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

Computer Network (CO3094)

Assignment 1

Develop A Network Application

Advisor: Nguyen Le Duy Lai

HO CHI MINH CITY, NOVEMBER 2024

Member list & Workload

No.	Full name	Student ID	
1	Doan Viet Tien Dat	2252141	50%
2	Nghiem Pham Vy Nghi	2252518	50%

Contents

1 Requirement Analysis

In this section, we shall go through the necessary requirements for the network application. With its attributes similar to a **torrent-like application**, this network application revolves around the communication between **Tracker** (or the **Server**) and **Peer** (Or the **Client**), and most important, the **Peer-to-Peer** communication and file sharing features. For a more organized view of the problems, we can divide the requirements into 2 types, the **Functional Requirements** and **Non-functional Requirements**. In each of these 2, we shall also divide them into smaller categories for Tracker and Peer individually.

1.1 Functional Requirements

1.1.1 Tracker

- The tracker keeps track of the connections from peers.
- The tracker keeps track of the available files information sent from each peer's repository.
- The tracker provides download information for the peers to establish connections between themselves.

1.1.2 Peer

- Peers can connect to the tracker.
- Peers can inform the application of their possessed files.
- Peers can connect to other peers whose information provided by the trackers.
- Peers can download a chunk of files from other peers and assemble them into a complete file.
- Peer can accept connections from other peers and process them

1.2 Non-Functional Requirements

1.2.1 Tracker

- Tracker can receive multiple requests from peers and response with a formatted, easy to understand answer.
- Tracker can identify each client through their unique identifiers.
- Tracker can print out information in their system in a clear manner.

1.2.2 Peer

- Peer can handle many connections and downloads at the same time, requiring them to be multi-threaded.
- Communication between peers must be easy to track and understand, requiring announcements must be made at each stage.

- There should be no loss in uploaded or downloaded data whatsoever.

2 Technology Stack

To build our app, we need to choose the infrastructure that can live up to our expectations. In the following subsections, we will discuss about our selections.

2.1 Language

In this project, we use **Python** as our main programming language. For starting, Python is on the easier side when it comes to a learning a new language. And in specific sense, It has many great libraries for building a network application.

2.2 Libraries

As mentioned, without the support from libraries we couldn't build a network app. There are multiple libraries used, however, we shall only describe some that plays a crucial role in the application

2.2.1 Flask

First to be mentioned, **Flask** is a library that supports **HTTP communications**. With its functionality, Flask is the core of communication between tracker and peers. Flask help define useful **routes** and **rule set** for the tracker based on the HTTP concept. We shall explain this in details later when it comes to the **Application Design** section.

2.2.2 Requests

However, Flask is only used to established HTTP API for **server-like application**, or in this case, is our Tracker. On the peer side, to communicate with the tracker using HTTP, we have to have another library named **Requests**. As its name suggested, Requests support us in **writing requests**, this involved locating **address** to send, specifying **HTTP method**, creating **headers** and appending content to the **body** of a complete **HTTP request**. We shall discuss the way to do this in the next subsection about **Network Protocols**.

2.2.3 Socket

While Flask established the core for communication between tracker and peers, **Socket** is the foundation of communication between peers. With the help from socket, every peer can participate in the network, exchange requests with the other peers.

2.2.4 Threading

As the requirements demand multiple things to be processed at the same time, we need a help from a library that can do multi-threading. This is where the **Threading** library comes into the play. With its help, creating threads is easy, meeting the specific requirements for the application.

2.3 Network Protocols

2.3.1 Transport Layer Protocols

As the requirements specified, the connection protocol between peers will be TCP. Yet, we don't need to program it as this is fully supported by the **Socket** library. By establish connection, the helpful socket library will handle the **handshake**, and also the **connection queue** for us. To implement it, simply do as follow:

```
import socket

# Create a socket instance
# AF_INET refers to the address-family ipv4
# SOCK_STREAM means connection-oriented TCP protocol.
socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Bind to an address
# Address containing a host(or ip) and a port (host:port)
socket.bind((host, port))
```

To **accept**, or **listen**, to connections:

```
# Listen to connection
socket.listen()
```

To connect to a remote socket at a specific address:

```
# Connect to an address
# Address containing a host(or ip) and a port (host:port)
socket.connect((host, port))
```

However, keep in mind that the socket must be either for **listening** or **connecting**, which means for both listening and connecting, we have to have **2** different sockets.

2.3.2 Application Layer Protocols

For the communication between the tracker and the peers, HTTP is involved in the process. We use Flask to help us with this task.

```
# Create a Flask app
app = Flask(__name__)

# Specifying routes for communications
```

```
# Here we have an example about a simple route
# The HTTP method used here is GET
# Route is '/'
@app.route('/', methods=['GET'])
def home():
    # Response written in JSON format
    response = {
        'message': 'Hello World!'
    }

    # Encode it into a JSON format that can be sent
    # The status code here is 200, meaning a successful response
    return jsonify(response), 200

# Run the app at the specified address
# Host is the IP address
# Port is the port number
app.run(host=host, port=port)
```

As said, on the peer side, we use the Requests library to send request:

```
# Function to send a hello request
def send_hello_request() :
    # The URL to send the request to
    # URL is the combination of:
    # ip(host): The IP Adress of the server(tracker)
    # port: Port number
    # route: The specific route to communicate
    url = 'http://trackerip:trackerport/route'

    # Sending a GET request
    # Providing the url
    # You can also specify the body here,
    # but since this is a GET request,
    # there is no body
    # This request will return a response from the server
    response = requests.get(url=url)

    return response
```

3 Functionality Analysis

With requirements and technology full analyzed, we now explore deeper into how we come up with the design of this application, including its **structure**, **methods**, **communication ruleset operations/interactions**.

3.1 Approach

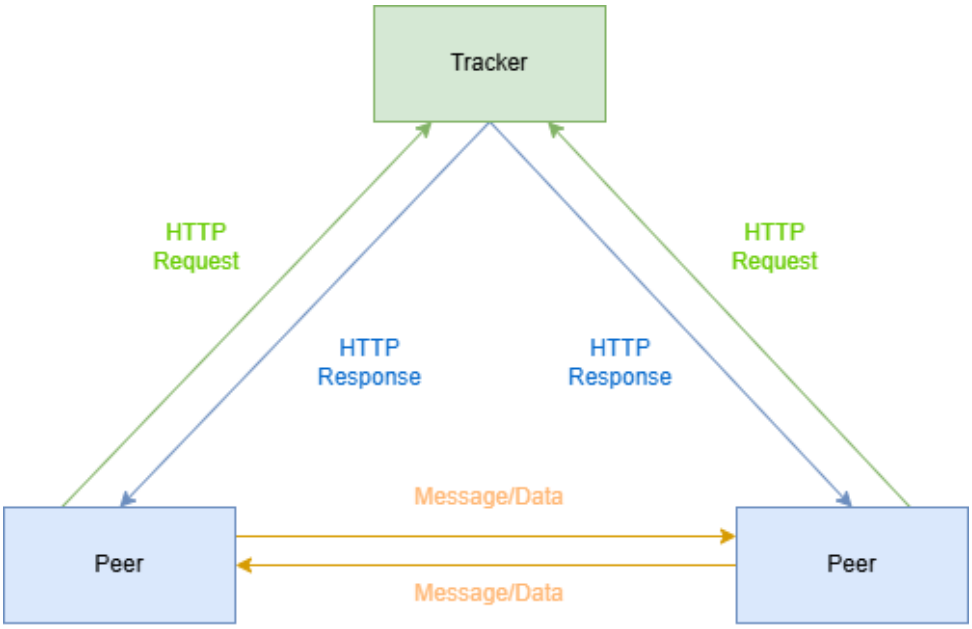

Figure 1: The simple structure of how the actors in system communicate

Let's start with how we approach the application. It is easy to see that there are currently 2 actors, the **tracker** and the **peer**, so we should design our application around them. Therefore, it is easier to separate them into **2 different files**.

3.2 Class & Diagram

Addressing this problem further, we shall integrate the **OOP approach** here and create a class for each actor. This help us manage the actor in an organized way. And accompanying each class, there shall be methods as the way we instruct the program to engage in a process. Let's go through a **class diagram** for our application

For a deeper explanation of how we implement these classes into the system and their methods, we shall convey them in the **Implementation** section. For now, let's move on to the flows.

3.3 Program Flows

In this subsection, we come up with the **flows** for how the peers communicate and carry out the important functions of a torrent-like system.

Figure 2: The class diagram of our application

3.3.1 Files seeding

Moving on to the core element of our torrent-like application, the **files**. As the application revolves around requesting and sending file, we have to establish a method for peers to publish the files they own so the others can see and get it.

This flow requires these steps:

1. **Command call**: The user called for the searching command along with the paths to the files they want to publish (The way to call the command will be described at a later section)

2. **Check existence**: After the call, the system will check for the existence of said files based on the provided paths. If a file does not exist, the process for that file is aborted and it will not be added to the payload of the request that will be sent to the tracker.

3. **Send request to server**: After ensuring the file existed, the system will move on to create a **POST** request and send it to the tracker

4. **Data parsing**: At the tracker side, as a POST request was sent to it, the tracker can process the request and parse it in a way the tracker can understand, therefore taking out important information such as the IP and (upload) port of the peer, along with file names. This step includes the validation and if the request

did not provide enough information, the process will obviously aborted, and an error response would be sent back.

5. **Data insertion**: For each file, the tracker check if such file has already provided by the same peer before. If the answer is yes, it abort the call, else it insert into the the file list.

6. **Response**: Each successfully inserted file data will be appended back to the response payload. At the end, the tracker sends a successful response back to the peer with status code 200.

7. **Response**: Back to the peer side, based on the response, the peer will make an announcement to the user

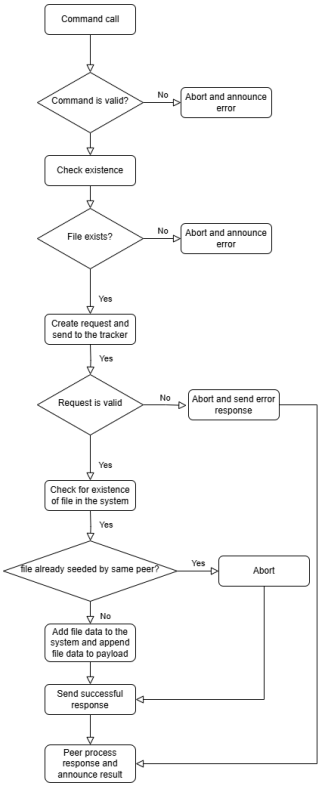

Figure 3: The flowchart for seeding flow

3.3.2 Files searching

To get a file, a peer must know the destination address so it can make connection and download. Therefore, a search flow is also provided.

This flow requires these steps:

1. **Command call**: The user called for the searching command along with the file name. Only 1 file name is accepted for this command so the process will be aborted if there is a number other than 1 file is provided

2. **Send request to server**: After processing the command, the system will create a **GET** request and send it to the tracker to ask for the requested file

3. **Data parsing**: At the tracker side, as a GET request was sent to it, the tracker can process the request and parse it in a way the tracker can understand, therefore taking out important information such as the IP and (upload) port of the peer, along with file identity. This step includes the validation and if the request did not provide enough information, the process will obviously aborted, and an error response would be sent back.

4. **Response**: Based on the file existence, the tracker will have an appropriate response. If it does exist, the tracker shall get the data about the peers who have that file and send add to the response

5. **Response**: Back to the peer side, based on the response, the peer will make an announcement to the user

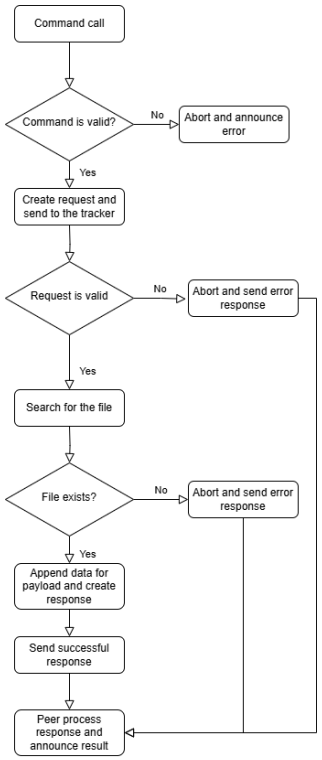

Figure 4: The flowchart for searching flow

3.3.3 Files downloading

Can't be a torrent-like application without the files downloading. This flow is enormous and involves the previous searching flow

This flow requires these steps:

1. **Command call**: The user called for the download command along with the file names.
2. **Threads creation**: This step does not contribute to the download process overall, but it is a crucial step to ensure **multiple files** can be processed at the same time as required. With each provided file, a **thread** will be created to handle them for the next steps.
3. **Check existence**: The system does not forget to check for the existence of a file first to see if a download request is needed
4. **Send search request**: For each file, a search request is sent to the tracker to get the needed information about peers who have the files
5. **Define download information**: if the requested file exist on the database of the tracker and the information about the other peers is sent back, based on the number of peers, the requesting peer shall define which chunk they want to receive from each destination peer.
6. **Sending request**: With enough information, the requesting send its requests to every provided peer and wait for the requested chunk to be sent back.
7. **Process request and response**: At the other ends, destination peers will each find the file on their system, separate the file into chunks, then sent the requested part back based on the request.
8. **Receive and assemble chunks**: After getting enough requested chunks, the requesting peer sort the chunks into the correct order, then start combining these chunks into a complete file by writing them to the repository.
9. **Announcement**: Last but not least, the peer will announce the completion so the user can grasp of the situation.

4 Implementation

In this section, we cover the implementation based on the said approach and flows.

4.1 Core Modules

As said, the tracker and peer will belong to different files, and for a better understanding of their concepts as entities inside the network, let's call them module from now on, namely the **Tracker module** and the **Peer module**.

4.1.1 Tracker module

The tracker is like a center for **data storing** and **provides initial necessary information** for the download of peers. The tracker does not process how the peers should download a file, as its name suggested, it only tracks and give out information when needed.

Properties

- **ip**: the ip address of the tracker, by default it is the 'localhost'.
- **port**: the port number of the tracker, by default it is 7000.
- **lock**: the lock provided by **Threading** to handle concurrency.
- **peer_list**: the list of peers, containing their id, address, and their seeded files.
- **file_list**: also called the seed list, containing seeded files information and which peers are holding onto them.
- **app**: short for Flask app to be precise, used to establish the HTTP application.

Methods

- **__init__(self, ip='localhost', port=7000,)**: the constructor, coming along with the desired address for the tracker server.
- **init(self)**: the *"start button"* for a Tracker **instance**, it initialize important **threads** including the **Flask thread**, where it handle the **Flask app**, and the **CLI thread**, thread to handle the **command line user interface**.
- **init_flask(self)**: initialize the Flask app, and define routes, therefore establish the **API** between the tracker and the peers.
- **init_cli(self)**: initialize the command line interface, where it accepts user inputs and process them.
- **shutdown(self, payload=None)**: stop the whole tracker system.
- **insert_file(self, infohash, name, size, path, ip, port, id)**: insert a new file data to the data storage. This function is a bit special since it locates files based on **infohash**, a **hashed** version of the file name.

- **view_peers(self)**:print the peers regconized by the system along with their contributed seeds.
- **view_files(self)**:print the file seeds along with their information and their contributing peers.
- **generate_error_response_json(self, message, status)**: a helper function for faster response generating.

4.1.2 Peer module

Compared to the Tracker, the Peer is more complicated. It contains many logic behind so that it can play both parts as the requesting peer and uploading peer.

Properties

- **id**: the identity of each peer, is a string for the name of each peer.
- **ip**: the ip address of the peer, by default it is the 'localhost'.
- **upload_port**: the port number used for uploading files(chunks) to other peers.
- **lock**: the lock provided by **Threading** to handle concurrency.
- **self_host**: the IP address of the peer.
- **tracker_host**: the IP address of the tracker, by default it is localhost.
- **tracker_port**: the port of the the tracker, by default it is 7000.

Methods

- **__init__(self, id, upload_port, self_host = 'localhost', tracker_host = 'localhost', tracker_port = 7000)**: the constructor, coming along with the desired input information from the user.
- **init(self)**: just like Tracker's init(), it first create socket, then initialize important **threads** including the **listen thread** to listen to connections and the **CLI thread**, thread to handle the **command line user interface**.
- **init_listen(self)**: initialize the listen thread. This thread is the first step to accept incoming request from other peers, parse the request and redirect the program what to do next. With each download request from peers, it create threads to handle, making it fully multi-threaded.
- **init_cli(self)**: initialize the command line interface, where it accepts user inputs and process them. With the help of the helper function **parse_command**, this thread process parsed command and choose what to do next based on the input.
- **init_download(self)**: another thread initializing function. This function initializes the download process, checking for **file's existence**, sends **search request**, becomes the center for **sending download requests** to other peers and **collecting chunks**, then finally **sorts** and **assembles** chunks into a complete file.

- **check_seed_files(self, names)**: check the existence of multiple files in the repository based on provided names.
- **check_existence(self, name)**: check the existence of a single file in the repository based on provided name.
- **get_chunk (self, path, chunk_size, index)**: get a chunk from a file based the size and offset, interpreted from index. It operates based on **memory map**, therefore returns a list of bytes, which is a part of the original file.
- **create_file_from_chunks(self, chunk_list, path)**: the reverse of get_chunk, this function assembles chunks into a complete file based on the provided list. Basically it takes encoded bytes from chunks and write them straight to a new file in the repository, therefore creating a replicate of the original one.
- **download_chunk_from_peer(self, id, ip, port, name, index, peer_count, file_chunks)**: this one creates connection to a peer, calculate the desired chunk size, then creates and sends a request to the destination peer. After that, it takes time to wait for the response containing desired chunk from the destination peer. At the end, it appends the newly received chunk back into the chunk list, which shall be sorted later.
- **handle_download_chunk_request (self, name, index, count, connection, ip, port)**: while the one above sends request, this one process such sent requests and then finds way to collect the asked chunk. After all of that, it sends the chunk back to the requesting peer and announce the end of the uploading flow.
- **shutdown(self, payload=None)**: stop the whole peer system.

4.2 Application interface

In this subsection we study the implementation of application interface, which is how peer can communicate with tracker, and how peer can communicate between themselves.

4.2.1 Tracker - Peer Communication

As said earlier, the tracker and peer communicate with each other through HTTP.

On the tracker side, using the HTTP philosophy we define some route used for communication. First is a **dummy** route, which serves as a test to see if the peer can send a request to the tracker. If succeeded, it gives back a "Hello World!" message

```
1 @self.app.route('/', methods=['GET'])
2     def home():
```

Second is a **search file** route, which is a GET request called to route "/file". As HTTP GET request does not come with a body, we also put the infohash, a hashed version of the file name as the path variable.

```
@self.app.route('/file/<infohash>', methods=['GET'])
    def search_file(infohash):
```

The third one is a **seed file** route, which is a POST request called to route "/file". Inside the body of this request containing the data in JSON format.

```
@self.app.route('/file/<infohash>', methods=['GET'])
    def search_file(infohash):
```

For 2 last functional routes, we have example requests for them as follow:

```
# The GET Request (search file)
infohash = sha256(name.encode()).hexdigest()
url = self.tracker_url + '/file/' + infohash

# Send request and receive response
response = requests.get(url=url)
```

```
# The POST Request (seed files)
url = self.tracker_url + '/file'
headers = {'Content-type': 'application/json', 'Accept': 'text/plain'}

for name in names :
    path = './data/' + name
    size = os.path.getsize(path)
    infohash = sha256(name.encode()).hexdigest()
    files.append({
        'name': name,
        'size': size,
        'path': path,
        'infohash': infohash
    })

data = {
    'files': files,
    'ip': self.self_host,
    'port': self.upload_port,
    'id': self.id
}
```

```

response = requests.post(url=url, json=data, headers=headers)
```

For the response from the tracker, there could be 2 types of response, the first one is for when it is an error:

```
# The error response
# Other fields are unnecessary since it is an error
response = {
    'failure_reason': message
}

# The tracker can specify a status code in range 400 - 500
return jsonify(response), status
```

And then there is the success response:

```
# List to hold peers that has the requested file
peers = []

# Append peer's information into the list
for seed in file_data['seeds'] :
    peers.append({
        'peer_id': seed['id'],
        'ip': seed['ip'],
        'port': seed['port'],
    })


# The success response for the search file request
# Tracker id is the address of the tracker
response_payload = {
    'tracker_id': f'{self.ip}:{self.port}',
    'peers': peers
}

# Here we choose 200 for a successful response
return jsonify(response_payload), 200
```

4.2.2 Peer - Peer Communication

Since the communication between peers does not follow HTTP so it does not need headers or status code, however, the content is also a string version of the data in **JSON format**. For this part, We only need a request for chunks and a corresponding response to send the chunks back, which is exchanged through socket, so there is not much communication between peers.

The content of the request for chunks is as follow:

```
# The content of the request
download_request = {
    'name': name,
    'id': id,
    'index': index,
    'count': peer_count
}

# Turn it into a string so it can be sent through the socket
dumped_download_request = json.dumps(download_request)
```

On the other side, the response is as follow:

```
# The content of the response
response = {
    'name': name,
    'id': self.id,
    'source_ip': self.self_host,
    'source_port': self.upload_port,
    'index': index,
    'chunk_map': chunk_map.decode(encoding='utf-8')
}

# Turn it into a string so it can be sent through the socket
connection.sendall(json.dumps(response).encode())
```

4.3 Multi-file download

As we have mentioned earlier, the application supports users download multiple files at the same time, this can be achieved through the use of Multi-threading.

For initializing the download threads:

```
# The list of created download threads
download_file_threads = []

# With each file name inside the command from user
# we create a distinct download thread for them
for name in content :
    download_file_thread = threading.Thread(target=self.init_download, args=(name,))
    download_file_thread.start()
    download_file_threads.append(download_file_thread)

# Join threads together so the program will wait for
# all the download threads to be completed
for download_file_thread in  download_file_threads :
    download_file_thread.join()
```

With each requested file, it can be separated into chunks for faster download at each destination peer, so for each of these chunks, we must also establish a thread to handle them:

```
# List of created chunk download threads
download_chunk_threads = []
# Download chunks by creating connections to peers and receive chunks sent by them
for index, peer in enumerate(peers) :
    id = peer['peer_id']
    ip = peer['ip']
    port = peer['port']
    download_chunk_thread = threading.Thread(target=self.download_chunk_from_peer,
    args=(id, ip, port, name, index, peer_count, file_chunks))
    download_chunk_thread.start()
    download_chunk_threads.append(download_chunk_thread)

# Join the threads together
for download_chunk_thread in download_chunk_threads :
    download_chunk_thread.join()
```

For handling multiple download requests:

```
# Create threads to handle each peer request
peer_request_handler = threading.Thread(
    target=self.handle_download_chunk_request,
```

```
    args=(name, index, count, connection, address[0], address[1])
)

peer_request_handler.start()
```

4.4 File separation & Chunks assembly

The handling of chunks is also important as it contributes to how torrent-like application can create faster download between peers. This process has 2 parts, the **separation of file** into smaller chunks and the **assembly of chunks** to create a complete file.

To separate a file, or rather, to take only a part of the file:

```
def get_chunk (self, path, chunk_size, index) :
    # Open a file in byte reading mode
    file = open(path, 'r+b')

    # The start and end index of the byte list we need
    start = chunk_size * index
    end = chunk_size * (index + 1)

    # With the help of mmap library, we create a memory map
    # then take only the bytes we need
    # which were put into a list
    chunk_map = mmap.mmap(file.fileno(), 0)[start:end]

    # Flush and close the opened file
    file.close()

    # This function return the list of taken bytes
    return chunk_map
```

To separate a file, or rather, to take only a part of the file:

```
def get_chunk (self, path, chunk_size, index) :
    # Open a file in byte reading mode
    file = open(path, 'r+b')

    # The start and end index of the byte list we need
    start = chunk_size * index
    end = chunk_size * (index + 1)
```

```
8
9       # With the help of mmap library, we create a memory map
10      # then take only the bytes we need
11      # which were put into a list
12      chunk_map = mmap.mmap(file.fileno(), 0)[start:end]
13
14      # Flush and close the opened file
15      file.close()
16
17      # This function return the list of taken bytes
18      return chunk_map
```

However, with these chunks sent back to the peer in an unordered manner, we have no way to know each chunk's location in the complete file. So we take an additional step to add the **index**, which will help locate the chunk. This index can be seen earlier in the peer to peer request subsection so we shall not elaborate further about it.

So with the list of chunks that is marked with index, we can sort them again to create an ordered chunk list:

```
1  # file\_chunks is a list of tuples (index, chunk)
2  # so we sort based on the first element of the tuple
3  file_chunks.sort(key=lambda tup: tup[0])
```

For the chunks assembly:

```
1  def create_file_from_chunks(self, chunk_list, path) :
2      # With the path to where the file shall be located
3      # we create a new byte file
4      file = open(path, 'bx+')
5
6      # With each chunk, continuously write it into the newly
7      # created file, and since the chunks have been sent back in
8      # string type, we have to encode them
9      for index, chunk in chunk_list :
10         encoded_data = str(chunk).encode()
11         file.write(encoded_data)
12
13     # At the end, close the file and the process is complete
14     file.close()
```

5 User Interface & Manual

For our project, we focus more on the logic so for now, we are only implementing the simple command line interface **(CLI)**. Users can interact with the application through the CLI by specific commands.

In this section, we shall guide you through the process of downloading file and some relevant commands.

A small note before we start the manual is that each OS may have their own way to compile a file, in the writer's case, is the "**py**" keywork. But if it is not the same for you, try other options like "**python3** [Filename.py] [args]" or "**python** [Filename.py] [args]".

5.1 Initialization

Open the CLI at the root directory where the file **tracker.py** resides and enter the following command to start the tracker app:

```
py tracker.py
```

After that, the CLI should show something like this that indicates the server has been started on the default ip **localhost** and port number **7000**:

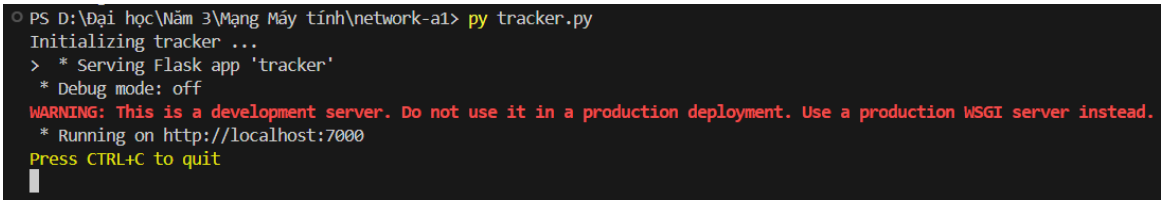

Figure 5: The tracker app has been started

At this part, let's leave the tracker for now and move on to the initialization of the peer apps. In the default repository, we have already provided you with 3 different folders named **peer1**, **peer2**, **peer3**, each contain a **data** folder which store the owned files of each peer and a file named **peer.py** that hold the source code of the peer app. For this test run, we will use all of them to better demonstrate the multi-file download aspect.

For each peer app, change directory inside the folder of each peer and then compile the **peer.py** file with arguments. The structure is:

```
cd [peer folder]
py peer.py [peer id] [peer ip] [peer listen/upload port]
```

For example: **Peer 1**:

```
cd peer1
py peer.py peer1 localhost 7001
```

Peer 2:

```
cd peer2
py peer.py peer2 localhost 7002
```

Peer 3:

```
cd peer3
py peer.py peer3 localhost 7003
```

After that, the CLI should announce the initialization of the peer apps:

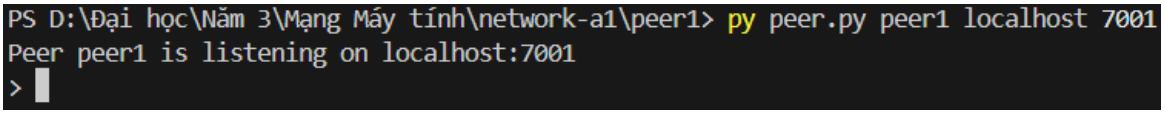

Figure 6: The peer app 1 has been started

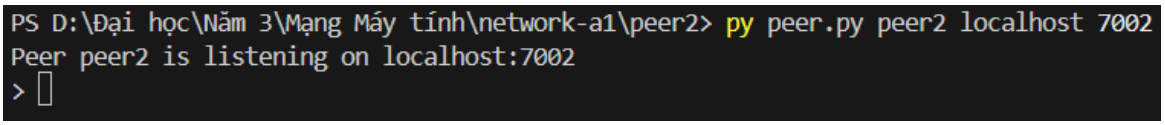

Figure 7: The peer app 2 has been started

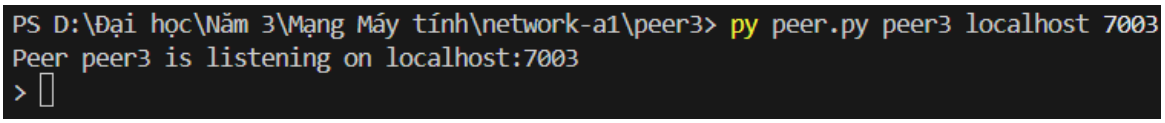

Figure 8: The peer app 3 has been started

Not that the new line has a sign, this indicates that we have enter the CLI thread, and now we must use specific commands to interact with the app.

5.2 Seeding file

For now, no file has been seeded to the tracker, so let's start seeding some. To do that, in the CLI of each peer, we can enter the following command:

```
SEED [file name]
```

Let's do that for **peer 1** and **peer 2**. Each of them has both files named **test1.txt** and **test2.txt** file in the data folder.

With peer 1, let's seed one file at a time:

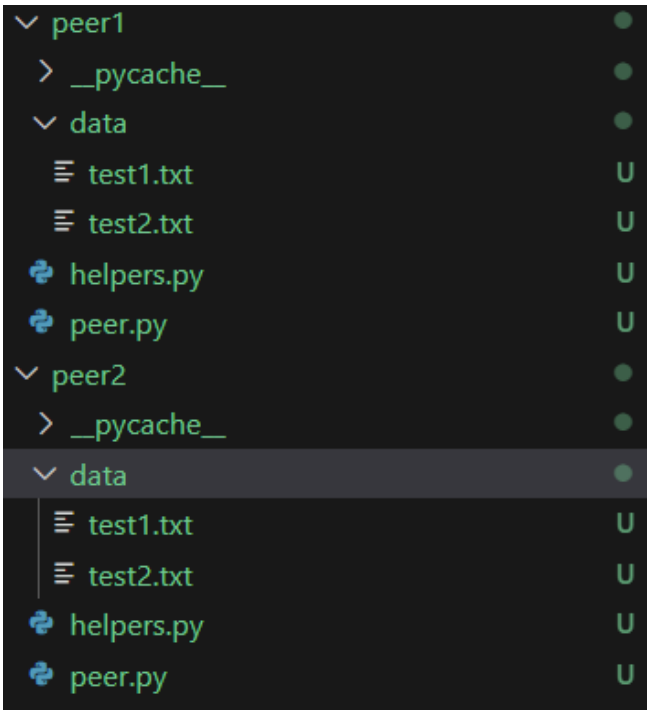

Figure 9: The structure of peer 1 and peer 2 folders

```
SEED test1.txt
```

After this first command, if you see something like the following picture then it was a success:

Figure 10: File test1.txt has been seeded successfully

Move back to the tracker, let's see its reaction to the request from peer 1:

Figure 11: Notes in the tracker for peer 1's request

Here we can see that the request from peer 1 has been logged onto the CLI, indicating that the tracker can receive **HTTP request**, in the case here is a **POST request**, from peer 1. Let's move forward with the seeding so that both peer 1 and peer 2 have seeded all their files to the tracker.

Peer 1:

```
SEED test2.txt
```

Peer 2:

```
SEED test1.txt test2.txt
```

Figure 12: Peer 2 sent 2 files as seeds at the same time

Note that here in the peer 2's CLI, we attempted to seed **2** files at the same time. The response from the tracker with the **status code 200** indicates that it can recognizes 2 files have been seeded.

5.3 View seeded files

Back to the tracker, we have a useful command to check the seeded files, let's give it a try:

```
view files
```

And the list of seeded files will be shown on the CLI along with its information, including the **name**, the **infohash**, the **size in bytes**, the **peers** that hold on the file. Each of the peer also has their **id** and **address displayed**. In the final line of printed information, we are also able to see the **total number** of files that have been seeded:

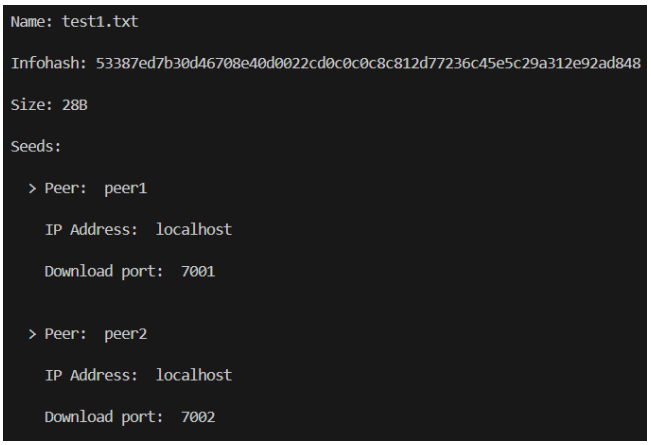

Figure 13: File test1.txt's data

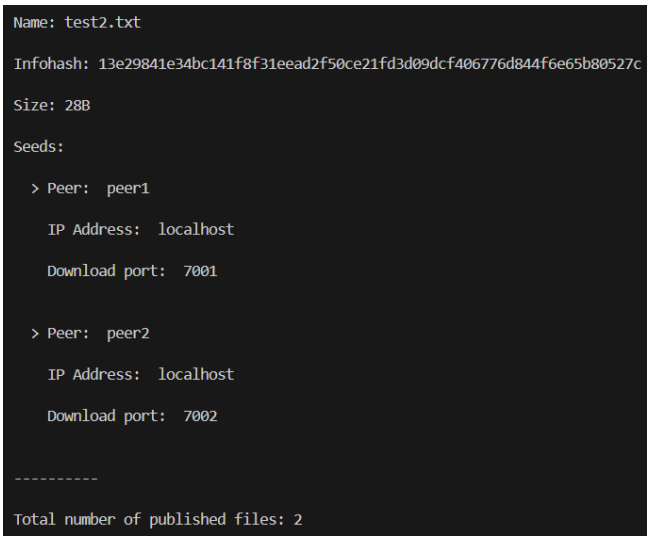

Figure 14: File test2.txt's data along with the total number of seeded files

5.4 Download file

Next we shall experience the core step of our application, the download command. With seeded files containing **test1.txt** and **test2.txt**, we shall choose another peer that does not have these 2 files, which is **peer 3**.

In peer 3 CLI, we will try the download for both of the 2 seeded files at the same time to see if it can live up to the task:

```
DOWNLOAD test1.txt test2.txt
```

After that, it should print a bunch of announcements out. Let's investigate its reaction to the command:

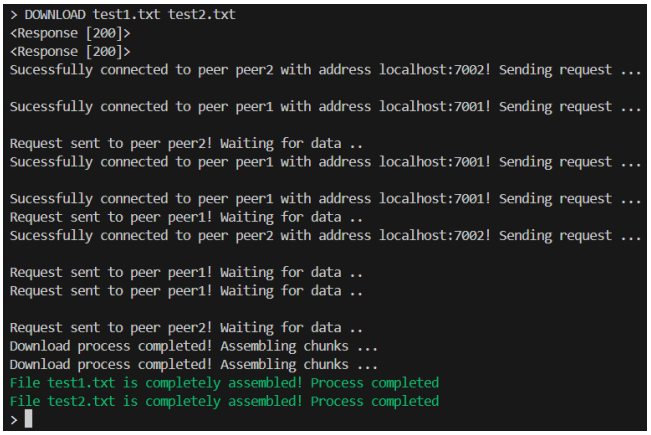

Figure 15: The printed announcements after the download of test1.txt and text2.txt

First, we can see 2 responses with status code 200 indicates that the tracker has acknowledged 2 requests, which were handled by 2 different threads, and send back the needed information.

Then, the peer app moved on to establish 2 connections to both peer 1 and peer 2. Note that since there are 2 files, each file was held by 2 peers so it established many requests **in parallel** and send them. At each **stage** of fetching chunks or assembling chunks, the application did us a favor of logging the information onto the screen.

On the other side, upon receiving requests, the application also logged out some information regarding the requests:

```
> Peer with address 127.0.0.1:51546 connected.

Request: {"name": "test1.txt", "id": "peer1", "index": 0, "count": 3}
Peer with address 127.0.0.1:51548 connected.

Request: {"name": "test1.txt", "id": "peer1", "index": 2, "count": 3}
10
Peer with address 127.0.0.1:51549 connected.

10
Request: {"name": "test2.txt", "id": "peer1", "index": 0, "count": 2}
14
A chunk has been sent back as requested by peer at 127.0.0.1:51546
A chunk has been sent back as requested by peer at 127.0.0.1:51548
A chunk has been sent back as requested by peer at 127.0.0.1:51549
```

Figure 16: Logged requests on the source socket side

5.5 Validate

In this step, we will do the checking manually to see if the application has run correctly. To do that, we shall compare the content of the files held by the original peer and the newly created file at the requesting peer:

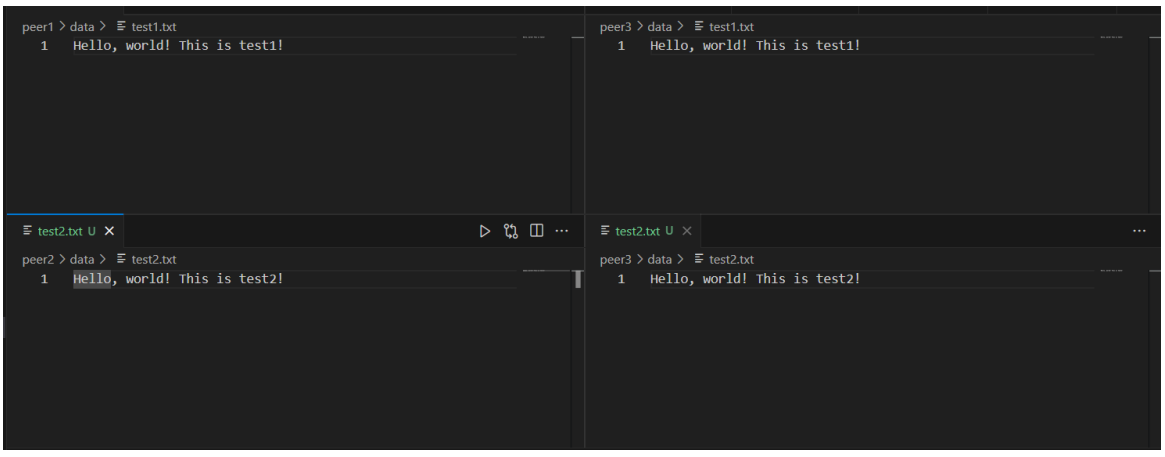

Figure 17: Comparison between destination and source files

On the right side, we choose 2 files from peer 1 and peer 2, which are the source, and on the right side, those 2 are corresponding version of peer 3.

By comparing the content inside, which are identical to each other, we can safely assume that the application has worked correctly.

5.6 Exit

Last but not least, after using the application, we can terminate the program by the following command, used for both tracker and peer:

```
exit
```

After that, the program will print an announcement saying that it is shutting down:

Figure 18: The exit of the program

6 Source code

Our program code is stored remotely using Github.

If you want to study it, please follow this https://github.com/Tonynghi/network-a1.

Or to try it locally on your PC, navigate with the CLI to desired directory for the clone and simply enter these commands:

```
1 git clone https://github.com/Tonynghi/network-a1.git
```

And it will help you clone a version to the local repository.

7 Conclusion

With the demonstration as the end for our description about a torrent-like network application, our report has come to an end.

7.1 Summary

In summary, we have covered the approach prior to the implementation, including requirements and system design. After that, we demonstrated the implementation of the application and it has fully functioned as required, both at the communication between tracker and peer or between peers themselves, and by making validation and comparison, the download process had been proven to be precise.

7.2 Improvements

While the application has run properly, there are still rooms for improvements. We can also further create more request for the peers to communicate with the tracker and get more information. Also, while the CLI can work fine, a GUI is always better and friendly to users. So for now we can see that there are many ways to develop our application in the future.