

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



COMPUTER NETWORKS LAB (CO3094)

---

Assignment 1

# DEVELOP A NETWORK APPLICATION

---

**Advisor:** Nguyễn Mạnh Thìn  
**Students:** Phạm Duy Tường Phước - 2252662  
Phan Hồng Quân - 2252685

HO CHI MINH CITY, MAY 2024



## Contents

<b>1</b>	<b>Member list &amp; Workload</b>	<b>2</b>
<b>2</b>	<b>Overview</b>	<b>2</b>
2.1	Requirement . . . . .	2
2.2	Implementation . . . . .	3
2.2.1	Server . . . . .	3
2.2.2	Client . . . . .	3
<b>3</b>	<b>Class Diagram</b>	<b>4</b>
<b>4</b>	<b>Application Instructions Manual</b>	<b>5</b>
4.1	Activating server . . . . .	5
4.2	Running peer . . . . .	6
<b>5</b>	<b>Define and describe the functions of the application</b>	<b>6</b>
5.1	Server (Tracker) . . . . .	6
5.2	Client (Peer) . . . . .	7
<b>6</b>	<b>Validation (sanity test) and evaluation of actual result</b>	<b>8</b>
<b>7</b>	<b>Future development direction</b>	<b>12</b>

## 1 Member list & Workload

No.	Fullname	Student ID	Problems	Percentage
1	Phạm Duy Tường Phước	2252662	- Client and Server architecture construction - Write report	50%
2	Phan Hồng Quân	2252685	- File and Piece construction - Write report	50%

## 2 Overview

### 2.1 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 try to connect to as many peer as possible in an attempt to download a file, all happen concurrently without server intervention. A peer is also capable of letting multiple peers downloading simultaneously files from it.

Figure 1. Illustration of file-sharing system.

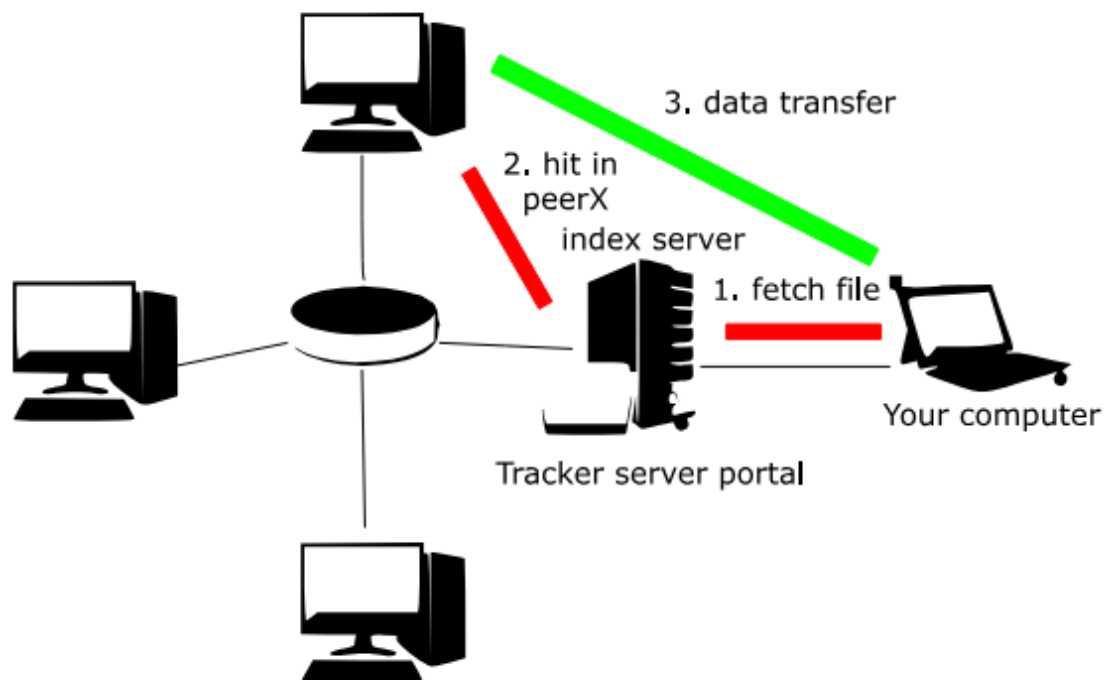


Figure 1: Applicaton summary

## 2.2 Implementation

In accordance with the project requirements, our group has successfully implemented two essential program files: the `tracker.py` (represent the server) and the `peer.py` (represent 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. And also, all the communication in this application is all conducted via TCP protocol.

### 2.2.1 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 would keep meta info of each file that a client has and would use that as the source for the response generated for each request from a client.
- The server would request other peers to update the status of their files in storage every time an update request sent from an arbitrary client.

### 2.2.2 Client

- Upon initiation, the client program embarks on a series of steps to ensure its seamless integration into the filesharing 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.
- Upon initiation, the client program would require user an input for the port number that the client program would run on. This would create or connect to the appropriate respiratory.
- Within the commandline 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 start connecting to the server, update the file available in the network, request and download a target file from the network.
- The command "start" would create a TCP connection from the peer to the server, giving information about files that are in the peer's respiratory. The server would send back information about what files are available in the network for the peer to download. This would complete the handshake progress for the two client and server.
- The command "update" would send the newest status of the client's respiratory to the server. When the server receive this message it would also send the update request to other clients in the network, each would in turn response with newest information of its respiratory. This command should be used before any decision of downloading a file is made to ensure the reliable synchronization with the file actually in the network.
- The command "request\_and\_download" would send the request to download the specific file to the server. In response, the server sends back a list of peer that has the file in its local directory. The, it is the responsibility of the client to contact to as many clients as

possible to get the target file, each client is concurrently required a piece until there is no piece left for downloading

- The command “stop” would stop the client from continuing downloading from other client by sending a closing connection for each client that it is downloading from.
- The command “end” would stop all the actions that the client is performing including connecting to server. It is recommended to use this function to shut down a client for reality concern

### 3 Class Diagram

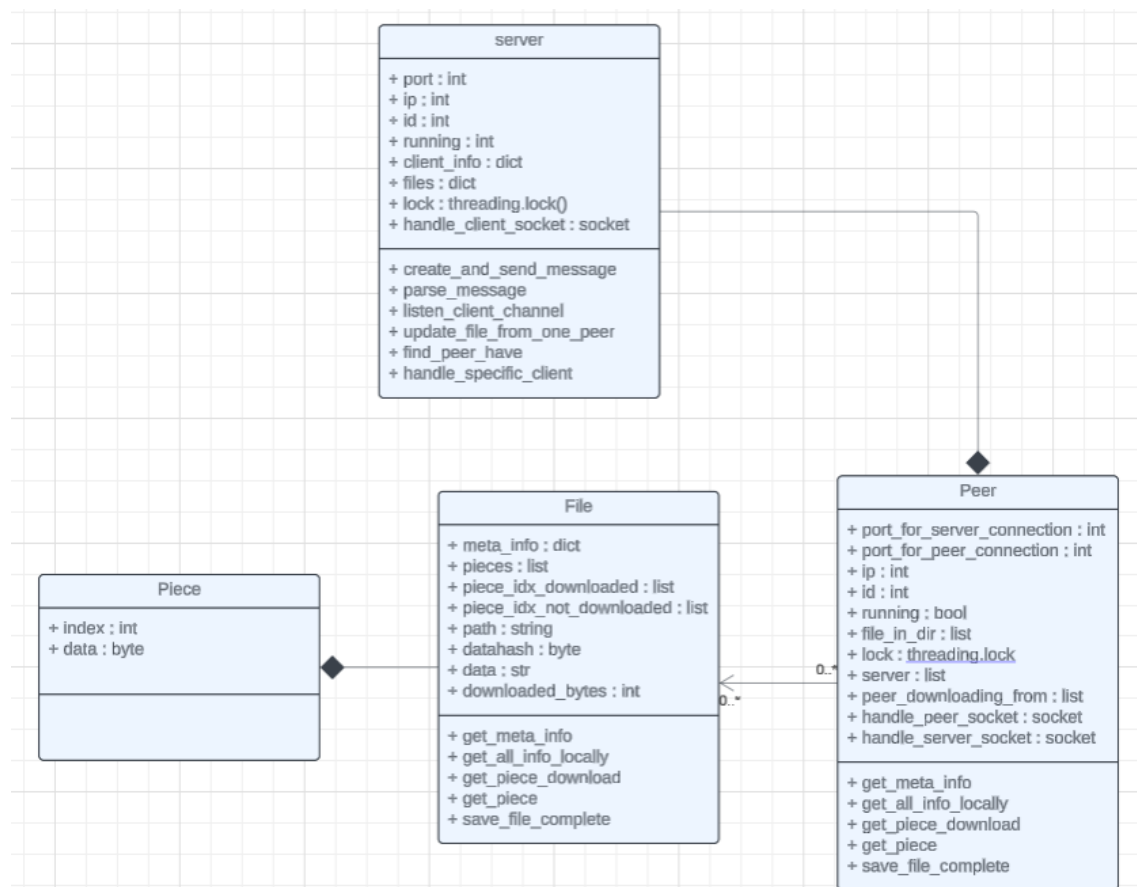


Figure 2: Class Diagram

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** in couple with all the information about each peer's sharing files.

- The **main attribute of the server is the client\_info** (including all the information about each client connecting to the server and information about files that each client is ready to share) and the files (this includes all the files that are available for downloading from other clients)
- The **Peer class has attributes for the host, ports, and file list**. It has some basic methods to connect to the main server, handle sending requests to the server, sending messages to other clients, receiving request or uploading file for other clients. Further, each client is also capable of reading files from its own respiratory.
  - **Important features of the peers is the file\_in\_dir where it keeps all the information about files** in its respiratory which is used for two most major behavior, submitting to the server and also for downloading pieces from others.
- The Piece class has the attributes to store its data of the file and index in the file chunks of pieces.
  - The most considerable factors of a piece is its index and the data it contains.
- The File class has the attributes to store and manage the pieces of a file. Also this class also helps to read and write files from/to the peer's respiratory.
  - The most vital aspect of this class is its list of `piece_idx_downloaded` and `piece_idx_not_downloaded`. These helps manage the status of the files

## 4 Application Instructions Manual

For efficiency, it is recommended to run the program with Python version 3.0.

### 4.1 Activating server

- It is important to firstly start the server before any steps can be taken further.
- The server needs just only the first kickstart, then it can be left untouched until the end of all other things
- To start the server, it is enough to run the file *tracker.py*: **python tracker.py**
- A successful invoking server would look like this on the screen:

```
a server with IP address 192.168.1.218, ID 12500 and port number 12500 is created
start listening...
start listening...
█
```

Figure 3: Invoke server

## 4.2 Running peer

- To start running a peer program, first run the file *peer.py*: **python peer.py**
- After the first step, follow several step printed out on the screen (requiring the input data for port number of the client and also the ip address, port number of the client for connecting) and there would be a peer ready for connecting to the server and download file from the network
- From that point on, “start”, “update”, “request”, “stop” and “end” are commands which should be typed to the terminal when an appropriate behavior is wished to be conducted. It is recommended to always have the “update” command run before the “request” command is made for data consistency on the network
- A successful invoking peer would look like this on the screen:

```
enter the port number for the peer: 12666
a peer with IP address 192.168.1.218, ID 12666 is created
Enter the command you want to execute: start
enter IP address of the server: 192.168.1.218
enter port number the server is listening on: 12500
Enter the command you want to execute: start listening for other peers...
{'type': 1, 'sid': '12500', 'file': {}}
files available for downloading:
type::1;sid::12666;file::{'hello_big.txt': [[0, 13], {'length': 1997, 'name': 'hello_big.txt', 'num_of_pieces': 14, 'piece_length': 150}],
'hello_big3.txt': [[0, 79], {'length': 11984, 'name': 'hello_big3.txt', 'num_of_pieces': 80, 'piece_length': 150}], 'hello_big2.txt': [[0,
, 26], {'length': 3994, 'name': 'hello_big2.txt', 'num_of_pieces': 27, 'piece_length': 150}], 'hello.txt': [[0, 0], {'length': 23, 'name':
'hello.txt', 'num_of_pieces': 1, 'piece_length': 150}]}
```

Figure 4: Running peer

## 5 Define and describe the functions of the application

In this section, we try to provide summarized usage of each function in the application. Note that every communication-purposed function in this application is implemented via TCP protocol.

### 5.1 Server (Tracker)

- **Create\_and\_send\_message():** Two compulsory arguments for this function is the socket of the connection to target address and the type of the message. This function would **automatically form a message in the form of string** consisting of data added in the argument, then, it would **encode it by 'utf-8'** and **send it to the socket for the target connection**. The message created would have the structure like a dictionary, each key-value pair would be in the form “key:value” and each pair would be separated by “;” for easy parsing later.
- **Parse\_message():** This function take in the message sent from clients and **decode it using 'utf-8'**. Then it would break the message into a real dictionary by split the message into pieces following the structure known beforehand in the *create\_and\_send\_message*. Several important fields would be “type” indicating the type of the message, “sid” tells information about the source id of the message.
- **Listen\_client\_channel():** This function represent a socket always waiting to accept any client that asks to set up a connection to server. After accepting the request, this

function would create a new distinct thread (named *handle\_specific\_client()*) to handle all the message sent from this client from that moment till the end

- ***Handle\_specific\_client()***: This function is created to handle messages from only one client. From here, server would receive information about each peer's respiratory status and build up a information network that would be used to support other client downloading need. In addition to collecting, this function is also where the server distribute the information for the request client.
- ***Update\_file\_from\_one\_peer()***: This function created as support for the main function *handle\_specific\_client()*. This would take information about the peer and the file of it and update this to the information network.
- ***Find\_peer\_have()***: This is another supportive function which is mainly used for find out which peer in the network is containing the information relating to the related file. It would return a list of peers that satisfy the requirement.

## 5.2 Client (Peer)

- ***Create\_and\_send\_message()***: Two compulsory arguments for this function is the socket of the connection to target address and the type of the message. This function would automatically form a message in the form of string consisting of data added in the argument, then, it would encode it by 'utf-8' and send it to the socket for the target connection. The message created would have the structure like a dictionary, each key-value pair would be in the form "*key::value*" and each pair would be separated by ";" for easy parsing later.
- ***Parse\_message()***: This function take in the message sent from clients and decode it using 'utf-8'. Then it would break the message into a real dictionary by split the message into pieces following the structure known beforehand in the *create\_and\_send\_message()*. Several important fields would be "type" indicating the type of the message, "sid" tells information about the source id of the message.
- ***Get\_file\_in\_dir()*** This function would dive into the peer's respiratory and list out the existing file in this place including the *meta\_info*, *current status of the file*, *information of each piece*.
- ***Connect\_server\_channel()*** This function would initially set the connection to the server, then if successful, all the remaining messages aiming to the server is all sent through this connection. Any notice produced by the server would also be receive through this connection only.
- ***Listen\_peer\_channel()*** This function represent a socket always waiting to accept any client that asks to set up a connection to this peer for downloading purpose. After accepting the request, this function would create a new distinct thread (named *handle\_specific\_peer()*) to handle all the message sent from this client from that moment till the end
- ***Handle\_specific\_peer()*** This function is created to handle messages from only one client. From here, the host client would receive request information about which piece of what file the request client is demanding. It then checks for the availability of the piece in its respiratory then would send it there exists, otherwise, a 'NO FOUND' message is sent.

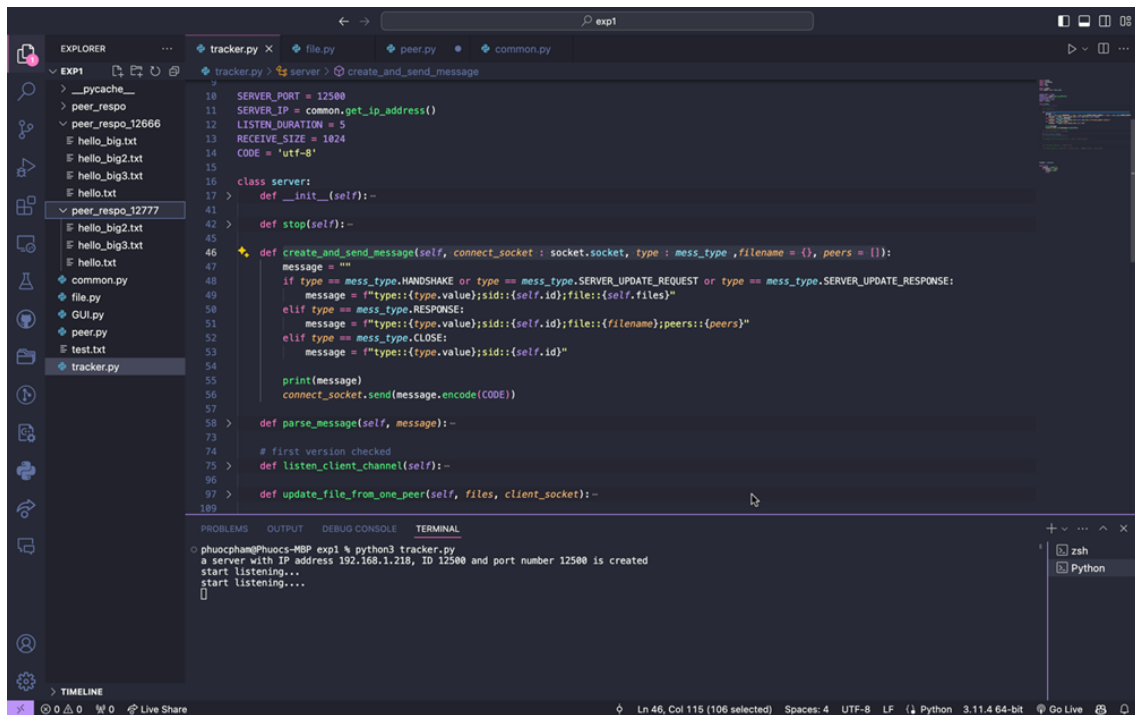


- ***Download\_from\_peer\_func()*** This function send request for downloading file pieces from target peer. Each time, a peer only request only one piece not downloaded yet from the target peer. If the needed data is not provided it would ask for another piece of data.
- ***Get\_download\_file()***: This is a support function for *download\_from\_peer\_func()*. This would return a file that the client want to download. If the file has already existed in its respiratory, it would return a file with all the status information about the file(what pieces have been already downloaded, meta\_info), otherwise, it create a new file in the respiratory return that file ready to be downloaded.
- ***Close\_connection()***: This would close the connection to all the peers that this peer is connecting to, preparing for shutdown
- ***Request\_and\_download\_file()***: This function would send a request for a target to the server. The server would response with a list of peer containing the appropriate data. Then in this function also, the peer would create many threads (the thread would run the *download\_from\_peer\_func()*) trying to connect to as many peers as possible in order to download the file.
- ***Start()***: This function would kickstart the peer by trying to connect to the server in order to join into the network.
- ***Stop()***: This would stop all the client's activity
- ***Update\_file\_for\_download()***: This function would send a request for updating the information about downloadable files to the server. The server would in turn send an update request to other peer requiring each to hand in the up-to-date information of its respiratory file. After receiving updated information from all the peers, the server would send back to the request client a new list of files available for downloading from the network. This function plays an important roles in maintaining the network data consistency

## 6 Validation (sanity test) and evaluation of actual result

In this section, we are going to demonstrate a sanity test sending and receiving files between three clients. This network can work well within a local WiFi network, however, for simplicity of the report we would demonstrate here how it run in one computer first.(This results in the same process and result throughout the demonstration)

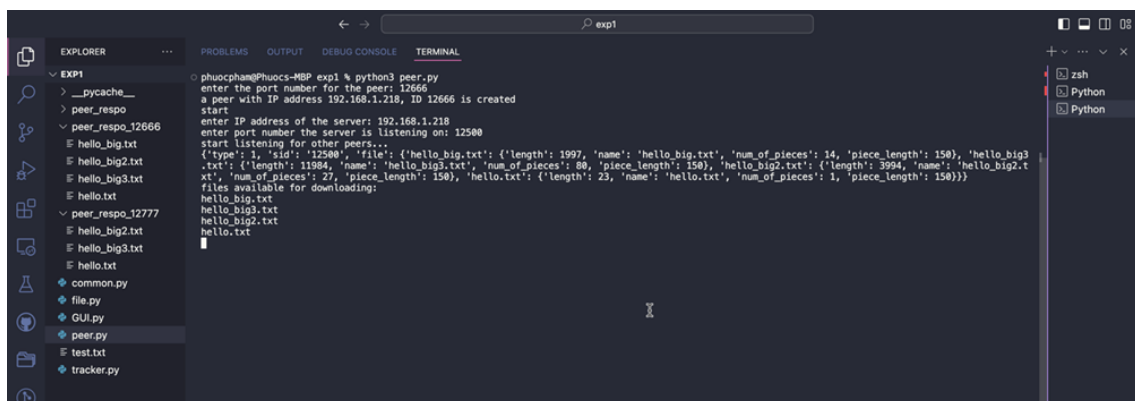
1. ***Starting the server (tracker).***



The screenshot shows the VS Code editor with the file explorer on the left. The file explorer shows a project named 'exp1' with a directory structure including 'peer\_respo\_12666', 'peer\_respo\_12777', and 'common.py'. The main editor displays the code for 'tracker.py'. The code defines a 'server' class with methods for initialization, stopping, creating and sending messages, parsing messages, and listening for client channels. The terminal at the bottom shows the command 'python3 tracker.py' being executed, and the output indicates that the server is listening on port 12500.

Figure 5: Start the server

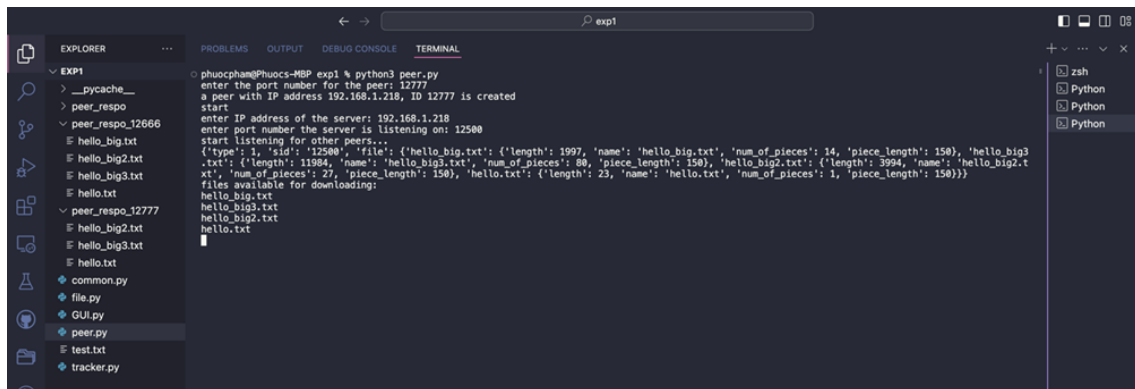
2. *Starting two peers 12666 and 12777 with already created directory with files for uploading. After successfully connect to the server, the peer receive a list of file available in the network*



The screenshot shows the VS Code editor with the file explorer on the left. The file explorer shows a project named 'exp1' with a directory structure including 'peer\_respo\_12666', 'peer\_respo\_12777', and 'common.py'. The main editor displays the code for 'peer.py'. The code defines a 'peer' class with methods for initialization, stopping, and listening for messages from the server. The terminal at the bottom shows the command 'python3 peer.py' being executed, and the output indicates that the peer is listening on port 12500. The peer receives a list of files available in the network, including 'hello\_big2.txt', 'hello\_big3.txt', and 'hello.txt'.

Figure 6: Peer received a list of files available

3. *Star a new piece with no already existed directory*

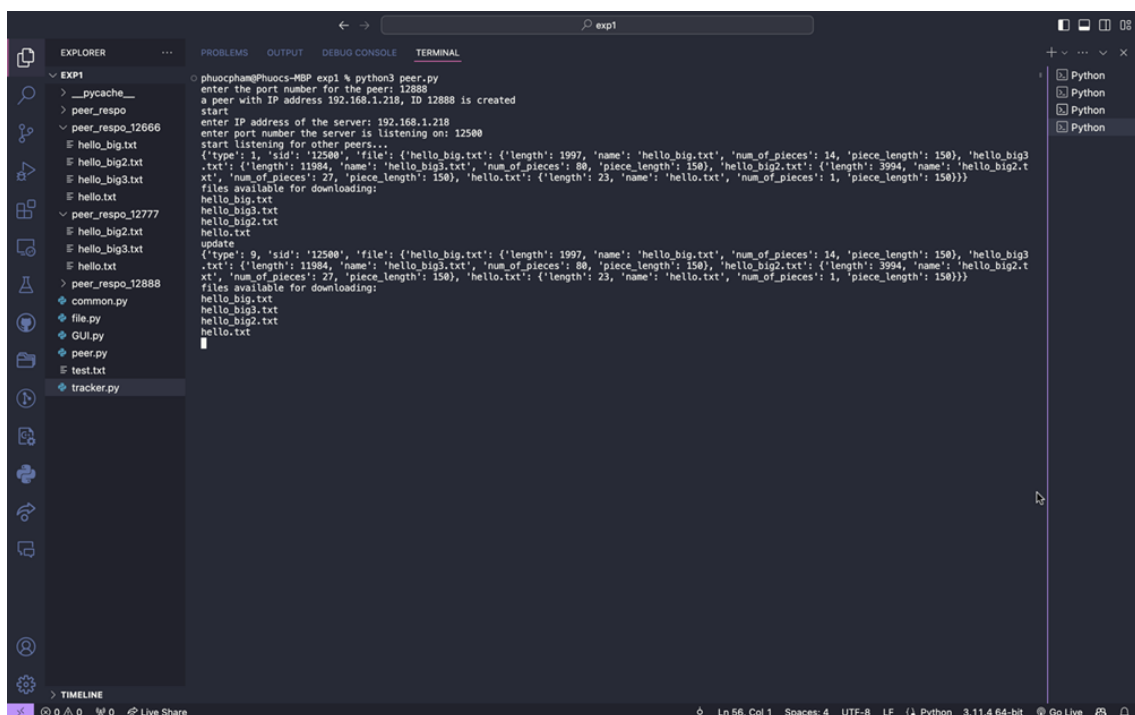


```

phucpham@Phuocs-MBP exp1 % python3 peer.py
enter the port number for the peer: 12777
a peer with IP address 192.168.1.218, ID 12777 is created
start
enter IP address of the server: 192.168.1.218
enter port number the server is listening on: 12500
start listening for other peers...
{'type': 1, 'sid': '12500', 'file': {'hello_big.txt': {'length': 1997, 'name': 'hello_big.txt', 'num_of_pieces': 14, 'piece_length': 150}, 'hello_big3.txt': {'length': 11904, 'name': 'hello_big3.txt', 'num_of_pieces': 80, 'piece_length': 150}, 'hello_big2.txt': {'length': 3994, 'name': 'hello_big2.txt', 'num_of_pieces': 27, 'piece_length': 150}, 'hello.txt': {'length': 23, 'name': 'hello.txt', 'num_of_pieces': 1, 'piece_length': 150}}}
files available for downloading:
hello_big.txt
hello_big3.txt
hello_big2.txt
hello.txt
  
```

Figure 7: Peer received a list of files available

#### 4. The update command is called before download a process



```

phucpham@Phuocs-MBP exp1 % python3 peer.py
enter the port number for the peer: 12888
a peer with IP address 192.168.1.218, ID 12888 is created
start
enter IP address of the server: 192.168.1.218
enter port number the server is listening on: 12500
start listening for other peers...
{'type': 1, 'sid': '12500', 'file': {'hello_big.txt': {'length': 1997, 'name': 'hello_big.txt', 'num_of_pieces': 14, 'piece_length': 150}, 'hello_big3.txt': {'length': 11904, 'name': 'hello_big3.txt', 'num_of_pieces': 80, 'piece_length': 150}, 'hello_big2.txt': {'length': 3994, 'name': 'hello_big2.txt', 'num_of_pieces': 27, 'piece_length': 150}, 'hello.txt': {'length': 23, 'name': 'hello.txt', 'num_of_pieces': 1, 'piece_length': 150}}}
files available for downloading:
hello_big.txt
hello_big3.txt
hello_big2.txt
hello.txt
update
{'type': 9, 'sid': '12500', 'file': {'hello_big.txt': {'length': 1997, 'name': 'hello_big.txt', 'num_of_pieces': 14, 'piece_length': 150}, 'hello_big3.txt': {'length': 11904, 'name': 'hello_big3.txt', 'num_of_pieces': 80, 'piece_length': 150}, 'hello_big2.txt': {'length': 3994, 'name': 'hello_big2.txt', 'num_of_pieces': 27, 'piece_length': 150}, 'hello.txt': {'length': 23, 'name': 'hello.txt', 'num_of_pieces': 1, 'piece_length': 150}}}
files available for downloading:
hello_big.txt
hello_big3.txt
hello_big2.txt
hello.txt
  
```

Figure 8: The update command is called before download a process

#### 5. Download request is made for the biggest file available. The peer outputs which piece is being requested from which peer. From the screen, we can see that this peer is requesting from concurrently two peers having the files in the network.

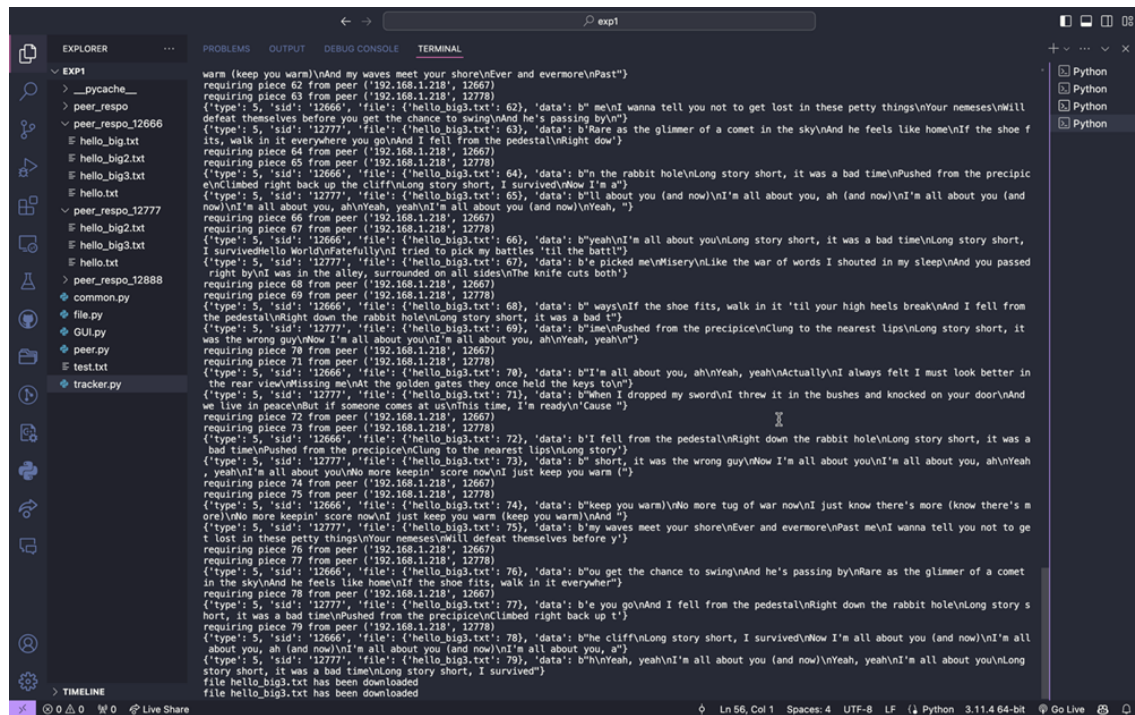


Figure 9: Download request

6. *Checking the receive file and see that it has the exactly same content as the source. This announce the success in conducting file sharing in the network*

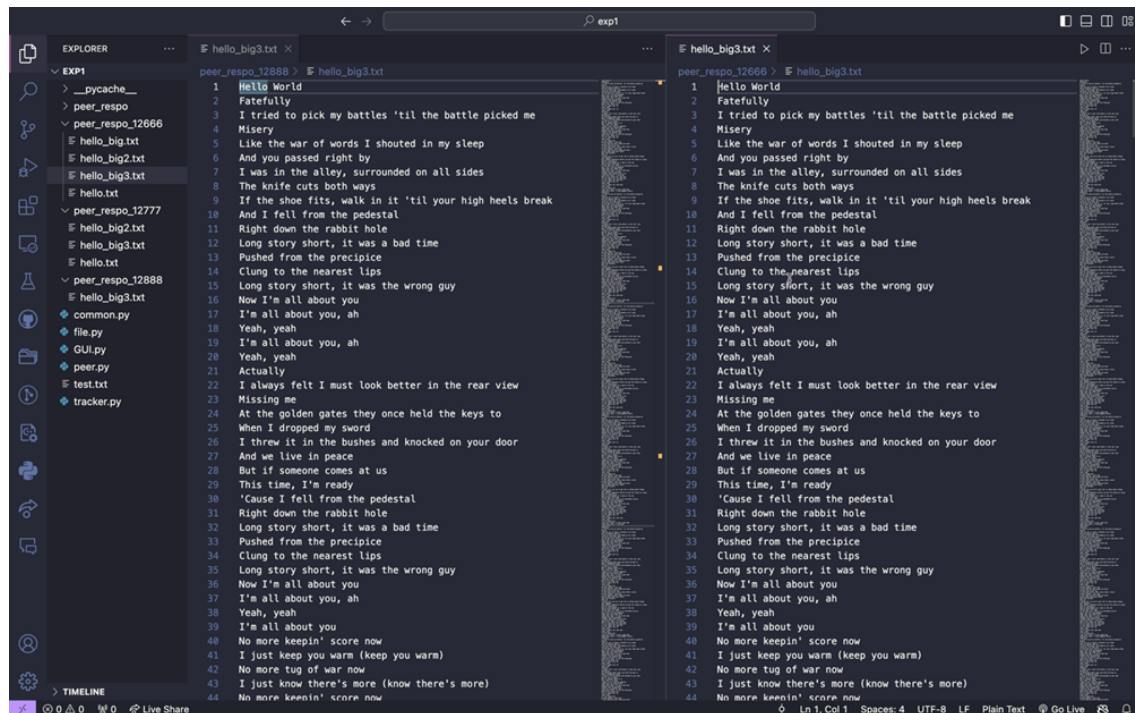


Figure 10: Checking the received file

### Evaluation of the requirement satisfaction of the application:

- The file is transferred correctly to the server.
- The file is correctly parsed into small pieces and then be put together correctly to produce the same file as the source.
- The peer successfully follows the multi-downloading pieces from many peers

## 7 Future development direction

- Design a proper UI for a better user experience as well as making space for other future functionalities.
- 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.
- The algorithm in request the piece from other peers is not optimized. We would love to enhance the efficiency of this algorithm