

**VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY**  
**HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY**

□ □  
**COMPUTER NETWORK**



**ASSIGNMENT 1**  
**DEVELOP A NETWORK APPLICATION**

**CC02 — SEMESTER 231**

**LECTURER : Prof. NGUYEN LE DUY LAI**

<b>Student</b>	<b>ID</b>	<b>Marks</b>
Bành Tân Thuận	2153011	100%
Lê Trần Nguyên Khoa	2152674	100%
Trần Minh Triết	2153057	100%
Nguyễn Vĩnh Huy	2152597	100%

**HO CHI MINH CITY – 2023**

# TABLE OF CONTENT

<b>1. Overview</b>	<b>2</b>
a. Requirement	2
b. Implementation	3
• Server	3
• Client	3
<b>2. Activity Diagram</b>	<b>5</b>
<b>3. Architecture Diagram</b>	<b>6</b>
<b>4. Class Diagram</b>	<b>7</b>
<b>5. Roles and responsibilities</b>	<b>8</b>
<b>6. Application Instructions Manual</b>	<b>9</b>
a. How to run the application	9
• Server: To start a server, we need to run the indexServer.py file.	9
• Client: To start a client, we need to run the user.py file.	9
b. How to use each command	9
• Server:	9
• Client:	10
<b>7. Define and describe the functions of the application</b>	<b>11</b>
a. Server	11
b. Peer	12
• Sender peer (inherit from Peer)	13
• Receiver peer (inherit from Peer)	14
<b>8. Define the protocols used for each function</b>	<b>15</b>
<b>9. Validation (sanity test) and evaluation of actual result (performance)</b>	<b>18</b>

# 1. Overview

This report details the design and implementation of a straightforward file-sharing application that operates on the TCP/IP protocol stack. The application is characterized by a centralized server that manages connected clients and their stored files. Clients communicate with the server to inform it about the contents of their local repositories without transmitting actual file data.

## a. Requirement

When a client needs a file not present in its repository, a request is sent to the server. The server identifies other clients storing the requested file name and shares their attributes with the requesting client. Subsequently, the client selects a suitable source client, and the file is directly fetched from that client without server intervention. The system is designed to support multiple clients downloading different files from a target client simultaneously, necessitating multithreading in the client code.

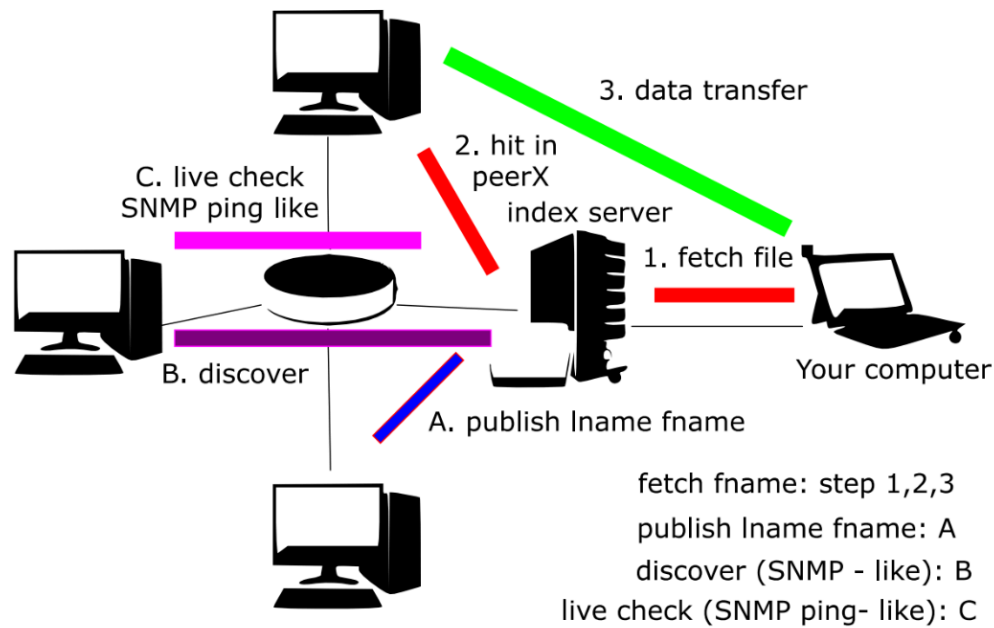
The client incorporates a simple command-shell interpreter accepting two types of commands:

- **publish lname fname**: Adds a local file (stored in the client's file system at lname) to the client's repository as a file named fname. This information is conveyed to the server.
- **fetch fname**: Requests a copy of the target file, which is then added to the local repository.

The server also features a command-shell interpreter with commands such as:

- **discover hostname**: Discovers the list of local files on the host named hostname.
- **ping hostname**: Performs a live check on the host named hostname.

The supported commands and system activities are illustrated in Figure 1. This report provides insights into the architecture, functionalities, and protocols employed in the development of this file-sharing application, offering a comprehensive understanding of its design and operation.



**Figure 1. Illustration of file-sharing system activities.**

## **b. Implementation**

In accordance with the project requirements, our group has successfully implemented two essential program files: the server and the client. Leveraging the versatility of the Python programming language, the project is executed through the terminal, aligning with the simplicity and efficiency of a command-line interface. It is noteworthy that our project is specifically designed for deployment within a local network, under the scope of this assignment.

### • **Server**

Upon initialization, the server strategically employs a multithreading approach to enhance its responsiveness. Initially, a dedicated thread is spawned to actively listen for incoming client connections. After which for each connected client, additional threads are dynamically created to serve their specific requests, systematically integrating them into a managed list.

The server's proactive engagement extends to regular pinging and discovery activities aimed at monitoring client activeness and detecting directory changes by getting the client to respond. This process occurs at intervals of 5-10 seconds, allowing the server to stay abreast of any alterations in client status or file directories. In cases where clients become disconnected, a systematic removal mechanism is activated, eliminating both the disconnected clients and any associated published files from the server's active list. This systematic approach ensures the server's adaptability and responsiveness to the dynamic nature of client interactions.

### • **Client**

Upon initiation, the client program embarks on a series of steps to ensure its seamless integration into the file-sharing ecosystem. Initially, it tries to retrieve its own directory or creates one if none exists for the shared files. Subsequently, the client binds to the socket that the server broadcasts, extracting information such as the server's IP and port. This preparatory phase lays the groundwork for user interaction.

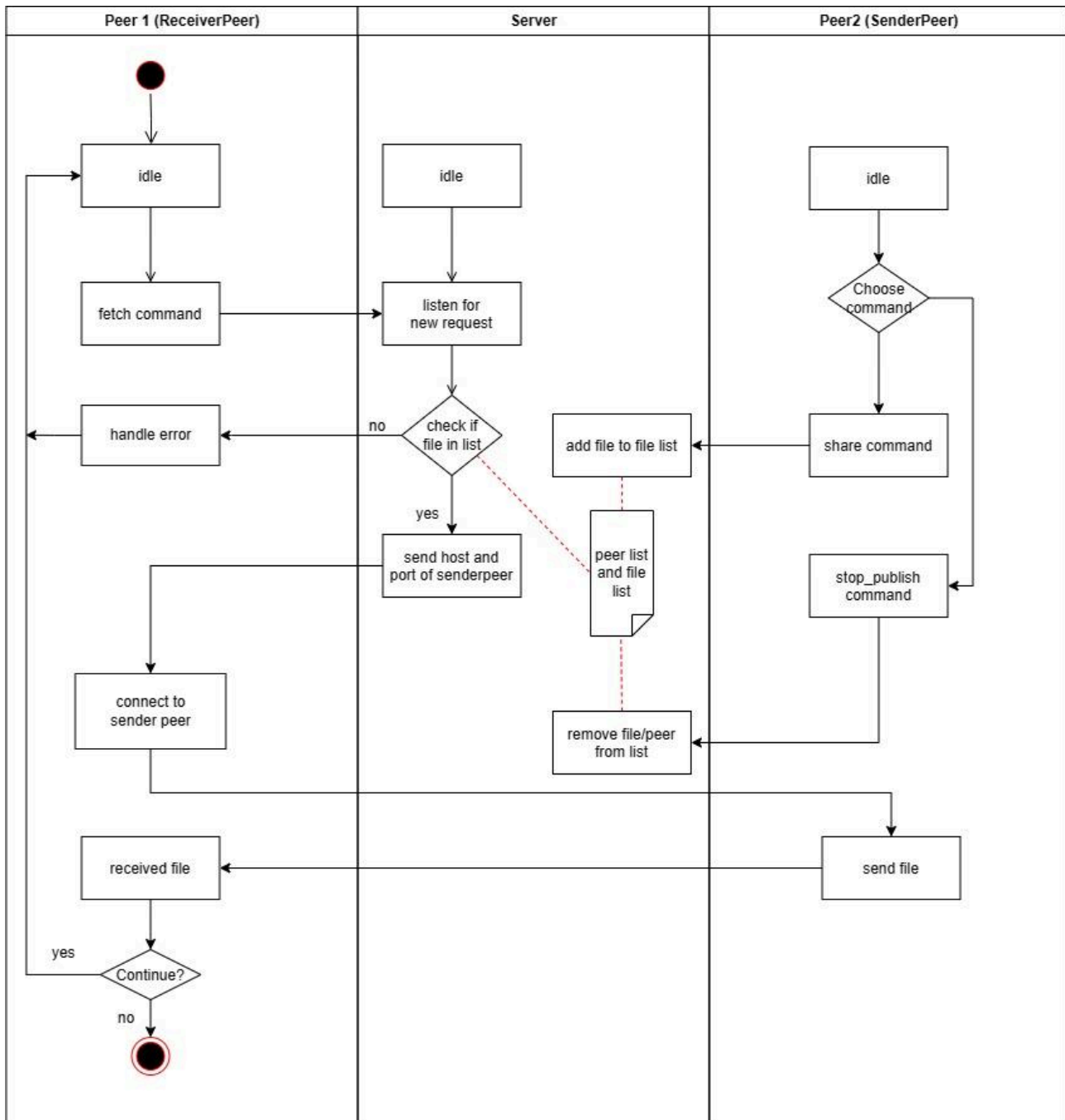
Within the command-line interface, the client seamlessly engages users by prompting input for commands, fostering an intuitive and dynamic interaction with the file-sharing system. Command options include the ability to publish or unpublish files residing in the local directory, providing users with versatile control over their shared resources.

The "fetch" command, a pivotal feature, retrieves a list of client IP addresses and ports possessing the desired file. Subsequently, the client intelligently selects the first client from this list and initiates a connection attempt. Each client maintains a dedicated IP and port for listening to fetching clients, creating individual threads to parallelly send files to multiple clients, optimizing the efficiency of the file-sharing process.

To gracefully conclude the client's interaction with the system, users are encouraged to utilize the "end" command rather than abruptly terminating the terminal. Upon executing this command, the client actively notifies the server, facilitating the immediate removal of the client from the system, thereby offering a more controlled and prompt conclusion to the client-server interaction. This approach contrasts with the passive removal mechanism occurring at regular intervals (every 5-10 seconds) and ensures a smoother and more responsive system operation.

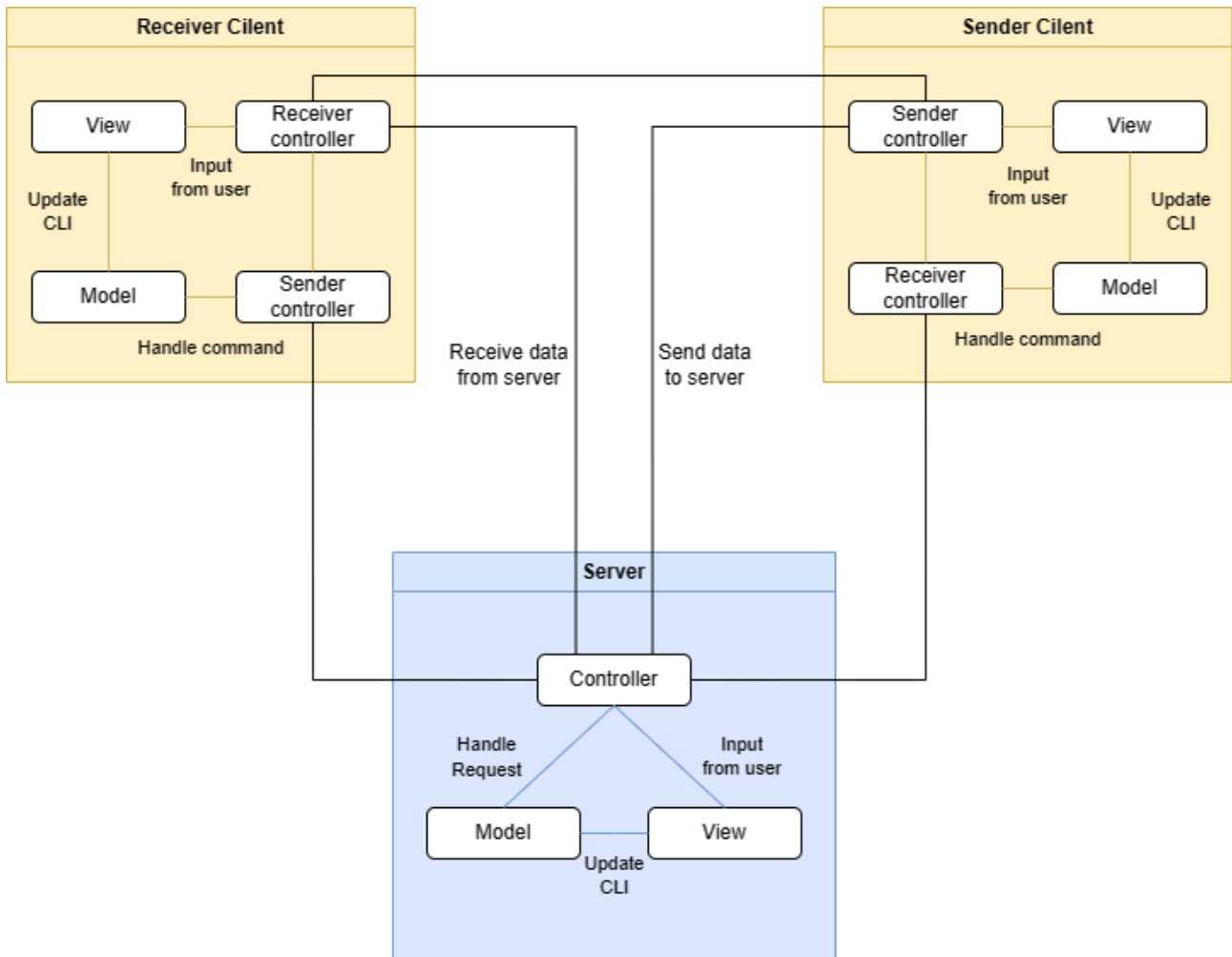
It is worth noting that for the client program functionality and connectivity, users are advised to disable their firewalls when sharing files through the client. This ensures peer to peer access for other users aiming to connect and engage in the collaborative sharing of files.

## 2. Activity Diagram



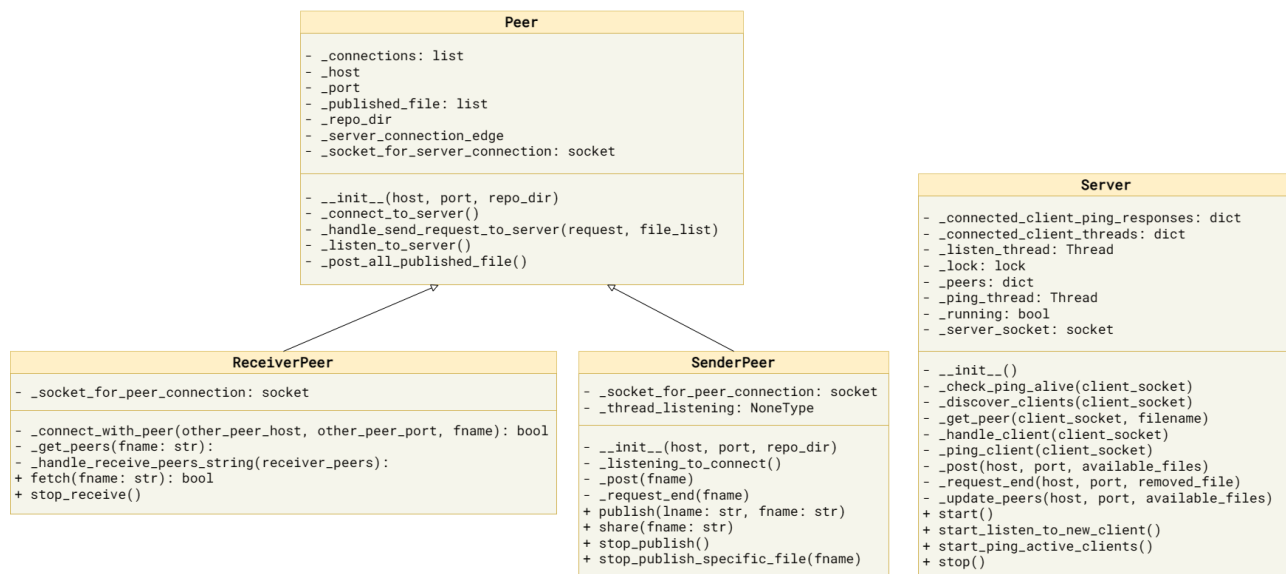
This diagram shows the main activity flow as well as the interaction between each component of the system.

### 3. Architecture Diagram



Our system uses MVP architecture for both client-side and server-side. The diagram above shows the interaction between each component in the system.

## 4. Class Diagram



The above diagram is a Class diagram of our P2P file sharing system. The main program consist of 4 classes:

- The Server class has attributes to store a list of peer, its
- The Peer class has attributes for the host, ports, and file list. It has some basic methods to connect to the main server and to handle sending requests to the server.
- The SendPeer class is a subclass of Peer. It inherits attributes and methods of the main Peer class as well as having distinct methods/attributes more suitable for publishing/sending files.
- The SenderPeer class is also a subclass of Peer with the inherited attributes. Its distinct methods are used to receive files from other peers and send a request to the server for a list of peers.



-

## 5. Roles and responsibilities

Name	Roles	Responsibilities
Lê Trần Nguyên Khoa	Project Manager Client side developer	Implementation of base peer class Implementation of 2 derived class senderPeer and receiverPeer Implementation of some peers basic functionality Maintain project progress and member workload
Bành Tân Thuận	Git flow Server side developer Report writer	Maintain a decent team's git flow Implementation of server's ping modules Implementation of server's discover modules Production Testing and quality assurance Report Writing
Trần Minh Triết	Server side developer Report writer	Implementation of server class Implementation of server's basic functionality Report Writing Diagram Construction
Nguyễn Vĩnh Huy	Client side developer Report writer	Implementation of sender to receiver connection Implementation of server to peer connection Implementation of publish and fetch command Report Writing

## 6. Application Instructions Manual

- Before starting with any of the below manuals, please make sure to have your python 3 installed. For best experience and compatibility, we recommended using python 3.10 .
- To check for your python version you can use the bash command:  
`python --v` Or: `py --v`
- You can install python 3.10 at: [download python](#)

### a. How to run the application

- **Server: To start a server, we need to run the *indexServer.py* file.**

- Method 1: Run with tools

If you have any code editor with python runner support (Visual Studio, VSCode CodeRunner extension,...), you can run *indexServer.py* with these tools.

- Method 2: using command line terminal

We execute these bash command:

!Note: DIRECTION is replaced with your actually store direction of the folder p2p-file-sharing

```
cd DIRECTION/p2p-file-sharing
python -u ./indexServer.py
```

!Some computer might not recognise the python keyword, you can instead use the command:

```
py -u ./indexServer.py
```

**You have successfully started the server!!**

- **Client: To start a client, we need to run the *user.py* file.**

- Method 1: Run with tools

If you have any code editor with python runner support (Visual Studio, VSCode CodeRunner extension,...), you can run *user.py* with these tools.

- Method 2: using command line terminal

We execute these bash command:

(Note that DIRECTION is replace with your actually store direction of the folder p2p-file-sharing)

```
cd DIRECTION/p2p-file-sharing
python -u ./user.py
```

Some computer might not recognise the python keyword, you can use the command:

```
py -u ./user.py
```

**You have successfully started your client!!**

!Note: Every-time you run *user.py*, a folder named *repo\_dir\_RANDOM* will be created. Please put your file in this folder if you want to publish this file.

!Note: We suggest you start the server script first to negate the chance of misbehavior.

### b. How to use each command

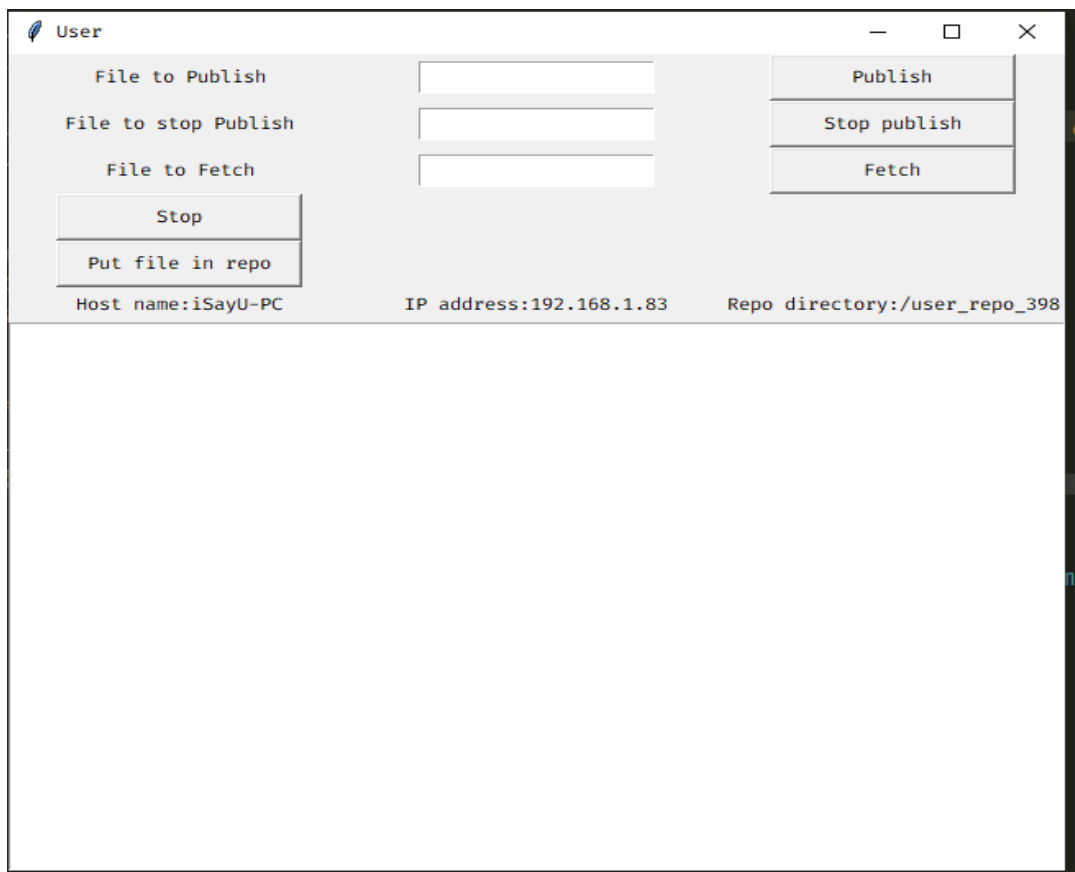
- **Server:**

- The Server only has 1 command which is “end”. “ping “ and “discover” already implemented as a server core (the server will automatically run these 2 command every 5 second to check connection continuously)
- Type “end” to the command prompt to shut the server down

- **Client:**

Client have 5 main functionality:

- publish: to publish a file to server, type the file name in the input field and press Publish button
  - Stop publish: to stop publishment of a file, type in the file name in the input field and press Stop Publish button, publishment will be remove from both server and client side, use this in case you want to delete a file from the user\_repo directory
  - Fetch: to fetch a file from another client via index server, type in the file name in the input field and press Fetch button. This file will be added to the user\_repo directory
  - Stop : to shut the client down
  - Put file in repo: use this when you want to put a file in your computer into repository for later publishment, fetched file from another client will also appeared in the same directory
  - Other information shown in the UI are: User's Hostname, IP address and repository directory
  - There is also a debug terminal to show debugging message
- Below is a screenshot of the client's UI



## 7. Define and describe the functions of the application

In this section, we detailly describe the main networking functions in the application.

### a. Server

#### - **`_handle_client_packet(self, client_socket):`**

This function handles every request sent from a `client_socket` to the server. Server should create a new thread to handle a newly connected client.

The clients can send 5 type of request to server:

“<publish>client\_host|client\_port|fname</publish>”: publish a file named `fname` to server.

“<unpublish>client\_host|client\_port|fname</unpublish>”: end publishment of a file named `fname`

“<get\_peer>client\_host|client\_port|fname</get\_peer>”: find the list of other clients with the published file `fname`

“<reveal>client\_host|client\_port|fnames</reveal>” : check if published files in the published list on the client side are actually published on the server side

“<pong>client\_host|client\_port</pong>” : response to the server’s ping request.

“<disconnect>client\_host|client\_port</disconnect>” : remove all client published files and end connection

#### - **`_publish_file(self, host, port, available_files):`**

This function handles the “<publish>” request

It publishes a file from a client with `host` and `port` to server. If this client is already in `_peers` dictionary\*, we will update the published files list of this client. If not, add this client in `_peers` dictionary and initialize the published files list

\*`_peers` dictionary: a dictionary with key is the (`host`, `port`) pair of the client, and the value is the published files list of this client. This dictionary keeps track of every client that has at least 1 published file.

#### - **`_update_peers(self, host, port, available_files):`**

This function handles the “<reveal>” request

It checks every file in `available_files` if it is in the published list of client with `host`, `port` in the dictionary, if not, add that file in to the list

#### - **`_unpublish_file(self, host, port, removed_file):`**

This function handles the “<unpublish>” request

It removes the specified file from the client's list of available files, which removes the peer from the `self._peers` dictionary if no file is specified (`removed_file = null`)

#### - **`_disconnect_client(self, client_socket)`**

This function handles the “<disconnect>” request

It removes the associated file from the disconnecting client's list of available files and removes the peer from the `self._peers` dictionary if no file is specified (`removed_file = null`). Then it closes the socket with that client.

- **`_handle_send_request_to_client(self, client_socket, request, data)`**

This method handles sending requests to the client. The requests are created by combining the request string, self host, self port and the list of data if necessary. It then encodes it into bytes and sends the package to the client.

- **`_get_peer(self, client_socket, filename):`**

This function handles the “<get\_peer>” request

It looks for every client host and port with a published file matched with filename and sends this list to the client\_socket.

- **`_ping_client(self, client_socket):`**

This function sends a “<ping>” request to the client\_socket.

It also unset the ponged flag for this client in the `_connected_client_ping_responses` list

- **`_check_ping_alive(self, client_socket):`**

It checks the aliveness of client\_socket. Returned the ponged flag for this client\_socket. If the client\_socket is not in the `_connected_client_ping_responses` list, return False

- **`_discover_clients(self, client_socket):`**

It sends a discover request to the client\_socket

- **`_update_listen_to_new_client(self):`**

Continuously checking for and accepting the connection with any client, if there is an accepted connection, create a new thread with target function `_handle_client()` to handle this connected client request.

- **`_update_ping_active_clients(self):`**

Continuously check every connected client aliveness by sending `_ping` requests and gathering `_pong` responses. If a client is shut down (server did not see the pong response), remove this client from the connection list.

- **`start(self):`**

Initiate the server

- **`stop(self):`**

Shut the server down

## **b. Peer**

- **`_handle_server_address_broadcast(self):`**

This method listens for server address broadcasts and connects to the server.

- **`_connect_to_server(self):`**

This method tries to connect to the server using the specified SERVER\_HOST and SERVER\_PORT. If the connection is successful, it creates a new thread to start listening to the server.

- **`_send_packet_to_server(self, request, file_list = ""):`**

This method handles sending requests to the server. The requests are created by combining the request string, self host, self port and the list of data if necessary. It then encodes it into bytes and sends the package to the server.

- **`_listen_to_server(self):`**

Continuously listens for messages from the server. If it receives a ‘\_ping’ message, it responds with a ‘\_pong’ message. If it receives a ‘\_discover’ message, it posts all published files.

- **`_reveal_all_published_file(self):`**

It constructs a string of all published files, separated by commas, and sends it to the server using the `_handle_send_request_to_server` method.

- **`_terminate_peer(self):`**

This function handles the server using “<disconnect>” request to the `server_socket`. It terminates the peer, closes connections, joins all threads.

- ***Sender peer (inherit from Peer)***

- **`_listening_to_connect(self):`**

This method creates a new socket and binds it to the host and port of the `SenderPeer` instance. It then starts listening for incoming connections. When a connection is accepted, it adds the connection to the list of connections, receives a file name from the connection, shares the file, removes the connection from the list, and closes the connection.

- **`publish(self, fname):`**

Constructs a <publish> request with the provided file name and sends it to the server using the `_handle_send_request_to_server` method from the `Peer` class.

- **`publish(self, lname: str = "", fname: str = ""):`**

This method checks if the file is already in the list of published files. If it is not, it adds the file to the list and sends a ‘\_post’ request to the server with the file name using the `_post` method.

- **`stop_publish_specific_file(self, fname):`**

This method checks if the file is in the list of published files. If it is, it constructs a ‘<unpublish>’ request with the provided file name and sends it to the server using the `_handle_send_request_to_server` method from the `Peer` class and removes the file from the list of published files.

- **`stop_publish(self):`**

Iteratively stops publishing all files in the list of published files using the `stop_publish_specific_file` method. It then closes the server connection.

- **`share(self, fname: str):`**

This method opens a file in binary read mode and sends the file to all connections in the list of connections. It uses the `sendfile` method of the socket object, which sends a file over a socket by copying it directly from the repo to the network, providing file data transfer over sockets.

- ***Receiver peer (inherit from Peer)***

- **`_handle_receive_peers_string(self, receiver_peers) -> [(str, int)]:`**

This method processes the string of peers received from the server. It decodes the string from bytes to a UTF-8 string, splits it into individual peers, and for each peer, it splits the host and port, converts the port to an integer, and appends the tuple of host and port to the list of peers. It then returns this list of peers.

- **`fetch(self, fname: str) -> bool:`**

This method fetches a file from another peer. It first gets a list of peers that have the file using the `_get_peers` method. It then tries to connect to the first peer in the list using the `_connect_with_peer` method. If the connection is successful, it opens the file in binary write mode and continuously receives data chunks from the peer and writes them to the file until no more data is received. It then prints a completion message and returns `True`. If the connection is not successful, it returns `False`.

- **`_get_peers(self, fname: str) -> [(str, int)]:`**

This method sends a ‘`_get_peer`’ request to the server with the filename `fname` using the `_handle_send_request_to_server` method inherited from the `Peer` class. It then continuously receives messages from the server until it receives a message that does not contain ‘`_ping`’. It processes this message using the `_handle_receive_peers_string` method and returns the resulting list of peers.

- **`_connect_with_peer(self, other_peer_host, other_peer_port, fname) -> bool:`**

This method creates a new socket and attempts to establish a connection with another peer using the provided `other_peer_host` and `other_peer_port`. If the connection is successful, it sends the filename `fname` to the peer and returns `True`.

- **`stop_receive(self):`** This method closes the server connection.

## 8. Define the protocols used for each function

Class	Method	Short Description	Protocol
Peer	init(self, host, port, repo_dir)	Initializes Peer with host, port, and repository directory.	None
	_handle_server_address_broadcast(self)	Listens for server address broadcasts and connects to the server.	UDP
	_get_broadcast_socket(self)	Returns a UDP socket for broadcasting server address.	UDP
	_connect_to_server(self)	Connects to the server and starts a listening thread.	TCP
	_parse_packet(self, data)	Parses received data packets into a list of dictionaries.	None
	_send_packet_to_server(self, request, data="")	Sends a formatted packet to the server based on the request type.	TCP
	_listen_to_server(self)	Abstract method to be implemented by subclasses for listening to the server.	TCP
	_reveal_all_published_files(self)	Sends a request to the server to reveal all published files by the current peer.	TCP
	_terminate_peer(self)	Terminates the peer, closes connections, and notifies the server.	TCP
Sender Peer	init(self, host, port, repo_dir)	Initializes SenderPeer with host, port, and repository directory.	None
	_listening_to_receiver_peer_connect(self)	Listens for incoming connections from other peers and handles file sharing.	TCP
	publish(self, lname="", fname="text.txt")	Publishes a file, adds it to the published file list, posts file information to the server, and starts a thread for listening to connections.	TCP
	stop_publish_specific_file(self, fname)	Requests the server to remove the current peer from the list of active peers for a specific file and closes associated socket connection.	TCP
	stop_publish(self)	Stops publishing all files by iterating through the published file list and calling stop_publish_specific_file for each file.	TCP
	share(self, fname)	Starts broadcasting the specified file to all connected peers.	TCP
Receiver Peer	_listen_to_server(self)	Listens to the server for incoming messages and handles different types of requests.	TCP



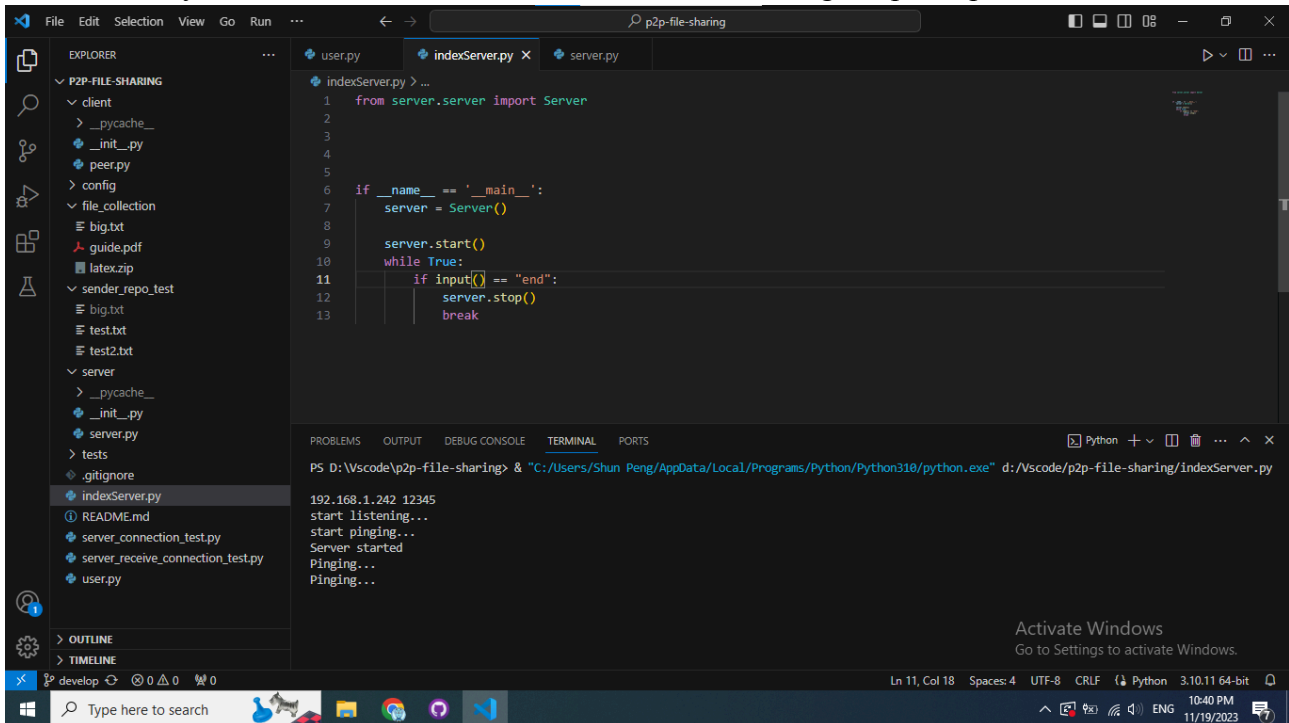
	<code>_handle_receive_peers_string(self, receiver_peers)</code>	Parses the string received from the server containing a list of peers and their information, returning a list of tuples (host, port).	TCP
	<code>fetch(self, fname) -&gt; bool</code>	Initiates the process of fetching a file from another peer. Sends a GET_PEER request to the server to get a list of peers with the specified file.	TCP
	<code>_contact_peer_and_fetch(self, peers_arr)</code>	Initiates a thread to connect with the first peer in the list and start the process of fetching the file.	TCP
	<code>_fetch_from_peer(self, peers_arr)</code>	Connects to a peer and receives the file data, writing it into the local repository.	TCP
	<code>_connect_with_peer(self, other_peer_host, other_peer_port, fname) -&gt; bool</code>	Establishes a connection with another peer to initiate the file transfer.	TCP
	<code>stop_receive(self)</code>	Terminates the receiver peer by stopping threads and closing sockets.	TCP
Server	<code>init(self)</code>	Initializes the Server class, sets up sockets, dictionaries to store peer information, and starts relevant threads.	None
	<code>_parse_packet(self, data)</code>	Parses incoming packet (XML data) to extract request, IP, port, and data information.	None
	<code>_handle_client_packet(self, client_socket)</code>	Handles client requests, including PUBLISH, UNPUBLISH, GET_PEER, PONG, REVEAL, DISCONNECT.	TCP
	<code>_publish_file(self, host, port, available_files)</code>	Adds or updates a peer's information in the dictionary with the list of available files.	TCP
	<code>_update_peers(self, host, port, available_files)</code>	Updates a peer's information in the dictionary with the list of available files.	TCP
	<code>_unpublish_file(self, host, port, removed_file)</code>	Removes a specified file from a peer's list of available files or removes the peer if no file is specified.	TCP
	<code>_disconnect_client(self, client_socket)</code>	Disconnects a client, removes it from dictionaries, and closes the socket.	TCP
	<code>_handle_send_request_to_client(self, client_socket, request, data)</code>	Sends a request with data to a specified client socket.	TCP
	<code>_get_peer(self, client_socket, filename)</code>	Retrieves a list of peers with the requested file and sends it back to the client.	TCP
	<code>_ping_client(self, client_socket)</code>	Sends a PING command to a client socket and waits for a response.	TCP
	<code>_check_ping_alive(self, client_socket)</code>	Checks if a client has responded to the PING command.	None

	<code>_discover_clients(self, client_socket)</code>	Sends a DISCOVER command to a client socket.	TCP
	<code>_update_listen_to_new_client(self)</code>	Listens for new client connections and handles them by creating new threads.	TCP
	<code>_update_ping_active_clients(self)</code>	Periodically pings active clients and removes inactive ones.	None
	<code>_update_broadcast_server_address(self)</code>	Periodically broadcasts the server address to facilitate client discovery.	UPD
	<code>start(self)</code>	Starts the server by initiating relevant threads.	None
	<code>stop(self)</code>	Stops the server by setting the running flag to False and joining threads.	None

## 9. Validation (sanity test) and evaluation of actual result (performance)

In this section, we are going to sanity test sending files. There are 2 PCs connected to the same wifi, they can be easily recognized by one is Window 10 (Not activated Windows yet) and the other is Window 11. Both PCs have been turned off the Windows firewall.

Initially, the Window 10 Pc starts the server, broadcasting its ip and port.



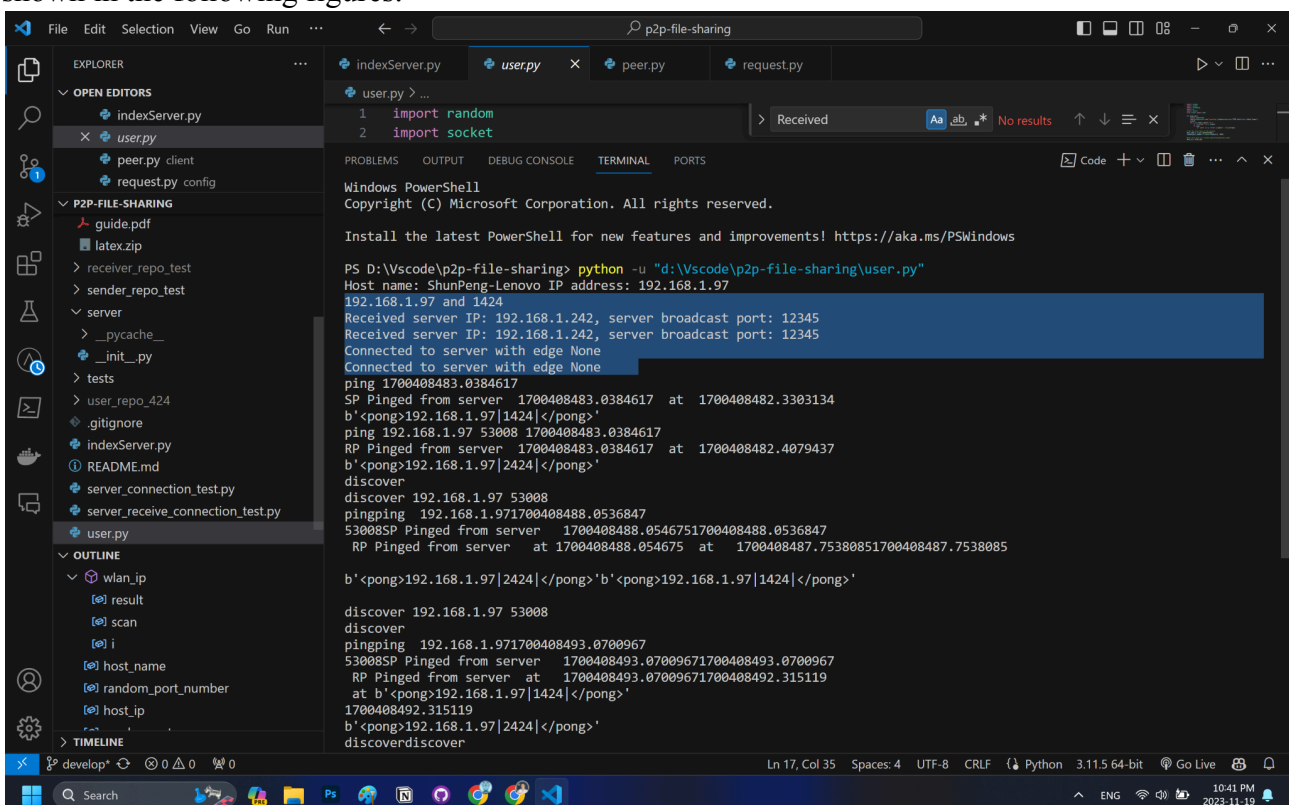
The screenshot shows a Visual Studio Code editor window titled 'p2p-file-sharing'. The Explorer panel on the left shows a project structure with folders like 'client', 'config', 'file\_collection', 'sender\_repo\_test', 'server', and 'tests'. The 'indexServer.py' file is open in the editor, showing the following code:

```
1 from server.server import Server
2
3
4
5
6 if __name__ == '__main__':
7     server = Server()
8
9     server.start()
10    while True:
11        if input() == "end":
12            server.stop()
13            break
```

The TERMINAL panel at the bottom shows the command prompt output:

```
PS D:\Vscode\p2p-file-sharing> & "C:/Users/Shun Peng/AppData/Local/Programs/Python/Python310/python.exe" d:/Vscode/p2p-file-sharing/IndexServer.py
192.168.1.242 12345
start listening...
start ping...
Server started
Pinging...
Pinging...
```

Then the Windows 11 PC connected the server broadcast ip and port, at which it got pinged after successfully connecting. A repository “user\_repo\_424” is created for this user. This can be shown in the following figures.



The screenshot shows a Visual Studio Code editor window titled 'p2p-file-sharing'. The Explorer panel on the left shows a project structure with folders like 'OPEN EDITORS', 'P2P-FILE-SHARING', 'receiver\_repo\_test', 'sender\_repo\_test', 'server', 'tests', and 'user\_repo\_424'. The 'user.py' file is open in the editor, showing the following code:

```
1 import random
2 import socket
```

The TERMINAL panel at the bottom shows the command prompt output:

```
PS D:\Vscode\p2p-file-sharing> python -u "d:\Vscode\p2p-file-sharing\user.py"
Host name: ShunPeng-Lenovo IP address: 192.168.1.97
192.168.1.97 and 1424
Received server IP: 192.168.1.242, server broadcast port: 12345
Received server IP: 192.168.1.242, server broadcast port: 12345
Connected to server with edge None
Connected to server with edge None
ping 1700408483.0384617
SP Pinged from server 1700408483.0384617 at 1700408482.3303134
b'<pong>192.168.1.97|1424|</pong>'
ping 192.168.1.97 53008 1700408483.0384617
RP Pinged from server 1700408483.0384617 at 1700408482.4079437
b'<pong>192.168.1.97|2424|</pong>'
discover
discover 192.168.1.97 53008
pingping 192.168.1.971700408488.0536847
53008SP Pinged from server 1700408488.0546751700408488.0536847
RP Pinged from server at 1700408488.054675 at 1700408487.75380851700408487.7538085
b'<pong>192.168.1.97|2424|</pong>'b'<pong>192.168.1.97|1424|</pong>'
discover
discover 192.168.1.97 53008
pingping 192.168.1.971700408493.0700967
53008SP Pinged from server 1700408493.07009671700408493.0700967
RP Pinged from server at 1700408493.07009671700408492.315119
at b'<pong>192.168.1.97|1424|</pong>'
1700408492.315119
b'<pong>192.168.1.97|2424|</pong>'
discoverdiscover
```

```

1  from server.server import Server
2
3
4
5
6  if __name__ == '__main__':
7      server = Server()
8
9      server.start()
10     while True:
11         if input() == "end":
12             server.stop()
13             break

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

Pinging...  
Pinging...  
Pinging...  
Pinging...  
Pinging...  
Pinging...  
Pinging...  
Pinging...  
Accepted socket with address ('192.168.1.97', 53009)  
Accepted socket with address ('192.168.1.97', 53008)  
Pinging...

Next, the Windows 10 PC connects to the server, a repository “user\_repo\_614” is created for this user.

```

1  from server.server import Server
2
3
4
5
6  if __name__ == '__main__':
7      server = Server()
8
9      server.start()
10     while True:
11         if input() == "end":
12             server.stop()
13             break

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

Pinging...  
Pinging...  
Pinging...  
Pinging...  
Pinging...  
Pinging...  
Pinging...  
Pinging...  
Accepted socket with address ('192.168.1.242', 61478)  
Accepted socket with address ('192.168.1.242', 61479)  
Pinging...

P5 D:\Vscode\p2p-file-sharing> & "C:/Users/Shun Peng/AppData/Local/Programs/Python/Python310/python.exe" d:/Vscode/p2p-f11 e-sharing/user.py  
Host name: ShunPeng-Acer IP address: 192.168.1.242  
192.168.1.242 and 1614  
Received server IP: 192.168.1.242, server broadcast port: 123  
45  
Received server IP: 192.168.1.242, server broadcast port: 123  
45  
Connected to server with edge None  
Connected to server with edge None  
ping 192.168.1.242 61478 1700408543.1730003  
ping 1700408543.1730003

At this time, this user publishes the “guide.pdf” file, which can be seen in the “user\_repo\_614” repository. The server is now storing the information of this user having the file.

The screenshot shows a VS Code window titled 'p2p-file-sharing'. The Explorer pane on the left shows a directory structure with 'user\_repo\_614' selected. The main editor shows 'guide.pdf' with PDF content. The TERMINAL pane at the bottom shows a series of 'Pinging...' messages and a successful connection to the server at 1700408693.5063145.

Then, the Window 11 user will fetch the file, since it is large, it takes many tcp segments to get the file. Now it can be shown in the “user\_repo\_424” having the guide.pdf file.

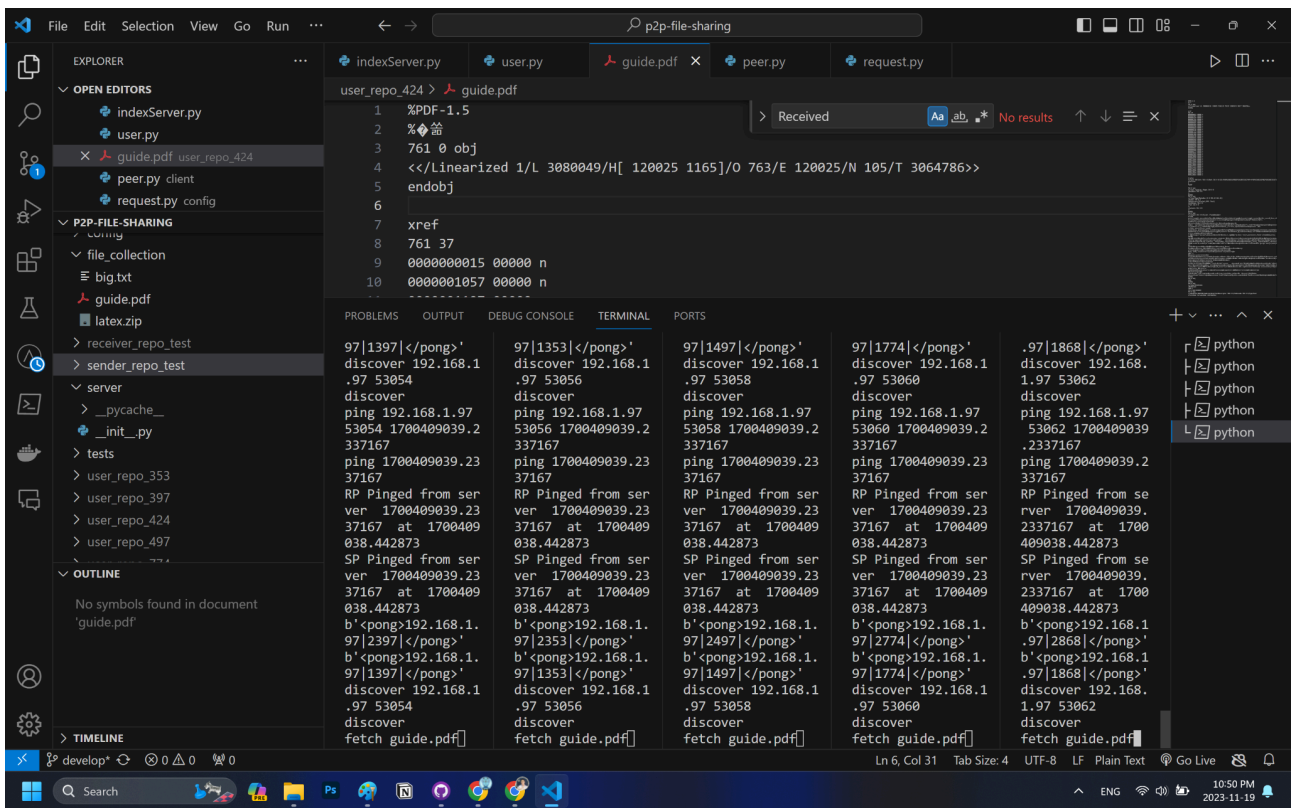
The screenshot shows a VS Code window titled 'p2p-file-sharing'. The Explorer pane on the left shows a directory structure with 'user\_repo\_424' selected. The main editor shows 'guide.pdf' with PDF content. The TERMINAL pane at the bottom shows a series of 'receive:' messages and a final 'Receiving file completed.' message.

Now this client tries to disconnect from the server. The associated directory is deleted and peer connection is closed.

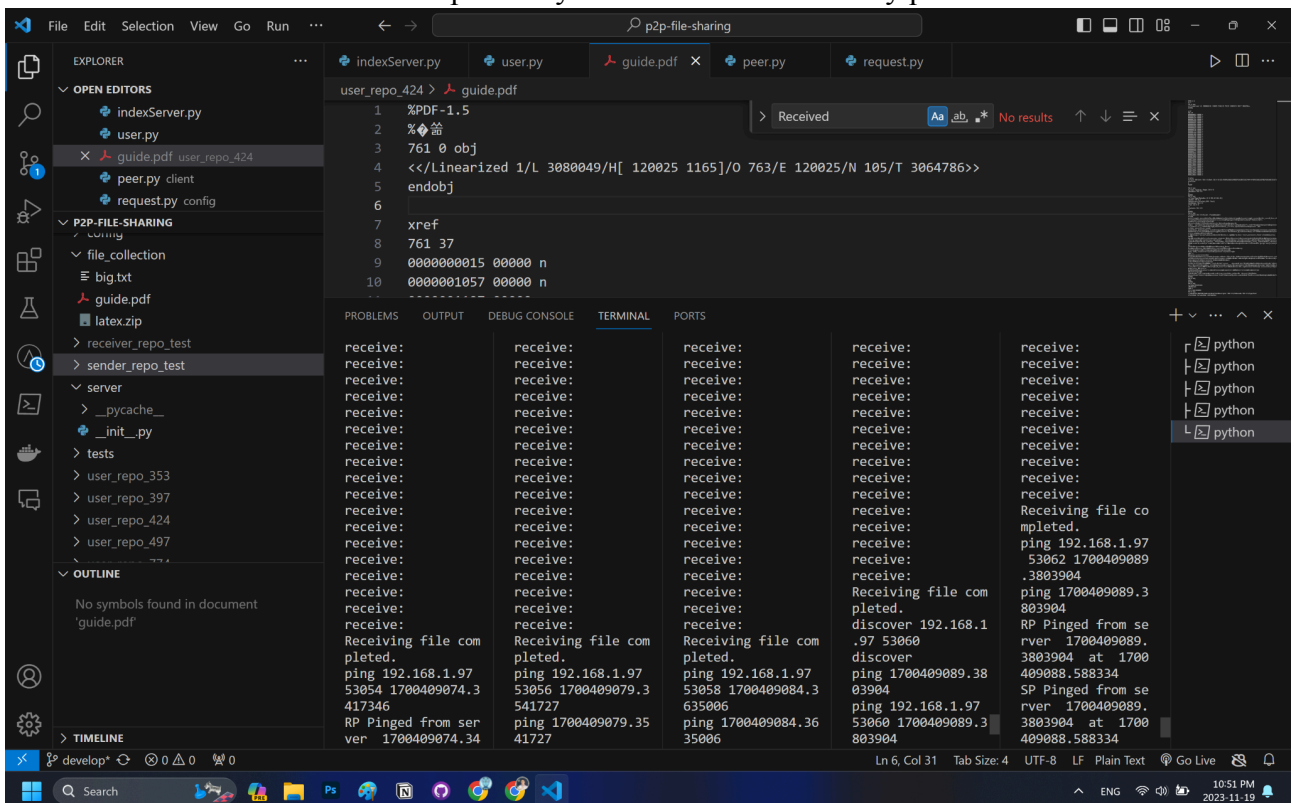
The screenshot shows a Visual Studio Code window with a project named 'p2p-file-sharing'. The Explorer panel on the left shows the project structure, including files like 'indexServer.py', 'user.py', 'guide.pdf', 'peer.py', and 'request.py'. The main editor displays the content of 'guide.pdf', which is a PDF document. The Terminal panel at the bottom shows the output of the application, including ping commands and connection status messages.

On the other hand, let's test 5 users at the same time to check the performance. All users are now trying to fetch the “guide.pdf” at the same time.

The screenshot shows a Visual Studio Code window with a project named 'p2p-file-sharing'. The Explorer panel on the left shows the project structure, including files like 'user.py', 'indexServer.py', 'guide.pdf', and 'server.py'. The main editor displays the content of 'guide.pdf'. The Terminal panel at the bottom shows the output of the application, including ping commands and connection status messages for multiple users.



It can be seen that all users parallelly fetch the file without any problems.



- **Evaluation:**

The connection between every clients and the server is established via 4G Wifi hotspot

File size	Time to transfer
6463KB	3 second
3138KB	2 second
12566KB	8 second

$$\text{Average throughput : } \frac{6463 + 3138 + 12566}{3+2+8} = 1.705 \text{ (KBps)}$$

## 10. Future development direction

- Design a proper UI for a better user experience as well as making space for other future functionalities.
- Current file sending method is linear, which is too slow if the user wants to send large files (>2Gb) over the network. We can improve on this by using multiple threads to send multiple segments of the file at the same time.
- Current application lacks the ability to verify if a peer is allowed to share/download a file from another peer which could lead to some security issues. We intend to fix this with a dedicated authentication system in the future.
- Documentation for the application right now is not what we wanted it to be. In the future, we want to add more in-depth details about how the application work as well as a better user tutorial to help future developers and users feel more comfortable developing/using the application.