

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



COMPUTER NETWORK (CO3093)

Assignment 1

Develop A Network Application

Supervisor: Mr. Nguyen Le Duy Lai
Students: Vương Khang - 2250008
Võ Hoàng Long - 2053192
Trần Đăng Khoa – 2252363

HO CHI MINH CITY, November 2024

Contents

1	Member list & Workload	2
2	Introduction	3
3	Theory	4
4	Application design	5
5	Implementation	7
5.1	Modules.....	7
5.1.1	Connect	7
5.1.2	Disconnect.....	8
5.1.3	Bencode and Bdecode module	9
5.1.4	Create torrent file.....	10
5.1.5	Upload metainfo	12
5.1.6	Seeding.....	14
5.1.7	Download file.....	15
6	How to use our network application	18
7	Conclusion	23

1 Member list & Workload

No.	Fullname	Student ID	Percentage of work
1	Vương Khang	2250008	100%
2	Võ Hoàng Long	2053192	100%
3	Trần Đăng Khoa	2252363	100%

2 Introduction

This assignment aims to provide practical experience in building a P2P (Peer-to-Peer) file-sharing application similar to a simplified version of BitTorrent, a protocol supporting the practice of peer-to-peer file sharing that is used to distribute large amounts of data over the Internet.

Objective:

- Design and implement a network application using the TCP/IP protocol stack.
- Develop an understanding of how file-sharing works at a technical level.
- Apply theoretical knowledge of networking to a practical, real-world problem.
- Enhance programming skills, particularly in network socket programming and multi-threading.

Application Overviews:

- **Centralized Server:** The application will utilize a centralized server (tracker) to keep track of connected clients and the files they store.
- **Client-Server Interaction:** Clients will register their files with the server but will not upload actual file data to it. Instead, they inform the server about the local availability of files.
- **File Requests:** When a client needs a file it does not possess, it will request this file from the server. The server will respond with information about which clients have the file, allowing the requesting client to download pieces of the file directly from its peers.
- **Multi-threading Requirement:** The client application must handle multiple download and upload processes simultaneously, necessitating a multi-threaded approach.

Key Components:

- **Magnet Text:** A simple string containing essential information needed to retrieve the metainfo file from the centralized tracker.
- **Metainfo File:** Similar to a .torrent file, this contains data about the torrent, such as tracker address and file segmentation details.
- **Pieces:** Files are divided into manageable pieces as specified in the Metainfo File, typically around 256KB each.
- **Files:** These are the actual data files being shared and downloaded through the network.

Tracker HTTP:

- **Requests:** Clients must be able to send structured requests to the tracker for actions like starting, stopping, and completing downloads.
- **Responses:** The tracker's responses will follow a simplified version of the BitTorrent specification, providing necessary information like peer lists and transaction statuses.

3 Theory

Networking Fundamentals:

- **TCP/IP Protocol Stack:** Describe the four layers (Application, Transport, Internet, Link) and their roles in data transmission across the network.
- **Sockets:** Explain how sockets serve as endpoints in network communication, distinguishing between TCP (connection-oriented, reliable) and UDP (connectionless, faster but less reliable).

Peer-to-Peer (P2P) Networking:

- **Architecture:** Contrast the decentralized nature of P2P networks with the centralized nature of client-server models, highlighting the advantages in terms of scalability and fault tolerance.
- **Resource Sharing:** Discuss how P2P networks leverage the resources of participating peers, reducing reliance on central servers and potentially improving system resilience and data availability.
- **Peer Discovery and Management:** Explain how peers in a network discover each other (e.g., via a central tracker or using distributed hash tables in trackerless systems) and manage connections.

File sharing protocols

- **BitTorrent Protocol:** Provide an overview of how the BitTorrent protocol manages data distribution among multiple peers, detailing the roles of torrent files, trackers, and the process of piece selection and transfer.
- **Chunk-based File Sharing:** Describe the method of breaking files into smaller, manageable pieces (chunks) for transmission, which allows for more efficient data handling and error recovery.

Concurrency and Multithreading

- **Importance in Networking:** Explain how multithreading can handle multiple user requests and data transfers simultaneously, enhancing the efficiency and responsiveness of network applications.
- **Challenges:** Discuss potential issues such as race conditions, deadlock, and the need for synchronization mechanisms to ensure data consistency and thread safety.

4 Application design

We implement the P2P network in our networking application as shown in the diagram below. The functionality of each module:

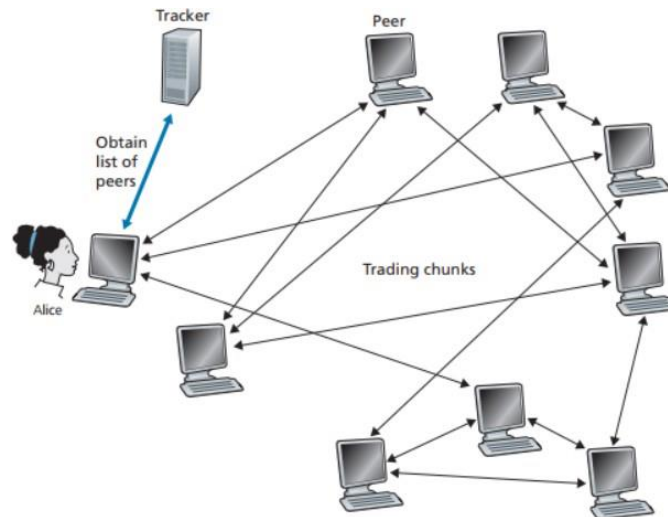


Figure 1: P2P model diagram

Peer Module:

- **Resource Sharing:**

1. **Providers and Consumers:** In a P2P network, every peer acts as both a provider and a consumer of resources. This means that each peer contributes to and utilizes the network's resources, such as bandwidth, storage, and processing power.
2. **Data Distribution:** Peers are responsible for hosting and distributing files. They share pieces of files they own or have downloaded with other peers, enhancing data availability and network resilience.

- **Data Management:**

1. **File Division:** Peers typically break down files into smaller segments or chunks to facilitate easier management, faster transfer, and more reliable downloading.
2. **Data Integrity:** Peers often manage checksums or hashes for file segments to ensure data integrity during transfers, preventing corruption or tampering.

- **Network Dynamics:**

1. **Dynamic Joining and Leaving:** Peers can join and leave the network freely, which necessitates robust mechanisms to handle such dynamic changes without affecting the overall network performance or data availability.

2. **Flow Control:** Effective load balancing is crucial, especially in popular networks to prevent any single peer from becoming a bottleneck due to excessive requests.

- **Error and Control Management:**

1. **Error Handling:** Peers need to handle errors gracefully, such as interruptions in data transfer or the unavailability of a file segment, by finding alternative sources for the required data.
2. **Flow Control:** They regulate data flow based on network conditions and peer capabilities to optimize throughput and reduce congestion.

Tracker Module:

- **Network Coordination:**

1. **Peer Discovery:** The tracker facilitates the discovery of peers. When a new peer wants to download a file, the tracker provides a list of peers that hold the pieces of the file, enabling the initiating peer to establish connections for data transfer.
2. **Maintaining Peer Lists:** Trackers maintain and update lists of active peers and their available resources, which are crucial for directing peers where to find file segments.

- **Resource Allocation:**

1. **Load Distribution:** Trackers monitor the load on different peers and can suggest more optimal connections to prevent overloading any single peer, thus balancing the network load.
2. **Efficiency Optimization:** By directing new peers to less busy or geographically closer peers, trackers can enhance the efficiency and speed of data transfers.

5 Implementation

5.1 Modules

5.1.1 Connect

```
1 def connect_to_server(server_host, server_port, client_ip, client_port,
2   client_id):
3     try:
4         payload = {'command': 'connect', 'port': client_port, 'ip':
5         client_ip, 'id': client_id}
6         tracker_url = f'http://{server_host}:{server_port}/announce'
7         response = requests.post(tracker_url, json=payload)
8
9         if response.ok:
10             data = response.json()
11             print("Status:", data.get('status'))
12             print("Message:", data.get('message'))
13         else:
14             data = response.json()
15             print("Failed to connect to server:", response.status_code)
16             print("Message:", data.get('message'))
17         return True
18     except (ConnectionError, HTTPError) as e:
19         print(f"Failed to connect to server: {e}")
20         reconnect_status = reconnect_to_server(tracker_url, payload)
21         if(reconnect_status == True):
22             return
23     except Exception as e:
24         print(f"An error occurred: {e}")
25         return False
```

Whenever a new peer wishes to join the BitTorrent network, it is required to connect to the tracker server (the central global tracker portal) to transmit its details, including its IP address, port, and identifier. As demonstrated in the provided code, the peer submits a request to the tracker via the HTTP protocol. The tracker then processes these requests and issues a response indicating whether the peer is permitted to connect. If the connection is successful, the tracker will retain the information corresponding to the peer. The code below demonstrates how tracker handle the connect request from peers.

```
1 def handle_connect(data):
2     with lock:
3         if len(peers) < MAX_PEERS:
4             client_ip = data['ip']
5             client_port = data['port']
6             client_id = data['id']
7             for peer in peers:
8                 if peer['ip'] == client_ip and peer['port'] ==
9                 client_port:
10                     print("This peer already connected")
```



```

10         return jsonify({'status': 'fail', 'message': 'You
have already connected'}), 400
11     peers.append({'ip': client_ip, 'port': client_port, 'id':
client_id})
12     print(f"Peer {client_ip}, {client_port} has connected to
server")
13     print("Current list of connected peers:")
14     for peer in peers:
15         print(f"Peer IP: {peer['ip']}, Port: {peer['port']}, ID:
{peer['id']}")
16     return jsonify({'status': 'success', 'message': 'Connection
established'}), 200

```

5.1.2 Disconnect

```

1 def disconnect_from_server(server_host, server_port, client_ip,
client_port, client_id):
2     try:
3         # Prepare the payload with disconnect command
4         payload = {'command': 'disconnect', 'ip': client_ip, 'port':
client_port, 'id': client_id}
5         # Send the POST request to the server's endpoint
6         tracker_url = f'http://{server_host}:{server_port}/announce'
7         response = requests.post(tracker_url, json=payload)
8
9         if response.ok:
10             # Parse the JSON response from the server
11             data = response.json()
12             print("Status:", data.get('status'))
13             print("Message:", data.get('message'))
14         else:
15             data = response.json()
16             # Handle non-200 responses
17             print("Failed to disconnect:", response.status_code)
18             print("Message:", data.get('message'))
19         except (ConnectionError, HTTPError) as e:
20             print(f"Failed to connect to server: {e}")
21             reconnect_status = reconnect_to_server(tracker_url, payload)
22             if(reconnect_status == True):
23                 return
24         except Exception as e:
25             # Handle exceptions that may occur during the request
26             print(f"An error occurred: {e}")

```

When a peer no longer wishes to remain connected to the tracker, it sends a disconnect request. Upon receiving this request, the tracker locates the relevant peer's information (IP address, port, and identifier) and proceeds to remove it. If the tracker successfully deletes the peer's data, it sends a response back to the peer to confirm that the connection has been terminated and the peer can leave the network. The code below demonstrates how tracker handle the disconnect request from peers.

```

1 def handle_disconnect(data):
2     with lock:
3         client_ip = data['ip']

```

```
4     client_port = data['port']
5     for peer in peers:
6         if peer['ip'] == client_ip and peer['port'] == client_port:
7             peers.remove(peer)
8             print(f"Peer {client_ip}, {client_port} has disconnected
from server")
9             print("Current list of connected peers:")
10            for peer in peers:
11                print(f"Peer IP: {peer['ip']}, Port: {peer['port']}")
12        )
13        return jsonify({'status': 'success', 'message': '
Connection terminated'}), 200
14    return jsonify({'status': 'error', 'message': 'Peer not found'})
, 404
```

5.1.3 Bencode and Bdecode module

```
1 def bencode(data):
2     """
3     Encode Python objects into bencoded bytes.
4     """
5     if isinstance(data, str):
6         # Convert Python str to bytes and then encode as bencoded byte
string
7         return f"{len(data)}:{data}".encode()
8     elif isinstance(data, bytes):
9         # Directly encode bytes to bencoded byte string
10        return f"{len(data)}:".encode() + data
11    elif isinstance(data, int):
12        # Encode Python integers to bencoded integers
13        return f"i{data}e".encode()
14    elif isinstance(data, list):
15        # Recursively bencode each item in the list
16        encoded_list = b"l" + b"".join(bencode(item) for item in data) +
b"e"
17        return encoded_list
18    elif isinstance(data, dict):
19        # Encode dictionaries, ensuring keys are sorted as raw strings
20        encoded_dict = b"d"
21        # Sort keys and ensure they are byte strings, encode values
recursively
22        for key, value in sorted(data.items(), key=lambda item: item[0]):
23            encoded_key = bencode(str(key))
24            encoded_value = bencode(value)
25            encoded_dict += encoded_key + encoded_value
26        encoded_dict += b"e"
27        return encoded_dict
28    else:
29        raise TypeError(f"Type {type(data)} not supported by bencode.")
30
31 def bdecode(data):
32     """
33     Decode bencoded bytes back into Python objects.
34     Handles byte strings as raw bytes to avoid decoding errors.
```

```

35 """
36 def decode_next(data, index):
37     start_char = chr(data[index])
38
39     if start_char.isdigit():
40         colon_index = data.index(b':', index)
41         length = int(data[index:colon_index])
42         start = colon_index + 1
43         end = start + length
44         # Only decode as string if it's safe to assume it's not
binary data
45         segment = data[start:end]
46         if all(32 <= byte < 127 for byte in segment): # Rough check
for text-safety
47             return segment.decode(), end
48             return segment, end # Return as bytes if potentially binary
49
50     elif start_char == 'i':
51         end_index = data.index(b'e', index)
52         num = int(data[index + 1:end_index])
53         return num, end_index + 1
54
55     elif start_char == 'l':
56         index += 1
57         lst = []
58         while data[index] != ord('e'):
59             item, index = decode_next(data, index)
60             lst.append(item)
61         return lst, index + 1
62
63     elif start_char == 'd':
64         index += 1
65         dic = {}
66         while data[index] != ord('e'):
67             key, index = decode_next(data, index)
68             value, index = decode_next(data, index)
69             dic[key] = value
70         return dic, index + 1
71
72     else:
73         raise ValueError("Invalid bencoded value")
74
75     result, _ = decode_next(data, 0)
76     return result

```

The description and usage of this module will be mentioned in the below modules

5.1.4 Create torrent file

```

1 def create_torrent_file(server_host, server_port, filename, client_id):
2     # Create torrent file
3     dir = "peer_" + str(client_id)
4     full_output_path = os.path.join(dir, filename)
5     if not os.path.exists(full_output_path):
6         print("You must have the file in the directory to create a
torrent file!")

```

```
7     return
8
9     piece_length = 512*1024 # 512kB piece size
10    file_size = os.path.getsize(full_output_path)
11    num_pieces = (file_size + piece_length - 1) // piece_length #
    Calculates the necessary number of pieces
12
13    # Initialize the 'pieces' as bytes
14    pieces = b''
15
16    # Progress bar setup
17    progress_bar = tqdm(total=num_pieces, unit='piece', desc='Creating
    torrent', leave=True)
18
19    # Calculate SHA1 hash for each piece and concatenate
20    with open(full_output_path, 'rb') as file:
21        for _ in range(num_pieces):
22            piece = file.read(piece_length)
23            pieces += hashlib.sha1(piece).digest() # Append the binary
    hash directly
24            progress_bar.update(1)
25    progress_bar.close()
26
27    # Metadata for the torrent
28    tracker_url = f'http://{server_host}:{server_port}/announce'
29    metadata = {
30        'announce': tracker_url,
31        'info': {
32            'name': filename,
33            'length': file_size,
34            'piece length': piece_length,
35            'pieces': pieces, # Assign the byte string containing all
    piece hashes
36        }
37    }
38    # Ensure you bencode the entire metadata dictionary
39    bencoded_data = bencode(metadata)
40
41    torrent_filename = f"{filename}.torrent"
42    full_output_path = os.path.join(dir, torrent_filename)
43    with open(full_output_path, 'wb') as torrent_file:
44        torrent_file.write(bencoded_data)
45
46    print("Torrent file created")
```

This module is the most important one in a Bittorrent network. When a peer (a seeder) holds a file and want to transmit it to other peers, it must create a ".torrent" file. This file will contain some specific fields:

- **announce:** The announce URL of the tracker (string)
- **info:** a dictionary that describes the file(s) of the torrent.

Inside the **info** field is another dictionary:

- **name:** the filename (string).

- **length:** length of the file in bytes (integer).
- **piece length:** number of bytes in each piece (integer).
- **pieces:** string consisting of the concatenation of all 20-byte SHA1 hash values, one per piece (byte string, i.e. not urlencoded).

As we can see in the above code, these information of torrent file is store as a dictionary in variable *metadata*. After that, we must use bencode module to specify and organize data in a terse format. The bencoding rules:

- **Byte Strings:** Byte strings are encoded as follows: <string length encoded in base ten ASCII>:<string data>. Note that there is no constant beginning delimiter, and no ending delimiter.
Example: "4:spam" represents the string "spam" with length 4. "o:" represents the empty string "".
- **Integers:** Integers are encoded as follows: i<integer encoded in base ten ASCII>e . The initial i and trailing e are beginning and ending delimiters.
Example: "i3e" represents the integer "3". "i-3e" represents the integer "-3"
- **Lists:** Lists are encoded as follows: l<bencoded values>e . The initial l and trailing e are beginning and ending delimiters. Lists may contain any bencoded type, including integers, strings, dictionaries, and even lists within other lists.
Example: "l4:spam4:eggse" represents the list of two strings: ["spam", "eggs"]. "le" represents an empty list: [].
- **Dictionaries:** Dictionaries are encoded as follows: d<bencoded string><bencoded element>e . The initial d and trailing e are the beginning and ending delimiters. Note that the keys must be bencoded strings. The values may be any bencoded type, including integers, strings, lists, and other dictionaries. Keys must be strings and appear in sorted order (sorted as raw strings, not alphanumerics). The strings should be compared using a binary comparison, not a culture-specific "natural" comparison.
Example: "d3:cow3:moo4:spam4:eggse" represents the dictionary "cow" => "moo", "spam" => "eggs" .
"d4:spam11:a1:bee" represents the dictionary "spam" => ["a", "b"] .

After bencoding successfully, we can write this bencoded data to a ".torrent" file. Then, this file is transmitted to other peer for further processing that will be discussed later.

5.1.5 Upload metainfo

```
1 def upload_info_hash_to_tracker(server_host, server_port, client_ip,
2   client_port, client_id, filename):
3     # Connect to the tracker and send torrent information
4     try:
5         dir = "peer_" + str(client_id)
6         # Check if file and torrent file exist
7         full_output_path = dir + "/" + filename
8         if not os.path.exists(full_output_path):
```

```

8         print("You don't have the file in the directory!")
9         return
10
11     torrent_filename = f"{filename}.torrent"
12     full_output_path = dir + "/" + torrent_filename
13     if not os.path.exists(full_output_path):
14         print("Create torrent file before uploading!")
15         return
16
17     with open(full_output_path, 'rb') as file:
18         metadata = bdecode(file.read())
19         bencoded_info = bencode(metadata['info'])
20
21     files.append({'filename': filename, 'pieces': metadata['info']['
pieces']})
22     headers = {'Content-Type': 'application/json'}
23     data = {
24         'command': 'upload info',
25         'peer_ip': client_ip,
26         'peer_port': client_port,
27         'peer_id': client_id,
28         'filename': filename,
29         'info_hash': hashlib.sha1(bencoded_info).hexdigest()
30     }
31
32     try:
33         tracker_url = f"http://{server_host}:{server_port}/announce'
34         response = requests.post(tracker_url, json=data, headers=
headers)
35
36         if response.ok:
37             print(f"Uploaded torrent info for {filename} to tracker"
)
38             print("Received from server:", response.json())
39         else:
40             print("Failed to upload torrent info:", response.
status_code)
41             print(response.text)
42     except (ConnectionError, HTTPError) as e:
43         print(f"Failed to connect to server: {e}")
44         reconnect_status = reconnect_to_server(tracker_url, data)
45         if(reconnect_status == True):
46             return
47     except Exception as e:
48         print(f"An error occurred: {e}")

```

When a seeder create a ".torrent" file successfully, it will contact the tracker to transmit its *info_hash* (the "info" dictionary of metadata is bencoded using SHA-1 to produce the *info_hash*). The *info_hash* is then used as a unique identifier for the torrent within the BitTorrent network. It is sent to the tracker when making requests to the tracker's announce URL, helping the tracker to identify and manage peers for this specific torrent. Whenever a peer want to download a file through a specific ".torrent" file, it will contact the tracker to get the list of seeders who have sent the *info_hash* to tracker.

5.1.6 Seeding

```
1 def send_file_piece(client_socket, file_name, peer_id, piece_index,
2 piece_size, file_pieces):
3     # Calculate the byte offset for the requested piece
4     offset = piece_index * piece_size
5     if piece_index < len(file_pieces):
6         dir = "peer_" + str(peer_id)
7         full_output_path = os.path.join(dir, file_name)
8         with open(full_output_path, 'rb') as file:
9             file.seek(offset)
10            piece_data = file.read(piece_size)
11            if piece_data:
12                client_socket.sendall(piece_data)
13                print(f"Sent piece index {piece_index}")
14            else:
15                print("No data read from file; possible end of file.")
16        else:
17            print(f"Piece index {piece_index} is out of range.")
18
19 def start_seeder_server(ip, port):
20     def client_handler(client_socket):
21         try:
22             while True:
23                 request = client_socket.recv(1024)
24                 if request.startswith(b'get_piece'):
25                     decoded_request = request.decode('utf-8')
26                     piece_index = int(decoded_request.split()[1])
27                     file_name = decoded_request.split()[3]
28                     seeder_id = int(decoded_request.split()[5])
29                     client_ip = decoded_request.split()[6]
30                     print("Connection from", client_ip, "for piece",
31 piece_index, "of", file_name)
32                     for file in files:
33                         if file['filename'] == file_name:
34                             file_pieces = file['pieces']
35                             piece_size = 256 * 1024
36                             send_file_piece(client_socket, file_name,
37 seeder_id, piece_index, piece_size, file_pieces)
38                             break
39
40                     return
41                 else:
42                     print("No valid request received.")
43                     return
44         except socket.error as e:
45             print("Socket error:", e)
46             client_socket.close()
47         finally:
48             client_socket.close()
49
50     server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
51     server_socket.bind((ip, port))
52     server_socket.listen(5)
53     print("Seeder listening on", ip, ":", port)
```

```
52
53     while True:
54         client_socket, addr = server_socket.accept()
55         Thread(target=client_handler, args=(client_socket,), daemon=True).start()
```

When a peer upload its *info_hash* to tracker successfully, it can enter the seeder mode to become a seeder for transmitting piece of files. This seeder will create thread to handle for connection to get pieces from other peers.

5.1.7 Download file

```
1 def download_torrent(torrent_filename, client_ip, client_id):
2     # Decode the torrent file
3     dir = "peer_" + str(client_id)
4     full_output_path = dir + "/" + torrent_filename
5     with open(full_output_path, 'rb') as file:
6         torrent_data = bdecode(file.read())
7         tracker_url = torrent_data['announce']
8         info_hash = hashlib.sha1(bencode(torrent_data['info'])).
9         hexdigest()
10         filename = torrent_data['info']['name']
11
12     all_hashes = torrent_data['info']['pieces']
13     hash_length = 20 # SHA-1 hashes are 20 bytes long
14     num_pieces = len(all_hashes) // hash_length
15     piece_length = torrent_data['info']['piece length']
16
17     validated_pieces = [None] * num_pieces
18     peer_idx = 0
19     i = 0
20     while i < num_pieces:
21         # Contact the tracker to get peers
22         payload = {
23             'command': 'get_peers',
24             'info_hash': info_hash
25         }
26         try:
27             response = requests.post(f'{tracker_url}', json=payload)
28             if response.ok:
29                 data = response.json()
30                 if data['status'] == 'success':
31                     print("Peers holding the file:", data['peers'])
32                 else:
33                     print("No peers found or error:", data['message'])
34                     break
35             else:
36                 print("Failed to contact tracker:", response.status_code)
37
38         )
39         print("Tracker recover, ask seeder connect again.")
40         return
41
42     except (ConnectionError, HTTPError) as e:
43         print(f"Failed to connect to server: {e}")
44         reconnect_status = reconnect_to_server(tracker_url, payload)
45         if(reconnect_status == True):
```



```
43         return
44
45     number_of_peers = len(data['peers'])
46     while number_of_peers == 0:
47         print("No peers found. Trying again in 5 seconds.")
48         time.sleep(5)
49         try:
50             response = requests.post(f'{tracker_url}', json=payload)
51             if response.ok:
52                 data = response.json()
53                 if data['status'] == 'success':
54                     print("Peers holding the file:", data['peers'])
55                     number_of_peers = len(data['peers'])
56                 else:
57                     print("No peers found or error:", data['message'])
58         except:
59             print("Failed to contact tracker:", response.
status_code)
60     except (ConnectionError, HTTPError) as e:
61         print(f"Failed to connect to server: {e}")
62         reconnect_status = reconnect_to_server(tracker_url,
payload)
63         if(reconnect_status == True):
64             return
65
66     if(peer_idx >= number_of_peers):
67         peer_idx = 0
68     seeder_ip, seeder_port, seeder_id = data['peers'][peer_idx]['ip'
], int(data['peers'][peer_idx]['port']), int(data['peers'][peer_idx]['id'])
69
70     client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM
)
71     try:
72         client_socket.connect((seeder_ip, seeder_port))
73     except ConnectionRefusedError:
74         print(f"Failed to connect to {seeder_ip}:{seeder_port},
trying next peer if available.")
75
76     peer_idx += 1
77     if(peer_idx == number_of_peers):
78         peer_idx = 0
79         time.sleep(5)
80         continue
81
82     peer_idx += 1
83     if(peer_idx == number_of_peers):
84         peer_idx = 0
85
86     client_socket.send(f"get_piece {i} of {filename} at_peer {
seeder_id} {client_ip}\n".encode())
87     piece = client_socket.recv(piece_length) # Use the full data
receiver function mentioned before
88     piece_hash = all_hashes[i * hash_length:(i + 1) * hash_length]
89     if hashlib.sha1(piece).digest() == piece_hash:
```

```

90         print(f"Received and validated piece {i} from {seeder_ip}:{seeder_port}")
91         validated_pieces[i] = piece
92         i = i + 1
93     else:
94         print(f"Piece {i} is corrupted")
95         continue
96
97     client_socket.shutdown(socket.SHUT_RDWR)
98     client_socket.close()
99     time.sleep(0.5)
100
101     #Create directory if not exists
102     directory = "peer_" + str(client_id)
103     if not os.path.exists(directory):
104         os.makedirs(directory)
105
106     full_output_path = os.path.join(directory, torrent_data['info']['name'])
107     with open(full_output_path, 'wb') as file:
108         for piece in validated_pieces:
109             if piece is not None:
110                 file.write(piece)
111     print(f"File has been successfully created")
112     print("Download completed and connection closed.")

```

Now, when a peer (a leecher) get the ".torrent" file for a file that it requests, it will have to decode the ".torrent" file to get information of tracker and the file name that it is going to download. As we can see in the code, after decoding the file, the leecher then announce the tracker through url in ".torrent" file to get the list of seeders who have the files to transmit. Then, leecher create socket for TCP connection with other seeders. It send "Get piece" request to corresponding seeders and download piece by piece. When a piece is downloaded successfully, it is checked against the corresponding SHA-1 hash piece contained in the ".torrent" file's metadata. This is crucial because it ensures the integrity and correctness of each piece of the file. If a piece doesn't match its hash, it's discarded and re-downloaded. If you want to observe the download piece process, uncomment the `time.sleep()` and adjust the time.

After downloading and checking the correctness of the pieces, these pieces will be stored and ordered in variable *validated_pieces* for writing to a complete file. The code below is for tracker return list of peers holding the file that leecher requires.

```

1 def handle_get_peers(data):
2     with lock:
3         info_hash = data['info_hash']
4         if info_hash in torrents:
5             peer_info = [peer for peer in torrents[info_hash]['peers']]
6             return jsonify({'status': 'success', 'message': 'Peer data
retrieved', 'peers': peer_info}), 200
7         return jsonify({'status': 'error', 'message': 'Torrent not found
'}), 404

```

6 How to use our network application

In the beginning, we have to identify number of peers we want to create connection with tracker. Then we create folder corresponds to id of peer we want to create.





 peer_1	11/19/2024 1:28 PM	File folder
 peer_2	11/19/2024 11:49 AM	File folder
 peer_3	11/19/2024 11:48 AM	File folder
 peer_4	11/19/2024 1:29 PM	File folder

Figure 2: *Peers folder*

Next, open terminal to run the tracker.py, define the ip address and port for tracker (ip address must be the public address, for example IPv4 Address of the wireless LAN adapter Wi-Fi, the address of my Wifi is 192.168.1.5). If it run successfully, we will get the message "Running on http://192.168.1.5:8000"

```
C:\Users\LEGION\Downloads\Computer Network\Assignment\BitTorrent-Python-main>python tracker.py 192.168.1.5 8000
The server is listening...
* Serving Flask app 'tracker'
* Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://192.168.1.5:8000
Press CTRL+C to quit
```

Figure 3: *Initiate tracker*

We only need to run the tracker once. After running tracker, open another terminal for peer. We run this command to create a peer connect to tracker:

"python peer.py server ip server port client ip client port client id"

If peer connects to tracker successfully, the message "Connection established" will be printed

```
C:\Users\PNN\Downloads\test>python peer.py 192.168.1.5 8000 192.168.1.78 8000 1
Status: success
Message: Connection established
Enter a command (create torrent, upload, download, disconnect, seeder, exit):
```

Figure 4: *Initiate peer*

The tracker also print the list of currently connected peers

file. We will receive the message "Torrent info registered successfully" if the seeder has completed sending info. Finally, enter the command "seeder" to inform other peers that peer 1 is ready for seeding pieces of its file and waiting for other peers' requests.

```
Enter a command (create torrent, upload, download, disconnect, seeder, exit): upload
Enter the filename to seed: test.txt
Uploaded torrent info for test.txt to tracker
Received from server: {'message': 'Torrent info registered successfully', 'status': 'success'}
Enter a command (create torrent, upload, download, disconnect, seeder, exit): seeder
Seeder listening on 192.168.1.78 : 8000
Enter a command (create torrent, upload, download, disconnect, seeder, exit): █
```

Figure 9: *Seeder upload its info and become seeder waiting for other peers' requests*

To create another peer that has demand to download the file from peer 1, open another terminal and run this command to create a peer connect to tracker:

"python peer.py server ip server port client ip client port client id"

For example, here I create a peer with ip address 192.168.1.5, port 8001, id 2.

```
C:\Users\LEGION\Downloads\Computer Network\Assignment\BitTorrent-Python-main>python peer.py 192.168.1.5 8000 192.168.1.5 8001 2
Status: success
Message: Connection established
Enter a command (create torrent, upload, download, disconnect, seeder, exit): |
```

Figure 10: *Initiate peer 2*

Before downloading pieces from peer 1, we must ensure that in folder peer_2 has ".torrent" file (in this example is the "test.txt.torrent" file that has been created by peer 1). Then, enter the command "download" and ".torrent" filename to start download pieces.

```
Enter a command (create torrent, upload, download, disconnect, seeder, exit): Connection from 192.168.1.5 for piece 0 of test.txt
Sent piece index 0
Connection from 192.168.1.5 for piece 1 of test.txt
Sent piece index 1
Connection from 192.168.1.5 for piece 2 of test.txt
Sent piece index 2
Connection from 192.168.1.5 for piece 3 of test.txt
Sent piece index 3
Connection from 192.168.1.5 for piece 4 of test.txt
Sent piece index 4
```

Figure 11: *Peer1 sending pieces*

```
Enter a command (create torrent, upload, download, disconnect, seeder, exit): Peers holding the file: [{'id': '1', 'ip': '192.168.1.78', 'port': 8000}]
Received and validated piece 0 from 192.168.1.78:8000
Peers holding the file: [{'id': '1', 'ip': '192.168.1.78', 'port': 8000}]
Received and validated piece 1 from 192.168.1.78:8000
Peers holding the file: [{'id': '1', 'ip': '192.168.1.78', 'port': 8000}]
Received and validated piece 2 from 192.168.1.78:8000
Peers holding the file: [{'id': '1', 'ip': '192.168.1.78', 'port': 8000}]
Received and validated piece 3 from 192.168.1.78:8000
Peers holding the file: [{'id': '1', 'ip': '192.168.1.78', 'port': 8000}]
Received and validated piece 4 from 192.168.1.78:8000
```

Figure 12: *Peer2 receiving pieces from Peer1*

As we can see from the picture, peer 1 send each piece to peer 2, when peer 2 receive a piece, it immediately validate the correction of piece. When all pieces download successfully without any corruption, peer 2 will merge validated pieces and write to a complete file.

```
Peers holding the file: [{'id': '1', 'ip': '192.168.1.78', 'port': 8000}]
Received and validated piece 26 from 192.168.1.78:8000
File has been successfully created
Download completed and connection closed.
```

Figure 12: *Download pieces complete*

When peer 2 get the completed file, it can become a seeder by using command "upload", "seeder" same as peer1.

```
Download completed and connection closed.
upload
Enter the filename to seed: test.txt
Uploaded torrent info for test.txt to tracker
Received from server: {'message': 'Torrent info registered successfully', 'status': 'success'}
Enter a command (create torrent, upload, download, disconnect, seeder, exit): seeder
Seeder listening on 192.168.1.5 : 8001
Enter a command (create torrent, upload, download, disconnect, seeder, exit): |
```

Figure 14: *Peer 2 become seeder after complete download*

Now when peer 3 enter the Bittorrent network and also want to download the file "test.txt", it can get pieces from peer 1 and peer 2 respectively by repeating the above process.

7 Conclusion

In conclusion, the development of the P2P file-sharing application as part of this project has provided invaluable practical experience in network application design using the TCP/IP protocol stack. The project has effectively demonstrated how theoretical knowledge in computer networking can be applied to real-world scenarios, enhancing both programming and problem-solving skills among team members.

The P2P application, inspired by the BitTorrent protocol, leverages a decentralized network architecture, allowing peers to act both as servers and clients, thus optimizing resource usage and enhancing the scalability and resilience of the network. This design is particularly beneficial for handling large-scale data distribution, which is common in today's data-driven environment.

The use of a centralized tracker to manage connections and data flow among peers has also illustrated important aspects of network management, including peer discovery and load balancing. The tracker plays a crucial role in maintaining the efficiency of the network, ensuring that files are accessible even if individual peers leave the network.

Throughout the project, the implementation of multi-threading was critical in allowing the application to handle multiple simultaneous data transfers efficiently, showcasing the practical challenges and solutions in concurrent programming.

Future improvements could focus on enhancing the security features of the application, implementing more robust error handling and recovery mechanisms, and exploring the use of more sophisticated algorithms for peer discovery and data distribution.

Overall, this project has not only solidified our understanding of networked applications and distributed systems but has also prepared us to tackle more complex networking challenges in the future, paving the way for further innovations in the field of computer networking.

References

- [1] *Computer Networking_A Top-Down Approach, Global Edition, 8th Edition, 2022.*
- [2] *Luis Soares, 2023, Implement Peer-to-Peer Exchange in Python,*<https://blog.devgenius.io/implementing-peer-to-peer-data-exchange-in-python-8e695134>