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

COMPUTER NETWORK

Assignment 1

Building a network application

Advisor:	Nguyễn Lê Duy Lai	
Students:	Nguyễn Thành Đạt	2252145
	Phạm Huỳnh Bảo Đại	2252139

HO CHI MINH CITY, DECEMBER 2024

Contents

1 Introduction

In Computer Networking, P2P (Peer-to-Peer) is a file-sharing technology that allows users to access mainly the multimedia files like videos, music, ebooks, games, etc. Individual users in this network are referred to as peers. The peers request files from other peers by establishing TCP or UDP connections.

This assignment requires to build a Peer-to-Peer file sharing application. The application that allows users to download and upload files concurrently from and to multiple peers in the network. This application will implement a centralized tracker server that keeps track of connected clients and the files they host, ensuring that users can locate and access the content.

By completing this assignment, we will not only gain practical experience in network programming and application development, but also deepen our understanding of the underlying mechanisms that drive P2P file sharing systems. This report will outline the design, implementation, and evaluation of our application.

2 Application Overview

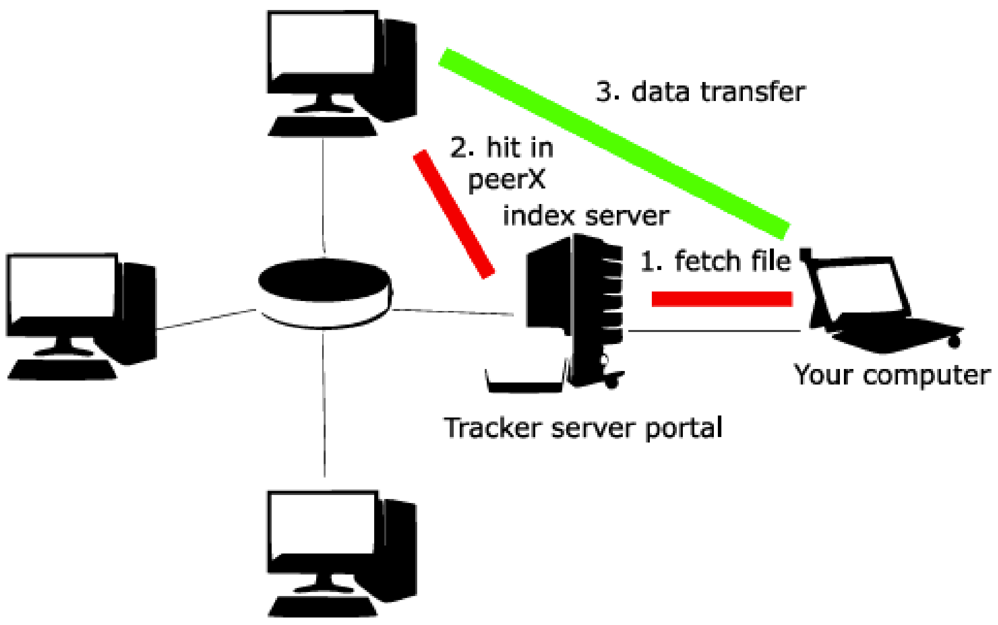


Figure 2.1: Visualize the process in P2P File sharing application.

Tracker Server: The tracker acts as a centralized server. It must always be running on a host machine, ready to accept connections from clients (peers). The clients don't send the actual file to the tracker. Instead, they simply announce their presence in the peer network and share information (metadata) about the files they are offering for download.

Peer (Client): When a client wants to join the peer network, it first connects to the tracker. There are two main types of interactions between the client (peer) and the tracker:

- **Fetch:** The peer sends a request to the tracker over the TCP protocol. This request includes the metadata (metainfo) of the file the peer wants to download. The tracker then searches the peer network for the requested file and responds with a list of peers that currently have the file available for sharing. This list contains the information needed for the peer to start downloading from those peers.

- **Publish:** In this interaction, the peer informs the tracker that it is now sharing a file. The peer sends the file's metadata to the tracker, which updates its records to indicate that the file is available from this peer. Importantly, only the metadata is sent to the tracker—not the actual file itself.

Communication between Peers: Once a peer receives the peer list from the tracker, the requesting peer (the one initiating the download) needs to establish connections with other peers and send them a request message to begin downloading. In this process, there are two key aspects to focus on:

- **Peer as a Host:** Each peer must also act as a server, waiting for incoming connections from other peers, similar to the role of the tracker. This functionality runs on a separate thread, allowing the main thread to continue handling fetch and publish operations without interruption.
- **Handling Multiple Peers for Downloading:** To efficiently handle multiple peers, we use a round-robin approach. A matching function is implemented to pair each peer with specific pieces of the file. The file is divided into smaller pieces (typically 512KB each) to speed up the download process. Once matched, the requests are sent to the peers, allowing for concurrent downloads from multiple peers.

3 Requirements

3.1 Functional Requirements

3.1.1 Tracker Requirements

Data Management of Clients: The server acts as a central manager for the file-sharing system, tracking connected clients and the files they possess. This requires the peer info to be stored in the server 's data correctly.

- peer_list : this will store the peer information include peer_id, peer_name, peer_host, peer_port and shared_file which store the files they shared.
- file_sharing : this will store the name of the file have been shared, and the peer_id that shared them.
- file_metadata : this will match the file with the information of the file, this information will be generated from the peer send the request to publish the file.

Multi-threading: The server is required to handle multiple peers simultaneously, so the code should be designed to support multithreading. Each incoming peer connection is handled in a separate thread, allowing the server to serve many peers concurrently without performance bottlenecks. This ensures that file upload and download processes can proceed in parallel, improving the overall efficiency of the file-sharing system.

Client Request Handling: The server responds to client requests in a server-client architecture. When a client sends a request, the server processes it and sends back the appropriate response.

3.1.2 Client Requirements

Notification of Shareable Files: Clients inform the server about the files they can share. If connected to the server, the client can publish file names one by one. Otherwise, an error is reported. The peers also send the metainfo of the file in the publish request.

File Search: Clients send a file name to the server for search. The server searches for the file among connected clients and in its database. If found, the server provides the client with the file's location (client hostname). If the requesting client wants to download the file, a direct peer-to-peer transfer is initiated without server intervention. If the file is not found, the system reports an error. The download process must be efficiently managed across multiple peers. To achieve this, we use a round-robin approach to distribute requests for different pieces of the file among the peers. This ensures that the workload is balanced, and all peers contribute to the download, which helps speed up the overall process.

Upload and Download with "publish" and "fetch" Commands: Clients support the "publish" and "fetch" commands:

- **publish file_name:** Adds the specified file to the server's database, including client and file information.
- **fetch file_name:** Sends a search request to the server. If found, the client receives connection information of the file's location and initiates a data transfer.

3.2 Non-Functional Requirements

- **Performance:** The system should support multiple concurrent connections by creating a thread for each connection, enhancing performance and throughput
- **User-Friendliness:** The system provides a simple interface with three commands ("publish", "fetch", "exit"), making it easy to use.
- **Graphical User Interface:** We use the command shell to process the the work, require the responses and messages send over each nodes clearly and understandable.

4 Architecture Design

4.1 Class Diagram

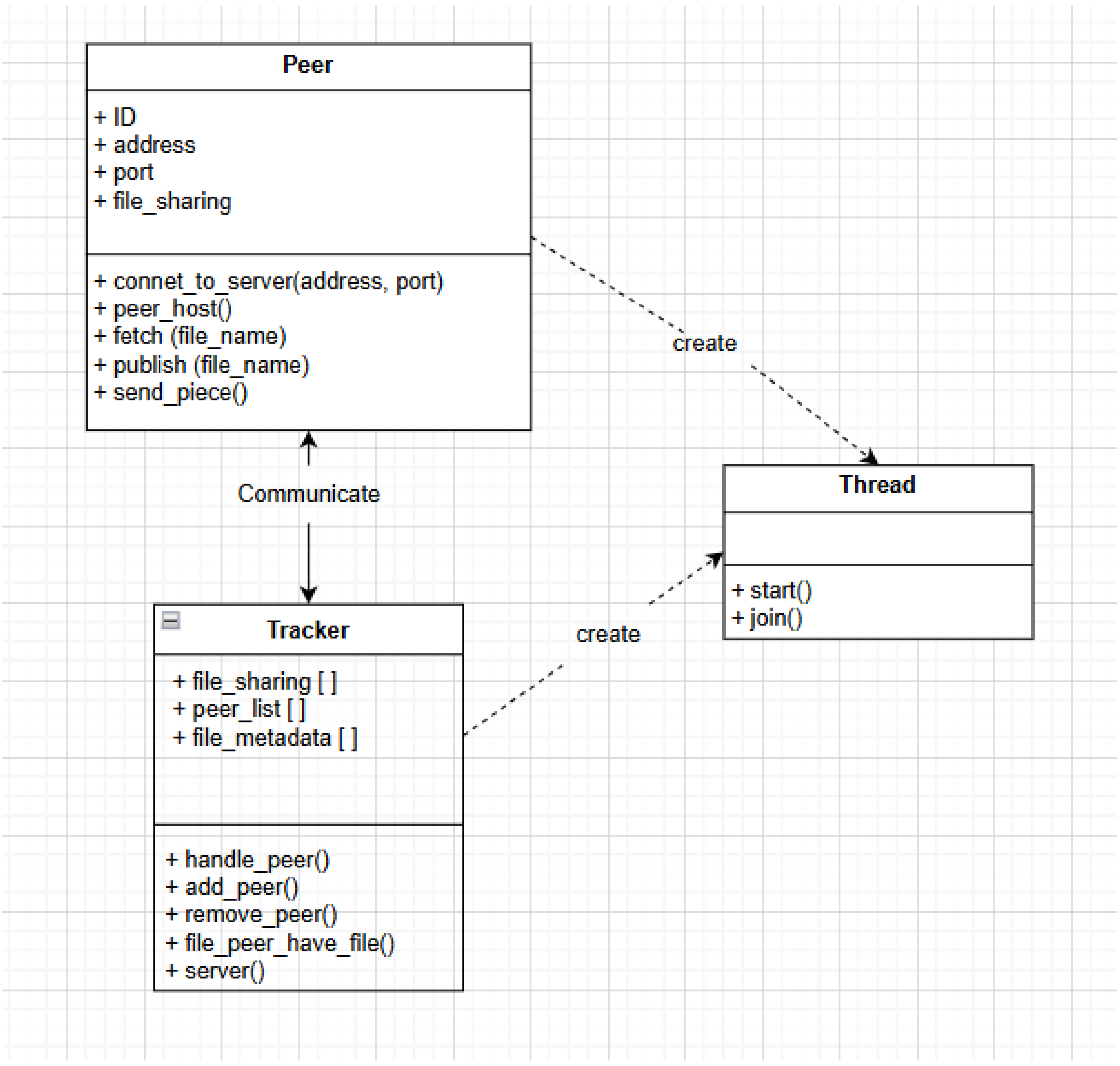


Figure 4.1: Class Diagram for application.

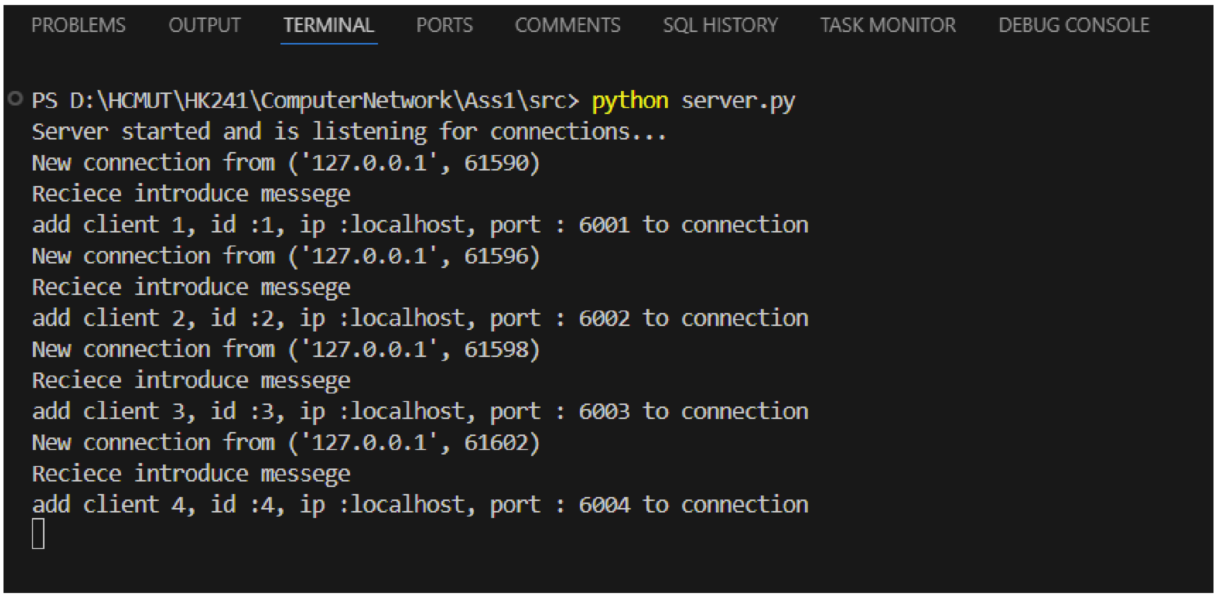
5 Result

The full code is on the GitHub link:
https://github.com/DatNguyen1402/ComputerNetwork_ass1
After cloning the repository, we can see the clients folder that will stand for the directory of each clients. In the assignment, I will use 4 clients to show how to application work.

5.1 Start up the server

First, to start the server, I run the command `python server.py`. Then, in separate terminal windows, I run `python client1.py`, `python client2.py`, `python client3.py`, and `python client4.py`, respectively.

In the image below, you can see that the server terminal shows the server starting up and waiting for connections. As we execute the commands on the clients, the clients will connect to the server one by one, establishing connections successfully.

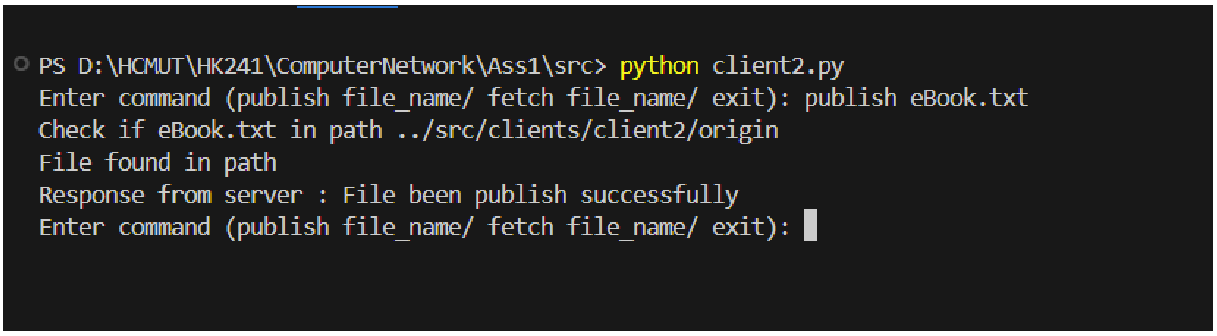


```
PS D:\HCMUT\HK241\ComputerNetwork\Ass1\src> python server.py
Server started and is listening for connections...
New connection from ('127.0.0.1', 61590)
Reciece introduce messege
add client 1, id :1, ip :localhost, port : 6001 to connection
New connection from ('127.0.0.1', 61596)
Reciece introduce messege
add client 2, id :2, ip :localhost, port : 6002 to connection
New connection from ('127.0.0.1', 61598)
Reciece introduce messege
add client 3, id :3, ip :localhost, port : 6003 to connection
New connection from ('127.0.0.1', 61602)
Reciece introduce messege
add client 4, id :4, ip :localhost, port : 6004 to connection
```

Figure 5.1: Start the connection and initialize the server.

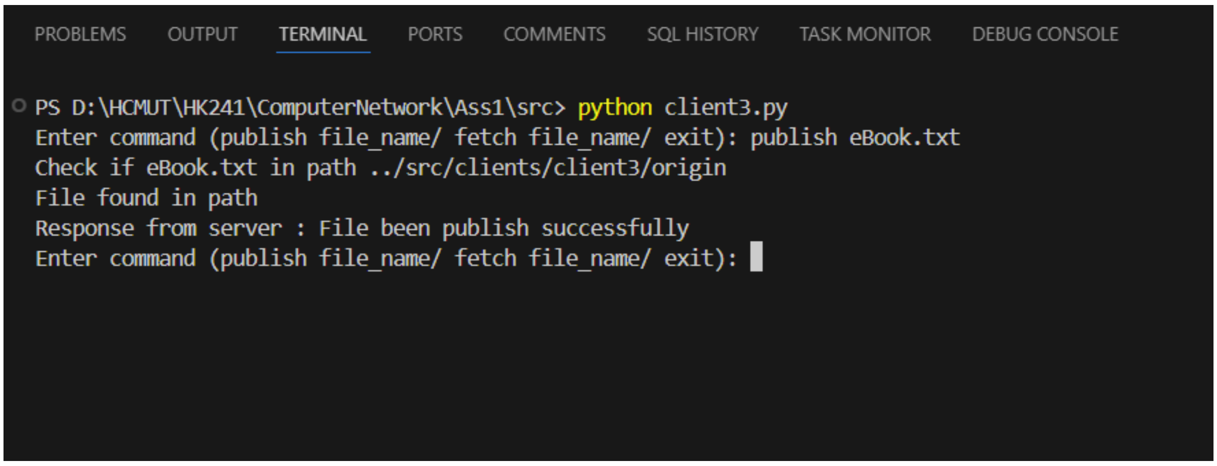
5.2 Case 1

In Case 1, I will demonstrate how the file is downloaded by client1. Initially, client1 does not have the file `eBook.txt`, but both client2 and client3 possess it. First, we run the `publish eBook.txt` command on both client2 and client3. As shown, the file is successfully published on both clients.



```
PS D:\HCMUT\HK241\ComputerNetwork\Ass1\src> python client2.py
Enter command (publish file_name/ fetch file_name/ exit): publish eBook.txt
Check if eBook.txt in path ../src/clients/client2/origin
File found in path
Response from server : File been publish successfully
Enter command (publish file_name/ fetch file_name/ exit):
```

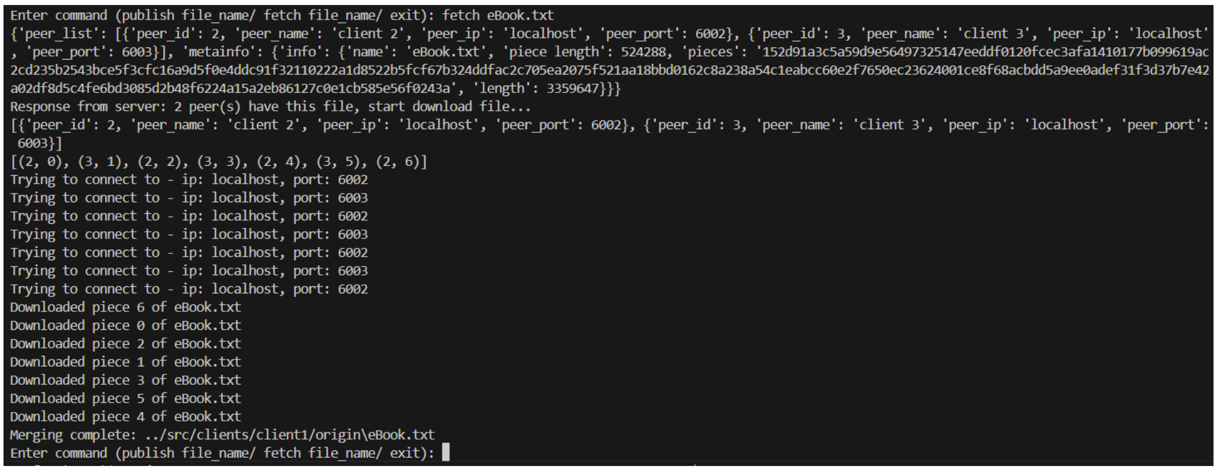
Figure 5.2: Peer 2 publishes the file.



```
PS D:\HCMUT\HK241\ComputerNetwork\Ass1\src> python client3.py
Enter command (publish file_name/ fetch file_name/ exit): publish eBook.txt
Check if eBook.txt in path ../src/clients/client3/origin
File found in path
Response from server : File been publish successfully
Enter command (publish file_name/ fetch file_name/ exit):
```

Figure 5.3: Peer 3 publishes the file.

Next, we run the `fetch eBook.txt` command on client1. Client1 then receives a response from the server tracker, which includes the peer list (IP addresses and ports of peers sharing the file) and the file's metadata (file size, pieces, and hash codes of the pieces). Client1 then matches the `peer_id` with the corresponding `piece_index` and begins generating requests to download the file from both peer2 (client2) and peer3 (client3). Once the download is complete, a message is printed confirming the successful transfer. Client1 then merges the downloaded pieces into the complete `eBook.txt` file and deletes the individual file pieces from its directory.



```
Enter command (publish file_name/ fetch file_name/ exit): fetch eBook.txt
{'peer_list': [{'peer_id': 2, 'peer_name': 'client 2', 'peer_ip': 'localhost', 'peer_port': 6002}, {'peer_id': 3, 'peer_name': 'client 3', 'peer_ip': 'localhost'
, 'peer_port': 6003}], 'metainfo': {'info': {'name': 'eBook.txt', 'piece length': 524288, 'pieces': '152d91a3c5a59d9e56497325147eeddf0120fcec3afa1410177b099619ac
2cd235b2543bce5f3cfc16a9d5f0e4ddc91f32110222a1d8522b5fcf67b324ddfac2c705ea2075f521aa18bbd0162c8a238a54c1eabcc60e2f7650ec23624001ce8f68acbdd5a9ee0adef31f3d37b7e42
a02df8d5c4fe6bd3085d2b48f6224a15a2eb86127c0e1cb585e56f0243a', 'length': 3359647}}}
Response from server: 2 peer(s) have this file, start download file...
[{'peer_id': 2, 'peer_name': 'client 2', 'peer_ip': 'localhost', 'peer_port': 6002}, {'peer_id': 3, 'peer_name': 'client 3', 'peer_ip': 'localhost', 'peer_port':
 6003}]
[(2, 0), (3, 1), (2, 2), (3, 3), (2, 4), (3, 5), (2, 6)]
Trying to connect to - ip: localhost, port: 6002
Trying to connect to - ip: localhost, port: 6003
Trying to connect to - ip: localhost, port: 6002
Trying to connect to - ip: localhost, port: 6003
Trying to connect to - ip: localhost, port: 6002
Trying to connect to - ip: localhost, port: 6003
Trying to connect to - ip: localhost, port: 6002
Downloaded piece 6 of eBook.txt
Downloaded piece 0 of eBook.txt
Downloaded piece 2 of eBook.txt
Downloaded piece 1 of eBook.txt
Downloaded piece 3 of eBook.txt
Downloaded piece 5 of eBook.txt
Downloaded piece 4 of eBook.txt
Merging complete: ../src/clients/client1/origin\eBook.txt
Enter command (publish file_name/ fetch file_name/ exit):
```

Figure 5.4: Peer 1 fetches the file.

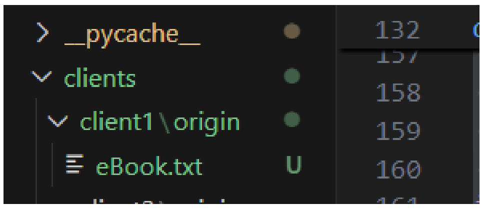
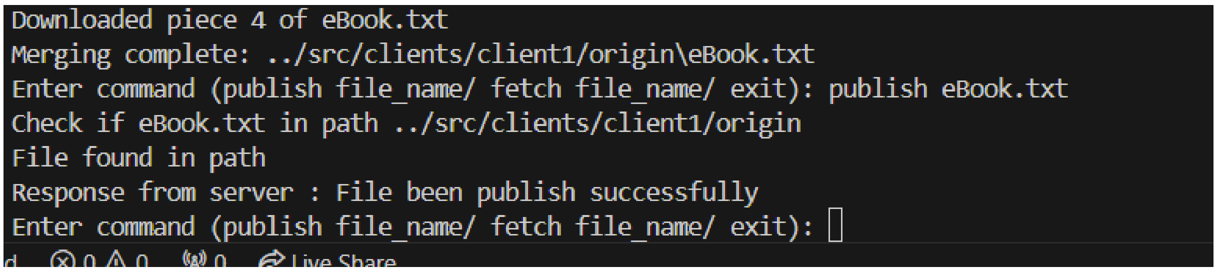


Figure 5.5: eBook.txt file appears in Peer 1's directory after fetch.

After this case, we can observe that the application successfully handles multi-peer downloads, ensuring the correct distribution of file pieces among peers. Additionally, the use of hash codes during the download process helps verify the integrity of the downloaded pieces. The tracker also responds to peer requests correctly, providing the necessary peer list and file metadata.

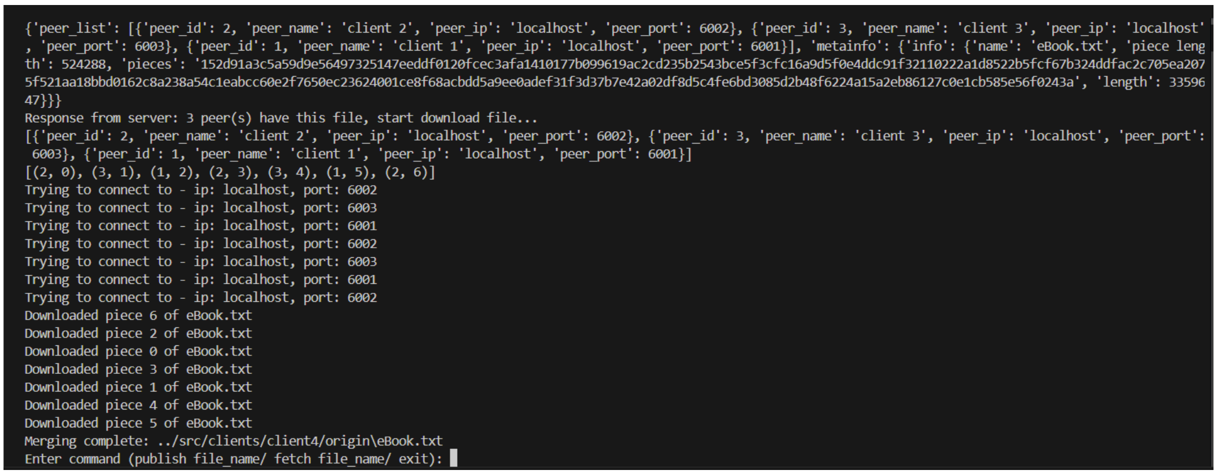
5.3 Case 2

In Case 2, we will test if the tracker is functioning correctly. We will publish the file that was fetched in Case 1 from client1 by running the **publish eBook.txt** command. A successful response will be printed out on client1, confirming that the file has been correctly published and demonstrating that Case 1 was executed successfully.



```
Downloaded piece 4 of eBook.txt
Merging complete: ../src/clients/client1/origin\eBook.txt
Enter command (publish file_name/ fetch file_name/ exit): publish eBook.txt
Check if eBook.txt in path ../src/clients/client1/origin
File found in path
Response from server : File been publish successfully
Enter command (publish file_name/ fetch file_name/ exit): 
```

Figure 5.6: Peer 1 publishes the file after fetching.



```
{'peer_list': [{'peer_id': 2, 'peer_name': 'client 2', 'peer_ip': 'localhost', 'peer_port': 6002}, {'peer_id': 3, 'peer_name': 'client 3', 'peer_ip': 'localhost'
, 'peer_port': 6003}, {'peer_id': 1, 'peer_name': 'client 1', 'peer_ip': 'localhost', 'peer_port': 6001}], 'metainfo': {'info': {'name': 'eBook.txt', 'piece leng
th': 524288, 'pieces': '152d91a3c5a59d9e56497325147eeddf0120fcec3afa1410177b099619ac2cd235b2543bce5f3cfc16a9d5f0e4ddc91f32110222a1d8522b5fcf67b324ddfac2c705ea207
5f521aa18bbd0162c8a238a54c1eabcc60e2f7650ec23624001ce8f68acbdd5a9ee0adef31f3d37b7e42a02df8d5c4fe6bd3085d2b48f6224a15a2eb86127c0e1cb585e56f0243a', 'length': 33596
47}}}
Response from server: 3 peer(s) have this file, start download file...
[{'peer_id': 2, 'peer_name': 'client 2', 'peer_ip': 'localhost', 'peer_port': 6002}, {'peer_id': 3, 'peer_name': 'client 3', 'peer_ip': 'localhost', 'peer_port':
 6003}, {'peer_id': 1, 'peer_name': 'client 1', 'peer_ip': 'localhost', 'peer_port': 6001}]
[(2, 0), (3, 1), (1, 2), (2, 3), (3, 4), (1, 5), (2, 6)]
Trying to connect to - ip: localhost, port: 6002
Trying to connect to - ip: localhost, port: 6003
Trying to connect to - ip: localhost, port: 6001
Trying to connect to - ip: localhost, port: 6002
Trying to connect to - ip: localhost, port: 6003
Trying to connect to - ip: localhost, port: 6001
Trying to connect to - ip: localhost, port: 6002
Downloaded piece 6 of eBook.txt
Downloaded piece 2 of eBook.txt
Downloaded piece 0 of eBook.txt
Downloaded piece 3 of eBook.txt
Downloaded piece 1 of eBook.txt
Downloaded piece 4 of eBook.txt
Downloaded piece 5 of eBook.txt
Merging complete: ../src/clients/client4/origin\eBook.txt
Enter command (publish file_name/ fetch file_name/ exit): 
```

Figure 5.7: Peer 4 fetches the file.

Next, we continue by fetching the file from client4. In the terminal, we can see that client4 receives a response from the tracker, which includes information about three peers (client1, client2, and client3). Client4 then correctly matches the peer information and generates the appropriate download requests. The file is successfully downloaded and appears in client4's directory.

5.4 Evaluation

Completed:

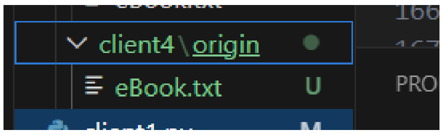


Figure 5.8: The file appears in Peer 4's directory after fetch.

- The application successfully manages connections with the central tracker. Multi-threading is implemented to handle multiple connections simultaneously.
- The tracker performs its role by returning a list of peers that hold the requested file.
- A round-robin approach is employed to distribute requests for file pieces across multiple peers.
- The ‘publish‘ and ‘fetch‘ functions are correctly implemented.
- The application enables downloading a file from multiple peers, and peers can also participate in the file-sharing process by serving pieces to others.

Incomplete:

- The application lacks a graphical user interface (GUI) and is operated through the command-line shell.
- It interacts only with file names, without handling metadata files (e.g., .torrent files), and is limited to sharing single files.
- Messages between peers are not fully structured, as they do not include timestamps, peer IDs, or status updates.

6 Conclusion

Our team successfully completed the project to implement a peer-to-peer file-sharing application using the Python programming language. This project provided us with valuable opportunities to enhance our skills in programming and data management.

Throughout the course of the project, the team learned how to implement socket programming in Python. We mastered techniques for establishing connections, transmitting, and receiving data between computers, applying them flexibly in building the file-sharing application.

In addition, the team gained strong knowledge in data management. We learned how to query, store, and transmit data to and from databases efficiently and scientifically, ensuring accuracy and security of the information.

Looking ahead, the team intends to further develop the application, focusing on the following aspects:

- **Improving security**: Encrypting data during file transmission and reception, while also encrypting sensitive user data stored in databases to ensure information security.
- **Enhancing functionality**: Adding new features such as managing a friends list, sharing files within groups, and more to meet the diverse needs of users.
- **Optimizing performance**: Improving file transfer efficiency and leveraging system resources more effectively to provide the best user experience.

The peer-to-peer file-sharing project has been a valuable learning experience for the entire team. It helped us improve our skills in programming, data management, and system security.

7 References

1. BitTorrent Specification. Accessed from:
   https://wiki.theory.org/BitTorrentSpecification

2. Tracker scrape. Accessed from:
   https://en.wikipedia.org/wiki/Tracker_scrape