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:

Name: Muztoba Hasan Sinha

Roll No : 15

Name: Bholanath Das Niloy

Roll No : 22

Submitted On :

January 31th, 2023

Submitted To :

Dr. Md. Abdur Razzaque

Md Mahmudur Rahman

Md. Ashraful Islam

Md. Fahim Arefin

Contents

1 Introduction

In this lab we have implemented file transfer using sockets which are a mediatory in between the application layer and the transport layer where we connect a server and client and enable the client to download files that are readily available in the same folder as the server is file is running. After this we have learnt to do the same thing but over http which was albeit easier since most of the connection issues were handled by python.

2 Objectives

The objective of this lab is to give hands-on experience with socket programming and HTTP file transfer. We will :

1. implement multithreaded chat from many clients to one server
2. set up an HTTP server process with a few objects
3. use GET and POST methods to upload and download objects in between HTTP clients and a server

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.

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.

3 Theory

3.1 HTTP

HTTP (Hypertext Transfer Protocol) is the protocol used for transferring data over the internet. It is a client-server protocol, where a client sends a request to a server and the server returns a response. HTTP is used for transmitting data such as HTML pages, images, and videos and is the foundation of data communication for the World Wide Web. HTTP supports various methods such as GET, POST, PUT, DELETE, etc. that allow clients to retrieve or manipulate data stored on a server. HTTP uses port 80 as the default port for communication, although other ports can also be used. The HTTP protocol is stateless, meaning it does not maintain information or context between requests and is designed to be fast, simple, and flexible.

3.2 HTTP/1.1 Limitations

HTTP 1.1, which is the most widely used version of the HTTP protocol, has several limitations. One limitation is the size of HTTP headers, which limits the amount of information that can be sent in a single request. Additionally, HTTP 1.1 does not support persistent connections, so each request/response must be established and closed separately, leading to increased latency and overhead. The limited number of simultaneous connections to a server also leads to scalability issues when handling large numbers of clients. Another limitation is the lack of built-in support for secure and encrypted communication, which has been addressed in later versions of the protocol. Despite these limitations, HTTP 1.1 remains widely used due to its simplicity, robustness, and compatibility with a wide range of devices and applications.

3.3 HTTP/2

HTTP/2 is the last version of the HTTP protocol(superseeded by HTTP/3), released in 2015. It aims to overcome the limitations of HTTP/1.1, such as low efficiency in processing parallel requests and slow page loading time. HTTP/2 achieves this through the use of multiplexing and binary encoding, which allow multiple requests and responses to be sent over a single connection. This results in a faster and more efficient transfer of data between the client and server. Additionally, HTTP/2 supports server push, allowing the server to proactively send resources to the client without having to wait for a request. This leads to a better user experience, as pages load faster and with fewer round-trip requests.

3.4 Usage of HTTP on the web:

HTTP (Hypertext Transfer Protocol) is a protocol used for transmitting and receiving information on the web. It is a stateless protocol, which means that each request and response is handled independently, without any memory of previous interactions. HTTP has two main methods for transmitting information, the GET method and the POST method.

The GET method is used to retrieve information from a server, while the POST method is used to send information to the server, such as submitting a form or uploading a file. The GET method is typically used for retrieving static content, such as a web page or an image, while the POST method is used for sending dynamic data to the server, such as submitting a form or uploading a file.

Both GET and POST methods are commonly used in web development, and choosing the right method depends on the use case and the type of data being transmitted.

4 Code

1. Sending files using socket programming

   (a) Server Side Code For handling multiple connections at once from different clients using socket programming:

```python
import socket
import tqdm
import os
import threading

SERVER_HOST = ""
SERVER_PORT = 5002
BUFFER_SIZE = 4096
SEPARATOR = "<SEPARATOR>"

s = socket.socket()
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((SERVER_HOST, SERVER_PORT))

def handle_client(client_socket, address):
    print(f"[NEW CONNECTION] {address} connected")

    connected = True
    while connected:
        arr = os.listdir('.')
        print(arr)

        client_socket.send(str(len(arr)).encode())

        for st in arr:
            st = st + "\n"
            client_socket.send(st.encode())

        filename = client_socket.recv(BUFFER_SIZE).decode()
        if filename == "quit":
            connected = False
            break
        filesize = os.path.getsize(filename)

        print(filename, filesize)

        client_socket.send(f"{filename}{SEPARATOR}{filesize}".encode())

        progress = tqdm.tqdm(range(filesize), f"Sending {filename}", unit="B",
    unit_scale=True, unit_divisor=1024)
        with open(filename, "rb") as f:
            while True:
                bytes_read = f.read(BUFFER_SIZE)
                if not bytes_read:
                    break
                client_socket.sendall(bytes_read)
                progress.update(len(bytes_read))
    client_socket.close()

def start():
    s.listen()
    print(f"[LISTENING] Server is listening on {SERVER_HOST}:{SERVER_PORT}")
    while True:
        conn, addr = s.accept()
        thread = threading.Thread(target=handle_client, args=(conn, addr))
        thread.start()
        print(f"[ACTIVE CONNECTIONS] {threading.activeCount() - 1}")

print("[STARTING] server is starting...")
start()
```

Listing 1: Server Side Code

(b) Client Side Code

```python
import socket
import tqdm
import os
import pickle

SEPARATOR = "<SEPARATOR>"
BUFFER_SIZE = 4096
host = "10.33.2.81"

port = 5002

s = socket.socket()

print(f"[+] Connecting to {host}:{port}")
s.connect((host, port))
print("[+] Connected.")
data = s.recv(BUFFER_SIZE).decode()
sz = int(data)
print(sz)
#for i in range(sz-1):
data = s.recv(BUFFER_SIZE).decode()
if data :
    print(data)
fn = input("select file by name ->")
s.send(fn.encode())
received = s.recv(BUFFER_SIZE).decode()
filename, filesize = received.split(SEPARATOR)

filename = os.path.basename(filename)
filesize = int(filesize)

progress = tqdm.tqdm(range(filesize), f"Receiving {filename}", unit="B", unit_scale
    =True, unit_divisor=1024)
with open(filename, "wb") as f:
    while True:
        bytes_read = s.recv(BUFFER_SIZE)
        if not bytes_read:
            break
        f.write(bytes_read)
        progress.update(len(bytes_read))

s.close()
```

Listing 2: Client Side Code

2. Sending Files over http

(a) Server Side Code for sending files over http

```python
from http.server import BaseHTTPRequestHandler, HTTPServer
import logging
import re

class S(BaseHTTPRequestHandler):
    def _set_response(val):
        val.send_response(200)
        val.send_header('Content-type', 'text/html')
        val.end_headers()

    def do_GET(val):
        logging.info("GET request,\nPath: %s\nHeaders:\n%s\n", str(val.path), str(
    val.headers))
        file_path = val.path[1:]
        try:
            with open(file_path, "rb") as f:
                file_content = f.read()

        except FileNotFoundError:
            logging.error("File not found: %s", file_path)
            val.send_error(404, "File not found")
            return
        val._set_response()
        val.wfile.write(file_content)

    def do_POST(val):
        content_length = int(val.headers['Content-Length'])
        post_data = val.rfile.read(content_length)

        logging.info("POST request,\nPath: %s\nHeaders:\n%s\n\nBody:\n%s\n",
                str(val.path), str(val.headers), post_data.decode('utf-8'))

        boundary = val.headers.get('Content-Type').split("=")[-1].encode()

        lines = post_data.split(boundary)
        for line in lines:
            if b"filename" in line:
                filename = re.findall(b'filename="(.*?)"', line)[0].decode()
                file_content = post_data.split(b'\r\n\r\n')[-1]

                with open(filename, "wb") as f:
                    f.write(file_content)

        val._set_response()
        val.wfile.write(f"File '{filename}' received and saved".encode('utf-8'))

def run(server_class=HTTPServer, handler_class=S, port=8080):
    logging.basicConfig(level=logging.INFO)
    server_address = ('', port)
    httpd = server_class(server_address, handler_class)
    logging.info('Starting httpd...\n')
    try:
        httpd.serve_forever()
    except KeyboardInterrupt:
        pass
    httpd.server_close()
    logging.info('Stopping httpd...\n')

if __name__ == '__main__':
    from sys import argv

    if len(argv) == 2:
        run(port=int(argv[1]))
    else:
        run()
```

Listing 3: Server Side Code

(b) Client Side Code

i. **GET**

```
import requests

x = requests.get('http://localhost:8080/')
if x.status_code == 200:
    with open("received_file.txt", "wb") as f:
        f.write(x.content)
    print("File successfully received")
    print("Content: ",x.text)
```

Listing 4: Client Side Code for GET

ii. **POST**

```
import requests

url = "http://localhost:8080/"
file_path = "nani.txt"
```

```python

with open(file_path, "rb") as f:
    files = {"file": f}
    response = requests.post(url, files=files)

print(response.text)
```
Listing 5: Client Side Code POST

5 Explaining the Code

5.1 Sending Files using Socket Programming

1. **Server Side Code**
   This code implements a simple file server in Python using sockets. The server runs on a specified host and port (**SERVER_HOST** and **SERVER_PORT**). It uses a socket object s to bind to the host and port, and listen for incoming connections.
   When a client connects, the **handle_client** function is called in a new thread to handle the client's request. The function sends a list of files in the current directory to the client, then receives a filename from the client. If the client sends **"quit"**, the connection is closed. If a valid filename is received, the function sends the filename and filesize to the client. Finally, the file is sent to the client in chunks using the **tqdm** library to display a progress bar.
   The start function is responsible for listening for incoming connections and starting a new thread for each client that connects. The number of active connections is displayed using **threading.activeCount()**.
   The script starts by calling **start()**, and the server starts listening for incoming connections.

2. **Client Side Code**
   This is a Python script that connects to a server using the socket module, and downloads a file from that server. The script performs the following steps, Imports required modules - **socket, tqdm, os, and pickle**. Defines the host and port of the server, and creates a socket object. Connects to the server using the connect method of the socket object, and prints a message indicating that it is connected. Receives the number of files present in the server's current working directory and the names of those files. Prompts the user to select a file by name and sends the name of the selected file to the server. Receives the file name and size from the server, and extracts them from the received message. Displays a progress bar for downloading the file using the **tqdm** module. Writes the contents of the file to the local system in binary format. Closes the connection to the server using the close method of the socket object.

5.2 Sending Files over HTTP

1. **Server Side Code:**
   This code is a simple implementation of a HTTP server using the Python standard library module http.server. The server is implemented as a custom handler, **S**, which inherits from **BaseHTTPRequestHandler**.
   The **do_GET** method handles incoming **GET** requests, which the server uses to return a file from the file system. The file path is obtained from the URL, and an attempt is made to open the file for reading. If the file is found, its contents are sent back to the client as the response. If the file is not found, a **404** error is returned to the client.

   The **do_POST** method handles incoming **POST** requests, which the server uses to receive a file from the client. The method first gets the content length of the request and reads the raw request data. Then it parses the request data to extract the file name and its contents. The file is then saved to the file system using the extracted name. A response is sent back to the client indicating that the file was received and saved.

   Finally, the **run** method starts the HTTP server and listens for incoming requests. The server will continue to run until interrupted by a keyboard interrupt.

   The script uses basic logging to output information about incoming requests and other log messages.

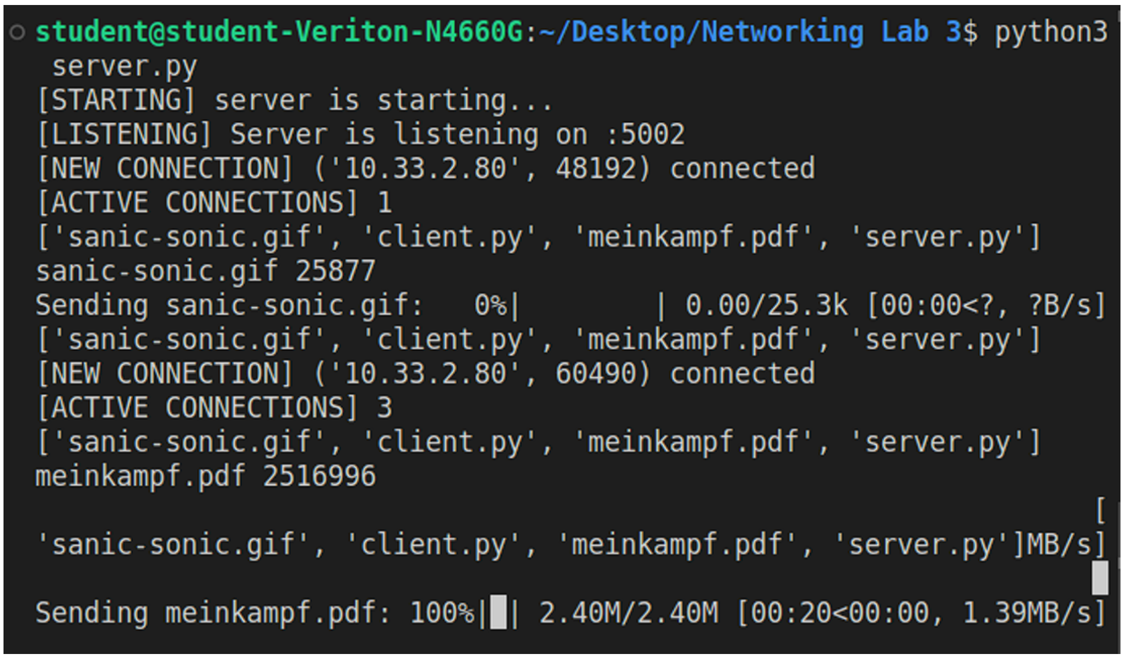
2. **Client Side Code:**

(a) **GET**
This code imports the **requests** library and uses it to make a **GET** request to a specified URL (**http://localhost:8080/hello.txt**). If the request returns with a status code of **200** (indicating a successful response), it opens a file named **hello.txt** and writes the contents of the response to it. If the response status code is not **200**, the code will print an error message indicating that the file download failed.

(b) **POST**
This code is a Python script that uses the **requests** library to make a **POST request** to an HTTP server running at **"http://localhost:8080/"**. The POST request includes the contents of a file, **"nani.txt"**, as the body of the request. The contents of the file are read and passed as a multi-part file in the "files" argument of the post request. The response from the server is then printed to the console.

6 Screenshots of Working Code

6.1 Sending Files using Socket Programming

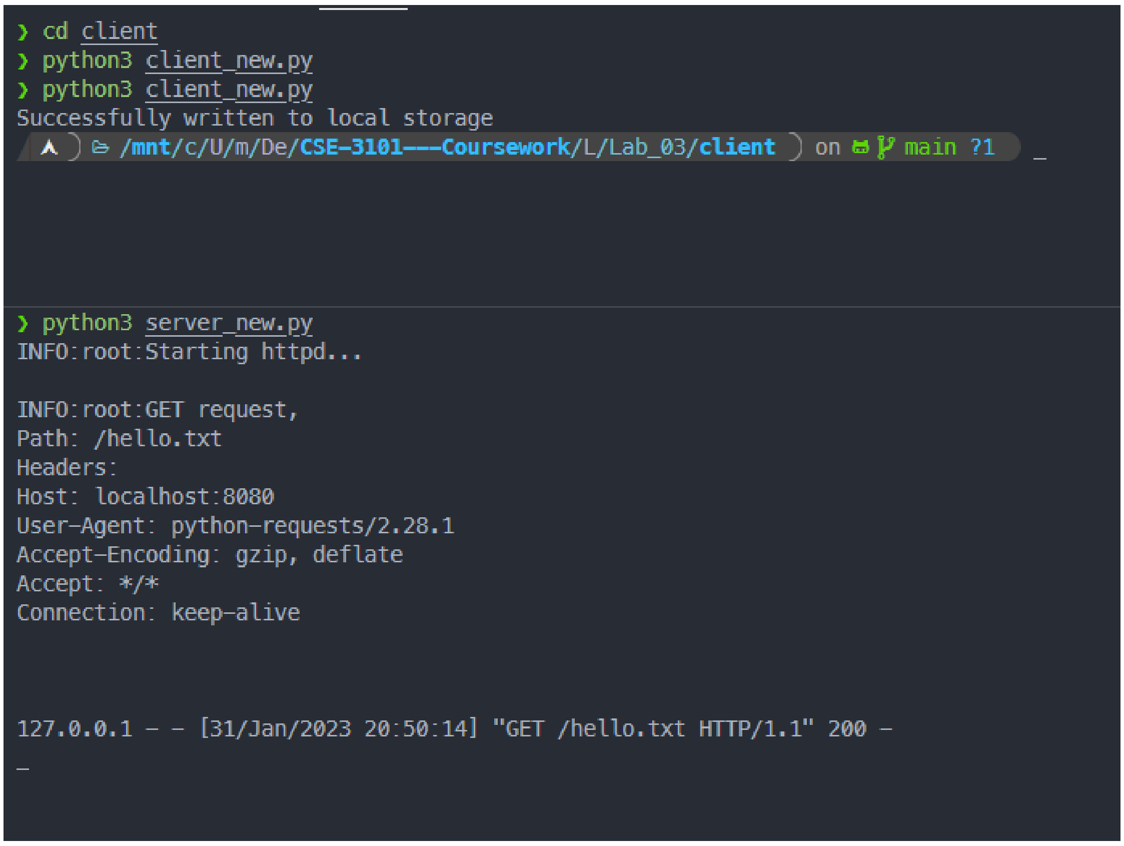


```
○ student@student-Veriton-N4660G:~/Desktop/Networking Lab 3$ python3
  server.py
 [STARTING] server is starting...
 [LISTENING] Server is listening on :5002
 [NEW CONNECTION] ('10.33.2.80', 48192) connected
 [ACTIVE CONNECTIONS] 1
 ['sanic-sonic.gif', 'client.py', 'meinkampf.pdf', 'server.py']
 sanic-sonic.gif 25877
 Sending sanic-sonic.gif:   0%|         | 0.00/25.3k [00:00<?, ?B/s]
 ['sanic-sonic.gif', 'client.py', 'meinkampf.pdf', 'server.py']
 [NEW CONNECTION] ('10.33.2.80', 60490) connected
 [ACTIVE CONNECTIONS] 3
 ['sanic-sonic.gif', 'client.py', 'meinkampf.pdf', 'server.py']
 meinkampf.pdf 2516996
                                                                    [
 'sanic-sonic.gif', 'client.py', 'meinkampf.pdf', 'server.py']MB/s]
                                                                    █
 Sending meinkampf.pdf: 100%|█| 2.40M/2.40M [00:20<00:00, 1.39MB/s]
```

Figure 1: Multiple Connections with Server

As can be seen here the server is sending the file to multiple different clients with the progress bars indicating as such.

6.2 Sending Files over http



```
❯ cd client
❯ python3 client_new.py
❯ python3 client_new.py
Successfully written to local storage
 ⅄ ❯ 🗁 /mnt/c/U/m/De/CSE-3101---Coursework/L/Lab_03/client ❯ on  main ?1  _
```

```
❯ python3 server_new.py
INFO:root:Starting httpd...

INFO:root:GET request,
Path: /hello.txt
Headers:
Host: localhost:8080
User-Agent: python-requests/2.28.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive



127.0.0.1 - - [31/Jan/2023 20:50:14] "GET /hello.txt HTTP/1.1" 200 -
_
```

Figure 2: HTTP connection GET request

As can be seen a simple GET request from the client is being executed here by the server. And the screenshots below show how the file is written to local storage.

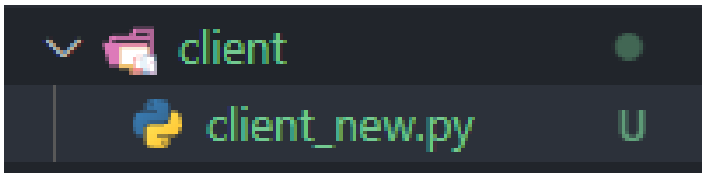


Figure 3: Before the GET call

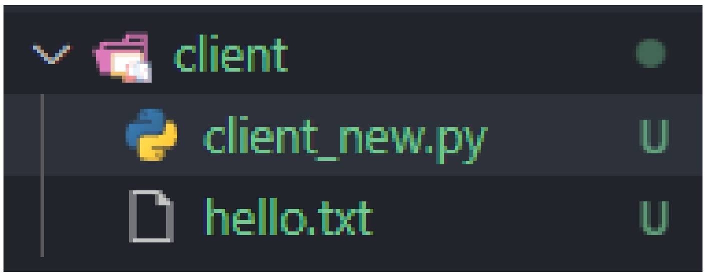
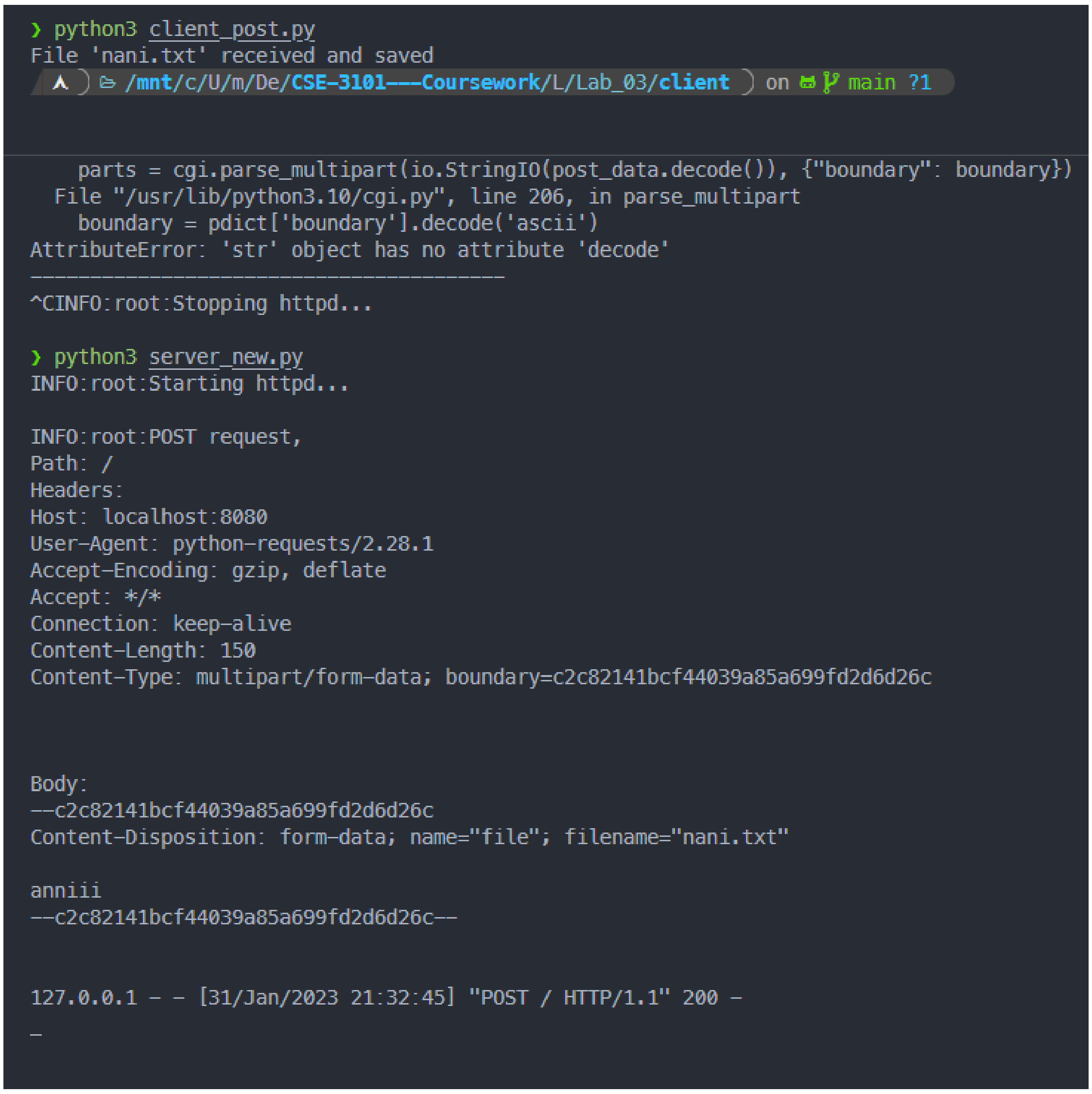


Figure 4: After the GET call

Finally we are doing the POST call and getting some file uploaded to the server.



```
❯ python3 client_post.py
File 'nani.txt' received and saved
 ⅄ 〉🗁 /mnt/c/U/m/De/CSE-3101---Coursework/L/Lab_03/client 〉on 🐱 main ?1


    parts = cgi.parse_multipart(io.StringIO(post_data.decode()), {"boundary": boundary})
  File "/usr/lib/python3.10/cgi.py", line 206, in parse_multipart
    boundary = pdict['boundary'].decode('ascii')
AttributeError: 'str' object has no attribute 'decode'
----------------------------------------
^CINFO:root:Stopping httpd...

❯ python3 server_new.py
INFO:root:Starting httpd...

INFO:root:POST request,
Path: /
Headers:
Host: localhost:8080
User-Agent: python-requests/2.28.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Content-Length: 150
Content-Type: multipart/form-data; boundary=c2c82141bcf44039a85a699fd2d6d26c



Body:
--c2c82141bcf44039a85a699fd2d6d26c
Content-Disposition: form-data; name="file"; filename="nani.txt"

anniii
--c2c82141bcf44039a85a699fd2d6d26c--


127.0.0.1 - - [31/Jan/2023 21:32:45] "POST / HTTP/1.1" 200 -
_
```

Figure 5: POST call

References

[1] 262588213843476. Simple Python 3 HTTP server for logging all GET and POST requests. https://gist.github.com/mdonkers/63e115cc0c79b4f6b8b3a6b797e485c7. [Online; accessed 2023-01-31].

[2] Digamber. How to create socket server with multiple clients in python. https://www.positronx.io/create-socket-server-with-multiple-clients-in-python/, sep 14 2020. [Online; accessed 2023-01-31].

[3] freeCodeCamp.org. Simplehttpserver explained: How to send files using python. *freeCodeCamp.org*, jan 9 2020. [Online; accessed 2023-01-31].

[4] Marcus Sanatan. How to Upload Files with Python's requests Library. *Stack Abuse*, dec 21 2020. [Online; accessed 2023-01-31].