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

COMPUTER NETWORK - Assignment 1

DEVELOP A NETWORK APPLICATION

Semester: 232

Advisors: Dr. Nguyen Le Duy Lai
Student: Ta Ngoc Nam 2152788
Nguyen Thinh Dat 2152507
Pham Huy Thien Phuc 2053346

HO CHI MINH CITY, APRIL 2024

Members list & Workload

No.	Full name	Student ID	Duty in the assignment	Percentage of work
1	Ta Ngoc Nam	2152788		100%
2	Nguyen Thinh Dat	2152507		100%
3	Pham Huy Thien Phuc	2053346		100%

Contents

Chapter 1

Assignment specifications

1.1 Objective

Build a Simple Like-torrent application with application protocols defined by each group, using the TCP/IP protocol stack.

1.2 Application description

- A centralized server keeps track of which clients are connected and storing what pieces of files.
- Through tracker protocol, a client informs the server as to what files are contained in its local repository but does not actually transmit file data to the server.
- When a client requires a file that does not belong to its repository, a request is sent to the server.
- 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.

The illustrated system in Figure 2.3.

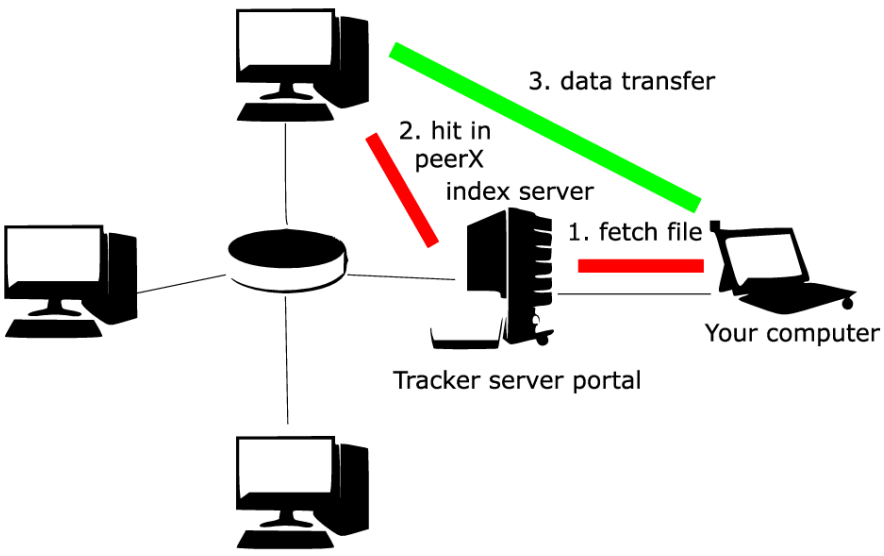

Figure 1.1: Illustration of like-torrent system activities

Chapter 2

Implementation

2.1 Project Files Structure

Overall, we have 3 + 1 source files and 2 folders to store data. The structure as shown in image

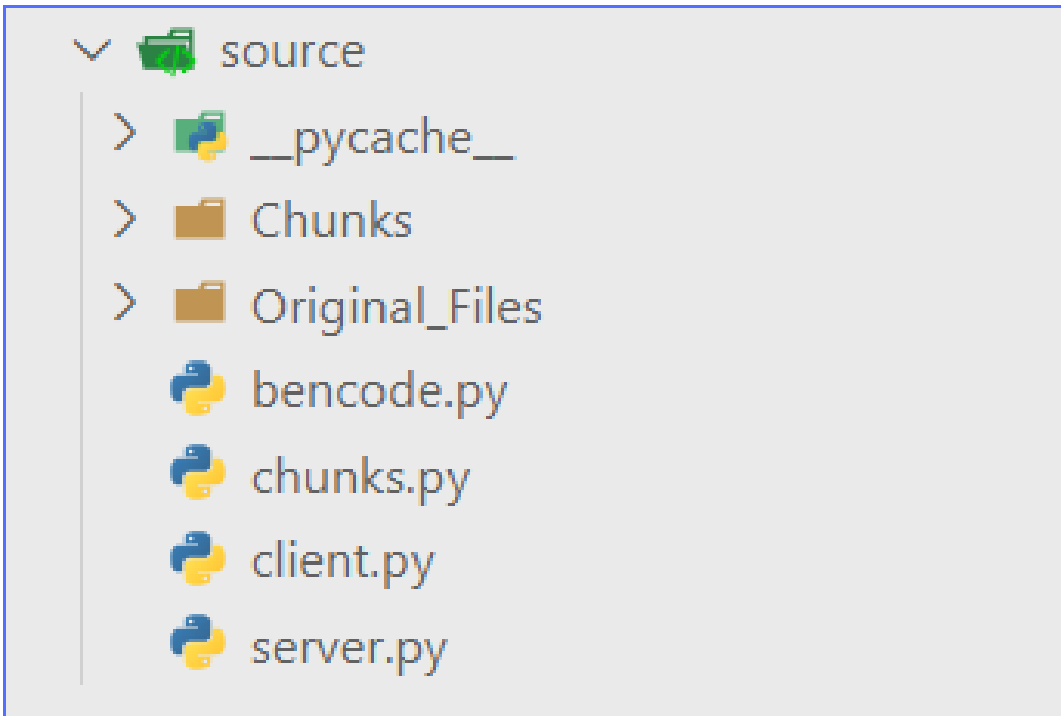

Figure 2.1: Project Structure

- **Folder "Chunks"**: contains all the chunks of original file and a torrent file which contains all information about the chunks.
- **Folder "Original_Files"**: contains all the data files which are shared. In this case, the origial files are **mv1.mp4** and **mv2.mp4**.
- **bencode.py**: is a method to convert torrent file (.json format) to string and then encoded ("utf-8") to send.

```python
import bencodepy
import re

decimal_match = re.compile('\d')

def bencode(value):
    return bencodepy.encode(value).decode()

def bdecode(data):
    '''Main function to decode bencoded data'''
    chunks = list(data)
    chunks.reverse()
    root = _dechunk(chunks)
    return root

def _dechunk(chunks):
    item = chunks.pop()

    if item == 'd':
        item = chunks.pop()
        hash = {}
        while item != 'e':
            chunks.append(item)
            key = _dechunk(chunks)
            hash[key] = _dechunk(chunks)
            item = chunks.pop()
        return hash
    elif item == 'l':
        item = chunks.pop()
        list = []
        while item != 'e':
            chunks.append(item)
            list.append(_dechunk(chunks))
            item = chunks.pop()
        return list
    elif item == 'i':
        item = chunks.pop()
        num = ''
        while item != 'e':
            num  += item
            item = chunks.pop()
        return int(num)
    elif decimal_match.search(item):
        num = ''
        while decimal_match.search(item):
            num += item
            item = chunks.pop()
        line = ''
        for i in range(int(num)):
            line += chunks.pop()
        return line
    raise "Invalid input!"
```

- chunks.py: is a file which helps us to separate the file into many *pieces*, each has maximun 512KB size.

```python
import os

SIZE = 524288


def chunk(file_name):
    original = "Original_Files"
    desti = "Chunks"
    chunk_size = SIZE

    with open(os.path.join(original, file_name), 'rb') as file:
        chunk_number = 1
        while True:
            chunk_data = file.read(chunk_size)
            if not chunk_data:
                break

            chunk_filename = f"{desti}/{os.path.basename(file_name)}.part{chunk_number}"

            with open(chunk_filename, 'wb') as chunk_file:
                chunk_file.write(chunk_data)

            chunk_number += 1


if __name__ == "__main__":
    chunk("mv1.mp4")
    chunk("mv2.mp4")
```

- **client.py** and **server.py** are two *main files* of this project. The details code provided in the github link below. In this report we are just describing the functions of them.

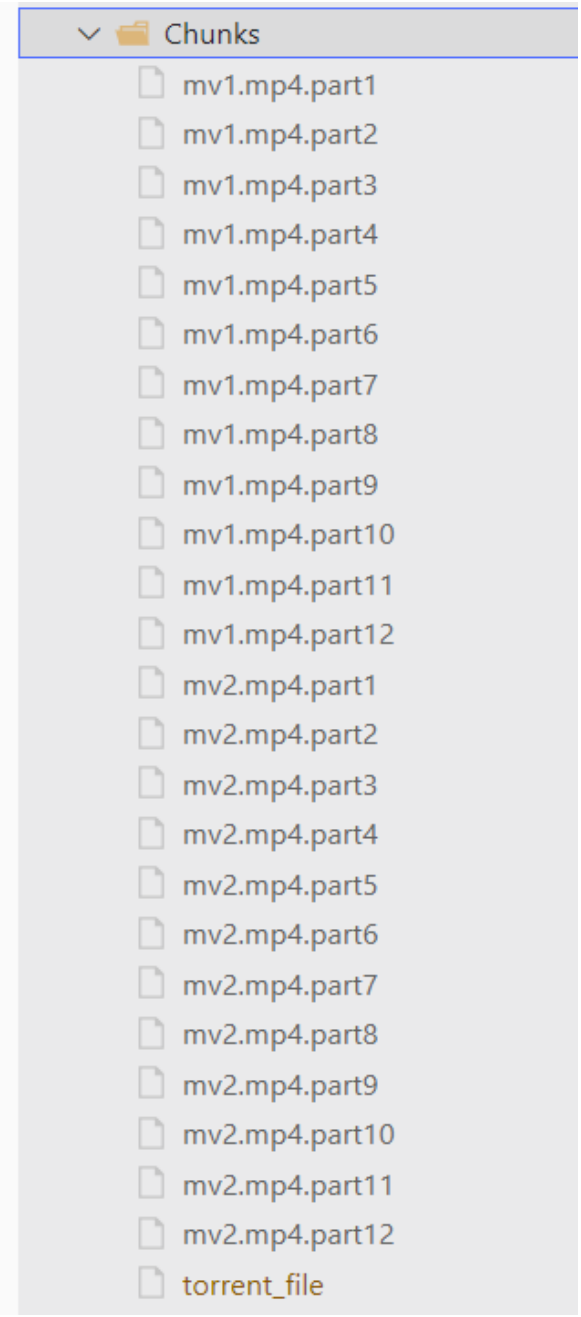

Figure 2.2: Chunks Folder

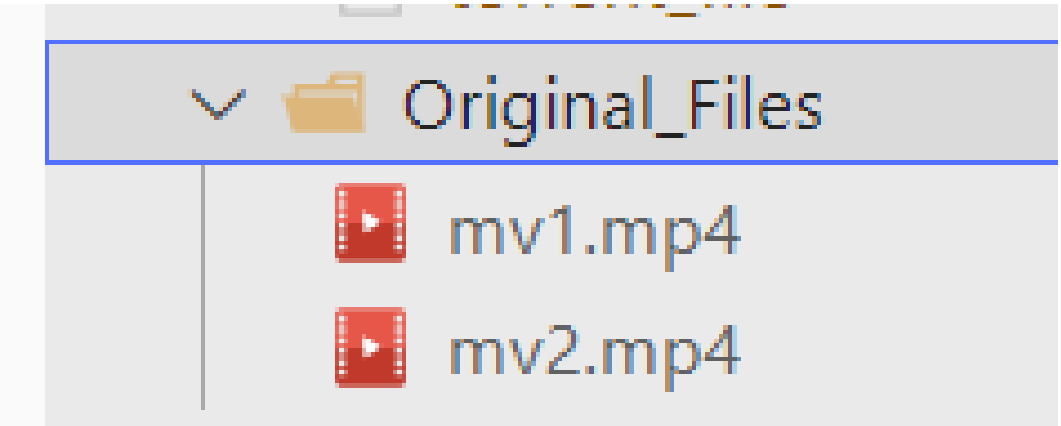

Figure 2.3: Original_Files Folder

2.2 Project Core

2.2.1 Server Side

- **Server Initialization:**
  - `run_server`: Initializes the server, sets it to listen for incoming connections, and handles each connection with a new thread.
- **Connection Management:**
  - `manage_client_connection`: Manages individual client connections. It decodes messages from clients, processes commands (like retrieving peer list, updating status, disconnecting, or exiting), and sends appropriate responses.
- **Peer List Management:**
  - `refresh_peer_list`: Updates the server's list of active peers when a client sends updated information. This is critical for maintaining an accurate view of the network's state.
- **Handling Client Commands:**
  - Commands handled include:

    get_peers Returns the current list of peers to the requesting client.
    update_status Updates the server's information for the client who sent the request.
    disconnect Properly handles the disconnection request by a client.
    exit Removes a client from the peer list and handles the client's exit from the network.
- **Utility Functions:**
  - `print_dict`: Utility function for printing dictionary contents, typically used for debugging or logging.

2.2.2 Client Side

- **Client-Server Communication:**
  - `connect_server`: Establishes a connection to the server, sends client info, and updates the local set of connected clients.
  - `get_client_set`: Requests the list of clients connected to the server.
  - `update_status_to_server`: Sends the current client's status to the server and refreshes the client set.
  - `disconnect_server`: Disconnects from the server and closes the socket.
  - `quit_torrent`: Shuts down the client's operation cleanly, ensuring all connections are terminated properly.
- **Client-Client Communication:**
  - `connect_client`: Connects to another client directly to initiate peer-to-peer communication.
  - `ping`: Checks connectivity with another client by sending a "ping" message.
  - `request_download`: Requests a specific file chunk from another client.
  - `disconnect_client`: Terminates the connection with another client.
- **File Handling and Chunk Management:**
  - `merge_chunks`: Combines downloaded chunks into a single file once all parts are received.
  - `send_file`: Sends a file over a socket, used in peer-to-peer file transfers.
  - `read_torrent_file`: Reads the torrent file which contains the structure of files and their chunks.
  - `update_chunk_status`: Updates the local record of which chunks of the file have been downloaded.
- **Utility and Helper Functions:**
  - `print_dict`: Prints the contents of a dictionary, used for debugging.

  - `check_server_connected`: Checks if the client is still connected to the server.
  - `check_target_client_connected`: Verifies whether there is an active connection to a specified client.

- **Threading and Main Execution:**
  - `command_handler`: Handles commands entered by the user, directing them to appropriate functions.
  - `command_thread`: Maintains a thread for receiving and handling user inputs.
  - `client_init`: Initializes client settings and prepares the environment.

Our code has been published here: https://github.com/namoi4009/NetCom_Ass1

Chapter 3

Realtime Testing

As in the below images, the client 1 who has the chunks file in the directory sent the files to client2 successfully through (P2P) and then the file is merge and can be watch in the client2's side.

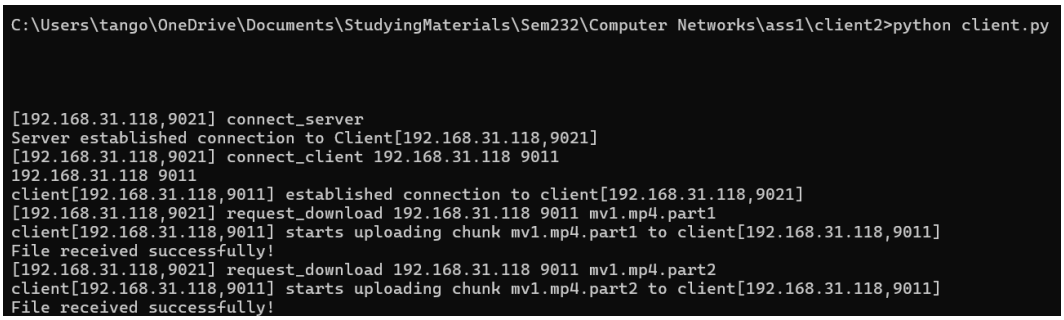

Figure 3.1: Client2 connect, request and receive files from client1

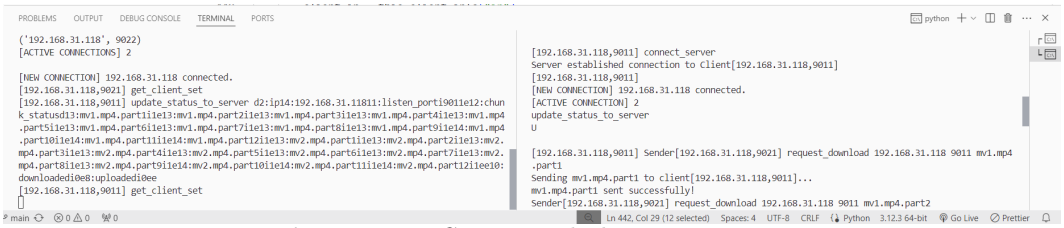

Figure 3.2: Server and client1 activities

Chapter 4

Conclusion

The development and deployment of a like-torrent application using Python for peer-to-peer (P2P) file sharing represents a significant advancement in decentralized data distribution. This application illustrates the robust capabilities of network programming and the practical application of threading and socket programming in Python. By mimicking the functionality of torrent systems, this project enables multiple users to connect and share files directly without relying on a centralized server, thereby enhancing data transfer speed and efficiency while also reducing server load.

The server component of the application plays a crucial role in maintaining the network's stability and coordination. It manages peer connections, updates peer statuses, and facilitates the efficient discovery of file resources among clients. This is instrumental in handling network dynamics where peers frequently connect and disconnect. Moreover, the server's ability to dynamically update and distribute the list of active peers ensures that the network remains scalable and can handle growth seamlessly. The use of bencoding for communication enhances data integrity and parsing simplicity across the network, reinforcing the robustness of the server-client communication protocol.

On the client side, the application showcases a comprehensive suite of functionalities that allow for intricate interaction between peers. Clients are not only able to download and upload file chunks but can also update their status on the network, request lists of other peers, and manage connections dynamically. This high level of interaction supports a resilient and flexible network where peers can autonomously manage their data exchanges. The threading implementation ensures that the client can handle multiple operations simultaneously, which is critical for the responsiveness and reliability of the P2P network.

In conclusion, this like-torrent application serves as a powerful demonstration of decentralized network applications and the potential of Python in creating sophisticated network interactions. The combination of effective server management and proactive client-side functionalities creates a robust platform for file sharing that mimics the efficiency and scalability of popular torrent applications. This project not only highlights the technical skills involved in network and concurrent programming but also paves the way for future innovations in decentralized data sharing technologies.

4.1 Missing Implementation

- Automatically executing necessary orders (connection, download and merge file
- Automatically choose and connect to peer who has the requested files
- User Interface