def record_sent_peers(self, sent_file): # peers will be peer;peer;peer;peer
    db_file = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'sent_peers.db')
    conn = sqlite3.connect(db_file)
    cursor = conn.cursor()
    cursor.execute("CREATE TABLE IF NOT EXISTS SentPeers (File TEXT, Peer TEXT)")

    # Record each peer that has been sent parts of the file
    cursor.execute("INSERT INTO SentPeers (File, Peer) VALUES (?, ?)",
        (sent_file.file_name, ';' + join(sent_file.sent_to)))

    conn.commit()
    conn.close()

def does_file_exist_in_db(self, file_name):
    # Query the database to check if the file exists
    db_file = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'sent_peers.db')
    conn = sqlite3.connect(db_file)
    cursor = conn.cursor()
    cursor.execute("CREATE TABLE IF NOT EXISTS SentPeers (File TEXT, Peer TEXT)")

    cursor.execute("SELECT * FROM SentPeers WHERE File=?", (file_name,))
    result = cursor.fetchone() # Fetch one row

    conn.close()

    print(f"result form sent_peers db is {result}")
    return result is not None

def main(self):
    server_thread = self.server
    a = mainThread()
    a.start()

    a = Thread(target=server_thread.main_server, args=())
    a.start()

    sleep(3)
    a = self.client
    a.start()

```

```

sleep(3)
b = Thread(target=self.send_server, args=())
b.start()

```

```

self.gui.main_loop()

```

```

def send_server(self):
    print("send reached")
    no_client = True
    while 1:
        if len(client_Peer.peers) != 0:
            try:
                val = self.gui.input_text[-1]
                self.gui.input_text.pop()
                if val == "":
                    break
            except:
                val = ""
            file_info = ""
            if val != "" and val != 'Enter file;Back':
                val = val.split(';')

                if val[0] == "Enter file":
                    file_path = val[1] # G:\file1\file 2\file 3\file 4\txx.txt
                    try:
                        with open(file_path, 'rb') as file:
                            file_info = str(file.read(10000)) # Read up to 1024 bytes from the file

```

CHANGE

```

except:
    self.gui.error_popup = True
    print("path doesnt exist")
    val = ""
    break
chunks = chunk_splitter().split_data(file_info, len(client_Peer.peers))
for c in chunks:
    print("chank chunk", c)
    if not self.does_file_exist_in_db(os.path.basename(val[1])):
        sent_to = sent_file(os.path.basename(val[1]))
        tmp_count = 1
        for i in client_Peer.peers:
            send_val = val[0] + ';' + val[1] + ';' + str(
                tmp_count) + ';' + chunks.pop(0)

```

```

        i.send_q.put(send_val) # Enter file;{file path};{the number of this chunk};{the
information of the chunk}
        sent_to.sent_to.append(str(i.ip)) # remember who we sent to
        tmp_count = tmp_count + 1
        self.record_sent_peers(sent_to)
    else:
        self.gui.error_popup = True
        print("file already entered")

    elif val[0] == "Request file":
        file_name = os.path.basename(val[1]) # xtx.txt
        print(f"in req file {file_name}")
        if not self.does_file_exist_in_db(file_name):
            self.gui.error_popup = True
            print("path doesnt exist")
            break
        send_val = val[0] + ';' + file_name # Request file;"name"
        for i in client_Peer.peers:
            i.send_q.put(send_val)
        try:
            print(f"sent value is {send_val}")
        except UnboundLocalError:
            pass #there is no value yet
    elif no_client:
        print("no client connected") # add ui
        no_client = False

if __name__ == "__main__":
    r = run()
    r.main()

```

client.py

```
import socket
from threading import Thread
from common import * # Assuming common contains shared variables like Globals, recv_q, etc.
import time
from encryption import SecureSocket # Importing the SecureSocket class from the encryption
module
import find_ip

connected_server = []

class Client(Thread):
    def __init__(self):
        Thread.__init__(self)
        self.peer = None
        print("client start")
        self.my_socket = socket.socket()

        encryption_key = b'Sixteen byte key' # Use the same key for encryption and decryption
        self.encryption = SecureSocket(encryption_key)

    def connect_to_server(self):
        while 1:
            ip_scan = find_ip.IP_scanner()
            self.server_ip_list = ip_scan.scan_all_ips()
            for ip in self.server_ip_list:
                try:
                    self.my_socket.connect((ip, Globals.port))
                    self.peer = client_Peer(ip)
                    client_Peer.add_peers(self.peer)
                    recv_q.put(f"found sever at {ip}")
                    print("338723", f"client connect to to server at {ip}")
                except Exception as e:
                    pass
                    # print(f"error on client is: {e}")
            if len(self.server_ip_list) == 0:
                #print("338723", "no servers found")
                pass
            time.sleep(20)
```

```

def client_send(self):
    while True:
        if self.peer is not None:
            if not self.peer.send_q.empty():
                msg = self.peer.send_q.get()
                msg = msg.encode('utf-8') # Encode the message as bytes
                encrypted_msg = self.encryption.encrypt(msg) # Encrypt the message
                self.my_socket.send(encrypted_msg)
                time.sleep(0.02)

def run(self):

    thread_conn = Thread(target=self.connect_to_server, args=())
    thread_conn.start()

    encryption_key = b'Sixteen byte key' # Use the same key for encryption and decryption
    secure_socket = SecureSocket(encryption_key)
    thread_send = Thread(target=self.client_send, args=())
    print("OG client : got to send thread")
    thread_send.start()

    while True:
        try:
            data = self.my_socket.recv(1024)
            decrypted_data = secure_socket.decrypt(data) # Decrypt received data
            decrypted_data = decrypted_data.decode('utf-8')
            recv_q.put(decrypted_data)
        except OSError:
            pass # no client connected

if __name__ == "__main__":
    print("usage: python main.py")

```


server.py

```
import os
import socket
import time
from threading import Thread
import sqlite3
from common import *
from dev_txt import chunk_splitter
from recreate_file import file_assembler
from encryption import SecureSocket # Import the SecureSocket class from the encryption
module

class server:
    def __init__(self):
        self.server_socket = socket.socket()
        self.port = Globals.port
        self.clients = set()

    def file_exists(self, file_name, paths_cursor):
        # Execute a SELECT query to check if the file name exists in the database
        paths_cursor.execute("SELECT COUNT(*) FROM paths WHERE name=?", (file_name,))
        result = paths_cursor.fetchone()[0]
        # If the result is greater than 0, it means the file name exists in the database
        return result > 0

    def handle_client(self, client_socket, encryption):
        """Handles a single client connection."""
        while True:
            paths_db_file = os.path.join(os.path.dirname(os.path.abspath(__file__)), f'paths.db')
            paths_conn = sqlite3.connect(paths_db_file)
            paths_cursor = paths_conn.cursor()

            client_info = client_socket.recv(1024)
            print(f"client info!!!!!!! + {client_info}")
            if client_info != b'':
                decrypted_info = encryption.decrypt(client_info).decode('utf-8')
            else:
                break
            print(f"info server got is {decrypted_info}")
            info_list = decrypted_info.split(';')
```

```

    if info_list[0] == "Enter file":
        data = (';').join(info_list)
        file_name = os.path.basename(info_list[1]) # i give the entire path, not actually
necesseray unless we dicdie to implement returning to the same path.
        file_info = info_list[2] + ';' + info_list[3]
        self.send_file(file_info, file_name, paths_conn, paths_cursor)
    elif info_list[0] == "Request file":
        print(f"entered req in server op, {info_list[1]}") # file name
        for row in paths_cursor.execute("SELECT path FROM paths WHERE name=?",
(info_list[1],)):
            path_list = row[0].split(';')
            for path in path_list:
                print("f", path)
            try:
                assemble = file_assembler(path_list).assemble_files_txt()
            except Exception as e:
                print(f"couldnt re-assemble: {e}")
                data = ""
                break
            print(assemble)
            data = str(info_list[0] + ';' + info_list[1] + ';' + assemble) # Request file;xtx.txt;1;he
            print(data, " is the data I'm sending back as req on server")
        else:
            data = "not in protocol. no answer ig"

        print(f"123 data sent is {data}")
        encrypted_data = encryption.encrypt(data.encode('utf-8'))
        client_socket.send(encrypted_data)
        data = ""
        paths_conn.close() # Close the connection at the end of each iteration

def main_server(self):
    while True:
        try:
            self.server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            self.server_socket.bind(("0.0.0.0", self.port))
            print("111", "server bind to port " + str(self.port))
            break
        except socket.error as e:
            print(e)
            time.sleep(1)

    self.my_name = "bb" + str(self.port)
    print(self.my_name, " server start")

```

```

self.server_socket.listen(1)

while True:
    (client_socket, client_address) = self.server_socket.accept()
    print("7167", "New client connecting")
    self.clients.add(client_socket)
    encryption_key = b'Sixteen byte key' # Use the same key for encryption and decryption
    secure_socket = SecureSocket(encryption_key)
    a = Thread(target=self.handle_client, args=(client_socket, secure_socket))
    a.start()

def send_file(self, data, file_name, paths_conn, paths_cursor):
    print(f"DATA IS {data}")
    chunks = chunk_splitter()
    path_list = (';').join(chunks.split_into_chunks(data))
    print(path_list)
    try:
        paths_cursor.execute("""CREATE TABLE IF NOT EXISTS paths
                                (name TEXT PRIMARY KEY NOT NULL,
                                 path TEXT NOT NULL)""")
        paths_cursor.execute("INSERT OR IGNORE INTO paths (name, path) VALUES (?, ?)",
                               (file_name, path_list))
        paths_conn.commit()
        print("Rows inserted:", paths_cursor.rowcount)
    except Exception as e:
        print("Error inserting data:", e)
    finally:
        paths_conn.close()

if __name__ == "__main__":
    print("usage: python main.py")

```

Common

```
from queue import Queue
from typing import List

inconspicuous_file_names = [
    "config",
    "cache",
    "log",
    "log",
    "index",
    "record",
    "audit",
    "inventory",
    "manifest",
    "template",
    "configuration_template",
    "log_template",
    "index_template",
    "record_template",
    "audit_template",
    "inventory_template",
    "manifest_template",
    "configuration_backup",
    "log_backup",
    "index_backup",
    "record_backup",
    "audit_backup",
    "inventory_backup",
    "manifest_backup",
    "settings.cfg",
    "temp.dat",
    "prefs.ini",
    "cache.data",
    "errors.log",
    "debug.log",
    "temp_data.tmp",
    "system_config.sys",
    "network_info",
    "resource_usage",
    "diagnostic_report",
```

```

"protocol_data",
"security_audit",
"hardware_inventory",
"configuration_data",
"system_logs",
"performance_metrics",
"system_errors.log",
"data_dump.dat",
"operational_report",
"incident_report",
"data_sheet",
"testing_results",
"protocol_summary",
"system_status",
"resource_summary",
"diagnostic_summary",
"equipment_inventory",
]

# send_q = Queue()
recv_q = Queue()

class client_Peer:
    peers: List['client_Peer'] = [] # Specify the type as List['client_Peer']

    def __init__(self, ip):
        self.ip = ip
        self.send_q = Queue()
        self.send_q.empty()

    @classmethod
    def add_peers(cls, peer):
        client_Peer.peers.append(peer)

class sent_file:
    def __init__(self, file_name):
        self.file_name = file_name # str
        self.sent_to: List[client_Peer] = [] # list

class file_paths_list:
    def __init__(self, name, list):

```

```
self.name = name  
self.list = list
```

```
class Globals:  
    # srcName=""  
    # dstName=""  
    port = 1000  
    sent_file_list: List[sent_file] = []
```

Dev_txt

```
import os
import random
import random_path

class chunk_splitter:
    def __init__(self):
        self.path = random_path.file_manager()
        self.chunk_count = 3

    def split_into_chunks_file(self, input_file):
        # read input
        with open(input_file, 'r') as f:
            text = f.read()

        # guess chunk size
        total_length = len(text)
        avg_chunk_size = total_length // self.chunk_count

        # split into chunks
        chunks = []
        start = 0
        for i in range(self.chunk_count - 1):
            chunk_size = random.randint(avg_chunk_size - avg_chunk_size // 2, avg_chunk_size +
avg_chunk_size // 2)
            chunks.append(text[start:start + chunk_size])
            start += chunk_size

        # last chunk
        chunks.append(text[start:])

        # write to files
        for i, chunk in enumerate(chunks):
            new_path = f"{self.path.generate_random_path()}.txt"
            with open(os.path.join(new_path), 'w') as f:
                f.write(chunk)
            print(f"new path save is : {new_path}")

    def split_into_chunks(self, text):
        self.chunk_count = random.randint(1, len(text)-1) # length of total text must be over 1
        self.chunk_count = 3
```

```

path_list = []
# guess chunk size
total_length = len(text)
avg_chunk_size = total_length // self.chunk_count

# split into chunks
chunks = []
start = 0
for i in range(self.chunk_count - 1):
    chunk_size = random.randint(avg_chunk_size - avg_chunk_size // 2, avg_chunk_size +
avg_chunk_size // 2)
    chunks.append(text[start:start + chunk_size])
    start += chunk_size

# last chunk
chunks.append(text[start:])

# write to files
for i, chunk in enumerate(chunks):
    new_path = f"{self.path.generate_random_path()}.txt"
    path_list.append(new_path)
    with open(os.path.join(new_path), 'w') as f:
        f.write(chunk)

return path_list

def split_data(self, txt, chunk_count):

    path_list = []
    # guess chunk size
    if chunk_count == 1:
        return [txt]
    total_length = len(txt)
    avg_chunk_size = total_length // chunk_count

    # split into chunks
    chunks = []
    start = 0
    for i in range(chunk_count - 1):
        chunk_size = random.randint(avg_chunk_size - avg_chunk_size // 2, avg_chunk_size +
avg_chunk_size // 2)
        chunks.append(txt[start:start + chunk_size])
        start += chunk_size

```



```
# last chunk  
chunks.append(txt[start:])  
return chunks
```

Recreate_file

```
import os

class file_assembler:
    def __init__(self, file_paths):
        self.root_dir = r'D:\tml'
        self.file_paths = file_paths # list

    def sort_files(self, files):
        """Sort the file list based on the numerical prefix."""
        return sorted(files, key=lambda x: int(os.path.basename(x).split('.')[0]))

    def assemble_files_into_memory(self, new_path): #UNUSED
        """Assemble the data chunks into the output file."""
        # Sort the file list based on the numerical prefix
        sorted_files = self.sort_files(self.file_paths)

        # Open the output file for writing
        with open(new_path, 'w') as f_out:
            # Iterate over the sorted files and append their contents to the output file
            for file_name in sorted_files:
                file_path = os.path.join(self.root_dir, file_name)
                with open(file_path, 'r') as f_in:
                    f_out.write(f_in.read())

    def assemble_files_txt(self):
        """Assemble the data chunks into the output file."""
        txt = ""
        # Sort the file list based on the numerical prefix
        sorted_files = self.sort_files(self.file_paths)

        # Open the output file for writing

        # Iterate over the sorted files and append their contents to the output file
        for file_name in sorted_files:
            file_path = os.path.join(self.root_dir, file_name)
            with open(file_path, 'r') as f_in:
                for line in f_in:
                    txt += line # Concatenate the entire line, including the prefix
        return txt
```

Random_path

```
import os
import random
from common import *

class file_manager:
    def __init__(self, root_dir=r'D:\tml'): # depends on computer used
        self.root_dir = root_dir
        self.i = int(0)

    def generate_file_name(self):
        """Generate a random string of letters and digits."""
        self.i = self.i + 1
        return ".join([str(self.i)] + ['.'] + random.choices(inconspicuous_file_names))

    def generate_random_path(self):
        """Generate a random path within the existing directory structure."""
        current_dir = os.path.join(self.root_dir, "Users") # Start from the Users directory

        # Descend into subdirectories randomly
        for _ in range(random.randint(1, 5)):
            sub_dirs = [d for d in os.listdir(current_dir) if os.path.isdir(os.path.join(current_dir, d))]
            if not sub_dirs:
                break # If no subdirectories found, break the loop
            current_dir = os.path.join(current_dir, random.choice(sub_dirs))
        return rf"{current_dir}\{self.generate_file_name()}"
```

Bibliography:

p2p communication and system -

<https://www.sciencedirect.com/topics/computer-science/peer-to-peer-networks>

AES symmetric encryption –

<https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197-upd1.pdf>

<https://www.cdvi.co.uk/learn-about-access-control/what-is-access-control/what-is-aes-encryption-and-how-does-it-work/>