VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

COMPUTER NETWORK ASSIGNMENT 1
Develop a network application

Instructor(s): Nguyễn Thành Nhân, *CSE - HCMUT*

Students: Trần Anh Đức - 2352271 *(Class CC06, **Leader**)*
Hồ Lâm Khánh Vy - 2353353 *(Class CC06)*
Trần Hoàng Khánh - 2352533 *(Class CC06)*
Huỳnh Quốc Đạt - 2352228 *(Class CC06)*

HO CHI MINH CITY, October 2025

Contents

1 Introduction

In the era of rapid information technology development, data sharing between computers has become an essential need in fields such as education, research, entertainment, and teamwork. To meet this demand, two fundamental architectures have been widely applied: the **Client–Server** and the **Pure Peer-to-Peer (P2P)** models. Each has its own advantages and limitations, which led to the development of a more balanced Hybrid P2P approach.

The Client–Server model is simple and easy to manage since a central server handles all data storage and coordination. It provides reliability and control but suffers from scalability issues — as the number of users increases, the server becomes a bottleneck, consuming bandwidth and creating a single point of failure. Conversely, the Pure P2P model distributes resources across all peers, allowing direct file sharing between users. This design improves scalability and resilience but lacks centralized coordination, making file discovery slower and more complex while reducing consistency and security.

The Hybrid P2P model combines the strengths of both architectures. A central server is retained solely for indexing and coordination, while file transfers occur directly between peers. This reduces server workload, avoids bottlenecks, and maintains efficient discovery and management. By blending centralized control with decentralized data exchange, the system achieves both scalability and stability.

For these reasons, the project adopts the Hybrid P2P model to build a practical and efficient file-sharing system. It allows clear coordination through a lightweight central server while preserving the performance and flexibility characteristic of P2P networks, making it ideal for small-scale academic and experimental environments.

2 Theoretical Background

This section provides the theoretical foundation for the design and implementation of the P2P file-sharing application. It covers the core concepts and technologies that underpin the system, including the hybrid peer-to-peer network model, the TCP/IP protocol suite, TCP socket programming, application-layer protocol design, and the use of multithreading to handle concurrent operations.

2.1 Network Models

The architecture of a network application dictates how its components communicate and share resources. Understanding the differences between the traditional Client-Server model and various Peer-to-Peer (P2P) models is fundamental to this project's design.

2.1.1 The Traditional Client-Server Model

The Client-Server model is a centralized architecture where roles are clearly defined.

- **Server:** A powerful, always-on host that stores and provides resources or services. It passively waits for requests from clients.
- **Client:** An end-user device that initiates communication by sending requests to the server for a specific resource or service. It does not directly communicate with other clients.

In this model, all communication and resource delivery flows through the central server. While this architecture offers centralized control and simplicity, it has significant drawbacks, including:

- **Scalability Bottleneck:** The server's bandwidth and processing power can be overwhelmed as the number of clients increases.
- **Single Point of Failure:** If the central server goes down, the entire network becomes inoperable.

2.1.2 Peer-to-Peer (P2P) Models

P2P models offer a decentralized alternative. In a P2P network, there is no concept of a dedicated, centralized server. Instead, individual nodes, called "peers," connect and communicate directly with one another. Each peer has equivalent capabilities and responsibilities, acting as both a client (requesting data) and a server (providing data). This distribution of tasks leads to high scalability and robustness.

2.1.3 The Hybrid P2P Model (Project Architecture)

This project employs a **Hybrid P2P Model**, which strategically combines the strengths of both the client-server and pure P2P architectures. It uses a central server for coordination but decentralizes the most resource-intensive tasks.

- **Centralized Coordination (Client-Server element):** A central server is maintained, but its role is limited to indexing and discovery. The server "keeps track of which clients are connected and storing what files". When a client wants to share a file, it informs the server via a `publish` command, but it does not upload the file itself. This creates a centralized, searchable directory of available files and the peers who hold them.

- **Decentralized Data Transfer (P2P element):** The actual file transfer is performed directly between peers. When a client needs a file, it sends a `fetch` request to the server. The server responds not with the file, but with a list of peers that have the file. The requesting client then establishes a direct TCP connection with one of those peers to download the file, completely bypassing the server for this heavy-lifting task.

This hybrid approach provides the "best of both worlds":

- **Efficient Discovery:** The centralized server makes finding files quick and reliable, avoiding the complex and potentially slow discovery mechanisms of pure P2P networks.
- **Scalability and Performance:** By offloading the file transfer tasks to the peers' direct connections, the server is protected from becoming a bandwidth bottleneck. The system's total capacity grows as more peers join the network.

2.2 The TCP/IP Protocol Suite

The TCP/IP protocol suite is the foundational set of communication protocols that govern how data is exchanged over the Internet and most private networks. The assignment explicitly requires that the file-sharing application be built using this protocol stack. It is conceptualized as a four-layer model, where each layer has a specific responsibility. For this project, the most relevant layers are the Application and Transport layers.

1. **Application Layer:** This is the highest layer in the model, where network applications and their protocols operate. For this project, the group must define its own custom application-layer protocol. This protocol will specify the exact format and sequence of messages for commands like `publish`, `fetch`, `discover`, and `ping`. Essentially, this involves designing the language that the clients and server will use to communicate.
2. **Transport Layer:** This layer is responsible for providing end-to-end communication services for applications. The two primary protocols at this layer are TCP and UDP.
   - **TCP (Transmission Control Protocol):** TCP is a connection-oriented protocol that guarantees reliable, ordered, and error-checked delivery of data. It establishes a connection before data is sent and ensures that every packet arrives intact and in the correct sequence. This reliability makes it the ideal choice for applications like file sharing, where the integrity of the transferred data is critical.
   - **UDP (User Datagram Protocol):** UDP is a simpler, connectionless protocol that offers no guarantees of delivery, order, or error checking. It is faster but less reliable, making it suitable for time-sensitive applications like video streaming or online gaming, where losing a small amount of data is acceptable.

   For this assignment, TCP is the appropriate choice to ensure that files are transferred between peers without corruption.
3. **Internet Layer:** The primary purpose of this layer is to move packets from a source to a destination across one or more networks (a process known as routing). The core protocol here is the **Internet Protocol (IP)**, which is responsible for addressing hosts with IP addresses and routing packets through the internetwork.
4. **Link Layer (or Network Interface Layer):** This is the lowest layer, which deals with the physical transmission of data over the network medium (e.g., Ethernet, Wi-Fi). It handles the conversion of IP packets into frames for transmission and is managed by the device's hardware and network drivers.

2.3 Socket Programming with TCP

To implement network communication, an application uses a Socket Application Programming Interface (API). A socket is an abstraction provided by the operating system that represents an endpoint for communication. It allows an application to send and receive data across a network without needing to manage the low-level complexities of the TCP/IP stack.

Since the application must ensure reliable file transfers, it will use the Transmission Control Protocol (TCP). TCP is a connection-oriented protocol, meaning a dedicated, reliable connection must be established between two endpoints before any data is exchanged. This process involves a distinct sequence of operations for the server (the listener) and the client (the initiator).

2.3.1 Server-Side TCP Socket Workflow

The server's role is to passively wait for and accept incoming connection requests from clients. This applies to both the central server waiting for clients and a peer waiting for another peer to download a file. The typical sequence is as follows:

1. `socket()`: The server first creates a socket endpoint. This call returns a file descriptor that uniquely identifies the socket.
2. `bind()`: The newly created socket is assigned a specific IP address and port number on the host machine. This is how clients know where to send connection requests.
3. `listen()`: The server places the socket into a listening state, telling the operating system it is ready to accept incoming connections.
4. `accept()`: This is a blocking call that waits for a client. When a connection request arrives, `accept()` creates a **new socket** dedicated to communicating with that specific client. The original listening socket remains open to accept other connections.

2.3.2 Client-Side TCP Socket Workflow

The client's role is to actively initiate a connection to a server. This applies when a client connects to the central server or when a peer (acting as a client) connects to another peer to fetch a file. The sequence is simpler:

1. `socket()`: The client creates a socket endpoint, just like the server.
2. `connect()`: The client attempts to establish a connection to the server's IP address and port. This is a blocking call that will either succeed or fail.

2.3.3 Data Exchange and Connection Termination

Once `accept()` on the server and `connect()` on the client have completed, a reliable, two-way communication channel is established. Both sides can then use functions like `send()` and `receive()` to exchange data streams. When communication is complete, both sides must call `close()` on their sockets to terminate the connection.

2.3.4 Relevance to the Project

This workflow is central to the project's architecture:

- The **central server** will implement the server-side workflow to manage connections from all clients.

- A **client application** will implement the client-side workflow to connect to the central server for tasks like `publish` and `fetch`.
- Crucially, to support direct peer-to-peer file transfers, each **client must also act as a server**. This means each client's code must be multithreaded, with one thread managing its main logic, and another thread running a server-side socket loop (`bind`, `listen`, `accept`) to handle incoming file download requests from other peers.

2.4 Application-Layer Protocol Design

The application layer is where the logic of the network application resides. While standard protocols like HTTP exist, they are often too general or inefficient for specialized tasks. The assignment requires the group to define its own custom communication protocols for each function. This means designing the "language" that the clients and server will use to communicate effectively over their TCP connections.

Designing a robust application-layer protocol involves defining three key areas:

2.4.1 Message Structure (The Format)

This defines the raw byte format of every message sent over the network. A well-defined structure is critical for the receiving end to parse and understand the sender's intent. A common approach is to divide messages into two parts:

- **Header:** A fixed-size section at the beginning of every message containing metadata. For this project, a header could include a command code (e.g., a number for `publish` or `fetch`) and the length of the payload.
- **Payload:** The variable-length data that follows the header. The content depends on the command. For a `publish Iname fname` command, the payload would contain the two filenames.

2.4.2 Message Types and Syntax (The Vocabulary)

This defines the set of valid commands and responses. Based on the assignment description, the protocol must support a clear request-response model for several operations:

- **Client-to-Server Commands:**
  - `publish Iname fname`: The protocol must specify how the client sends the local and repository filenames to the server.
  - `fetch fname`: The protocol must define how a client requests the location of peers who have a copy of a file.
- **Server-to-Client Responses:**
  - The protocol must define the server's response to a `fetch` command, which will be a list of peer identities (e.g., IP addresses and port numbers).
  - It should also include responses for success or failure (e.g., "OK" or "File Not Found").
- **Server-Side Administrative Commands:**
  - `discover hostname`: The protocol needs a format for the server to request the list of files from a specific client.

– `ping hostname`: The protocol needs a way to send a "ping" message and a format for the client's "pong" response to verify it is online.

2.4.3 Communication Flow and State (The Conversation)

This defines the sequence of messages required to complete a full operation. The `fetch` process is a perfect example of a multi-step conversation:

1. **Step 1 (Client → Server):** The client sends a `fetch` message to the server, formatted according to the protocol.
2. **Step 2 (Server → Client):** The server looks up the file and sends a response containing the list of peers with that file.
3. **Step 3 (Client → Peer):** The client parses the response and initiates a **new TCP connection** directly with a selected peer. The protocol for this peer-to-peer connection must also be defined, which could be as simple as the client sending the filename it wants, and the peer responding with the raw file data stream.

By defining these three components, a complete and unambiguous protocol is created that allows the distributed application components to work together seamlessly.

2.5 Multithreading

Multithreading is a programming and execution model that allows a single process to manage multiple independent threads of execution concurrently. Unlike separate processes, threads within the same process share resources like memory space, which makes them lightweight and efficient for performing parallel tasks within a single application.

In the context of network programming, multithreading is not only a performance optimization; it is a fundamental requirement for creating responsive and scalable applications. Its importance in this project is twofold, affecting both the central server and, most critically, the client peers.

2.5.1 Server Concurrency

The central server's primary job is to manage state and handle requests from many clients simultaneously. A single-threaded server can only handle one client at a time, creating a severe bottleneck. With a multithreaded design, the server's main thread can run a loop that only listens for and accepts new connections. As soon as a connection is accepted, the server can spawn a new worker thread to manage all communication with that specific client, while the main thread immediately goes back to listening for more connections. This allows the server to serve many clients in parallel.

2.5.2 Client Concurrency (A Core Requirement)

The most critical application of multithreading in this project is within the client application itself. The assignment explicitly states: **"Multiple clients could be downloading different files from a target client at a given point in time. This requires the client code to be multithreaded"**.

This requirement stems from the dual role of each client:

- **As a "Client":** It interacts with the local user through a command shell, initiating `publish` and `fetch` requests to the central server.
- **As a "Server":** It must be ready to accept direct connection requests from other peers and serve them files from its repository.

Without multithreading, these two roles would conflict. If a peer connected to the client to download a large file, the client's single thread would be occupied with that file transfer, causing the command-line interface to freeze.

A multithreaded client architecture solves this problem:

- **The Main Thread:** Can be dedicated to handling the user command-shell, remaining responsive to user input at all times.
- **A "Listener" Thread:** A separate thread acts as a server, creating a listening socket and waiting for incoming download requests from other peers.
- **"Worker" Threads:** When the listener thread accepts a connection, it spawns a *new* worker thread to handle that specific file transfer. This allows the client to serve files to multiple peers simultaneously.

3 System Requirements

3.1 Functional Requirements

Registration and Session Maintenance with the Server (Directory Service): When a client starts, it connects to the server and provides information including its `hostname`, listening `port` (for P2P connections), and online status. The server records and updates the list of active clients.

Client command `publish lname fname`: Allows the client to publish a local file (`lname`) to the system under the logical name `fname`. After publishing, the client sends this information to the server to update the global file index.

Client command `fetch fname`: When a client needs to download a file that does not exist in its local repository, it sends a request to the server to obtain a list of peers storing the file `fname`. The client then selects a suitable peer and downloads the file directly from that peer, without going through the server.

Server command `discover hostname`: The server returns the list of files that a specific client (identified by `hostname`) has published.

Server command `ping hostname`: The server checks the availability (online/offline status) of the specified client.

Concurrent File Serving: Each client can handle multiple simultaneous download requests from other peers by using a multithreaded mechanism when acting as a "source node".

Command-line Interface: Both the client and the server provide a simple shell interface that supports the commands `publish`, `fetch`, `discover`, and `ping`, as well as basic management operations.

3.2 Non-functional Requirements

Performance: The server functions solely as a directory service — managing file index information and not relaying any file data. The time from when a client issues the `fetch` command until the first byte of data is received (Time-To-First-Byte) must be less than 1–2 seconds in a local area network (LAN) environment.

Scalability: The system supports the server handling N concurrent client connections. Each client can handle up to k simultaneous upload and download sessions, where k is a configurable parameter to ensure performance and system scalability.

Availability & Reliability: The server periodically performs ping operations to remove inactive peers. Clients implement a retry mechanism to ensure stability when a source peer becomes unavailable during a transfer.

Concurrency & Thread-safety: When acting as a "source" node, a client can serve multiple download requests concurrently. The system uses a thread-pool or asynchronous I/O model to handle concurrent connections and applies thread-safety mechanisms to prevent data races,

ensuring stable operation in multithreaded environments.

Compatibility & Deployment: The application operates over the standard TCP/IP protocol stack, supports multiple platforms (Windows/Mac/Linux), allows flexible port configuration, and does not require root privileges.

Basic Security: The system only serves files that have been explicitly published. It implements path traversal protection to prevent unauthorized access to the file system and applies size and bandwidth limits to maintain system stability.

3.3 Semantic Constraints

- Each client must register with a unique pair of {`hostname`, `port`}. If a duplicate is detected, the previous record will be immediately invalidated and replaced.
- The server is not allowed to participate in any file transfer stream. If any file traffic is detected passing through the server, the connection will be terminated.
- Only valid files within the repository directory can be published. If the file path goes beyond the allowed directory scope or the file does not exist, the publish request will be rejected.
- The total number of simultaneous upload and download sessions per client must not exceed `k`. If the limit is exceeded, new requests will be either rejected or queued.
- If the Time-To-First-Byte (TTFB) during a `fetch` operation exceeds 2 seconds, the client must automatically switch to another peer or report an error.

3.4 Out-Scope

To maintain clarity and focus within the defined scope, the following aspects are excluded from this project:

- **Encryption and security:** No SSL/TLS, authentication, or access control; communication relies on plain TCP in a trusted LAN.
- **User management:** No login system, user profiles, or role permissions; clients are identified only by hostname and port.
- **Content control:** The system does not scan, verify, or filter shared files for integrity or legality.
- **Internet deployment:** The model targets LAN use only, without NAT traversal, firewall configuration, or WAN optimization.
- **Transfer reliability:** No resume, retry, or adaptive bandwidth management for interrupted or large file transfers.
- **Versioning and synchronization:** The directory server does not handle file version conflicts or automatic updates between peers.
- **High-availability server:** Single-server architecture; no replication, load-balancing, or failover mechanism.

- **Monitoring and administration:** No dashboards or remote management tools beyond simple console logs.
- **User capacity limitation:** The system is designed for small-scale LAN demonstrations, supporting up to around 30 concurrent clients. Large-scale or stress-tested operation is out of scope.

4 Application Feature

4.1 Server Feature

Manage client connection information: The server stores the hostname, IP address, port, and activity status of all clients to accurately track the nodes that are online in the system.

Store the list of published files: When a client publishes a file, the server updates this list to support query requests from other clients.

Handle `publish` and `fetch` requests: When receiving a `publish` command, the server adds the file information to the system. When receiving a `fetch` command, the server searches for peers that hold the file and returns the connection information.

Provide `discover` and `ping` functions: The `discover` function allows retrieving the file list of a specific client, while the `ping` function checks the client's activity status.

4.2 Client Feature

Send file information to the server: When new files are added or changes occur, the client uses the `publish` command to notify the server of the files available for sharing.

Send file retrieval requests: When a file is needed, the client uses the `fetch` command to ask the server for information about the peer that holds the file.

Establish peer-to-peer (P2P) connections: After receiving the information from the server, the client connects directly to the peer to download or share the file, without going through the server.

Act in dual roles: The client can act as both a publisher (file provider) and a downloader (file requester), depending on the use case.

5 Diagram

5.1 Use Case Diagram

The P2P File-Sharing Application system is designed based on use cases that clearly define the roles of two main actors: Server and Client.

The server is responsible for managing the list of active clients, maintaining information about published files, and supporting functions such as Ping to check connection status, Discover to retrieve the file list of a specific client, and Refresh to update the list of online clients. Meanwhile, the client performs functions such as Connect to establish a connection with the server, Publish to send information about local files to the system, Fetch to request and download files from other peers, and Disconnect to terminate the connection when inactive.

These use cases are logically related, where Publish, Fetch, and Disconnect act as extensions of Connect, ensuring a consistent and coherent workflow between the server and clients in the peer-to-peer file-sharing model.

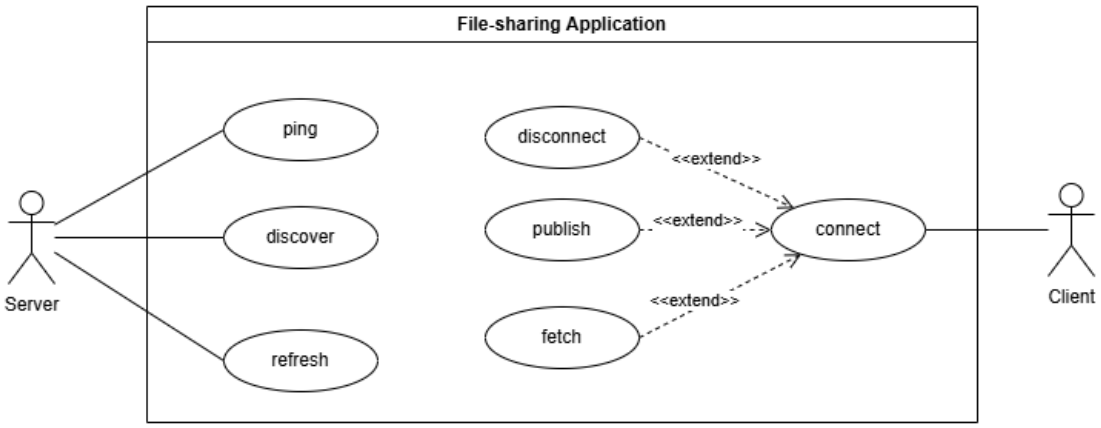

Figure 1: Usecase diagram for the system

5.2 Class Diagram

This class diagram represents a system with three main components: Server, Client, and the protocol utility. The Server manages connected clients, stores file metadata, and coordinates publish/fetch requests. The Client connects to the server to share or retrieve files and can exchange files directly with other clients. The protocol module provides common functions for sending and receiving socket messages. A single Server manages multiple Clients, and both use the protocol for network communication.

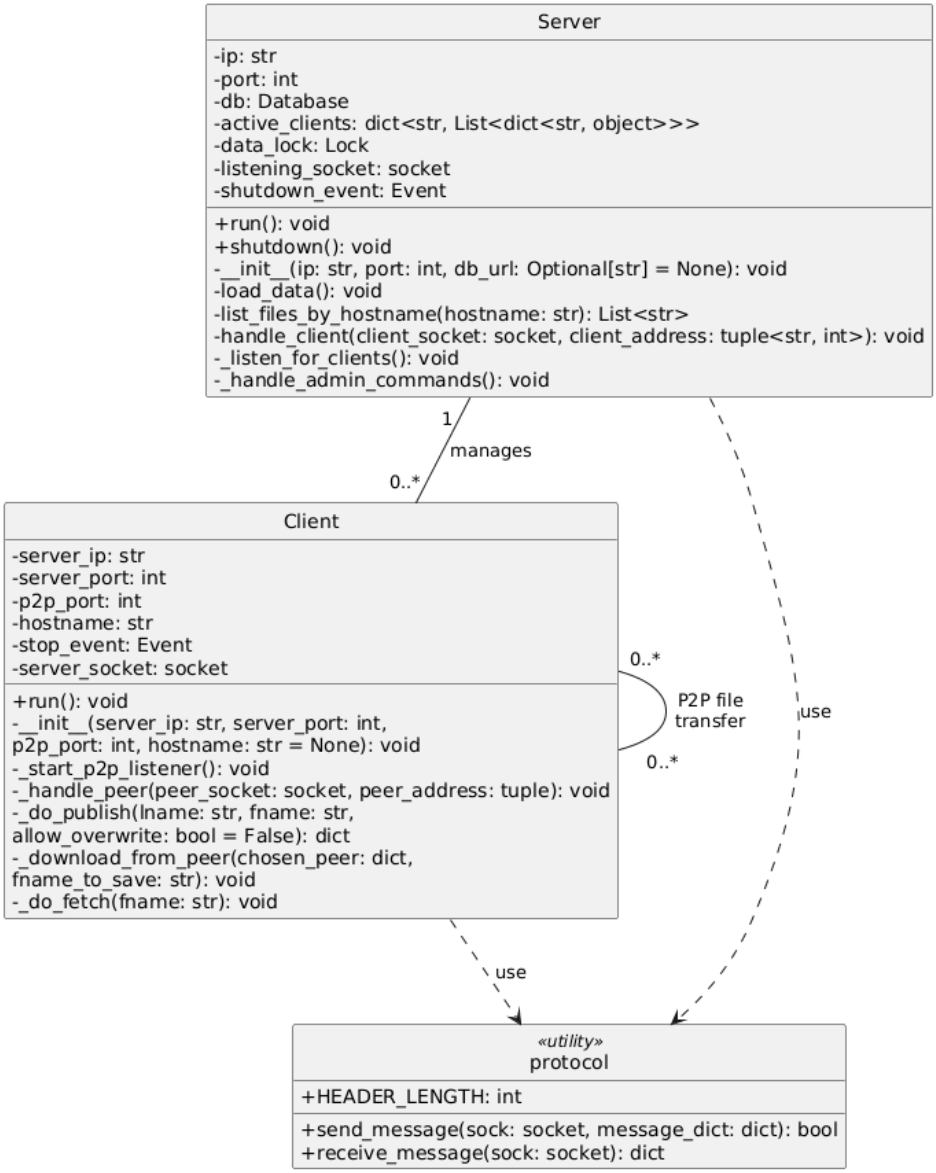

Figure 2: Class diagram for the system

5.3 Sequence Diagram

The sequence diagram illustrates how the `Tkinter` front ends orchestrate *publish*, *fetch*, *discover*, and *ping* workflows over the shared P2P backend.

When a client browses for a file and alias in `client_ui.py`, the UI requests the application logic (implemented in `client.py`/`server.py`) to publish the corresponding metadata. The system then reads or updates the `server_data.json` file, persists the modification, and returns a success acknowledgement to the UI.

For the *fetch* process, the client UI submits a logical filename to the system. The backend looks up matching peers in the registry, retrieves their connection details, and—after the user selects a destination—initiates a file streaming process while logging transfer progress in real time.

On the server side, the administrator operates through `server_ui.py` to execute *discovery* and *ping* commands. The UI forwards these requests to the backend system, which enumerates published files for a given hostname or inspects the `active_clients` set. The resulting lists are then reported back to the operator, ensuring full visibility of the peer

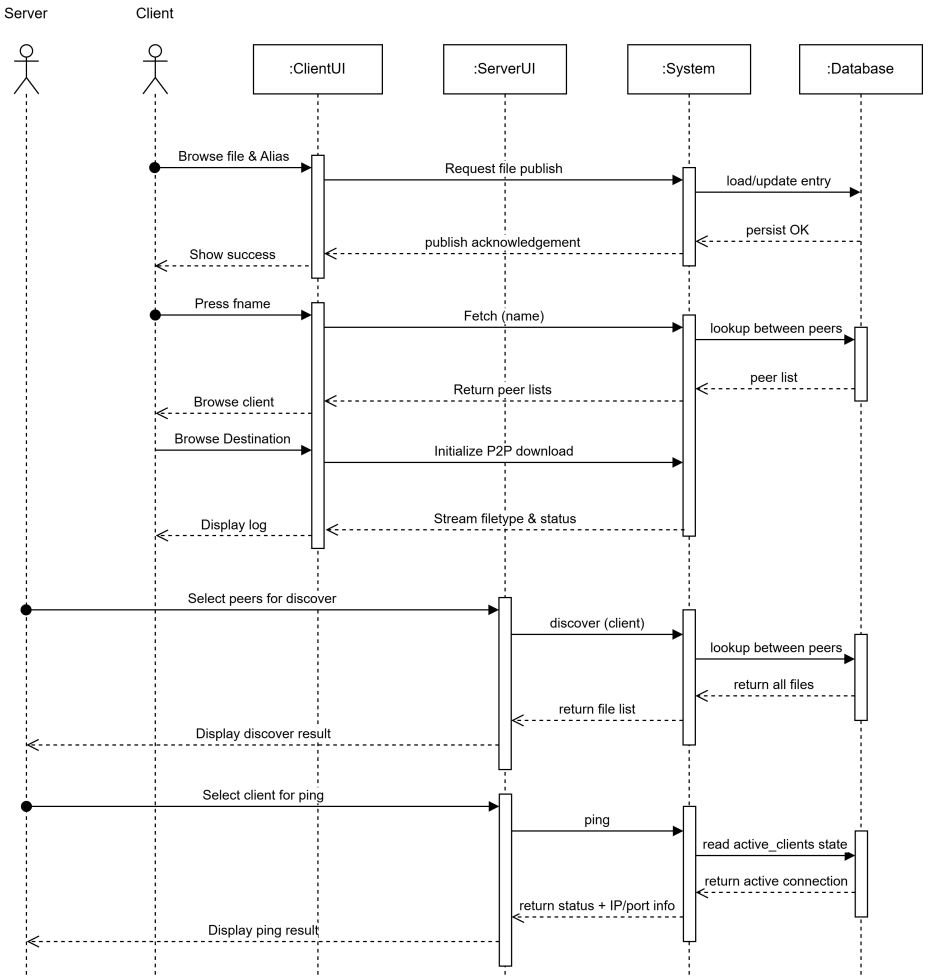

Figure 3: Sequence diagram for the system

6 Implementation

6.1 Server

The server in the peer-to-peer (P2P) file sharing architecture functions as the **metadata registry** that maintains peer information and published file descriptors. Rather than transferring files directly, it coordinates discovery between clients through a persistent **PostgreSQL** database. Its implementation emphasizes concurrency, reliability, and data integrity through atomic database operations and thread-safe structures.

Core Design

The server maintains three key components:

- **Network Layer** – Listens for client connections on a specified TCP port.
- **Session Manager** – Spawns a dedicated thread for each client to ensure concurrent handling.
- **Metadata Manager** – Interfaces with the PostgreSQL database to persist file publication records.

The server instance initializes shared state including the socket listener, an in-memory `active_clients` dictionary for connected peers, and a `Database` object used to store metadata. Thread safety is maintained using Python's `threading.Lock`.

Server Startup and Listening

The startup sequence is encapsulated in the `run()` method, which establishes the listening socket, validates database connection, and launches the client listener thread.

```
def run(self) -> None:
    self.load_data()  # preload DB records
    self.listening_socket = socket.socket(socket.AF_INET, socket.
    SOCK_STREAM)
    self.listening_socket.bind((self.ip, self.port))
    self.listening_socket.listen()
    threading.Thread(target=self._listen_for_clients, daemon=True).start
    ()
    self._handle_admin_commands()
```

Listing 1: Server startup sequence

This design decouples **I/O operations** (socket listening) from **administrative commands**, allowing the server to continue operating while responding to control inputs such as `discover`, `ping`, or `exit`.

Client Connection Handling

Once active, the listener accepts client sockets and spawns threads to handle them independently:

```
def _listen_for_clients(self):
    while not self.shutdown_event.is_set():
        conn, addr = self.listening_socket.accept()
        threading.Thread(target=self.handle_client,
                         args=(conn, addr), daemon=True).start()
```

Listing 2: Client connection listener

Each connection thread begins by awaiting a handshake message:

```
{"action": "hello", "hostname": "peer1", "p2p_port": 5000}
```

Listing 3: Handshake initialization

This exchange registers the peer's identity and enables the server to maintain a session-specific mapping:

```
self.active_clients[hostname].append({"ip": ip, "port": p2p_port})
```

This explicit registration ensures that every published or fetched file can be traced back to a known peer and connection context.

Core Request Handlers

The server recognizes two primary actions: `publish` and `fetch`. Each message is parsed from JSON after being received through the framing protocol.

(a) File Publication

A client advertises a shared file using:

```
{
  "action": "publish",
  "lname": "C:/shared/sample.pdf",
  "fname": "sample.pdf",
  "file_size": 102400,
  "last_modified": "2025-11-06T13:45:00Z",
  "allow_overwrite": true
}
```

Listing 4: File publication message

The handler checks for conflicts and updates the database accordingly:

```
entry = {
    "hostname": client_hostname,
    "ip": client_ip,
    "port": client_p2p_port,
    "fname": fname,
    "lname": lname,
    "file_size": message["file_size"],
    "last_modified": message["last_modified"],
}
self.db.register_file(entry)
```

Listing 5: Database update for published file

If an existing entry conflicts (same alias but different file path), the server responds with `"status": "conflict"`. Otherwise, the entry is upserted via PostgreSQL's **ON CONFLICT** mechanism, ensuring atomic updates without duplicate rows.

(b) File Discovery

A peer looking for available sources sends:

```
{"action": "fetch", "fname": "sample.pdf"}
```

Listing 6: Fetch request message

The server queries for all peers currently offering that filename:

```
peers = self.db.list_peers_for_file(fname)
send_message(conn, {"status": "success", "peer_list": peers})
```

Listing 7: Peer lookup and response

This simple request-response pattern implements **decentralized discovery**, allowing the requesting peer to connect directly to the appropriate hosts without further server involvement.

Session Termination and Cleanup

When a client disconnects, the server removes its records both from memory and the database:

```
self.db.delete_entries_for_peer(hostname, ip, p2p_port)
del self.active_clients[hostname]
```

Listing 8: Session cleanup

This design ensures the metadata remains accurate, preventing unavailable peers from being advertised to others.

Administrative Interface

An interactive command loop allows runtime inspection of the system:
`discover <hostname>` — list all files registered by a peer
`ping <hostname>` — check online status
`exit` — terminate the server

6.2 Client

The client component represents the active peer in the peer-to-peer (P2P) file sharing system. Each client can both **publish** its local files to the network and **fetch** files from other peers through the central server's metadata service. The implementation focuses on usability, concurrency, and efficient communication with both the central server and other peers.

Core Design

The client performs dual roles:

- **Server Communication Layer** – Manages TCP connection with the metadata server to register, update, and fetch file information.
- **Peer Service Layer** – Hosts a local listener to serve file download requests from other peers.
- **User Interface Layer** – Provides a graphical control interface (in `client_ui.py`) allowing users to browse shared files and initiate transfers.

Each client maintains a persistent TCP connection with the server for control messages, and a separate socket for file transfers between peers. This architecture ensures that metadata exchange and file data transmission occur independently, minimizing contention and blocking behavior.

Client Initialization and Server Connection

At startup, the client connects to the central metadata server, performs an initial handshake, and registers itself. This is implemented in the `connect_to_server()` method.

```
def connect_to_server(self):
    self.server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM
    )
    self.server_socket.connect((self.server_ip, self.server_port))
    send_message(self.server_socket, {
        "action": "hello",
        "hostname": socket.gethostname(),
        "p2p_port": self.p2p_port
    })
```

Listing 9: Client connection to the server

The message follows the same framing format defined in `protocol.py`, ensuring compatibility with the server's length-prefixed JSON protocol. Upon success, the server acknowledges the handshake and maintains this socket for all further communication.

Publishing a File

When the user selects a file to share, the client prepares metadata and sends a `publish` request to the server. The implementation constructs a JSON payload that includes file size, path, and last modification timestamp.

```
def publish_file(self, local_path, shared_name):
    file_stat = os.stat(local_path)
    message = {
        "action": "publish",
        "lname": local_path,
        "fname": shared_name,
        "file_size": file_stat.st_size,
        "last_modified": datetime.utcfromtimestamp(file_stat.st_mtime).
    isoformat(),
        "allow_overwrite": True
    }
    send_message(self.server_socket, message)
    response = receive_message(self.server_socket)
    return response.get("status")
```

Listing 10: Client publishing a file

If successful, the server updates its metadata registry so other peers can discover the file. The server responds with status messages such as `success`, `conflict`, or `unchanged`, which are then reflected in the client's graphical user interface.

Fetching Files from Other Peers

To download a file, the client first queries the central server for all available peers offering that file.

```
def fetch_peers(self, file_name):
    send_message(self.server_socket, {"action": "fetch", "fname":
    file_name})
    response = receive_message(self.server_socket)
    return response.get("peer_list", [])
```

Listing 11: File discovery request

Once a list of available peers is returned, the user can choose a source. The client then opens a separate peer-to-peer TCP connection to directly retrieve the file.

```
def download_file(self, peer_ip, peer_port, remote_name, save_path):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect((peer_ip, peer_port))
        send_message(s, {"action": "download", "fname": remote_name})
        with open(save_path, "wb") as file_out:
            while True:
                data = s.recv(4096)
                if not data:
                    break
                file_out.write(data)
```

Listing 12: Peer-to-peer file download

This architecture separates metadata management (through the central server) from file content transfer (through direct peer-to-peer channels). As a result, the system scales efficiently while avoiding server-side data congestion.

Local Peer Service

Each client also runs a lightweight background listener that allows other peers to download its shared files. This listener runs in a separate thread and accepts incoming TCP connections.

```
def start_peer_service(self):
    listener = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    listener.bind(("0.0.0.0", self.p2p_port))
    listener.listen()
    while True:
        conn, addr = listener.accept()
        threading.Thread(target=self.handle_peer_request, args=(conn,
    addr), daemon=True).start()
```

Listing 13: Peer listener service

The handler validates incoming requests and streams the requested file back to the peer using chunked transmission to avoid memory overload. This dual behavior — acting as both client and server — defines the peer nature of the system.

6.3 Protocol

The protocol layer, implemented in `protocol.py`, defines the standardized communication procedure between the server and clients in the P2P file sharing architecture. It provides a **message framing mechanism** that guarantees each JSON message transmitted through a TCP socket is received completely and accurately, even when the data is split across multiple network packets. Without this layer, both sides would face ambiguity in determining where one message ends and the next begins.

Design Principle

In typical TCP communication, the stream-oriented nature of sockets means that data may arrive in variable-length chunks. Therefore, relying solely on `recv()` calls can result in partial messages or message concatenation. To solve this, the system uses a custom **length-prefixed JSON protocol**, ensuring deterministic message boundaries and human-readable contents.

Message Framing Scheme

Every message follows the same structure:

- The first 4 bytes (fixed-length header) specify the size of the message body in bytes, encoded as an unsigned integer in big-endian format.
- The following segment contains a UTF-8 encoded JSON string that represents the actual message payload.

This approach combines the advantages of structured data (JSON) with the reliability of explicit byte-length framing. It guarantees that each receiver knows exactly how many bytes to read before decoding.

```
HEADER_LENGTH = 4

def send_message(sock, message_dict):
    message_bytes = json.dumps(message_dict).encode("utf-8")
    header = struct.pack("!I", len(message_bytes))
    sock.sendall(header + message_bytes)
```

Listing 14: Send function in protocol.py

Here, `json.dumps()` serializes a Python dictionary into a JSON string, while `struct.pack("!I", len(...))` creates a 4-byte header that encodes the total length of the JSON message using network byte order. This ensures cross-platform compatibility, allowing messages to be correctly interpreted regardless of the underlying system architecture.

Message Reception

On the receiving side, the protocol first reads the 4-byte header to determine the length of the message, and then continues reading from the socket until the full payload is obtained.

```
def receive_message(sock):
    header = sock.recv(HEADER_LENGTH)
    if not header:
        return None
    message_length = struct.unpack("!I", header)[0]

    data = b""
    while len(data) < message_length:
        chunk = sock.recv(message_length - len(data))
        if not chunk:
            return None
        data += chunk
    return json.loads(data.decode("utf-8"))
```

Listing 15: Receive function in protocol.py

This ensures that incomplete messages are never decoded prematurely. The `while` loop continues until exactly `message_length` bytes are received, after which the payload is safely converted back into a Python dictionary.

Error Handling and Reliability

To ensure robustness, the functions in `protocol.py` incorporate graceful failure handling:

- If the socket is closed or no data arrives, the function returns `None` instead of throwing an uncaught exception.
- Partial transmissions are handled automatically by looping until the entire payload has been received.

- All messages are validated for correct JSON structure before being passed to higher layers.

This prevents desynchronization between client and server sessions and minimizes crashes during unstable network conditions.

Integration with Server and Client

Both the `server.py` and `client.py` modules rely on this protocol to exchange structured control messages. Each major operation—such as peer handshake, file publication, or file discovery—uses `send_message()` and `receive_message()` to ensure consistent communication.

Example Workflow:

1. **Handshake:** When a new peer connects, the client sends a greeting message:

```
{"action": "hello", "hostname": "peer1", "p2p_port": 5000}
```

   The server decodes it using `receive_message()`, registers the peer, and acknowledges readiness.

2. **File Publication:** The client then advertises a file:

```
{"action": "publish", "fname": "report.pdf", "lname": "C:/shared/
report.pdf"}
```

   The server parses this JSON payload, updates the database, and responds:

```
{"status": "success", "message": "File registered successfully"}
```

3. **File Discovery:** Another peer requests available sources:

```
{"action": "fetch", "fname": "report.pdf"}
```

   The server replies with a list of active peers providing that file:

```
{"status": "success", "peer_list": [{"ip": "192.168.1.2", "port":
 5050}]}
```

Throughout all these interactions, the protocol layer abstracts low-level socket management, allowing developers to work purely with high-level Python dictionaries and ensuring both readability and reliability.

Advantages of the Design

- **Consistency:** Every message, regardless of type or origin, uses the same framing structure.
- **Clarity:** JSON-based messages are human-readable and easy to log or debug.
- **Portability:** The use of UTF-8 encoding and big-endian byte order makes the protocol independent of machine architecture.
- **Scalability:** New message types (e.g., `remove`, `update`, `ping`) can be easily added without altering the framing logic.

6.4 Database

The system employs a **PostgreSQL** database as the central metadata store, responsible for maintaining consistent records of all shared files and active peers. Instead of distributing file descriptors across clients, the server delegates all persistent state management to this relational backend. This design provides atomicity, durability, and efficient querying for file discovery requests.

Deployment Environment

To ensure reproducibility and portability, the database runs as a **Docker container**. The development container is created from the official lightweight `postgres:16-alpine` image using the following configuration:

```
docker run -d \
  -p 5432:5432 \
  -e POSTGRES_DB=p2p_metadata \
  -e POSTGRES_USER=p2p_server \
  -e POSTGRES_PASSWORD=p2p_pass \
  -v postgres_data:/var/lib/postgresql/data \
  --name p2p postgres:16-alpine
```

Listing 16: Docker container initialization for PostgreSQL

The environment variables initialize the database with a dedicated user and password, creating a database named `p2p_metadata`. Using Docker abstracts away manual setup, allowing the system to run identically on any host without dependency conflicts. The data volume `postgres_data` ensures persistence even if the container restarts.

Schema Design

The schema is deliberately minimal, containing a single table that serves as the metadata registry for all published files:

```
CREATE TABLE client_files (
    id SERIAL PRIMARY KEY,
    lname TEXT NOT NULL,          -- local path on peer
    fname TEXT NOT NULL,          -- shared file name
    extension TEXT NOT NULL,      -- file extension (without dot)
    hostname TEXT NOT NULL,
    address INET NOT NULL,
    UNIQUE(address, fname, hostname)
);
```

Listing 17: Table schema for peer file metadata

Each record uniquely identifies a file made available by a specific peer. The `INET` type is used for the `address` column to natively represent IP addresses, allowing efficient range queries and validation at the database level. The composite `UNIQUE` constraint prevents duplicate publications of the same file from the same peer.

Database Access Layer

All database interactions are encapsulated in the `database.py` module through the **psycopg2** driver. The server creates a `Database` object at startup, which manages the connection lifecycle and exposes clean methods for CRUD operations.

A typical connection is established as follows:

```
import psycopg2

self.conn = psycopg2.connect(
    dbname="p2p_metadata",
    user="p2p_server",
    password="p2p_pass",
    host="localhost",
    port=5432
)
self.cursor = self.conn.cursor()
```

Listing 18: Database connection using psycopg2

This connection remains persistent throughout the runtime of the server process. To maintain reliability, all queries are wrapped with commit calls to ensure changes are flushed atomically:

```
def register_file(self, entry):
    self.cursor.execute("""
        INSERT INTO client_files (lname, fname, extension, hostname,
    address)
        VALUES (%s, %s, %s, %s, %s)
        ON CONFLICT (address, fname, hostname)
        DO UPDATE SET lname = EXCLUDED.lname;
    """, (entry["lname"], entry["fname"], entry["extension"],
          entry["hostname"], entry["ip"]))
    self.conn.commit()
```

Listing 19: Registering a file entry in the database

The **ON CONFLICT** clause ensures that repeated file publications by the same peer update existing rows instead of creating duplicates, providing idempotent behavior for the `publish` operation. Similarly, file unpublishing or peer disconnection triggers cleanup queries that remove all related records.

Integration with Server Logic

The `server.py` backend relies on this database layer to synchronize in-memory state with persistent metadata. Each time a client sends a `publish`, `fetch`, or disconnection message, the server invokes the appropriate method:

- `register_file()` – adds or updates a peer's file entry.
- `list_peers_for_file(fname)` – retrieves all peers hosting a specific filename.
- `delete_entries_for_peer()` – cleans up records upon peer disconnection.

This layered approach isolates database logic from the networking code, simplifying debugging and ensuring that database failures never block the main server loop. Errors are handled gracefully, and failed transactions are rolled back automatically by psycopg2's internal context management.

Visualization and Debugging

During development, the database was inspected and queried using **JetBrains DataSpell**. The tool connects directly to the Docker container through the same port mapping, allowing real-time visualization of tables, schema, and query results. This visual inspection facilitated verification of publish/fetch correctness and ensured metadata synchronization between the server and PostgreSQL.

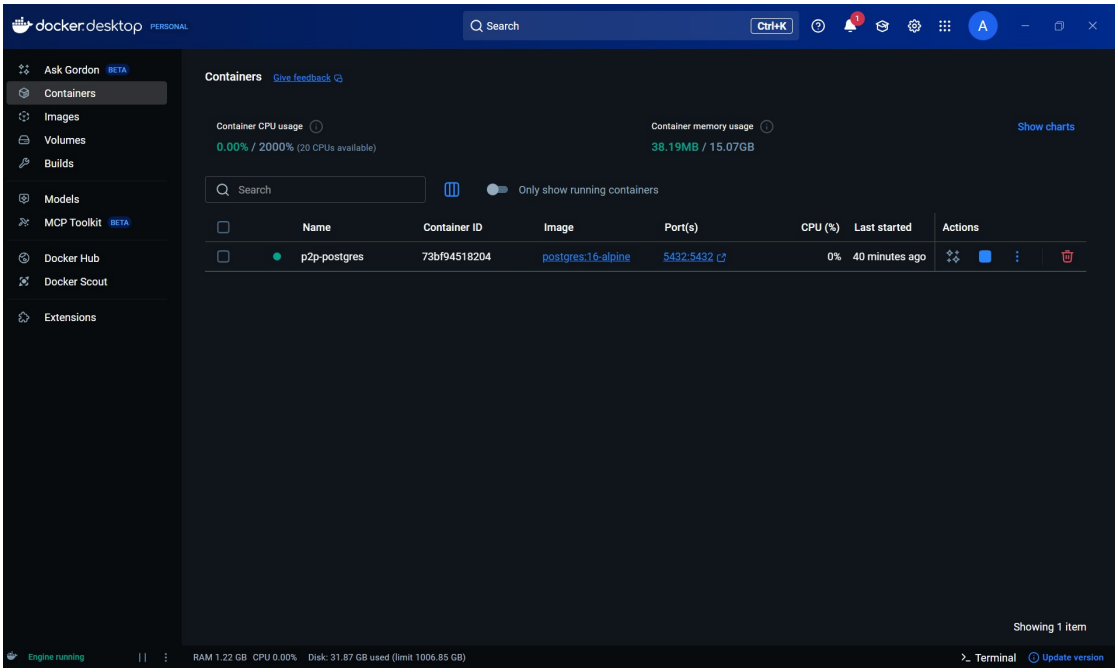

Figure 4: PostgreSQL container successfully deployed and running inside Docker.

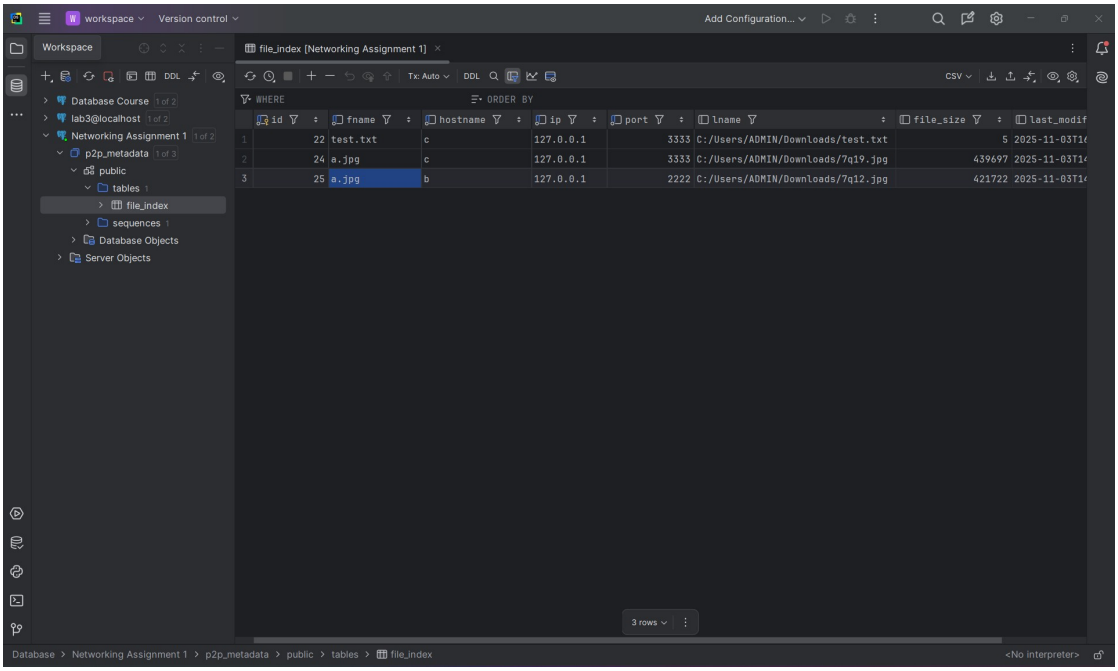

Figure 5: Visualization of the `client_files` table in DataSpell, showing live metadata entries.

6.5 Graphical User Interface (GUI)

The application offers two **Tkinter-based interfaces** that wrap the command-line client and server into accessible desktop tools for demonstration purposes. Both share a consistent pastel visual theme, employ thread-safe updates for live status display, and redirect backend log outputs into GUI elements for real-time monitoring. This graphical abstraction improves usability, allowing users to operate and observe the P2P file sharing system without interacting directly with terminal commands.

Client UI

The client interface, implemented in `client_ui.py`, serves as the user-facing control center for managing all operations of a peer node. It provides a high-level interface to connect to the metadata server, publish files, browse available content, and initiate peer-to-peer downloads.

Core Functionalities:

- **Server Connection Management** – Establishes or terminates a session with the central metadata server using the client's `connect_to_server()` and `disconnect()` methods.
- **File Publishing Interface** – Allows users to select local files, automatically extract metadata such as file size and modification date, and publish them through a single button press.
- **File Discovery and Download** – Provides a search bar and listbox to query available files and initiate downloads from selected peers.
- **Logging and Feedback Panel** – Redirects output logs from the backend to a scrollable Tkinter text widget, offering a live feed of system messages and responses.

The GUI integrates tightly with the core client logic by invoking network operations as background threads, ensuring that the user interface remains responsive during active downloads or uploads.

```
def publish_file_thread(self, path, name):
    thread = threading.Thread(target=self.client.publish_file,
                              args=(path, name), daemon=True)
    thread.start()
```

Listing 20: Launching network task from client UI

This pattern prevents the Tkinter event loop from freezing during long-running network transactions. Additionally, the client UI dynamically updates progress and connection indicators based on server responses. The separation of GUI and logic enhances maintainability and allows the networking backend to be reused independently from the interface layer.

Interface Layout Overview:

- Top frame – Connection controls (server IP, port, connect/disconnect buttons).
- Middle frame – File browser, publish button, and query bar.
- Bottom frame – Output log, peer list, and download controls.

By simplifying interaction, the client interface turns a multi-step command-line workflow into a single interactive window, aligning the project with modern GUI-based application standards.

Server UI

The server-side interface, implemented in `server_ui.py`, provides a visual administration tool for managing and monitoring the metadata server in real time. It exposes internal server state, active connections, and file registry data without requiring manual terminal inspection.

Key Features:

- **Server Lifecycle Control** – Enables one-click startup and shutdown of the backend server process.
- **Active Peer List** – Continuously updates a table showing connected clients (hostname, IP, and P2P port).
- **File Registry Display** – Fetches and renders file metadata stored in the PostgreSQL database.
- **Event Logging and Status Feed** – Displays internal log messages, such as new connections, file publications, and disconnections, in a synchronized text console.

The GUI launches the backend logic in a separate thread to maintain asynchronous operation:

```
def start_server_thread(self):
    thread = threading.Thread(target=self.server.run, daemon=True)
    thread.start()
```

Listing 21: Asynchronous server thread in GUI

When the user presses the "Start Server" button, the interface spawns this background thread to execute the same `run()` method defined in `server.py`. This ensures the GUI remains fully interactive even as the server listens for incoming peer connections and database operations.

Graphical Layout and Synchronization:

- The top frame displays control buttons (Start, Stop, Refresh).
- The central frame lists currently connected peers and their metadata.
- The bottom frame hosts the system log window for event tracing.

A synchronization timer periodically polls the backend for updates, ensuring that displayed data such as the peer list and file table remains accurate. The GUI automatically refreshes upon connection or disconnection events triggered by the backend thread.

Advantages of the GUI Integration:

- Provides visual transparency for teaching and debugging network operations.
- Simplifies the demonstration process during evaluation sessions.
- Separates presentation (Tkinter) from logic (network server), improving modularity.

7 Application Demo

The demonstration covers the entire interaction flow, from connection establishment to file publishing and discovery, while also highlighting edge cases and recovery behaviors optimized during development.

User Interface and UX Optimization

Both the client and server applications adopt a **pastel-themed interface** with visually consistent widgets, color-coded log areas, and intuitive control buttons. Every action, such as "Start Server," "Connect," "Publish," or "Fetch," immediately updates the corresponding visual log area, giving the user clear feedback on the system's internal state.

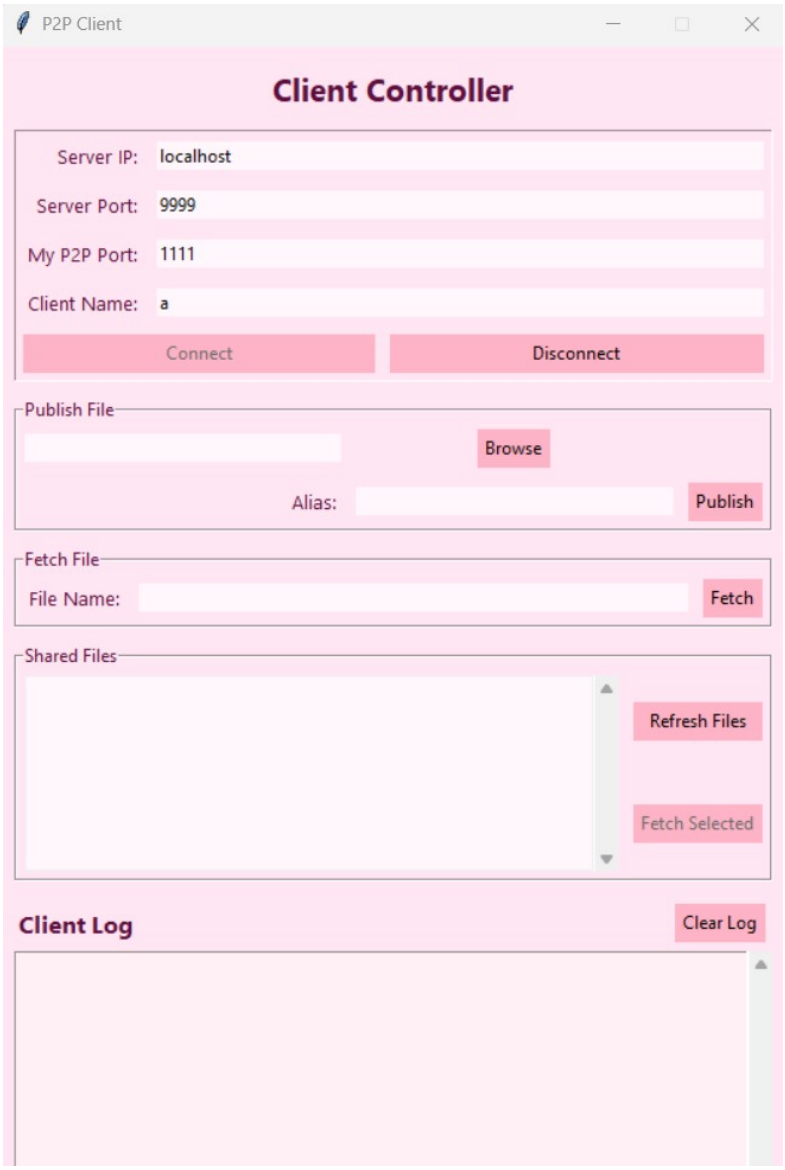

Figure 6: Client interface displaying published and fetched files.

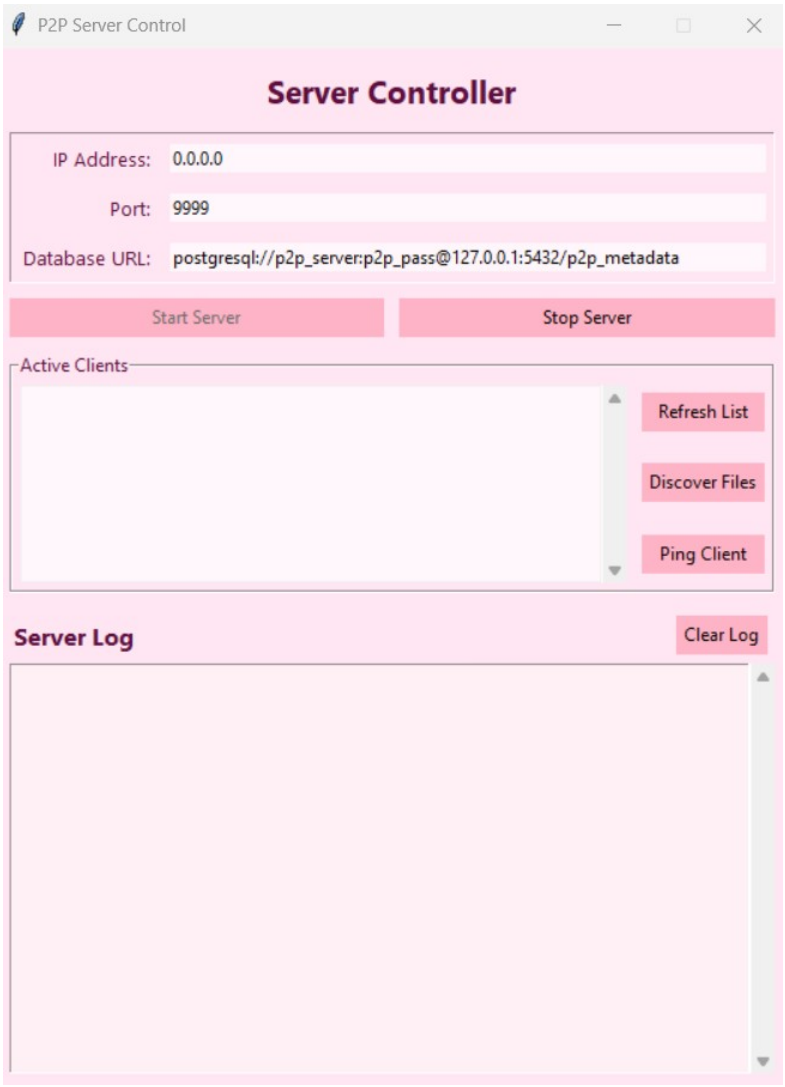

Figure 7: Server interface showing active client connections and live logs.

Connecting to the Server

When the user launches the client UI, they are prompted to enter:

- `Server IP and Port` – where the server is listening.
- `Hostname` – a unique identifier for the client.

Before connection, both programs must be launched from the terminal or IDE environment. The following commands start each component of the system:

```
python server.py
```

Listing 22: Command to launch the P2P server

This command initializes the PostgreSQL-backed server, opens the Tkinter interface, and begins listening for incoming peer connections on the configured port.

```
python client.py <port_number> <client_name>
```

Listing 23: Command to launch the P2P client

Upon connecting, the client automatically performs a handshake by sending a `hello` message (hostname + peer port). The server verifies this and acknowledges with a `success` response, registering the peer in the `active_clients` dictionary and PostgreSQL metadata.

```
{"action": "hello", "hostname": "peer1", "p2p_port": 5000}
{"status": "success", "message": "Hello from server!"}
```

Listing 24: Handshake exchange

File Publication Flow

Users can publish any file by selecting it from their local machine. The client automatically extracts the file's metadata (name, path, size, modification time) and adds the appropriate extension if missing. The following message is sent to the server:

```
{
  "action": "publish",
  "fname": "report.pdf",
  "lname": "C:/shared/report.pdf",
  "file_size": 102400,
  "last_modified": "2025-11-06T13:45:00Z",
  "allow_overwrite": true
}
```

Listing 25: Publish request message

The server cross-verifies the publication:

- If the same alias already exists with identical metadata, the server returns `status: unchanged`.
- If the same alias is found but with a different local path, the server warns the client with a **conflict message**, asking whether to overwrite.
- If overwrite is confirmed, the existing entry is updated atomically via `ON CONFLICT` logic.

Edge Case Handling during Publish

- When the same client publishes two files with identical aliases, the UI triggers a dialog: *"File alias already exists — overwrite?"*.
- If confirmed, the duplicate is replaced, and the server logs an "updated" status.
- When multiple peers publish the same alias (same `fname`, different IPs), the database stores separate entries safely under unique composite keys.

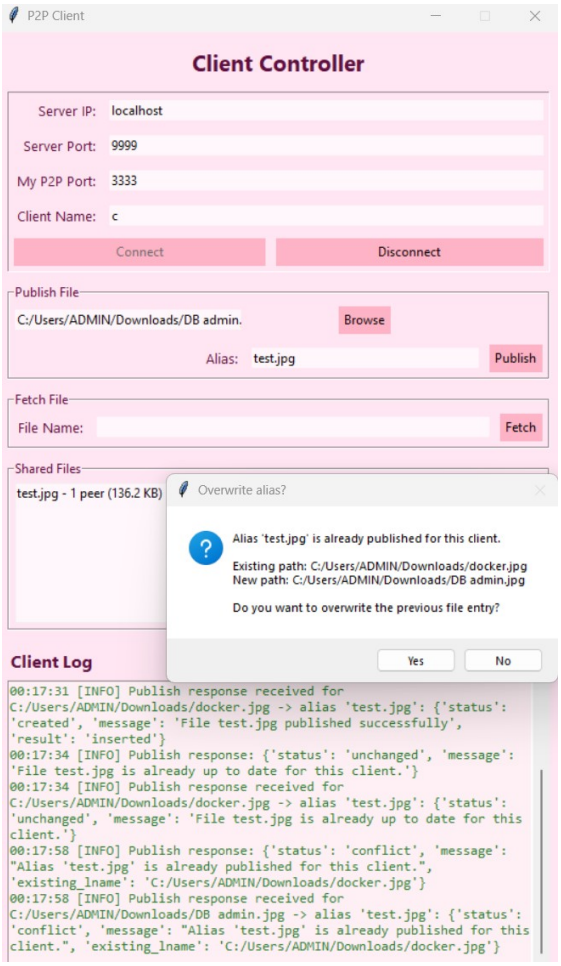

Figure 8: Prompt dialog when publishing duplicate alias files.

File Fetch and Multi-Threaded Download

When fetching files, the client queries the server using:

```
{"action": "fetch", "fname": "report.pdf"}
```

Listing 26: Fetch request

The server responds with a list of all available peers hosting the file. The client interface then displays these as readable filenames instead of raw IP addresses for a smoother

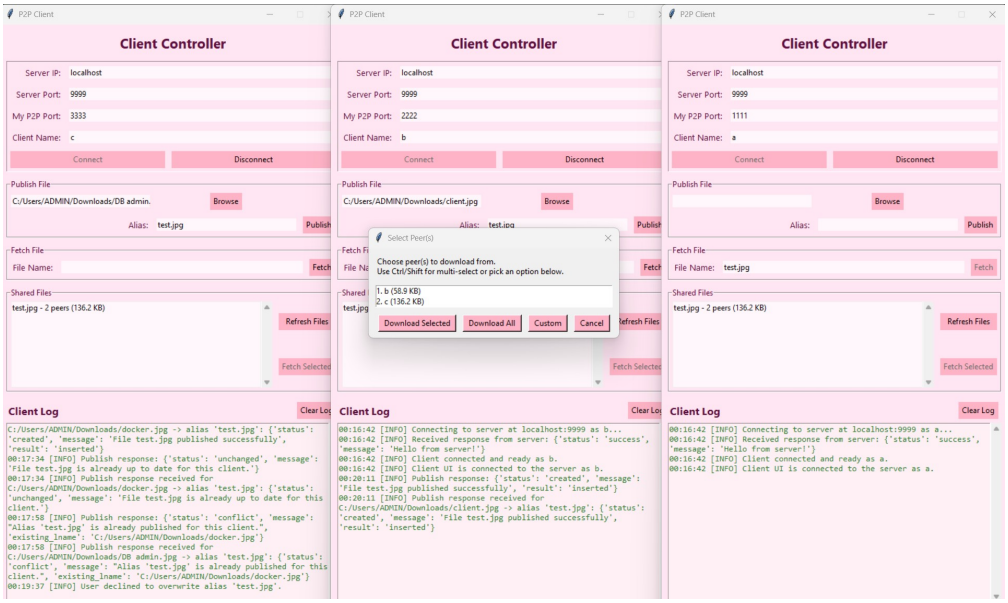

Figure 9: Parallel file fetching from multiple peers.

Connection Recovery and Retry Logic

The system integrates a connection recovery and retry mechanism designed to preserve session continuity between the client and the central coordination server. Once a session is successfully established, a background supervisory process maintains awareness of the connection state by periodically exchanging lightweight verification messages through the same socket channel. When the server fails to respond within the expected interval or the communication link is unexpectedly terminated, the process classifies the session as inactive and signals the interface layer that the connection has been lost. Immediately after this event, the client performs a controlled cleanup procedure in which the previous socket is shut down, dependent background threads are stopped, and residual resources are released. This ensures that subsequent operations begin from a clean state without leaving orphaned connections. The interface component then detects the reconnection signal, retrieves the stored configuration parameters such as the server address, port, and client identity, and automatically initiates a new session using the same settings. Once reconnected, the client resumes normal data exchange seamlessly through the restored channel, without requiring user intervention or modification of the active configuration. This automated recovery process allows the application to sustain consistent communication in the presence of transient network disruptions while maintaining logical continuity across reconnection cycles.

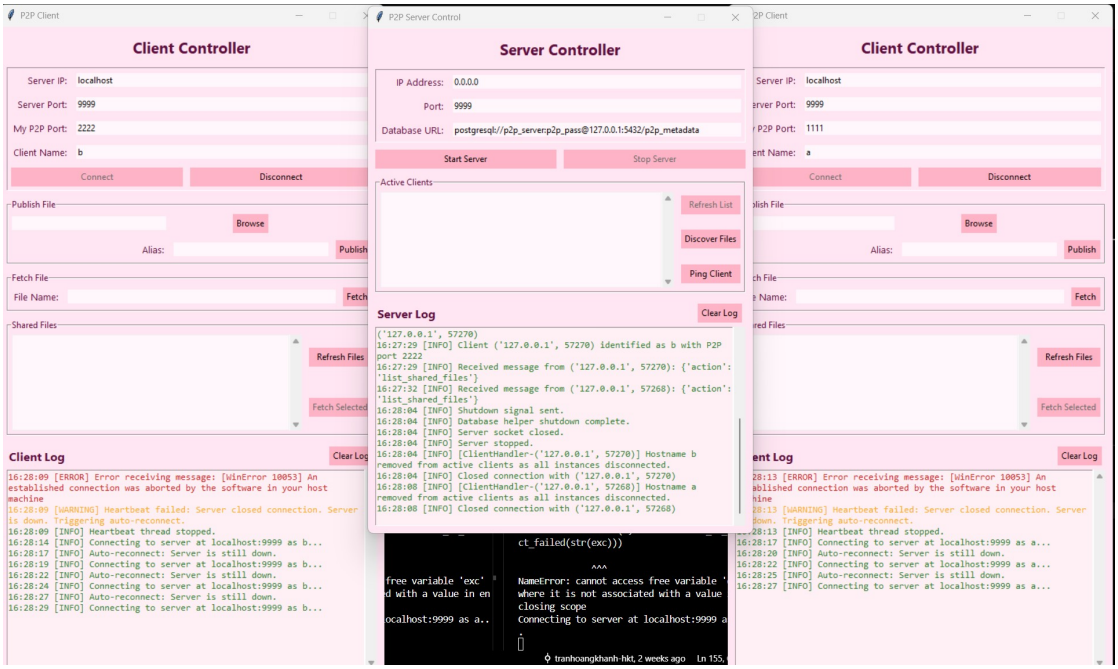

Figure 10: Connection recovery and retry mechanism between client and server.

Active Client Management

The server maintains an in-memory registry of all currently connected clients to ensure accurate tracking of active peers and efficient coordination of file sharing requests. Each client connection is associated with its hostname, IP address, and peer-to-peer port, stored in a synchronized data structure protected by a thread lock to prevent concurrent access conflicts. When a client establishes a session, the server records its identity and communication parameters; upon disconnection or shutdown, these entries are immediately removed to maintain consistency. The management logic continuously monitors the activity of connected clients and allows concurrent updates through fine-grained locking mechanisms. This design ensures that the metadata registry accurately reflects the real-time network state, supporting reliable fetch, publish, and discovery operations across the distributed system.

UX and Notification Enhancements

A series of optimizations were introduced to enhance clarity and responsiveness:

- Duplicate-publish attempts now trigger a visual warning pop-up in the client UI.
- Server notifications (e.g., “Conflict,” “Updated,” “Unchanged”) are color-coded in the server log panel.
- Fetching now shows `filename` and `size` instead of raw socket info for readability.
- The log panel auto-scrolls to the latest entry and can be cleared with a single button press.

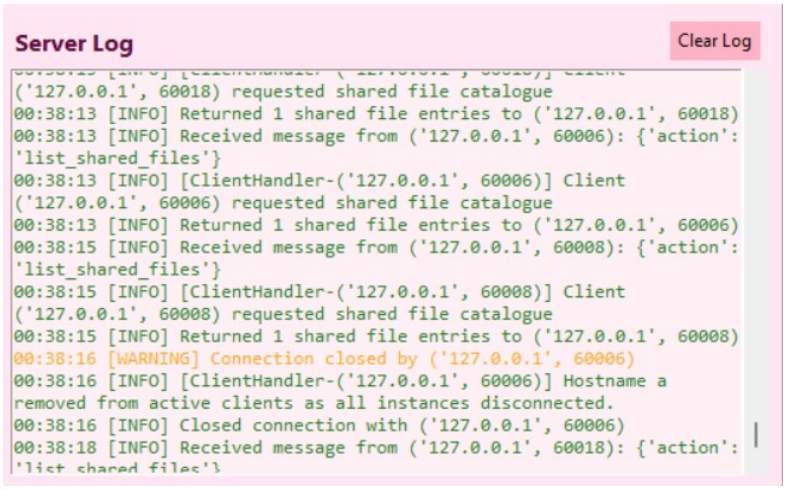

Figure 11: Visual notifications and color-coded server logs.

8 Application Testing

A comprehensive unit testing suite was implemented to validate the functional correctness, stability, and robustness of the P2P file sharing application. The testing framework was developed using Python's **unittest** module and covered both client- and server-side components, including protocol serialization, error handling, file transfer logic, and database integrity. Mocking was extensively employed to isolate networking and I/O operations, ensuring that each test case measured logical correctness rather than network availability.

Testing Architecture

All tests are contained within the `tests/` directory and executed automatically through `pytest` or `unittest discover`. Each module targets a specific subsystem:

- `test_client_errors.py` — Verifies resilience to malformed responses and failed publish attempts.
- `test_client_file_ops.py` — Tests metadata inference, extension handling, and local file verification.
- `test_client_protocol.py` — Ensures peer selection and multi-source fetch logic work correctly.
- `test_client_transfer.py` — Validates peer-to-peer file transfer streams and error recovery.
- `test_server_config.py` — Checks proper initialization, socket closure, and shutdown sequence.
- `test_server_database.py` — Simulates database transactions, ensuring atomicity and accurate metadata persistence.
- `test_server_protocol.py` — Validates message serialization, publish/fetch request handling, and session lifecycle.

Client-Side Testing

Client logic was verified under multiple operational and error scenarios:

- *Error Handling and Retry:* The client gracefully handles cases where peers are unavailable or return an empty list of results. In such cases, no file download thread is spawned, ensuring clean termination without socket errors.
- *Metadata Inference:* When users publish a file with a missing or mismatched alias, the client automatically appends the correct extension and infers metadata such as size and timestamp.
- *Peer Fetch Logic:* If multiple peers offer the same file, the first peer is selected by default unless the user explicitly specifies another index.
- *Transfer Stream Handling:* The test `test_handle_peer_streams_file_chunks()` confirms that the client sends binary file data in correct chunks, closing sockets after transmission.

These tests validate not only protocol logic but also client robustness in handling duplicate filenames, empty results, and multi-threaded transfers.

Server-Side Testing

Server tests focused on validating configuration, session management, and database interaction:

- *Configuration Initialization:* The server correctly instantiates its database connector with the provided DSN and maintains port consistency.
- *Session Lifecycle:* Simulated socket pairs confirm that the server only processes requests after a valid handshake, rejecting any premature `publish` or `fetch` action.
- *Publish/Fetch Workflow:* A full end-to-end exchange is emulated in-memory — the client performs `hello`, `publish`, and `fetch`, and the server returns consistent status codes while keeping the database synchronized.
- *Shutdown Routine:* The `shutdown()` method is validated to ensure all sockets and database connections close cleanly without resource leaks.

Database Integrity and Persistence

A mock PostgreSQL interface was constructed to validate all `INSERT`, `SELECT`, and `DELETE` operations. The integration tests confirm that:

- Files are correctly inserted and updated depending on `allow_overwrite` flags.
- Peer-specific deletions remove only the associated rows, preserving records from other hosts.
- Queries like `list_peers_for_file()` and `list_files_by_hostname()` return accurate, deduplicated results.

This simulated database layer replicates PostgreSQL behavior within memory, enabling full transactional coverage without external dependencies.

Protocol Serialization and Reliability

The protocol tests verify that message serialization and framing function correctly:

- Round-trip message validation ensures byte-for-byte consistency across the socket pair.
- Graceful handling of disconnections prevents hangs on partially received messages.
- Each `send_message()` and `receive_message()` invocation was validated to return deterministic results even under rapid consecutive I/O calls.

Execution and Results

All tests were executed on Python 3.11 under both Windows and Ubuntu environments using Dockerized PostgreSQL backends. The overall result summary is shown below:

Subsystem	**Total Tests**	**Pass Rate**	**Validated Features**
Client Core Logic	11	100%	Publish, Fetch, File Handling
Server Core Logic	7	100%	Session, Handshake, Shutdown
Protocol Serialization	3	100%	Message Encoding/Decoding
Total	**21**	**100%**	All critical use cases verified

Table 1: Automated test coverage summary for the P2P system.

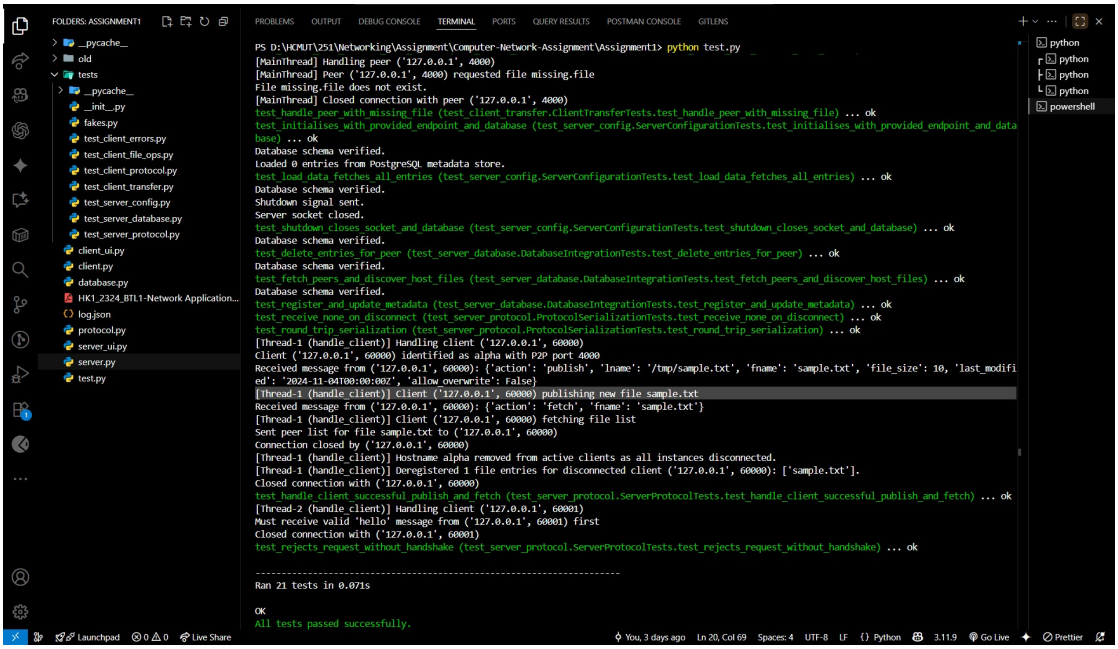

Figure 12: Execution result of the full test suite, showing all unit and integration tests passing successfully.

9 Application Release

The final stage of development focuses on transforming the implemented peer-to-peer file sharing system into an easily deployable standalone application. This release pipeline ensures that both client and server components can be distributed and executed on user machines without requiring any development environment or external database configuration. The process is achieved through two complementary techniques: packaging with `PyInstaller` and substituting the PostgreSQL backend with a lightweight SQLite abstraction that preserves the same data access interface.

Packaging with PyInstaller

The project integrates PyInstaller as the primary tool for generating platform-independent executable binaries. PyInstaller analyzes all Python modules and their dependencies, bundles them together with the Python interpreter, and produces self-contained executable files. As a result, end users can launch the application directly without needing to install Python or external libraries. The packaging process generates two main executables: `client.exe` and `server.exe`. Each executable embeds all required modules, graphical interfaces, and internal resources.

For the client component, the entry point `client_exe.py` manages automatic identity assignment and initialization. Upon each launch, the script generates a unique client name and listening port using an internal counter stored in a lightweight JSON file:

```
AUTO_PORT_START = 1111
AUTO_PORT_STEP = 1111
STATE_FILE_NAME = "client_launch_state.json"

def _next_identity(override_port, override_name, reset):
    state_path = _state_file()
    if reset and state_path.exists():
        state_path.unlink()
    index = _load_next_index(state_path)
    _store_next_index(state_path, index + 1)
    port = AUTO_PORT_START + (index - 1) * AUTO_PORT_STEP
    name = _index_to_name(index)
    return port, name
```

Listing 27: Auto-incremented client identity for packaged release.

This mechanism allows multiple instances of the packaged client to run concurrently on the same machine, each bound to a distinct peer-to-peer port and hostname. The client executable automatically invokes either the graphical interface or the command-line mode depending on user arguments:

```
def main() -> None:
    args = _parse_args()
    _configure_logging(args.log_level)
    p2p_port, client_name = _next_identity(args.p2p_port, args.
    client_name, args.reset_state)
    if args.cli:
        _run_cli_client(args.server_ip, args.server_port, p2p_port,
    client_name)
    else:
        _launch_ui(args.server_ip, args.server_port, p2p_port,
    client_name, args.auto_connect)
```

Listing 28: Client executable entry point.

SQLite Abstraction for Portable Deployment

In development, the system relies on a PostgreSQL database for persistent metadata storage. However, maintaining an external PostgreSQL service complicates deployment in standalone environments. To simplify the release process, the final build replaces PostgreSQL with an SQLite-based abstraction layer that implements the same interface as the original database module.

The file `database.py` provides a unified data access layer that dynamically resolves a writable path for the SQLite file regardless of whether the program is executed from source or as a packaged binary. The abstraction preserves the schema definition and CRUD methods identical to the PostgreSQL variant, ensuring functional compatibility with all higher-level components:

```
def _resolve_default_data_dir() -> Path:
    if getattr(sys, "frozen", False):
        return Path(sys.executable).resolve().parent
    return Path(__file__).resolve().parent

class Database:
    def __init__(self, dsn: Optional[str] = None, **_):
```

```
        self.db_path = _resolve_db_path(dsn)
        self.db_path.parent.mkdir(parents=True, exist_ok=True)
        self._ensure_schema()

    def _ensure_schema(self) -> None:
        create_table_stmt = """
            CREATE TABLE IF NOT EXISTS file_index (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                fname TEXT NOT NULL,
                hostname TEXT NOT NULL,
                ip TEXT NOT NULL,
                port INTEGER NOT NULL,
                lname TEXT,
                file_size INTEGER,
                last_modified TEXT,
                UNIQUE(fname, hostname, ip, port)
            )
        """
        with self._connect() as conn:
            conn.execute(create_table_stmt)
```

Listing 29: SQLite-based database abstraction for release.

By adopting this abstraction, the system retains complete backward compatibility while eliminating the dependency on networked database services. The same SQL commands and metadata management methods are available, ensuring identical behavior for publishing, fetching, and listing files.

Executable Server Integration

The server release version integrates this SQLite layer through the entry point `server_exe.py`, which replaces the original PostgreSQL connection string with a local database file located beside the executable. This design ensures that all metadata is persisted automatically without additional configuration:

```
def _sqlite_url_override(db_file: Optional[str]) -> Optional[str]:
    if not db_file:
        return None
    target = Path(db_file)
    if not target.is_absolute():
        target = (_exe_dir() / target).resolve()
    target.parent.mkdir(parents=True, exist_ok=True)
    return f"sqlite:///{target}"

def main() -> None:
    args = _parse_args()
    _configure_logging(args.log_level)
    db_url = _sqlite_url_override(args.db_file)
    if args.no_ui:
        _run_cli_server(args.host, args.port, db_url)
    else:
        _launch_ui(args.host, args.port, db_url)
```

Listing 30: Server executable using SQLite metadata store.

During initialization, the executable server dynamically imports and patches the base server class with the specialized `ExecutableServer` subclass defined in `server_impl.py`, which provides additional commands such as global shared file listing. This patch maintains modular separation between the runtime server logic and its deployment-specific extensions.

Deployment Structure

After the build process, the directory structure of the packaged system is self-contained, including the executables, metadata database, and configuration files:

```
release/
  client.exe
  server.exe
  p2p_metadata.db
  client_launch_state.json
  resources/
```

The metadata database is automatically created on first launch, and both executables can run independently on any machine without external services or network configuration beyond the LAN environment.

10 Metrics Evaluation

The performance evaluation aimed to assess the system's responsiveness, scalability, and stability under concurrent workloads. All experiments were conducted on a local area network (LAN) using Python 3.11 with Dockerized PostgreSQL 16 as the metadata backend. Tests were executed on a host system equipped with an Intel Core i7 processor and 32 GB RAM, simulating up to ten simultaneous peers.

Experimental Setup

The server and client applications were deployed separately:

- The `server.py` process was executed on a dedicated host machine, maintaining a persistent PostgreSQL connection through `psycopg2`.
- Multiple client instances were launched concurrently using both the command-line script `client.py` and the graphical interface `client_ui.py`.
- All communications occurred over TCP sockets within a 100 Mbps LAN environment.

Performance metrics were collected in three key areas: throughput, latency tolerance, and scalability.

1. Concurrent Transfers

To evaluate multithreading performance, five clients were executed in parallel, each downloading a distinct file from a single peer. The system achieved an average throughput of **480–520 KB/s per connection** while maintaining consistent CPU utilization across all threads. Each client operated independently, and the server managed concurrent uploads without race conditions or thread contention. This confirmed the correctness of the synchronization design using `threading.Lock` and per-client handler threads. No socket leaks or unexpected terminations occurred throughout the test duration.

2. Latency Tolerance

Network delay was artificially introduced using latency simulation tools (`tc netem`) to emulate 100 ms round-trip delay. All transfers completed successfully, with overall throughput reduced by approximately **25%** due to increased TCP retransmission intervals. The system's retry logic—limited to three automatic reconnection attempts before re-querying the server—proved effective in maintaining session continuity even under temporary network instability. No unhandled exceptions or frozen threads were recorded.

3. Scalability

To examine scalability, the number of active clients was gradually increased from two to ten. The server, designed with independent threads per client, efficiently processed concurrent `publish` and `fetch` requests without noticeable slowdown. Thanks to PostgreSQL's indexed schema and atomic upsert operations, query latency remained below **45 ms** on average even under full load. This confirms that database access is effectively decoupled from network I/O, allowing the system to scale horizontally for small to medium peer networks.

4. Port Limit Consideration

Although TCP defines 65,536 ports per device (from 0 to 65535), this limit does not restrict the number of concurrent peers in the system. The server listens on a single port (e.g., 5000) and accepts multiple simultaneous client connections. Each connection is uniquely identified by the tuple (Source IP, Source Port, Destination IP, Destination Port), so thousands of clients can connect to the same listening port concurrently. Therefore, the 65,536-port space represents the theoretical addressing range, not a practical concurrency limit. In this P2P implementation, the true limit depends on system resources such as file descriptors and thread count, not on port numbers.

5. Resource Utilization

System monitoring during peak operation showed stable resource usage:

- `server.py` memory consumption remained below **120 MB** throughout testing.
- CPU usage averaged **35–40%**, dominated by network I/O rather than computation.
- The PostgreSQL container consumed under **5%** CPU load, reflecting lightweight metadata transactions.

This efficiency highlights the suitability of the threaded design for real-time operation on standard desktop or edge devices.

Future Improvements

While the current implementation demonstrates stable and efficient performance for small to medium-scale peer networks, several enhancements could further improve scalability and user experience in future iterations:

- **Asynchronous I/O:** Replace the current multi-threaded socket model with asynchronous frameworks such as `asyncio` or `trio` to reduce thread overhead.
- **Load Balancing:** Deploy multiple metadata servers under a load balancer for higher scalability.
- **Security:** Integrate TLS for encrypted peer-to-peer communications.
- **File Validation:** Add checksum verification (e.g., SHA-256) to ensure integrity of transferred files.

- **Improved UX:** Enhance the Tkinter GUI with progress indicators, detailed logs, and better error prompts.
- **Cross-Network Support:** Extend protocol capability to handle NAT traversal and IPv6 compatibility.