



UNIVERSITY OF DHAKA

Department of Computer Science and Engineering

CSE-3111 : Computer Networking Lab

Lab Report 3: **Implementing File transfer using Socket Programming and HTTP GET/POST requests**

Submitted By:

Joty Saha (Roll-51)

Ahona Rahman (Roll-59)

Submitted To:

Dr. Saifuddin Md. Tareeq

Redwan Ahmed Rizvee

Submitted On: February 7, 2024

Contents

1	Introduction	3
1.1	Objectives	3
2	Theory	4
2.1	Socket Programming	4
2.1.1	Socket Creation	4
2.1.2	Binding	4
2.1.3	Listening	4
2.1.4	Accepting Connections	4
2.1.5	Communication	4
2.1.6	Closing Sockets	4
2.2	HTTP (Hypertext Transfer Protocol)	4
2.2.1	HTTP Methods	4
2.2.2	HTTP Requests	5
2.2.3	HTTP Responses	5
2.2.4	File Transfer with HTTP	5
3	Methodology	6
3.1	Socket Programming	6
3.2	HTTP GET/POST Requests	6
3.3	Task 1: File Transfer via Socket Programming	7
3.4	Task 2: File Transfer via HTTP	10
4	Experimental result	15
4.1	Task 1	15
4.1.1	After Running Socket Server Code:	15
4.1.2	After Running Socket Client Code:	15
4.1.3	File downloaded via Socket programming:	16
4.1.4	Showing Downloaded file in the Folder :	16
4.1.5	Opening downloaded file :	17
4.1.6	Server side after download from any client :	17
4.1.7	Error: File not found	18
4.2	Task 2	18
4.2.1	After Running HTTP Server Code:	18
4.2.2	After Running HTTP Client Code:	18
4.2.3	Select Get:	19
4.2.4	Enter the name of the downloaded file and rename it:	20
4.2.5	File downloaded via HTTP programming:	21
4.2.6	Showing Downloaded file in the Folder :	21
4.2.7	Opening downloaded file :	22
4.2.8	Server side after download from any client :	23
4.2.9	Error: File not found	24
4.2.10	File upload via HTTP programming:	24
4.2.11	Showing uploaded file in the folder:	25
4.2.12	Opening uploaded file :	26
4.2.13	Server side after upload from any client :	27
4.2.14	Error: File not found	28

5	Experience	29
5.1	Socket Programming Exploration	29
5.2	HTTP Server Setup	29
5.3	Utilizing HTTP GET/POST Methods for File Transfer	29
5.4	Challenges and Learnings	29

1 Introduction

The integration of socket programming and HTTP protocols is paramount in modern network communication systems. Through socket programming, developers can establish communication channels between devices over a network, while HTTP (Hypertext Transfer Protocol) governs how data is exchanged between clients and servers on the web. In this experiment, we explore the fusion of these technologies to facilitate file transfer between clients and servers, utilizing both raw socket communication and the higher-level HTTP GET and POST methods.

1.1 Objectives

The primary objective of this lab is to provide participants with hands-on experience in socket programming and HTTP file transfer, fostering a deep understanding of network communication concepts and protocols. Throughout this lab, participants will engage in a series of tasks aimed at achieving the following objectives:

1. Implement Multithreaded Chat:

- Participants will develop a multithreaded chat system to facilitate communication between multiple clients and a single server. This task will involve the creation of a server application capable of accepting and managing connections from numerous clients concurrently.
- Through the implementation of multithreading, participants will ensure efficient handling of client messages, enabling seamless communication in real time.

2. Set Up an HTTP Server Process:

- Participants will establish an HTTP server process equipped to handle HTTP requests from clients. This server will act as a central hub for storing and serving various objects, such as files or data.
- Configuration of the HTTP server will include defining endpoints for handling GET requests, allowing clients to retrieve objects hosted on the server. Additionally, participants will implement functionality to process POST requests, enabling clients to upload objects to the server.

3. Utilize GET and POST Methods for File Transfer:

- Participants will utilize HTTP GET requests to enable clients to download objects hosted on the server. By implementing proper handling of GET requests, participants will ensure seamless retrieval of files by clients.
- The implementation of HTTP POST requests will allow clients to upload objects, such as files or data, to the server. Participants will develop robust mechanisms to receive, process, and store uploaded objects securely on the server.

By engaging in these tasks, participants will gain practical experience in socket programming and HTTP file transfer, honing their skills in network communication and protocol implementation. Through the completion of these objectives, participants will be equipped with the knowledge and expertise to develop robust, scalable, and efficient network applications.

2 Theory

2.1 Socket Programming

Socket programming is a fundamental concept in network communication, allowing processes to communicate with each other over a network. A socket is an endpoint for communication between two machines. In socket programming, a server listens for incoming connections on a specific port, while a client initiates a connection to the server.

2.1.1 Socket Creation

In socket programming, sockets are created using the `socket` system call. This call returns a socket descriptor that represents the endpoint for communication.

2.1.2 Binding

After creating a socket, the server binds it to a specific address and port combination using the `bind` system call. This allows clients to connect to the server at the specified address and port.

2.1.3 Listening

Once bound, the server socket enters a listening state using the `listen` system call. In this state, the server waits for incoming connection requests from clients.

2.1.4 Accepting Connections

When a client sends a connection request to the server, the server socket accepts the connection using the `accept` system call. This creates a new socket for communication with the client while the server socket continues to listen for new connections.

2.1.5 Communication

After establishing a connection, both the client and server can send and receive data through their respective sockets using `read` and `write` operations.

2.1.6 Closing Sockets

When communication is complete, sockets are closed using the `close` system call, releasing the associated resources.

2.2 HTTP (Hypertext Transfer Protocol)

HTTP is an application-layer protocol used for transmitting hypermedia documents, such as HTML pages, over the Internet. It serves as the foundation of data communication on the World Wide Web, facilitating the exchange of information between clients and servers in a standardized manner.

2.2.1 HTTP Methods

HTTP defines a set of methods that clients can use to interact with resources on a server. These methods dictate the type of operation to be performed on the specified resource. The two most commonly used methods are:

- **GET:** This method is used to request data from a specified resource on the server. It retrieves the content of the resource identified by the given URL. GET requests are typically used for retrieving web pages, images, or other static content.
- **POST:** The POST method is used to submit data to be processed to a specified resource on the server. It sends data in the request message body, which is typically used for submitting form data, uploading files, or performing other actions that require data to be sent to the server for processing.

2.2.2 HTTP Requests

An HTTP request is composed of several components:

- **Request Line:** The request line contains the HTTP method, the URL of the requested resource, and the HTTP version.
- **Headers:** HTTP headers provide additional information about the request, such as the user agent, content type, and cookies. Headers are key-value pairs separated by a colon and a space.
- **Message Body:** The message body is an optional component of an HTTP request and is used to send additional data to the server, such as form data or file uploads. Not all requests contain a message body.

2.2.3 HTTP Responses

Similarly, an HTTP response consists of the following components:

- **Status Line:** The status line includes the HTTP version, a status code indicating the outcome of the request, and a status message providing a brief description of the status code.
- **Headers:** Like in requests, HTTP headers in responses provide additional information about the response, such as the content type, server information, and caching directives.
- **Message Body:** The message body contains the actual content of the response, such as HTML markup, JSON data, or binary file data. The presence of a message body depends on the nature of the response and the request that triggered it.

2.2.4 File Transfer with HTTP

HTTP can be leveraged for file transfer by utilizing the GET and POST methods:

- **GET Requests:** Clients can use GET requests to download files from a server. By specifying the URL of the desired file, clients can retrieve the file's contents as part of the response body.
- **POST Requests:** Clients can upload files to a server using POST requests. The file data is included in the request's message body, allowing clients to submit files to the server for processing or storage.

By leveraging standard HTTP protocols for file transfer, clients and servers can seamlessly exchange files over the web, providing a convenient and widely adopted method for transferring data.

3 Methodology

3.1 Socket Programming

Socket programming provides a low-level mechanism for data communication between processes running on different networked computers. In this experiment, we implemented file transfer using socket programming in Python.

- **Establishing the Server:**

We utilized Python's `socket` module to instantiate a server-side socket. This socket was configured to listen for incoming connections on a specified port. Upon receiving a connection request, the server socket accepts the connection and establishes communication with the client.

- **Creating the Client:**

A client-side application was developed to connect to the server socket using the server's IP address and port number. The client initiates a connection request to the server and upon successful connection, begins the file transfer process.

- **File Transfer Implementation:**

Once the connection is established, the client reads the file to be transferred from the disk. The file data is then sent to the server using socket send operations. On the server side, the server socket receives the file data through socket receive operations. The received data is written to the disk, thereby completing the file transfer process.

- **Measurement of File Transfer Time:**

To evaluate the performance of file transfer using socket programming, we measured the time taken for the entire transfer process. This includes the time taken from the client's initiation of the connection to the server's reception and storage of the file. We recorded the time elapsed for the transfer and noted any discrepancies or anomalies.

3.2 HTTP GET/POST Requests

HTTP (Hypertext Transfer Protocol) is a widely used protocol for communication between web clients and servers. In this experiment, we implemented file transfer using HTTP GET and POST requests in Python.

- **Setting up the HTTP Server:**

We instantiated a simple HTTP server using Python's `http.server` module. This module provides a basic HTTP server framework that listens for incoming HTTP requests on a specified port. The server was configured to handle both HTTP GET and POST requests from clients.

- **Creating the Client:**

A client-side application was developed to send HTTP GET or POST requests to the server. The client initiated a connection to the server's IP address and port using standard HTTP protocols. Depending on the chosen method (GET or POST), the client constructed an appropriate HTTP request.

- **File Transfer Implementation using GET/POST Requests:**

For file transfer using HTTP GET requests, the client appended the file data to the URL as parameters. On the server side, the HTTP server parsed the request parameters to retrieve the file data and stored it on the disk.

For file transfer using HTTP POST requests, the client constructed an HTTP POST request with the file data included in the request's body. The server received the HTTP POST request, parsed the body of the request to extract the file data, and stored it on the disk.

- **Measurement of File Transfer Time:**

Similar to the socket programming approach, we measured the time taken for the file transfer process using HTTP requests. We recorded the time elapsed from the client's request initiation to the server's processing and storage of the file.

3.3 Task 1: File Transfer via Socket Programming

1. Implement a simple file server that listens for incoming connections on a specified port. The server should be able to handle multiple clients simultaneously.
2. When a client connects to the server, the server should prompt the client for the name of the file they want to download.
3. The server should then locate the file on disk and send it to the client over the socket.
4. Implement a simple file client that can connect to the server and request a file. The client should save the file to disk once it has been received.

Listing 1: Socket Programming Server Code

```

1 import socket
2 import os
3 import time
4 from threading import Thread
5
6 IP = "10.42.0.44"
7 port = 4381
8 ADDR = (IP, port)
9
10
11 def handle_client(conn, addr):
12     print(f"[NEW CONNECTION] {addr} connected...")
13
14     # Get the list of files in the server directory
15     files = os.listdir('.')
16     file_list = '\n'.join(files)
17
18     # Print the list of files available
19     # print("List of files available for download:")
20     # for file_name in files:
21     #     print(file_name)
22
23     # Send the file list to the client
24     conn.send(file_list.encode())
25
26     conn.send("Enter the filename you want to download: ".
27               encode())
28     filename = conn.recv(1024).decode()
29     file_path = os.path.join("/home/hp/Downloads/
30                               Lab2_File_Transfer/Socket_Programming", filename)
31
32     if os.path.exists(file_path):
33         download_time = time.strftime('%Y-%m-%d %H:%M:%S')
34         with open(file_path, "rb") as file:
35             data = file.read()
36
37             conn.sendall(data)
38             print(f"[SENT] File '{filename}' sent to {addr} at {
39                   download_time}.")
40     else:
41         conn.send("File not found.".encode())
42
43     print(f"[DISCONNECTED] {addr} disconnected...")
44     conn.close()
45
46 def start_server():
47     print("[STARTING] Server is starting...")
48     server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
49     server.bind(ADDR)

```

```

47     server.listen()
48     print(f"[LISTENING] Server is listening on {IP} : {port}
        ...")
49
50     while True:
51         conn, addr = server.accept()
52         client_handler = Thread(target=handle_client, args=(
            conn, addr))
53         client_handler.start()
54
55 if __name__ == "__main__":
56     start_server()

```

Listing 2: Socket Programming Client Code

```

1  import socket
2  import os
3
4  IP = "10.42.0.44"
5  port = 4381
6  ADDR = (IP, port)
7
8
9  def main():
10     client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
11     client.connect(ADDR)
12
13     # Receive prompt for filename from the server
14     prompt = client.recv(1024).decode()
15     print("List of files available for download from [SERVER]:
        ")
16     print(f"{prompt}")
17
18     # Receive list of files from server
19     file_list = client.recv(1024).decode()
20
21     print(file_list)
22
23     # Get the filename from the user
24     filename = input()
25     client.send(filename.encode())
26
27     # Receive file data from the server
28     data = b""
29     while True:
30         chunk = client.recv(1024)
31         if not chunk:
32             break
33         data += chunk

```

```

34
35     # Check if the server sent an error message
36     if data == b"File not found.":
37         print(f"[SERVER]: File not found.")
38     else:
39         # Save the received data to a file
40         with open(f"downloaded_{filename}", "wb") as file:
41             file.write(data)
42
43         file_size_bytes = os.path.getsize(f"downloaded_{
44             filename}")
45         file_size_mb = file_size_bytes / (1024 * 1024)
46         print(f"[SERVER]: File '{filename}' downloaded
47             successfully. Size: {file_size_mb:.2f} MB")
48
49     client.close()
50
51 if __name__ == "__main__":
52     main()

```

3.4 Task 2: File Transfer via HTTP

1. Implement a simple HTTP file server that listens for incoming connections on a specified port. The server should be able to handle multiple clients simultaneously.
2. When a client sends a GET request to the server with the path of the file they want to download, the server should locate the file on disk and send it to the client with the appropriate HTTP response headers. The client should save the file to disk once it has been received.
3. When a client sends a POST request to the server with a file, the server saves it.

Listing 3: HTTP Server Code

```

1 import time
2 from http.server import BaseHTTPRequestHandler, HTTPServer
3 import logging
4 import os
5
6 class S(BaseHTTPRequestHandler):
7     def _set_response(self):
8         self.send_response(200)
9         self.send_header('Content-type', 'text/html')
10        self.end_headers()
11
12    def do_GET(self):
13        logging.info("GET request,\nPath: %s\nHeaders:\n%s\n",
14                     str(self.client_address), str(self.path), str(self.
15                     headers))
16        file_path = self.path[1:]
17
18        if file_path == '':
19            self._list_files()
20        else:
21            self._serve_file(file_path)
22
23    def _list_files(self):
24        files = '\n'.join(os.listdir())
25        self._set_response()
26        self.wfile.write(files.encode())
27
28    def _serve_file(self, file_path):
29        if os.path.exists(file_path):
30            try:
31                download_time = time.strftime('%Y-%m-%d %H:%M
32                :%S')
33                with open(file_path, "rb") as f:
34                    file_content = f.read()
35                print(f"[SENT] File '{file_path}' sent at {
36                    download_time}.")
37            except FileNotFoundError:
38                logging.error("File not found: %s", file_path)
39                self.send_error(404, "File not found")
40                return
41
42            self._set_response()
43            self.wfile.write(file_content)
44        else:
45            logging.error("File not found: %s", file_path)
46            self.send_error(404, "File not found")
47
48    def do_POST(self):
49        content_type = self.headers['Content-Type']

```

```

46         content_length = int(self.headers['Content-Length'])
47         post_data = self.rfile.read(content_length)
48
49         file_name = 'upload.in'
50         if 'filename' in content_type:
51             file_name = content_type.split('filename=')[1].
52                 strip('"')
53
54         with open(file_name, 'wb') as local_file:
55             local_file.write(post_data)
56
57         logging.info("POST request,\nPath: %s\nHeaders:\n%s\n\n\
58             nBody:\n%s\n", str(self.client_address),
59                 str(self.path), str(self.headers))
60
61         self._set_response()
62         self.wfile.write("POST request for {}".format(self.
63             path).encode('utf-8'))
64
65 def start(server_class=HTTPServer, handler_class=S, port=8080)
66 :
67     logging.basicConfig(level=logging.INFO)
68     server_address = ('', port)
69     httpd = server_class(server_address, handler_class)
70     # Set a larger value for max_size to handle larger file
71     uploads
72     httpd.max_size = 10 * 1024 * 1024 # 10 MB
73     logging.info('Starting httpd...\n')
74     try:
75         httpd.serve_forever()
76     except KeyboardInterrupt:
77         pass
78     httpd.server_close()
79     logging.info('Stopping httpd...\n')
80
81 if __name__ == '__main__':
82     from sys import argv
83
84     if len(argv) == 2:
85         start(port=int(argv[1]))
86     else:
87         start()

```

Listing 4: HTTP Client Code

```

1 import os
2
3 import requests
4 import mimetypes
5
6
7 def list_files():
8     file_list = requests.get('http://localhost:8080/').text
9     print("Available files for download or posting:")
10    print(file_list)
11
12
13 def get_file():
14     list_files()
15     fil = input('Enter File Name: ')
16
17     response = requests.get(f'http://localhost:8080/{fil}')
18
19     if response.status_code == 200:
20         # Prompt the user for the name to save the file as
21         filename = input("Enter the name to save the file as (
            including extension): ")
22         with open(filename, "wb") as f:
23             f.write(response.content)
24             file_size_bytes = os.path.getsize(filename)
25             file_size_mb = file_size_bytes / (1024 * 1024)
26             print(f"[SERVER]: File '{filename}' downloaded
                successfully. Size: {file_size_mb:.2f} MB")
27     else:
28         print("Error: Could not receive file")
29
30
31 def post_file():
32     list_files()
33     file_path = input('Enter File Path for Posting: ')
34
35     if not os.path.exists(file_path):
36         print("Error: File does not exist.")
37         return
38
39     with open(file_path, 'rb') as file:
40         files = {'file': (os.path.basename(file_path), file)}
41         response = requests.post('http://localhost:8080/',
            files=files)
42
43     if response.status_code == 200:
44         print("File successfully posted")
45     else:
46         print(f"Error: Could not post file. Server response: {

```

```

47         response.status_code}")
48     print(response.text)  # Print out the response content
49                             for debugging
50
51
52 def main():
53     choice = input("Choose operation (1 for GET, 2 for POST):
54                     ")
55
56     if choice == '1':
57         get_file()
58     elif choice == '2':
59         post_file()
60     else:
61         print("Invalid choice. Please choose 1 for GET, 2 for
62             POST.")
63
64 if __name__ == '__main__':
65     main()

```

4 Experimental result

Some snapshots of the client-side queries can be seen in the following figures:

4.1 Task 1

4.1.1 After Running Socket Server Code:

```
/usr/bin/python3.11 /home/asus/Documents/Python/NetworkingLab/Lab2_File_Transfer/Socket_Programming/server.py
[STARTING] Server is starting...
[LISTENING] Server is listening on 127.0.0.1 : 3381...
```

Figure 1: Content of Server

4.1.2 After Running Socket Client Code:

The client is successfully connected to the server. The client can now download any file from the given list.

```
/usr/bin/python3.11 /home/asus/Documents/Python/NetworkingLab/Lab2_File_Transfer/Socket_Programming/client.py
List of files available for download from [SERVER]:
tom.jpg
client.py
downloaded_big.txt
.idea
server.py
big.txt
sherlock.pdf
downloaded_Sherlock.pdfEnter the filename you want to download:
```

Figure 2: Server Connect to Client

4.1.3 File downloaded via Socket programming:

File transfer successfully occurs via socket programming; the file size is also mentioned to the clients.

```
/usr/bin/python3.11 /home/asus/Documents/Python/NetworkingLab/Lab2_File_Transfer/Socket_Programming/client.py
List of files available for download from [SERVER]:
tom.jpg
client.py
downloaded_big.txt
.idea
server.py
big.txt
sherlock.pdf
downloaded_Sherlock.pdf
Enter the filename you want to download:
sherlock.pdf
[SERVER]: File 'sherlock.pdf' downloaded successfully. Size: 0.71 MB

Process finished with exit code 0
|
```

Figure 3: File downloaded successfully

4.1.4 Showing Downloaded file in the Folder :

Here, we can see the file we just downloaded.

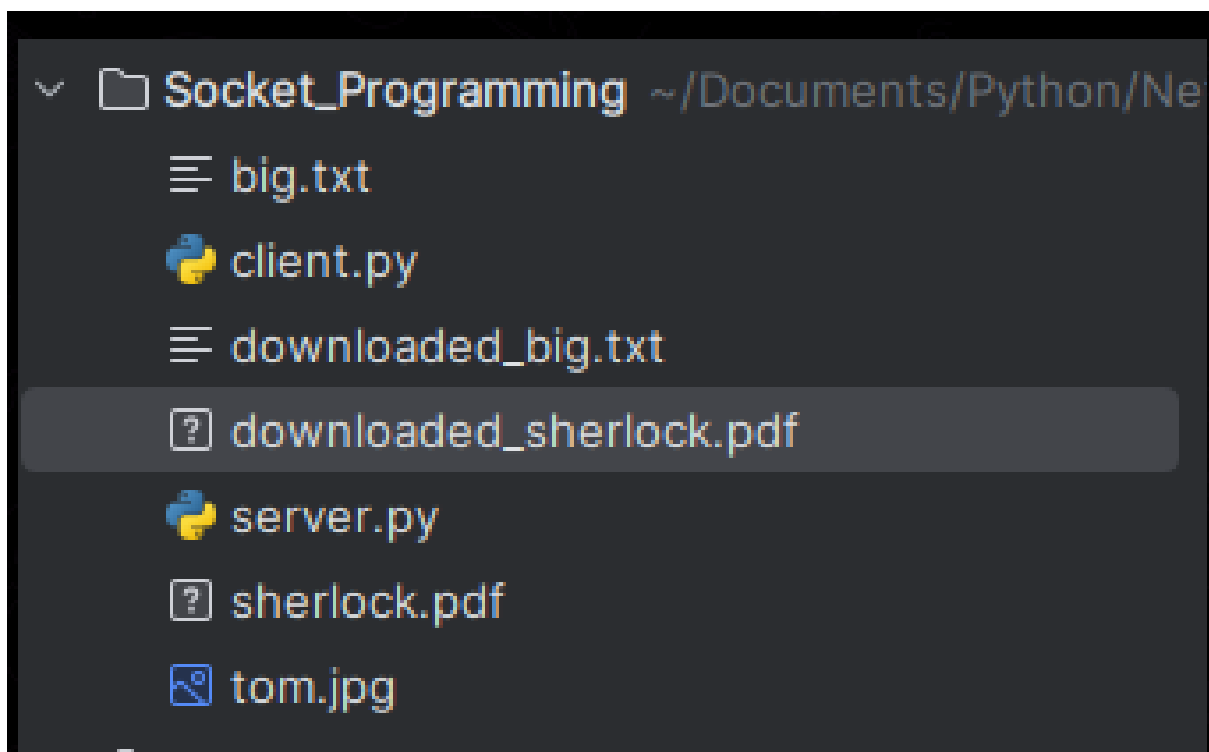


Figure 4: File downloaded successfully

4.1.5 Opening downloaded file :

Now, let's open the downloaded file.

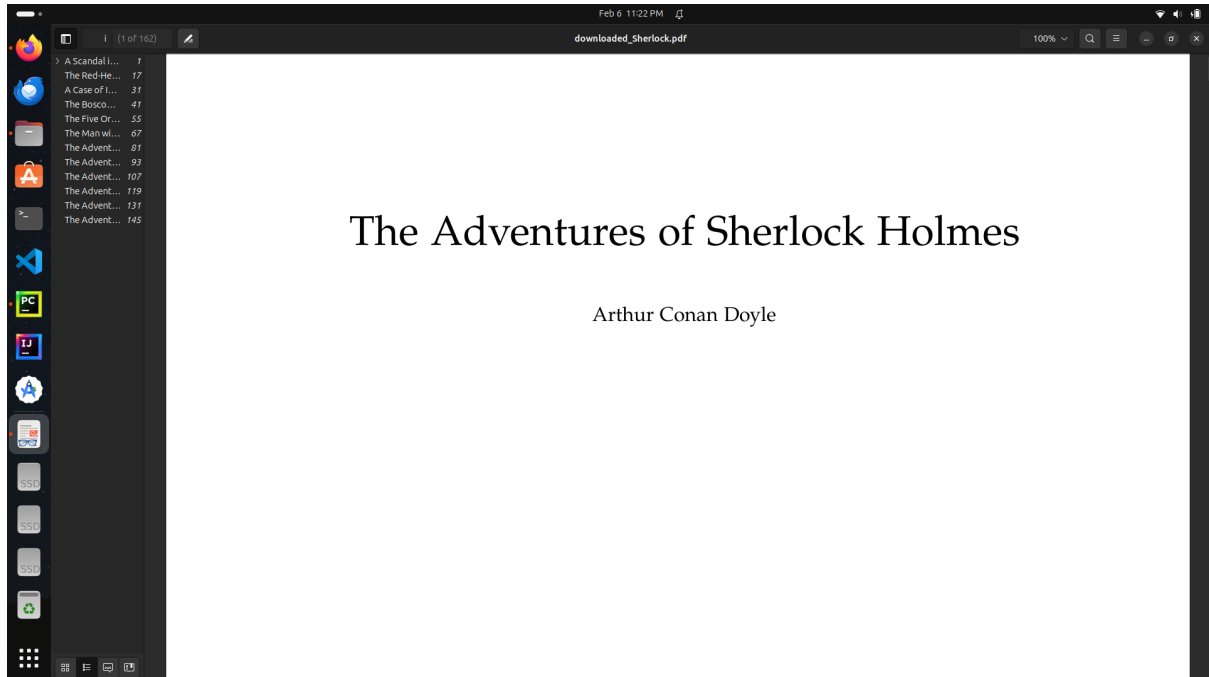


Figure 5: File opened

4.1.6 Server side after download from any client :

The server is tracking records, such as which file is downloaded by which client. Here, we also tried to handle multiple clients.

```
/usr/bin/python3.11 /home/asus/Documents/Python/NetworkingLab/Lab2_File_Transfer/Socket_Programming/server.py
[STARTING] Server is starting...
[LISTENING] Server is listening on 127.0.0.1 : 3381...
[NEW CONNECTION] ('127.0.0.1', 32808) connected...
[SENT] File 'sherlock.pdf' sent to ('127.0.0.1', 32808) at 2024-02-08 20:27:03.
[DISCONNECTED] ('127.0.0.1', 32808) disconnected...
|
```

Figure 6: Server Side

4.1.7 Error: File not found

If we want to download any file that is not in the given list, then it will show a file not found error.

```
/usr/bin/python3.11 /home/asus/Documents/Python/NetworkingLab/Lab2_File_Transfer/Socket_Programming/client.py
List of files available for download from [SERVER]:
tom.jpg
client.py
downloaded_big.txt
.idea
server.py
big.txt
downloaded_sherlock.pdf
sherlock.pdf
Enter the filename you want to download:
book.pdf
[SERVER]: File not found.

Process finished with exit code 0
```

Figure 7: Error

4.2 Task 2

4.2.1 After Running HTTP Server Code:

```
/usr/bin/python3.11 /home/hp/Downloads/Lab2_File_Transfer/HTTP/server.py
INFO:root:Starting httpd...
```

Figure 8: Content of Server

4.2.2 After Running HTTP Client Code:

The client is successfully connected to the server. The client can now download any file from the given list.

```
/usr/bin/python3.11 /home/hp/Downloads/Lab2_File_Transfer/HTTP/client.py
Choose operation (1 for GET, 2 for POST): |
```

Figure 9: Server Connect to Client

4.2.3 Select Get:

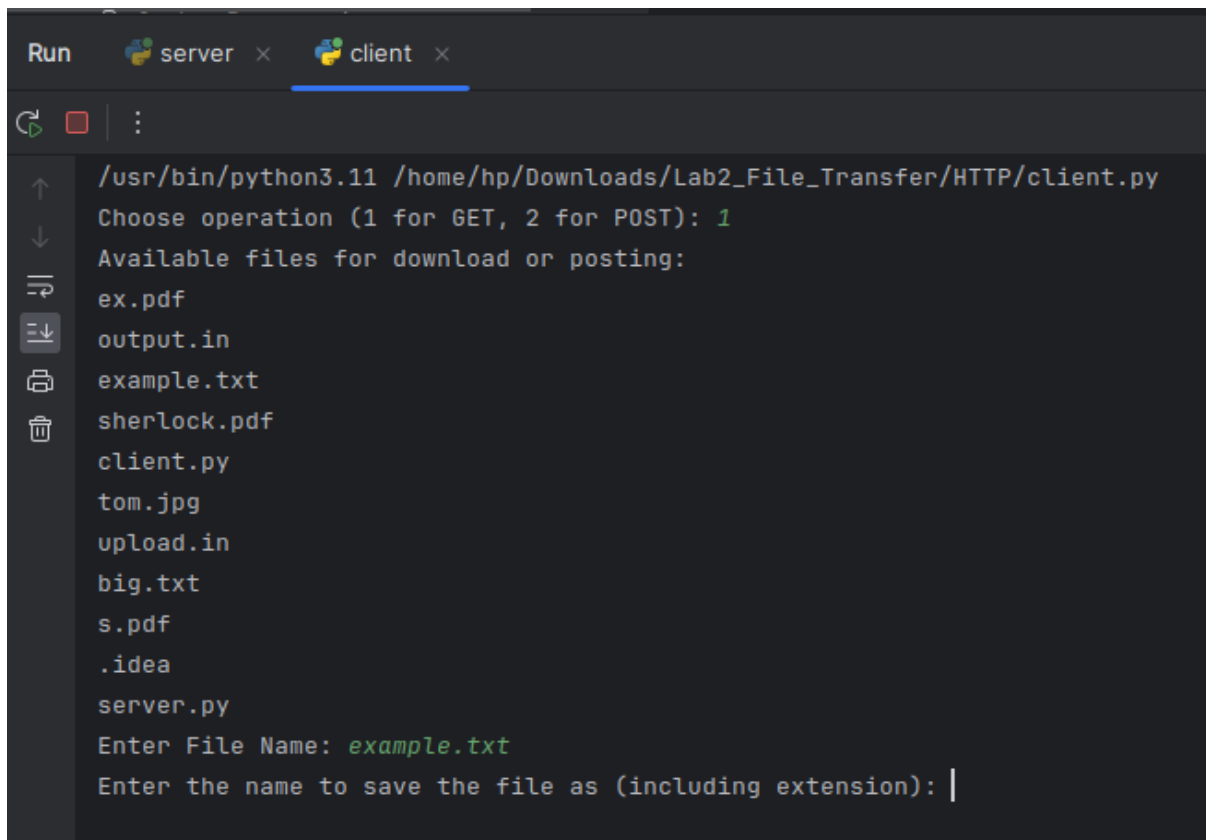
Select option 1 to Download file from server via http.

```
/usr/bin/python3.11 /home/hp/Downloads/Lab2_File_Transfer/HTTP/client.py
Choose operation (1 for GET, 2 for POST): 1
Available files for download or posting:
output.in
example.txt
sherlock.pdf
client.py
tom.jpg
upload.in
big.txt
s.pdf
.idea
server.py
Enter File Name:
```

Figure 10: File downloaded successfully

4.2.4 Enter the name of the downloaded file and rename it:

The client should enter the desired file name to download. The client can rename it before downloading.



```
Run  server x  client x
/usr/bin/python3.11 /home/hp/Downloads/Lab2_File_Transfer/HTTP/client.py
Choose operation (1 for GET, 2 for POST): 1
Available files for download or posting:
ex.pdf
output.in
example.txt
sherlock.pdf
client.py
tom.jpg
upload.in
big.txt
s.pdf
.idea
server.py
Enter File Name: example.txt
Enter the name to save the file as (including extension): |
```

Figure 11: Choose file name and rename it

4.2.5 File downloaded via HTTP programming:

File transfer successfully occurs via HTTP programming; the file size is also mentioned to the clients.

```
/usr/bin/python3.11 /home/hp/Downloads/Lab2_File_Transfer/HTTP/client.py
Choose operation (1 for GET, 2 for POST): 1
Available files for download or posting:
output.in
example.txt
sherlock.pdf
client.py
tom.jpg
upload.in
big.txt
s.pdf
.idea
server.py
Enter File Name: example.txt
Enter the name to save the file as (including extension): ex.pdf
[SERVER]: File 'ex.pdf' downloaded successfully. Size: 0.09 MB

Process finished with exit code 0
|
```

Figure 12: File downloaded successfully

4.2.6 Showing Downloaded file in the Folder :

Here, we can see the file we just downloaded.

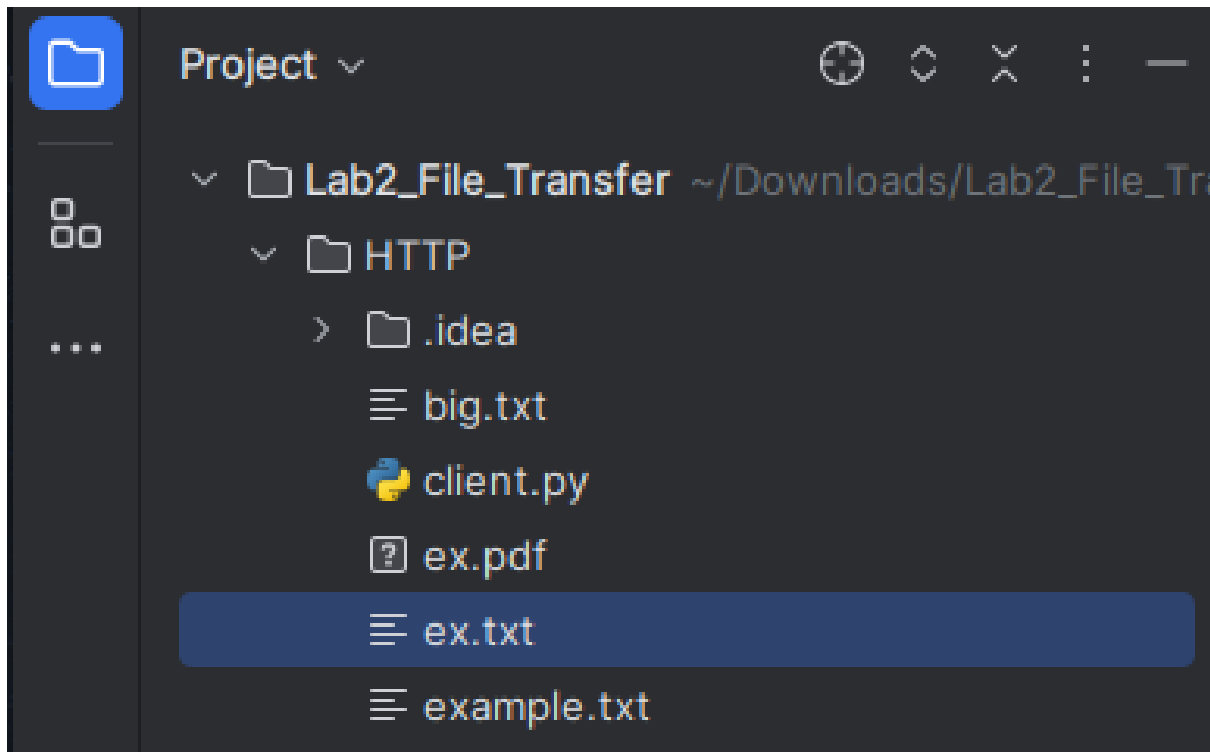


Figure 13: File downloaded successfully

4.2.7 Opening downloaded file :

Now, let's open the downloaded file.

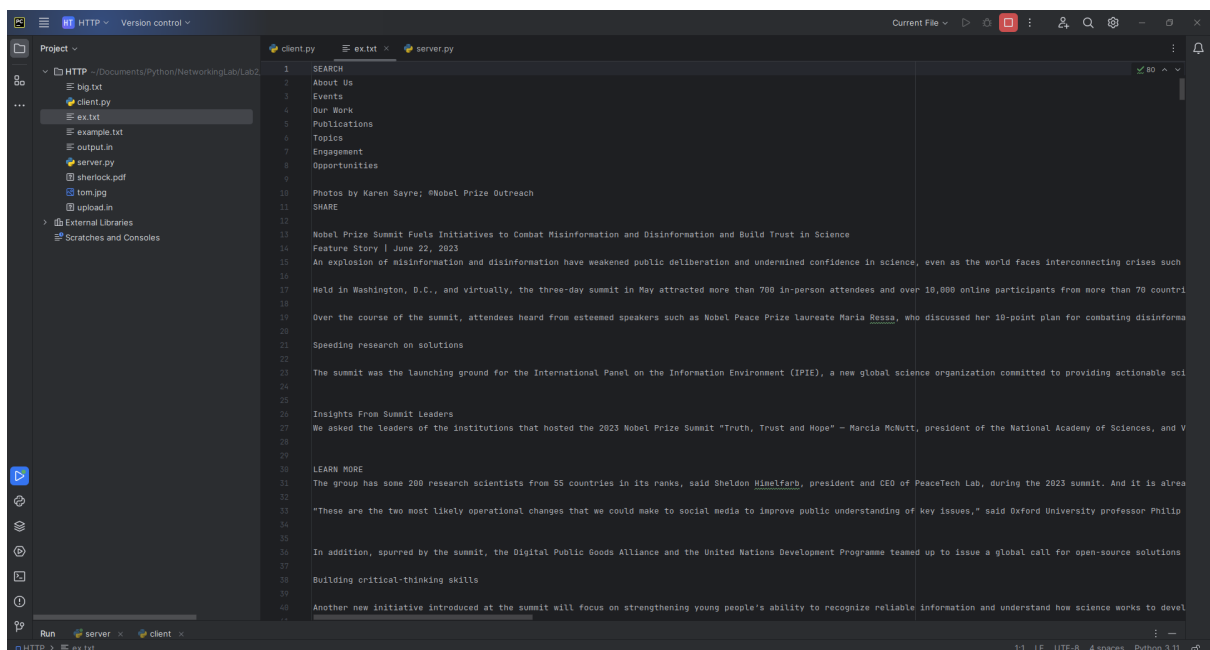


Figure 14: File opened

4.2.8 Server side after download from any client :

The server is tracking records, such as which file is downloaded by which client. Here, we also tried to handle multiple clients.

```
/usr/bin/python3.11 /home/asus/Documents/Python/NetworkingLab/Lab2_File_Transfer/Socket_Programming/server.py
[STARTING] Server is starting...
[LISTENING] Server is listening on 127.0.0.1 : 3381...
[NEW CONNECTION] ('127.0.0.1', 32808) connected...
[SENT] File 'sherlock.pdf' sent to ('127.0.0.1', 32808) at 2024-02-08 20:27:03.
[DISCONNECTED] ('127.0.0.1', 32808) disconnected...
|
```

Figure 15: Server Side

4.2.9 Error: File not found

If we want to download any file that is not in the given list, then it will show a file not found error.

```
/usr/bin/python3.11 /home/asus/Documents/Python/NetworkingLab/Lab2_File_Transfer/Socket_Programming/client.py
List of files available for download from [SERVER]:
tom.jpg
client.py
downloaded_big.txt
.idea
server.py
big.txt
downloaded_sherlock.pdf
sherlock.pdf
Enter the filename you want to download:
book.pdf
[SERVER]: File not found.

Process finished with exit code 0
```

Figure 16: Error

4.2.10 File upload via HTTP programming:

The client is successfully connected to the server. The client can now upload any file from the given list.

```
/usr/bin/python3.11 /home/hp/Downloads/Lab2_File_Transfer/HTTP/client.py
Choose operation (1 for GET, 2 for POST): 2
Available files for download or posting:
ex.pdf
output.in
example.txt
sherlock.pdf
client.py
tom.jpg
upload.in
big.txt
s.pdf
.idea
server.py
Enter File Path for Posting: tom.jpg
File successfully posted

Process finished with exit code 0
```

Figure 17: File uploaded successfully

4.2.11 Showing uploaded file in the folder:

Here, we can see the file we just uploaded.

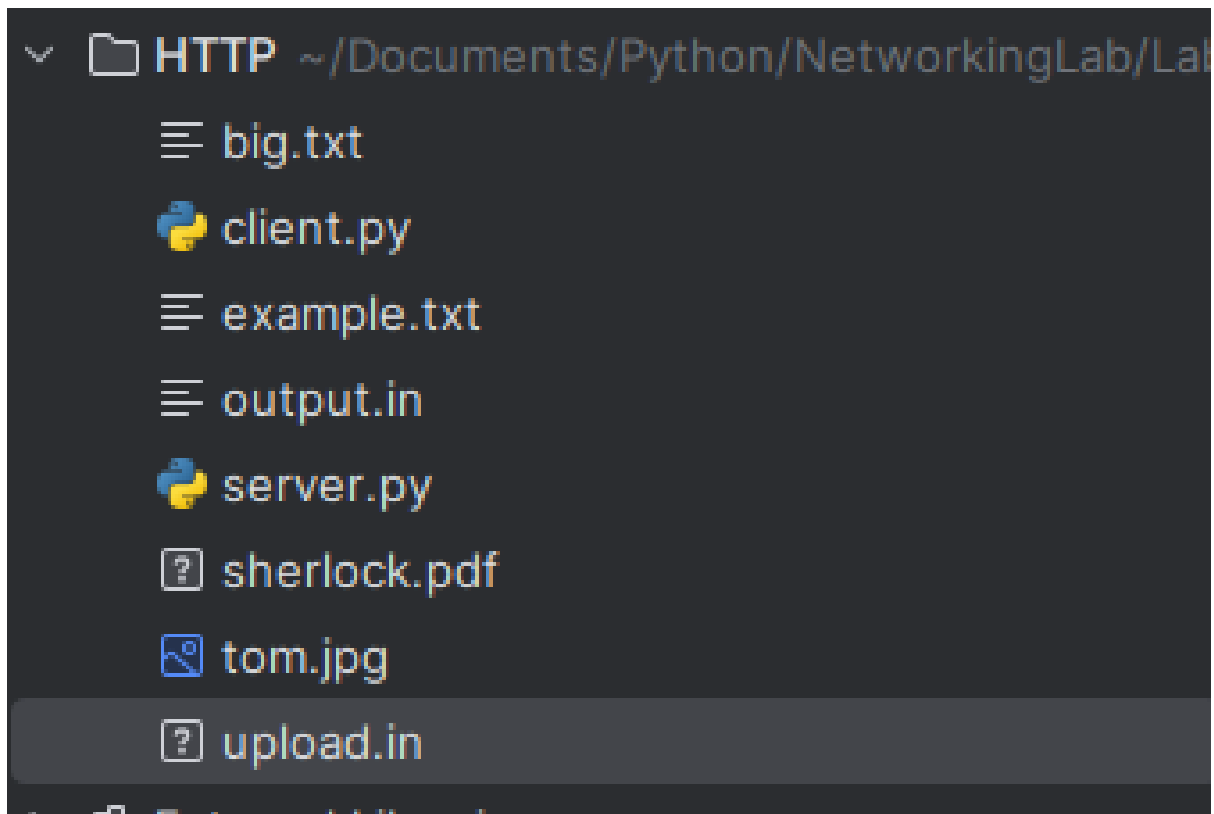


Figure 18: File uploaded successfully

4.2.12 Opening uploaded file :

Now, let's open the uploaded file.

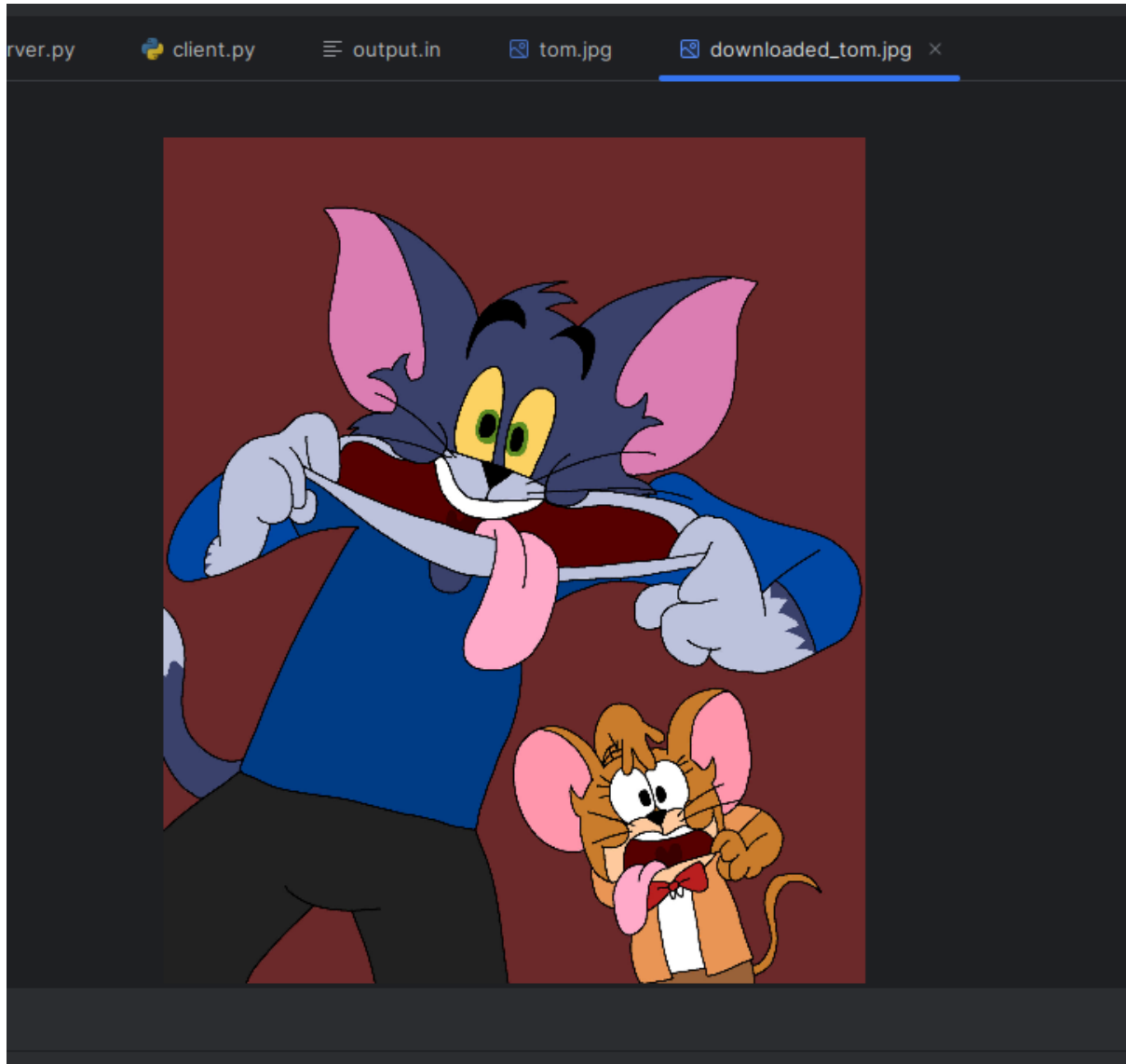
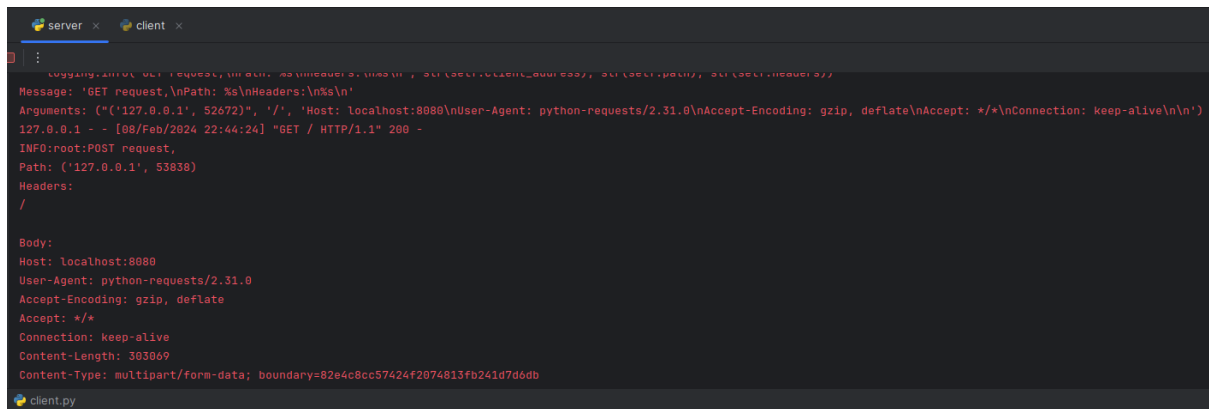


Figure 19: File opened

4.2.13 Server side after upload from any client :

The server is tracking records, such as which file is uploaded by which client. Here, we also tried to handle multiple clients.



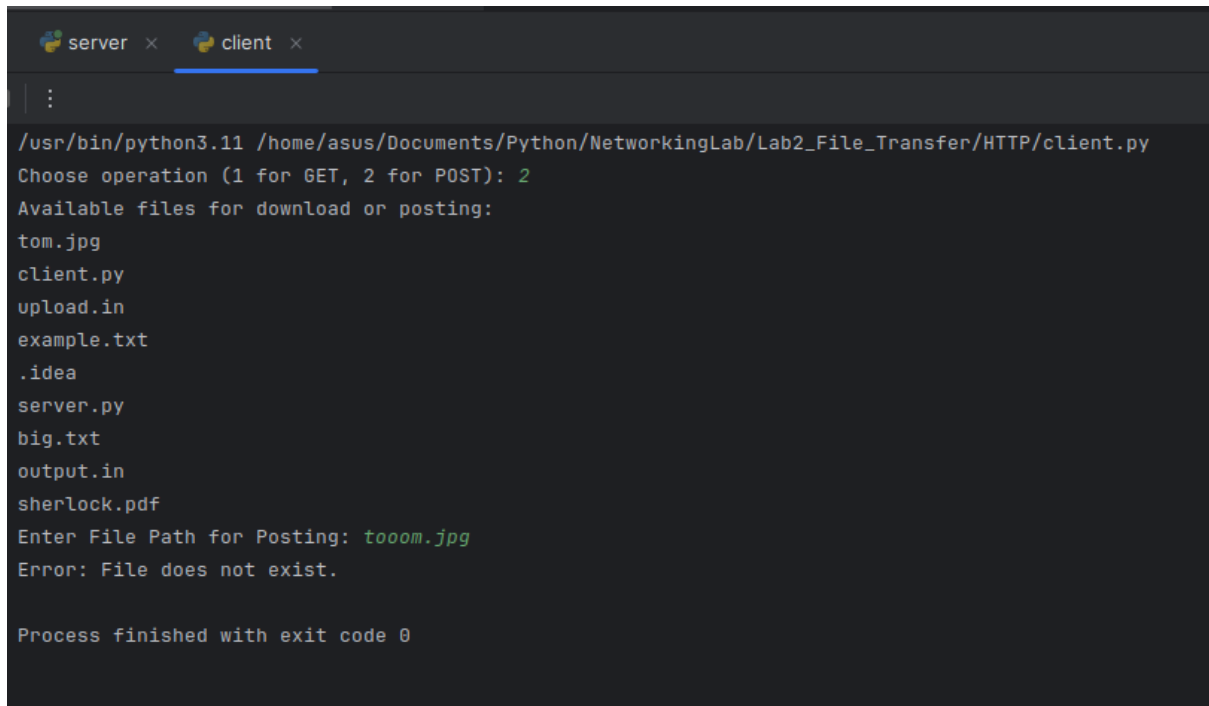
```
server client
Logging started, GET request, method: GET, message: GET request, path: /, client: client, body: /, client: client, body: /
Message: 'GET request,\nPath: %s\nHeaders:\n%s\n'
Arguments: (('127.0.0.1', 52672), '/', 'Host: localhost:8080\nUser-Agent: python-requests/2.31.0\nAccept-Encoding: gzip, deflate\nAccept: */*\nConnection: keep-alive\n\n')
127.0.0.1 - - [08/Feb/2024 22:44:24] "GET / HTTP/1.1" 200 -
INFO:root:POST request,
Path: ('127.0.0.1', 53030)
Headers:
/

Body:
Host: localhost:8080
User-Agent: python-requests/2.31.0
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Content-Length: 303069
Content-Type: multipart/form-data; boundary=82e4c8cc57424f2074813fb241d7d6db
client.py
```

Figure 20: Server Side

4.2.14 Error: File not found

If we want to upload any file that is not in the given list, then it will show a file not found error.



```
server x client x
| :
/usr/bin/python3.11 /home/asus/Documents/Python/NetworkingLab/Lab2_File_Transfer/HTTP/client.py
Choose operation (1 for GET, 2 for POST): 2
Available files for download or posting:
tom.jpg
client.py
upload.in
example.txt
.idea
server.py
big.txt
output.in
sherlock.pdf
Enter File Path for Posting: tooom.jpg
Error: File does not exist.

Process finished with exit code 0
```

Figure 21: Error

5 Experience

5.1 Socket Programming Exploration

- The initial phase of the experiment focused on socket programming, where participants gained familiarity with concepts such as socket creation, binding, listening, and accepting connections.
- By implementing a multithreaded chat system, participants honed their skills in handling communication between multiple clients and a single server concurrently.
- This task enabled participants to understand the nuances of multithreading and ensure efficient message handling in real-time scenarios.

5.2 HTTP Server Setup

- In the subsequent phase, participants set up an HTTP server process capable of handling HTTP requests from clients.
- Configuring the server involved defining endpoints for handling HTTP GET requests to download files from the server and implementing functionality to process HTTP POST requests for uploading files.
- This phase provided participants with insights into configuring and managing HTTP servers and understanding how clients interact with servers to exchange data.

5.3 Utilizing HTTP GET/POST Methods for File Transfer

- One of the highlights of the experiment was the implementation of file transfer mechanisms using HTTP GET and POST methods.
- Participants leveraged HTTP GET requests to enable clients to download files from the server, ensuring seamless retrieval of files by implementing proper request handling.
- Similarly, the implementation of HTTP POST requests enabled clients to upload files to the server, demonstrating robust mechanisms for receiving, processing, and storing uploaded files securely.

5.4 Challenges and Learnings

- Throughout the experiment, participants encountered various challenges, including handling concurrent connections, parsing HTTP requests, and ensuring data integrity during file transfer operations.
- However, these challenges served as valuable learning opportunities, allowing participants to deepen their understanding of network communication principles and develop effective strategies for addressing common issues encountered in real-world scenarios.

References

- [1] GeeksforGeeks. (n.d.). Introducing Threads in Socket Programming (Java). Retrieved from <https://www.geeksforgeeks.org/introducing-threads-socket-programming-java>
- [2] GeeksforGeeks. (n.d.). Transfer the file (client side) to the server (socket programming in Java). Retrieved from <https://www.geeksforgeeks.org/transfer-the-file-client-socket-to-server-socket-in-java/>
- [3] DigitalOcean. (n.d.). Java HttpURLConnection Example: Java HTTP Request GET/POST. Retrieved from <https://www.digitalocean.com/community/tutorials/java-httpurlconnection-example-java-http-request-get-post>
- [4] JavaTpoint. (n.d.). Java GET and POST. Retrieved from <https://www.javatpoint.com/java-get-post>