

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



## COMPUTER NETWORKS (CO3093)

---

Assignment 1 - Semester 241

# Develop a Network Application

---

Instructor: Nguyễn Mạnh Thìn, *CSE-HCMUT*

Students: Cao Ngọc Lâm - 2252419  
Nguyễn Châu Hoàng Long - 2252444  
Trịnh Anh Minh - 2252493  
Hồ Khánh Nam - 2252500

HO CHI MINH CITY, NOVEMBER 2023



## Contents

<b>1</b>	<b>Assignment Specifications</b>	<b>3</b>
1.1	Objective . . . . .	3
1.2	Application description . . . . .	3
<b>2</b>	<b>Phase 1</b>	<b>5</b>
2.1	System description . . . . .	5
2.1.1	Core functionalities . . . . .	5
2.1.2	Additional functionalities . . . . .	5
2.2	Diagram . . . . .	6
2.3	Layered architecture . . . . .	7
2.4	Tracker Protocol . . . . .	8
2.5	Peer-Downloading . . . . .	14
2.6	Peer-Uploading . . . . .	18
<b>3</b>	<b>Phase 2</b>	<b>20</b>
3.1	Manual document . . . . .	20
3.2	Basic Testing . . . . .	21
<b>4</b>	<b>Conclusion</b>	<b>27</b>



## Member list & workload

No.	Full name	Student ID	Assignment	Effort
1	Cao Ngọc Lâm	2252419	Implement CLI for Node and Tracker Implement Protocol: uploading and downloading	100%
2	Nguyễn Châu Hoàng Long	2252444	Write Report & implement request queue algorithm	100%
3	Trịnh Anh Minh	2252493	Write Report & Sanity Test	100%
4	Hồ Khánh Nam	2252500	Write Report & Sanity Test	100%

# 1 Assignment Specifications

## 1.1 Objective

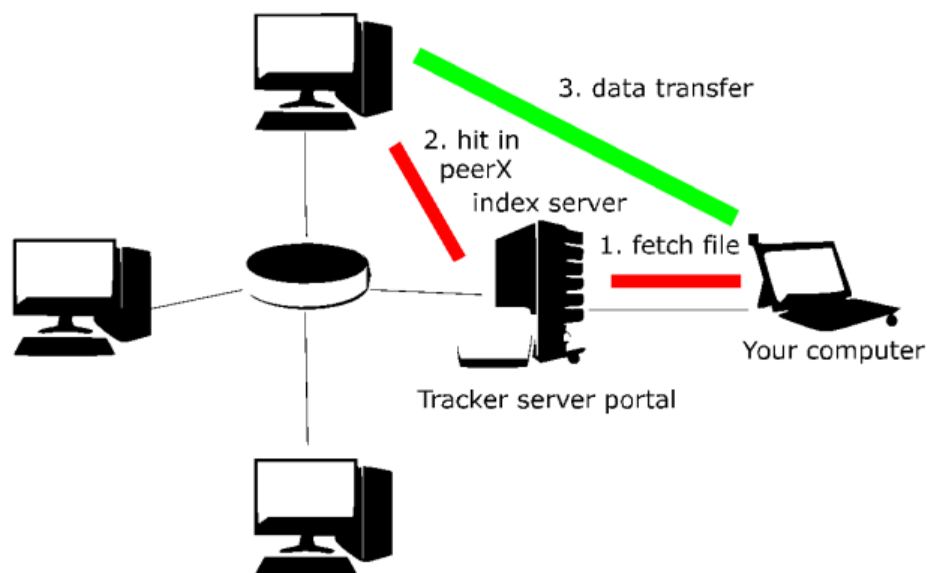
Build a **Simple Torrent-like Application (STA)** with the protocols defined by each group, using the TCP/IP protocol stack and must support **multi-direction data transferring (MDDT)**.

## 1.2 Application description

### Application overview

- The application includes the two types of hosts: tracker and node.
- A centralized tracker keeps track of multiple nodes and stores what pieces of files.
- Through tracker protocol, a node 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 node requires a file that does not belong to its repository, a request is sent to the tracker.
- **MDDT**: The client can download multiple files from multiple source nodes at once, simultaneously.

This requires the node code to be a multithreaded implementation.



**Figure 1.1** Illustration of file-sharing system.

## The simple Torrent-like essential components

Before starting with the network transmission, you need to know something about all these small crucial components:

- **Magnet text:** is simple text that contains all of the necessary information to metainfo file on the tracker. For minimum requirement is a hash code point to the metainfo file/information stored on your centralized tracker portal (aka your-tracker-portal.local).
- **Metainfo File:**(also known as .torrent file) holds all the details about your torrent, including where the tracker address (IP address) is, what is the piece length, and piece-count. It can be considered as the info where magnet text points to and is combined with the current list of trackers and nodes.
- **Pieces:** are the unit of file' parts. Pieces are equally-sized parts of the download. The usual size of the piece is around 512KB. Piece info is listed in Metainfo File
- **Files:** are also specified in the Metainfo File. File includes multiple pieces. There can be more than one file in a torrent. This implies that you may need to map the piece address space to the file address space if you have N parts and M files.
  - ∅ Simple math mistakes are very likely possible here, but you **MUST** carry out the entire mapping between File and the associated pieces by hand and by yourself definition for later referring.
  - ∅ For starting point, you can implement one file per torrent as a minimum score but stopping at this stage could limit your scoring evaluation.

In the Simple Torrent-like network we implemented, the overall architecture is described as follows:

## 2 Phase 1

### 2.1 System description

#### 2.1.1 Core functionalities

- **Tracker:** stores one Metainfo File which holds all the details about all files shared in the system, typically are correspond pieces information. Also, Tracker has one socket server called sock in order to listen for connections from other nodes in the Torrent. For each incoming connection from each client, tracker will create one child socket server and one thread, in which it runs a while loop to continuously handle requests from the connected node. This thread will not terminate until node send message to announce the connection closure.
- **Node:** Each node will have two main sockets, the first one for communicating with the tracker and the second one for handling the upload and downloading process. Each socket will be initialized based on the pre-configured tracker address and the current IP address of each node. The node will create a thread: one for listening requests to other peers to handle two tasks: respond to the information of its related pieces and upload the pieces for the nodes that request it. In the listening request thread, for each request going to a node from other nodes, it will create a corresponding sub-thread for handling it, therefore multiple peers can connect to a node to request and the node can handle those requests simultaneously. For the main process, there will be a while loop acting as a command-line functionality to wait for the user typing their commands: fetch for request files, exit for exit the program. The fetch command will request the tracker to get the information of current peers that may own its requesting files, then the node will start communicating with those peers to get the necessary pieces of those nodes that are related to its requesting files. After a filter algorithm, an optimized list of request pieces array corresponding to each peer will be generated and node will start a thread for each peer to download necessary pieces from them. The filter algorithm will generate the pieces with the criteria: the node will not send a request for a piece *the peer does not have*, the node will not send a request for a piece that it *already has*, and it is guaranteed that *no duplicates will exist between each request queues*.

#### 2.1.2 Additional functionalities

- **Graceful disconnect:** When a node closes the connection, it send a message to the tracker to notify the connection closure, tracker can quickly remove that node's related information from the Metainfo File and freeing any resources belonging to that client.
- **Resume downloading:** Given that the mechanism for downloading pieces in a request queue based system. During the file downloading process, for some unpredictable reasons that might prevent the node from downloading the pieces (network issues, signal strength fluctuation, unstable network, etc.), the downloading process can be temporarily pause within a pre-configured amount of time until the downloading connection is closed. If that connection is closed, the next connection used to download pieces from the same file will only download the not-yet-downloaded portion in the request queue. For example, if Node

A request to download pieces from Node B following the request queue whose format is ["piece1.txt", "piece2.txt", "piece3.txt", "piece4.txt"]. If Node A only requests to download completely two pieces including "piece1.txt", "piece2.txt" before the network interruption, the recovery next time Node A continue to request for the last two pieces, given that the Node B has not closed the connection.

## 2.2 Diagram

### Class Diagram of the system

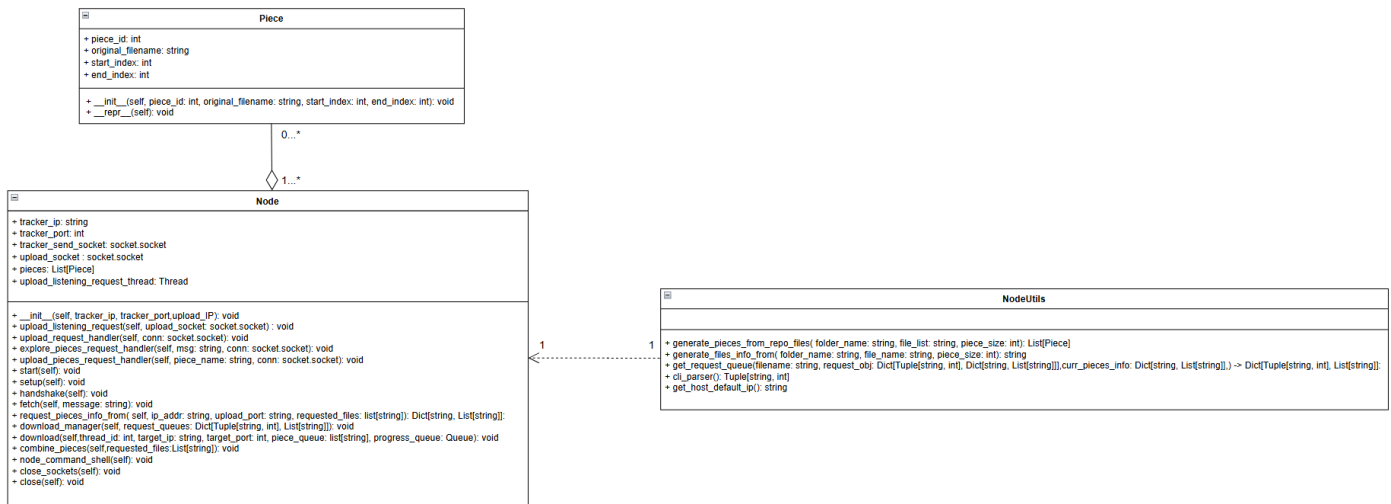


Figure 2.1 Node Class Diagram

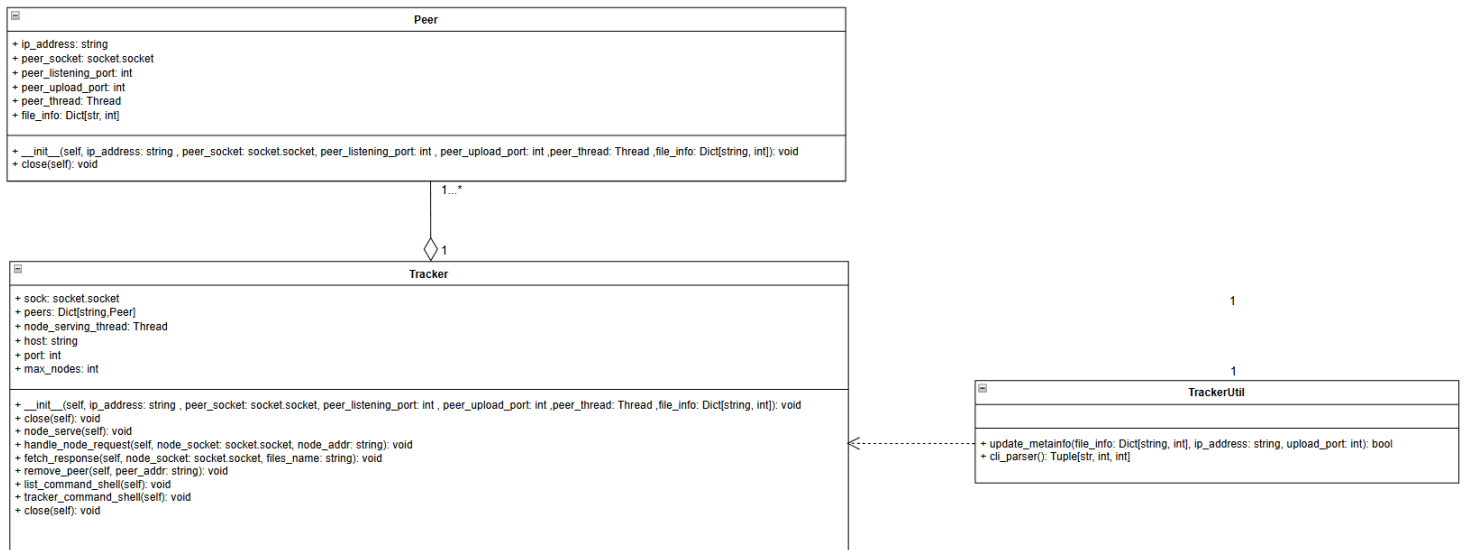


Figure 2.2 Tracker Class Diagram

## 2.3 Layered architecture

For the Node Layered architecture diagram, the component in presentation layered represent a kind of command (**fetch** and **exit**) when the user typing in the command line.

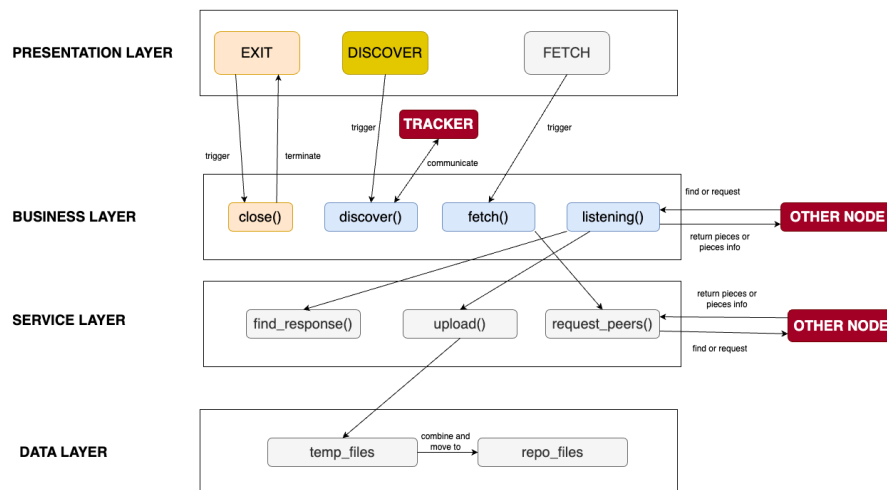


Figure 1: Node layered architecture

For the Tracker Layered architecture diagram, we make a "Logging" component to represent the logging action into the terminal when an event triggered into the tracker (Connection requests,



close announcement from the node,...). `list` and `exit` components represent command as in the Node layered architecture diagram above.

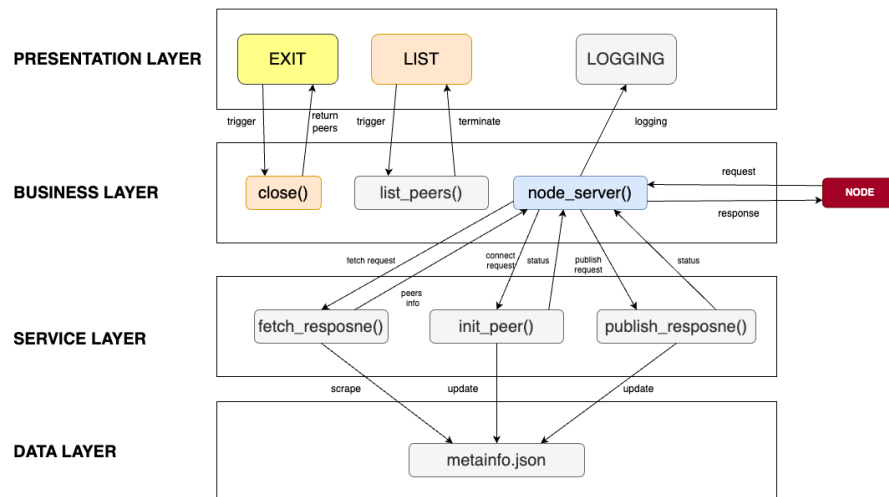


Figure 2: Tracker layered architecture

## 2.4 Tracker Protocol

The application protocol for transferring file will contain two main phases

### 1. Handshake

Initially, the Node Application will create two sockets:

- **tracker\_send\_socket**: Socket to communicate with tracker
- **upload\_socket**: Socket to listen to upload requests from other nodes and responds to them.

On the other hand, the Tracker Application will initialize a socket for handling requests from Node with the `ip_address` and port based on it current ip address and the CLI param:

```

1 self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
2 self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
3 self.sock.bind((host, port))
  
```

In the handshake phase, we define the following protocol for connecting establishment between node and tracker as follow:

- Node send connection request to tracker and sends **First Connection** to indicating first-time-connection

---

```
1 self.tracker_send_socket.connect((self.tracker_ip, self.tracker_port))
2 self.tracker_send_socket.send("First Connection".encode())
```

---

- On the tracker, its socket will receive the request connection and determine the first-time-connection based on the message **First Connection** receiving from socket

---

```
1 while True:
2     try:
3         node_socket, node_addr = self.sock.accept()
4     except Exception as e:
5         continue
6
7     data = node_socket.recv(BUFFER_SIZE).decode()
8     if data == "First Connection":
9         # Handle peer settings and metainfo on tracker-side...
```

---

- Next, the node will generate its info (including its ip\_address, sending port, upload port and its file info) then sending to tracker:

---

```
1 node_info = (
2     self.upload_ip
3     + " "
4     + str(self.tracker_send_socket.getsockname()[1])
5     + " "
6     + str(self.upload_socket.getsockname()[1])
7     + " "
8     + file_info
9 )
10 self.tracker_send_socket.sendall(node_info.encode())
```

---

- On the tracker, it will receive a **First Connection** message, indicating that the **conn** is a newly established socket for communication with the node. Following this, it will process the file information sent by the node, initialize the appropriate **Peer** objects to manage Node connections more efficiently, and update the metainfo file accordingly.

---

```
1 if data == "First Connection":
2     try:
3         peer_info: list[str] = (
4             node_socket.recv(BUFFER_SIZE).decode().split(" ", 3)
5         )
6
7         TrackerUtil.update_metainfo(
8             json.loads(peer_info[3]), peer_info[0], int(peer_info[2])
9         )
10
11         peer_thread: Thread = Thread(
```

```
12         target=self.handle_node_request,
13         args=[node_socket, node_addr],
14         daemon=True,
15     )
16
17     self.peers[node_addr] = Peer(
18         ip_address=peer_info[0], # IP Address of the peer
19         peer_socket=node_socket, # Node socket for communication with that peer
20         peer_thread=peer_thread, # Thread for handling that peer
21         peer_listening_port=int(
22             peer_info[1]
23         ), # Listening port of the peer
24         peer_upload_port=int(peer_info[2]), # Upload port of the peer
25         file_info=json.loads(
26             peer_info[3]
27         ), # File information for the peer
28     )
29
30     print(
31         f"[Connection]: {peer_info[0]}:{peer_info[1]} joined the network"
32     )
33     node_socket.send("Connected".encode())
34     peer_thread.start()
35     except Exception as e:
36         node_socket.send(
37             "Some error occurred while updating metadata on tracker".encode()
38         )
39     node_socket.close()
```

---

If the node is closing, it will send a `close` message to the tracker, then tracker will remove the corresponding peer object referencing it and update the metainfo file again.

The sequence diagram for illustrating the *handshake* phase can be presented as follow:

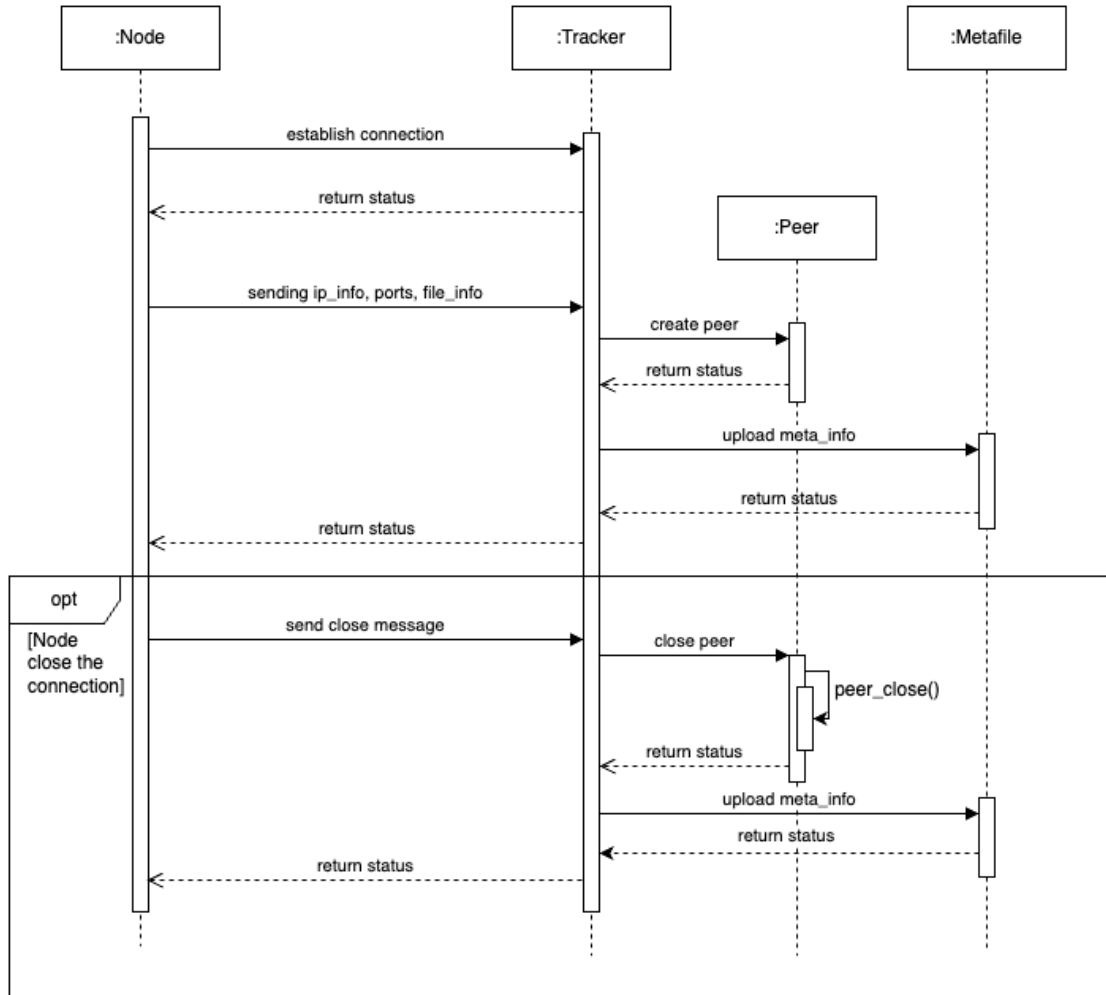


Figure 3: Handshake sequence diagram

## 2. Fetch

- For fetching phases, node will type the `fetch` command line with its desired files to download. To ensure the requested file must not contain the file that node already have, a filter loop shall be executed:

```
1 available_files = os.listdir(REPO_FOLDER)
2 for file in message.split()[1:]:
3     if file in available_files:
4         print(f"[Warning]: You already have file {file}")
5
6 requested_files = [
7     file for file in message.split()[1:] if file not in available_files
8 ]
9
10 if len(requested_files) == 0:
11     return
12
13 # Send the fetch message to the tracker to get the peers information related to the requested files
14 print(
15     "[Warning]: Fetching to tracker to get peers information may contain pieces of requested files..."
16 )
17
18 self.tracker_send_socket.sendall(message.encode()) # fetch 1MB.txt 2MB.txt
```

- Although the requested files array not be passed into the message sent to tracker, it will be used for optimize the requested queue in the download section. After sending message, tracker will parse this message and finding the nodes may contain these files and return the message to the node with the json form:

```
{
    "tracker_ip": "172.20.10.2:8000",
    "172.20.10.2:60937": {
        "peer_id": "172.20.10.2:60937",
        "ip_addr": "172.20.10.2",
        "upload_port": "60936"
    },
    ...
}
```

- The JSON response will include the tracker's IP address for handling node requests and information about nodes that may have the requested files. For simplicity, we use a combination of the peer's IP address and port as the key for each peer object in the response. Each peer object contains the IP address of the node and its upload port, which contain the necessary details for the next step.
- After obtaining the dictionary containing peers that may have the requested file, the node will send the message `find [requested_files]` on each of the upload sockets of those peers. Each peers will return to the node their `pieces_name` information. The Node will used the those pieces name array and combine with its pieces information to generate the object requested queue with the form:

- Overall, the sequence diagram for fetch protocol can be presented as below. For more detail about Peer - Downloading (including the algorithm for requesting the necessary pieces from other nodes and the download process) and Peer - Uploading (the implementation that can help the node serving multi nodes request at the same time), we leave it to section 2.2.5 and 2.2.6.

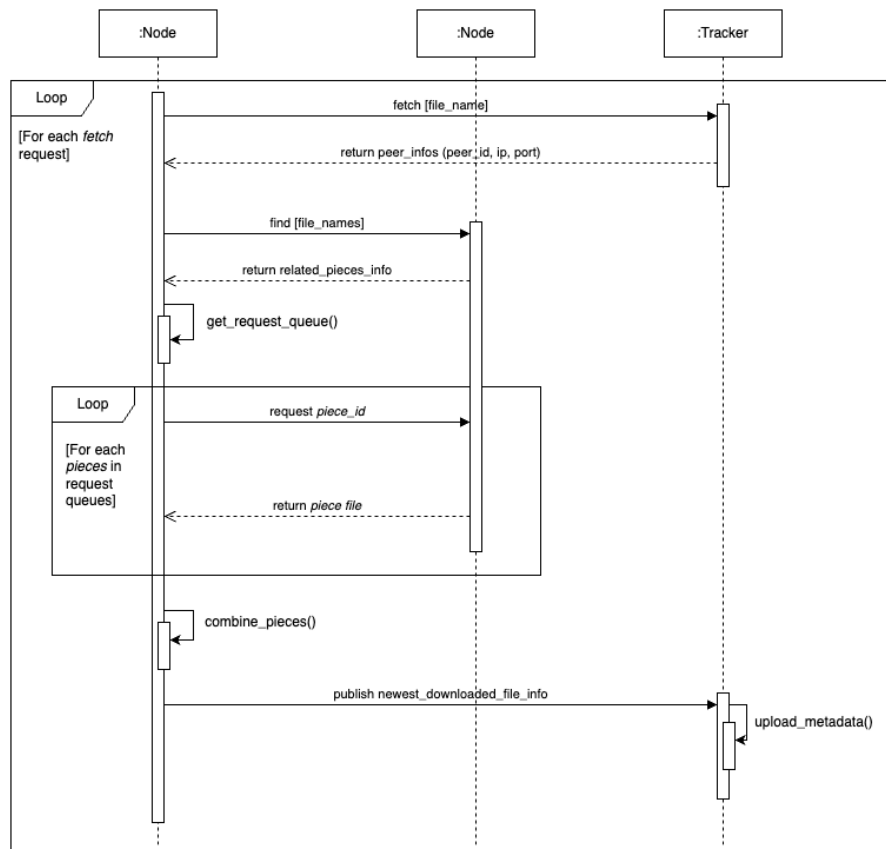


Figure 4: Fetch sequence diagram

## 2.5 Peer-Downloading

To optimize bandwidth usage and eliminate duplicate requests for existing file pieces, we establish specific request queues for each node involved in downloading a file. Each queue holds details about the pieces that one node needs to request to facilitate the download process. For scenarios involving multiple file downloads, request queues are created for each file. The final download plan for a node is determined by aggregating all the request queues associated with that node across the files being downloaded.

The concept of the request queue algorithm is to allocate one piece to each node based on the list of pieces available to that node. Specifically, the first piece in a node's list is assigned to it, and then removing that pieces in other node's lists if include. This process is repeated sequentially for other nodes involved in downloading the file. The end of last node process constitutes the first iteration. The algorithm continues iterating in this manner until all lists are empty, resulting in a final request queue for each node.

The initial implementation is described below:

---

```
1 //Get the request queue
2 result = {key: [] for key in data}
3 keys = list(data.keys())
4 total_value = 0
5 for value in data.values():
6     total_value += len(value)
7 while total_value > 0:
8     listkey = keys.copy()
9
10    # Loop through remaining keys(node's port) to update correspond request queue
11    while len(listkey) > 0:
12        min_key = get_min_key(data, listkey)
13
14        # Remove key whose value is an empty list
15        if len(data[min_key]) == 0:
16            keys.remove(min_key)
17            listkey.remove(min_key)
18            continue
19
20        # Append request queue of corresponding node
21        piece = data[min_key][0]
22        piece_name = f"{filename}_{piece}.{file_extension}"
23        result[min_key].append(piece_name)
24
25        # remove ${piece} in each of keys'value (if any) and decrease total_value
26        for eachkey in keys:
27            if piece in data[eachkey]:
28                data[eachkey].remove(piece)
29                total_value -= 1
30        listkey.remove(min_key)
31 return result
```

---

However, a challenge arises when there is a significant disparity in the lengths of the nodes' piece lists, potentially leading to imbalanced request queues. This imbalance can cause some nodes to take significantly longer to complete the download process. To address this, the algorithm

prioritizes assigning pieces to nodes with the shortest lists in each iteration, optimizing the overall download time.

The process to get the min-length node in each iteration is described as:

---

```
1 def get_min_key(d, keys):
2     # Initialize min_key as None and min_length as infinity to find the minimum
3     min_key = None
4     min_length = float("inf")
5
6     for key in keys:
7         list_length = len(d[key])
8         if list_length < min_length:
9             min_length = list_length
10            min_key = key
11    return min_key
```

---

By integrating the two algorithms described above, we can develop the optimized queue algorithm as follow:

---

```
1     def get_min_key(d, keys):
2         # Initialize min_key as None and min_length as infinity to find the minimum
3         min_key = None
4         min_length = float("inf")
5
6         for key in keys:
7             list_length = len(d[key])
8             if list_length < min_length:
9                 min_length = list_length
10                min_key = key
11        return min_key
12
13    # Get the request queue
14    result = {key: [] for key in data}
15    keys = list(data.keys())
16    total_value = 0
17    for value in data.values():
18        total_value += len(value)
19    while total_value > 0:
20        listkey = keys.copy()
21
22        # Loop through remaining keys(node's port) to update correspond request queue
23        while len(listkey) > 0:
24            min_key = get_min_key(data, listkey)
25
26            # Remove key whose value is an empty list
27            if len(data[min_key]) == 0:
28                keys.remove(min_key)
29                listkey.remove(min_key)
30                continue
31
```

---



```
32         # Append request queue of corresponding node
33         piece = data[min_key][0]
34         piece_name = f"{filename}_{piece}.{file_extension}"
35         result[min_key].append(piece_name)
36
37         # remove ${piece} in each of keys' value (if any) and decrease total_value
38         for eachkey in keys:
39             if piece in data[eachkey]:
40                 data[eachkey].remove(piece)
41                 total_value -= 1
42         listkey.remove(min_key)
43     return result
```

---

- Once the optimized request queues object is obtained, each request queue within it will be assigned to a separate thread to manage the downloading process, ensuring that the downloading for each node is handled concurrently.

```
1     download_threads = []
2     for peer, queue in request_queues.items():
3         thread = Thread(target=self.download, args=(peer[0], peer[1], queue))
4         download_threads.append(thread)
5         thread.start()
6
7     for thread in download_threads:
8         thread.join()
9
10    print("Download completed")
```

---

- In the download method, a socket will be created for each piece in the request queue to handle the downloading process. While this approach may be inefficient and potentially memory-intensive, a more optimized implementation for the downloading method could be explored in the future.

```
1     for piece_name in piece_queue:
2         with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as download_socket:
3             download_socket.connect((target_ip, target_port))
4             download_socket.sendall(f"request {piece_name}".encode())
5             piece_data = b""
6             while True:
7                 chunk = download_socket.recv(PIECE_SIZE)
8                 if not chunk:
9                     break
10                piece_data += chunk
11
12            if piece_data:
13                piece_path = os.path.join(TEMP_FOLDER, f"{piece_name}")
14                with open(piece_path, "wb") as piece_file:
15                    piece_file.write(piece_data)
```

```
16         else:
17             print(
18                 f"[Error]: Failed to download {piece_name}, no data received"
19             )
20             continue
```

---

- All the downloaded pieces will be temporarily stored in the `temp` folder. A subsequent process will then merge these pieces in the correct order to reconstruct the complete file, which will be saved in the `repo` folder:

---

```
1  for file_name in requested_files:
2      combined_file_path = os.path.join(REPO_FOLDER, file_name)
3      with open(combined_file_path, "wb") as combined_file:
4          piece_prefix = f"{file_name.split('.')[0]}_" # e.g "1MB_"
5          pieces = sorted(
6              [f for f in os.listdir(TEMP_FOLDER) if f.startswith(piece_prefix)],
7              key=lambda x: int(x.split("_")[1].split(".")[0]),
8          )
9          for piece in pieces:
10             piece_path = os.path.join(TEMP_FOLDER, piece)
11             with open(piece_path, "rb") as piece_file:
12                 with mmap.mmap(
13                     piece_file.fileno(), length=0, access=mmap.ACCESS_READ
14                 ) as mmapmed_file:
15                     combined_file.write(mmapmed_file)
16             print(f"Combined file {file_name} created successfully in {REPO_FOLDER}")
```

---

- After combined files successfully, the `self.pieces` of Node will be updated based on the downloaded file and an message will be send to Tracker to update its metainfo files:

---

```
1  new_file_info = NodeUtils.generate_files_info_from(REPO_FOLDER,requested_files)
2  msg = f"publish {new_file_info}"
3  self.tracker_send_socket.send(msg.encode())
```

---

## 2.6 Peer-Uploading

- As be described in above, each Node will have an `upload_socket` responsible for serving the request from other nodes. Those requests can be categorized in two types:
  1. Finding the information of its pieces related to the files in the message.
  2. Request to upload the pieces to other nodes.
- In our implementation, each request from other nodes will be executed in separate thread, which **allow several peers to connect to it so that it can serve multiple request streams simultaneously**:

---

```
1 while True:
2     try:
3         conn, addr = upload_socket.accept()
4     except Exception as e:
5         break
6     upload_handler_thread = Thread(
7         target=self.upload_request_handler, args=(conn,), daemon=False
8     )
9     upload_handler_thread.start()
```

---

- For each request connection, `upload_request_handler` will parse the message and categories it into one of the above two types:

---

```
1 msg = conn.recv(1024).decode()
2 if msg.startswith("find"):
3     self.explore_pieces_request_handler(msg, conn)
4 elif msg.startswith("request"):
5     self.upload_pieces_request_handler(msg.split()[1], conn)
```

---

- The `explore_pieces_request_handler` will scrape its `Pieces` (which mapping to the real pieces can be splitted from `repo` folder) and generate the corresponding response containing the name of all the pieces related to the request:

---

```
1 response = {}
2 requested_files = msg.split()[1:]
3 for file_name in requested_files:
4     for piece in self.pieces:
5         if piece.original_filename == file_name:
6             response.setdefault(file_name, []).append(f"{piece.piece_id}")
7 conn.sendall(json.dumps(response).encode())
```

---

- Otherwise, the `upload_pieces_request_handler` will transfer the corresponding piece to the requesting node:

---

```

1 piece_path = os.path.join(PIECES_FOLDER, piece_name)
2 with open(piece_path, "rb") as piece_file:
3     with mmap.mmap(
4         piece_file.fileno(), length=0, access=mmap.ACCESS_READ
5     ) as mmapped_file:
6         chunk = mmapped_file.read(PIECE_SIZE)
7         conn.sendall(chunk)

```

---

Given the limited timeline, the seeding process cannot be implemented at this stage. This means the application will not automatically start sharing the file with other peers interested in downloading it after completing the download. A more advanced implementation may be considered in the future.

In the end, the overall illustration for the network can be presented as follow:

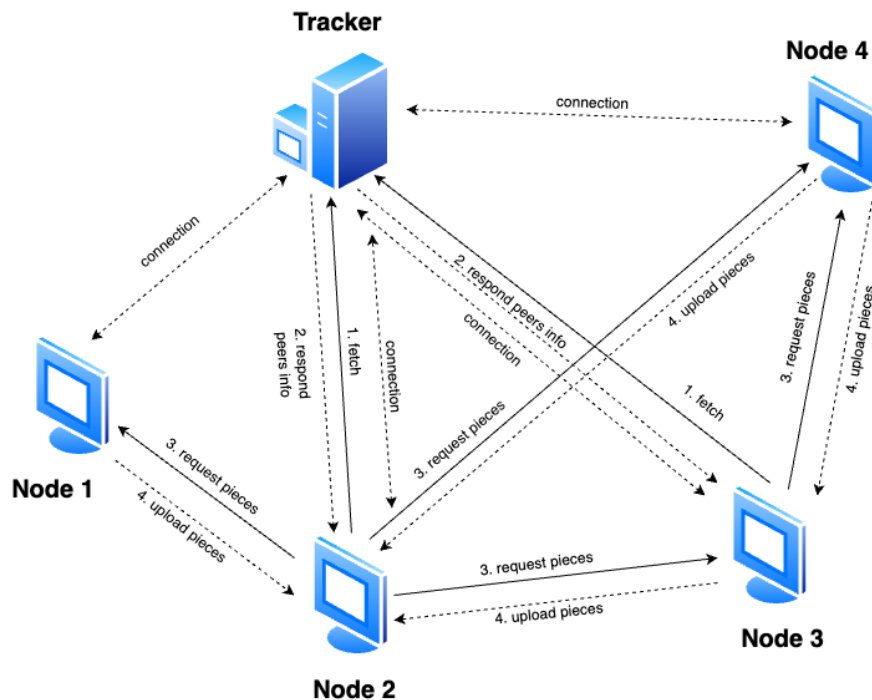
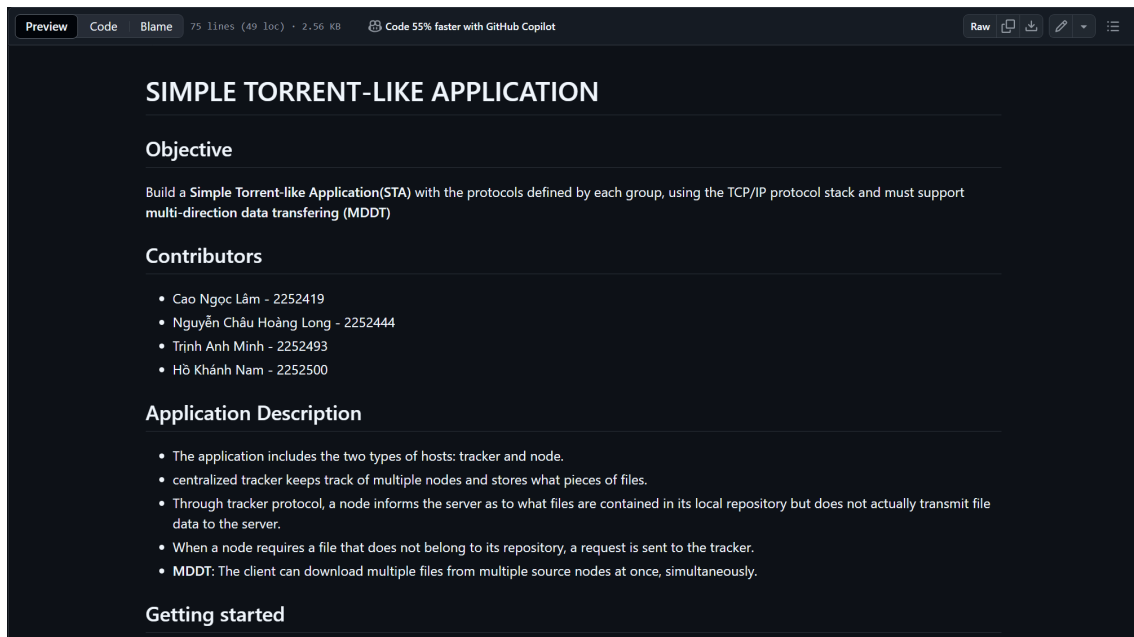


Figure 5: Network

## 3 Phase 2

We have created a repository for our project on Github with the following link: <https://github.com/lamcao1206/simple-torrent-application>

### 3.1 Manual document



**Figure 2.2** File README on Github

For more details, access this link: <https://github.com/lamcao1206/file-sharing-system/blob/main/README.md>

#### Prerequisites

Make sure you already have installed Python on your system.

Otherwise following this guide to fulfill it: <https://www.geeksforgeeks.org/how-to-install-python-on-windows/>

#### Running the application

- Clone the repository and navigate the project directory

```
1 git clone https://github.com/lamcao1206/file-sharing-system 241-computer-  
2 network-assignment1
```

- Connect local devices in a public wireless LAN.

- Make sure to configure the IP and port that the tracker bind to and ensure that the IP and port that other nodes connect to matches that tracker IP and port, also configures the IP that the upload socket of each node bind to in the Torrent-like network so that it can be reached from other clients for fetching files
- One device run as tracker in folder tracker

```
1 python tracker.py
```

- Other devices run as nodes in folder node

```
1 python node.py
```

**NOTE:** When tracker listening connection from nodes, if failed, temporarily turning off your firewall and antivirus software, then try again.

## 3.2 Basic Testing

For the convenience of making report, we conducted the following test on a single laptop instead of multiple devices due to the fact that we have little time for meeting, executing tests between multiple laptops and making full document about those tests. But basically the logic are the same, and we will demonstrate this situation in Demo day (31th November 2024).

### Setting up the files

We will create three node folder, representing three node applications with their files:

node	Yesterday at 22:55	--	Folder
node.py	Today at 13:26	24 KB	Python Script
pieces	Today at 13:54	--	Folder
repo	Today at 14:16	--	Folder
1.pdf	Yesterday at 22:10	4,6 MB	Adobe...cument
2.pdf	Yesterday at 22:13	5 MB	Adobe...cument
temp	Today at 13:42	--	Folder
node1	Today at 14:18	--	Folder
node.py	Today at 14:18	24 KB	Python Script
pieces	Today at 13:27	--	Folder
repo	Today at 14:17	--	Folder
3.pdf	Today at 14:17	4,3 MB	Adobe...cument
4.pdf	Today at 14:17	3,3 MB	Adobe...cument
temp	Yesterday at 22:28	--	Folder
node2	Today at 14:18	--	Folder
node.py	Today at 14:17	24 KB	Python Script
pieces	Yesterday at 22:32	--	Folder
repo	Today at 14:17	--	Folder
5.pdf	Today at 14:17	4,3 MB	Adobe...cument
6.pdf	Today at 14:17	3,8 MB	Adobe...cument

Figure 6: Setup files

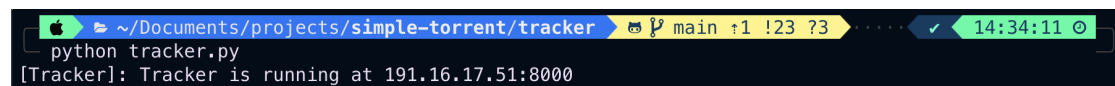
From the image, we get the following files information of each Node:

Node	File Name	File Size
Node A	1.pdf	4.6 MB
	2.pdf	5 MB
Node B	3.pdf	4.3 MB
	4.pdf	3.3 MB
Node C	5.pdf	4.3 MB
	6.pdf	3.8 MB

Table 1: File distribution across nodes with file sizes.

### Handshake phase

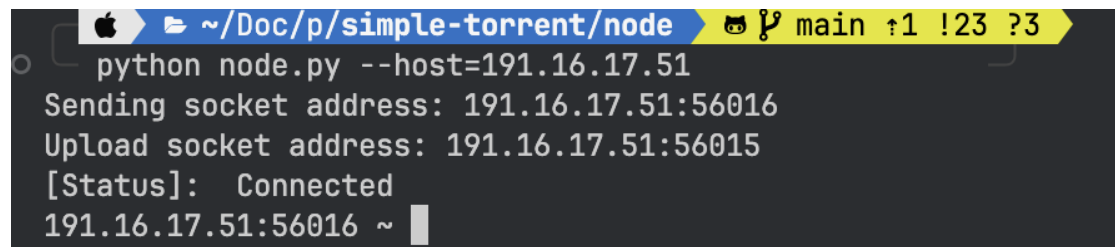
In Tracker folder, we run tracker.py and receive its IP information through the logging on Command Line:



```
~/Documents/projects/simple-torrent/tracker main ↑1 !23 ?3 14:34:11
python tracker.py
[Tracker]: Tracker is running at 191.16.17.51:8000
```

Figure 7: Tracker Initialization

Initialize each node with the IP and port parameter of tracker:



```
~/Doc/p/simple-torrent/node main ↑1 !23 ?3
python node.py --host=191.16.17.51
Sending socket address: 191.16.17.51:56016
Upload socket address: 191.16.17.51:56015
[Status]: Connected
191.16.17.51:56016 ~
```

Figure 8: Node Init

The [Status]: Connected indicates that the tracker has already update its file information on the metainfo.json in Tracker folder:

```
{
  "tracker_addr": "191.16.17.51:8000",
  "1.pdf": {
    "file_size": 4581703,
    "piece_size": 524288,
    "piece_count": 9,
    "nodes": [
      "191.16.17.51:56015"
    ]
  },
  "2.pdf": {
    "file_size": 4953576,
    "piece_size": 524288,
    "piece_count": 10,
    "nodes": [
      "191.16.17.51:56015"
    ]
  }
}
```

Figure 9: Metainfo

After init all three Node, we have the following information:

Node	Sending Address	Upload Addr
node	191.16.17.51:56016	191.16.17.51:56015
node1	191.16.17.51:56036	191.16.17.51:56035
node2	191.16.17.51:56038	191.16.17.51:56037

Table 2: Nodes information

We can also check those connections in tracker side by `list` command:



```
python tracker.py
[Tracker]: Tracker is running at 191.16.17.51:8000
[Connection]: 191.16.17.51:56016 joined the network
[Connection]: 191.16.17.51:56031 joined the network
[Connection]: 191.16.17.51:56033 joined the network
[Close]: 191.16.17.51:56031 offline
[Close]: 191.16.17.51:56033 offline
[Connection]: 191.16.17.51:56036 joined the network
[Connection]: 191.16.17.51:56038 joined the network
list
- [0] ('191.16.17.51', 56016)
- [1] ('191.16.17.51', 56036)
- [2] ('191.16.17.51', 56038)
```

Figure 10: Tracker List Connection

### Fetch phase

In the node with ip address 191.16.17.51:56016, we will begin to download all the files 1.pdf to 6.pdf through the `fetch` command:

```
191.16.17.51:56016 ~ fetch 1.pdf 2.pdf 3.pdf 4.pdf 5.pdf 6.pdf
[Warning]: You already have file 1.pdf
[Warning]: You already have file 2.pdf
Fetching to tracker to get peers information may contain pieces of requested files...
[Result]:
{
  "191.16.17.51:56016": {
    "peer_ip": "191.16.17.51:56016",
    "ip_addr": "191.16.17.51",
    "upload_port": 56015
  },
  "191.16.17.51:56036": {
    "peer_ip": "191.16.17.51:56036",
    "ip_addr": "191.16.17.51",
    "upload_port": 56035
  },
  "191.16.17.51:56038": {
    "peer_ip": "191.16.17.51:56038",
    "ip_addr": "191.16.17.51",
    "upload_port": 56037
  },
  "tracker_ip": "191.16.17.51:8000"
}
Requesting pieces information from peers... Ok
{
  "('191.16.17.51', 56035)": ["3_0.pdf", "3_1.pdf", "3_2.pdf", "3_3.pdf", "3_4.pdf", "3_5.pdf",
    "3_6.pdf", "3_7.pdf", "3_8.pdf", "4_0.pdf", "4_1.pdf", "4_2.pdf", "4_3.pdf", "4_4.pdf", "4_5.pdf", "4_6.pdf"],
  "('191.16.17.51', 56037)": ["5_0.pdf", "5_1.pdf", "5_2.pdf", "5_3.pdf", "5_4.pdf", "5_5.pdf",
    "5_6.pdf", "5_7.pdf", "5_8.pdf", "6_0.pdf", "6_1.pdf", "6_2.pdf", "6_3.pdf", "6_4.pdf", "6_5.pdf", "6_6.pdf", "6_7.pdf"]
}
Start downloading...
Download completed
Combined pieces ok
```

Figure 11: `fetch`

First, node will send request to tracker to find other nodes that contains the file, then the

tracker will response with an object json, containing information of 3 peers, including itself. Then it will request to those peers, getting the pieces information related to the requested files and return to that node. It will generate the requested queue pieces for requesting downloading to each peers. As can be observed, the pieces that the node requested satisfy the criteria: don't send a request for a piece the peer does not have, don't send a request for a piece that the node already have and the pieces mustn't be duplicated between request pieces queues for each node.

After downloading the pieces and stores in **temp** folder, the node will combine those files and storing in **repo** folder. It also send the request to tracker to update the metafile:

```
{
  "tracker_addr": "191.16.17.51:8000",
  "1.pdf": {
    "file_size": 4581703,
    "piece_size": 524288,
    "piece_count": 9,
    "nodes": ["191.16.17.51:56015"]
  },
  "2.pdf": {
    "file_size": 4953576,
    "piece_size": 524288,
    "piece_count": 10,
    "nodes": ["191.16.17.51:56015"]
  },
  "4.pdf": {
    "file_size": 3296397,
    "piece_size": 524288,
    "piece_count": 7,
    "nodes": ["191.16.17.51:56035", "191.16.17.51:56015"]
  },
  "3.pdf": {
    "file_size": 4290877,
    "piece_size": 524288,
    "piece_count": 9,
    "nodes": ["191.16.17.51:56035", "191.16.17.51:56015"]
  },
  "6.pdf": {
    "file_size": 3790153,
    "piece_size": 524288,
    "piece_count": 8,
    "nodes": ["191.16.17.51:56037", "191.16.17.51:56015"]
  },
  "5.pdf": {
    "file_size": 4279599,
    "piece_size": 524288,
    "piece_count": 9,
    "nodes": ["191.16.17.51:56037", "191.16.17.51:56015"]
  }
}
```

Figure 12: Updated metafile on tracker

As can be seen, the ip address of node is updated on the entry containing the file 3.pdf, 4.pdf, 5.pdf and 6.pdf.

node	Yesterday at 22:55	--
node.py	Today at 13:26	24 KB
pieces	Today at 15:15	--
repo	Today at 15:15	--
1.pdf	Yesterday at 22:10	4,6 MB
2.pdf	Yesterday at 22:13	5 MB
3.pdf	Today at 15:15	4,3 MB
4.pdf	Today at 15:15	3,3 MB
5.pdf	Today at 15:15	4,3 MB
6.pdf	Today at 15:15	3,8 MB
temp	Today at 15:15	--
node1	Today at 14:18	--
node.py	Today at 14:18	24 KB
pieces	Today at 14:47	--
repo	Today at 14:17	--
3.pdf	Today at 14:17	4,3 MB
4.pdf	Today at 14:17	3,3 MB
temp	Yesterday at 22:28	--
node2	Today at 14:18	--
node.py	Today at 14:17	24 KB
pieces	Today at 14:47	--
repo	Today at 14:17	--
5.pdf	Today at 14:17	4,3 MB
6.pdf	Today at 14:17	3,8 MB

Figure 13: Node repo after `fetch`

The downloaded file in repo folder of node containing the sizes exactly as the original files from other nodes, which indicating that the downloading process is success.

## 4 Conclusion

The file-sharing application has been successfully developed using the TCP/IP protocol stack, based on the defined application protocols. Its architecture is centered around a centralized server that manages client connections and file information efficiently. Nodes communicate with the tracker to share details about their local file repositories without sending the actual file data.

A key feature of the application is the ability for nodes to request files they do not have. When this happens, the server identifies available sources, allowing the requesting client to directly download the file from the chosen source without tracker involvement. This decentralized approach reduces server load and improves efficiency.

To enable simultaneous file transfers, the node code supports multithreading, allowing multiple clients to download different files from a target client at the same time. This design ensures better use of resources and improves the application's responsiveness.

On the node side, a simple command-shell interpreter provides two main commands:

`fetch`: to request and download files from other clients. `exit`: to send `close` message to tracker to indicate closing connection and exit the program. On the server side, the command-shell interpreter allows file discovery and real-time monitoring of connected client hosts.

This implementation demonstrates effective communication between clients and the server, showcasing the reliability of the defined application protocols. Using the TCP/IP protocol stack ensures secure and dependable data transfer across the network. This project offered valuable learning experiences in designing and building a distributed file-sharing system, addressing challenges such as decentralized file retrieval, multithreading, and command handling. The collaborative effort in developing this application has provided a deeper understanding of networked systems and application protocols.

## References

- [1] Bittorrent Protocol Specification, <https://wiki.theory.org/BitTorrentSpecification>
- [2] How to install Python on Windows?, <https://www.geeksforgeeks.org/how-to-install-python-on-windows/>
- [3] James F. Kurose, Keith W. Ross, *Computer Networking: a Top Down Approach*, 8th Edition