CSE-3111: Computer Networking Lab

Lab Report 2: Implementing File Transfer using HTTP GET/POST requests

Submitted by:

Tazkia Malik
Roll: 07

Taif Ahmed Turjo
Roll: 45

Submitted to:

Dr. Ismat Rahman
Associate Professor, Dept. of CSE, University of Dhaka

Mr. Palash Roy
Lecturer, Dept. of CSE, University of Dhaka

Mr. Jargis Ahmed
Lecturer, Dept. of CSE, University of Dhaka

Department of Computer Science & Engineering

University of Dhaka

Sunday 18th May, 2025

Contents

1 Introduction

Efficient and reliable file transfer mechanisms are essential in the design of modern web-based applications. The Hypertext Transfer Protocol (HTTP) has emerged as a dominant standard for such data exchanges due to its simplicity, scalability, and broad platform compatibility. Utilizing HTTP methods—specifically GET for retrieving resources and POST for submitting data—enables structured and secure file transfer operations within the client-server model. Unlike low-level socket programming, HTTP abstracts much of the underlying complexity, allowing developers to implement file upload and download functionalities in a more accessible and interoperable manner. This laboratory work focuses on implementing file transfer using HTTP GET and POST requests, demonstrating their practical application in facilitating seamless communication between clients and web servers.

2 Objective

The main objectives of this lab are:

- To understand and implement file transfer using the HTTP protocol through GET and POST methods.
- To develop a client-server model where the client can upload and download files via HTTP requests.
- To analyze the practical advantages and limitations of using HTTP for file transfer in comparison to lower-level communication methods.

3 Theory

3.1 What is HTTP?

The Hypertext Transfer Protocol (HTTP) is an application-layer protocol used for transmitting hypermedia documents, such as HTML, over the Internet. It follows a client-server architecture, where the client sends a request and the server returns a corresponding response. HTTP is stateless, meaning each request is processed independently without knowledge of previous interactions. It is the foundation of data communication on the World Wide Web and supports various request methods for resource interaction.

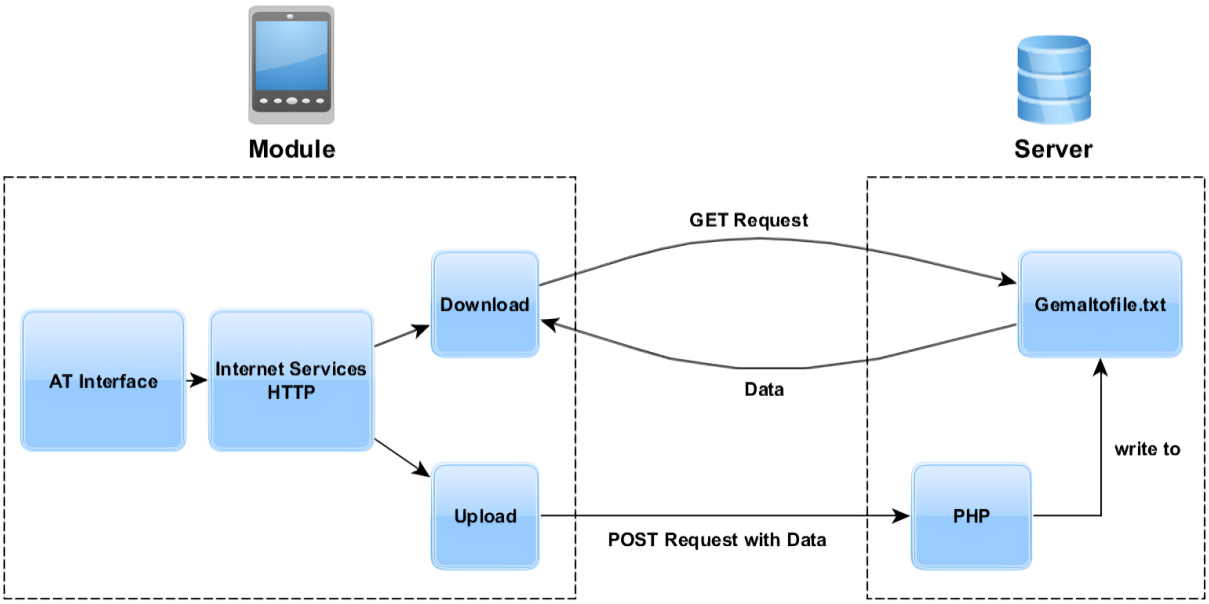

Figure 1: Illustration of a GET and POST request

3.2 GET Method

The GET method is used to request data from a specified resource. It appends parameters to the URL and is primarily used for retrieving information without making any modifications on the server side. GET requests are idempotent and safe, meaning repeated requests yield the same result without side effects.

3.3 POST Method

The POST method is used to submit data to the server, often resulting in a change in server state or database content. Unlike GET, POST transmits data within the request body, allowing for the secure and efficient transfer of large payloads, such as files or form submissions. POST requests are not idempotent, as each request may create or modify resources.

4 Design Details

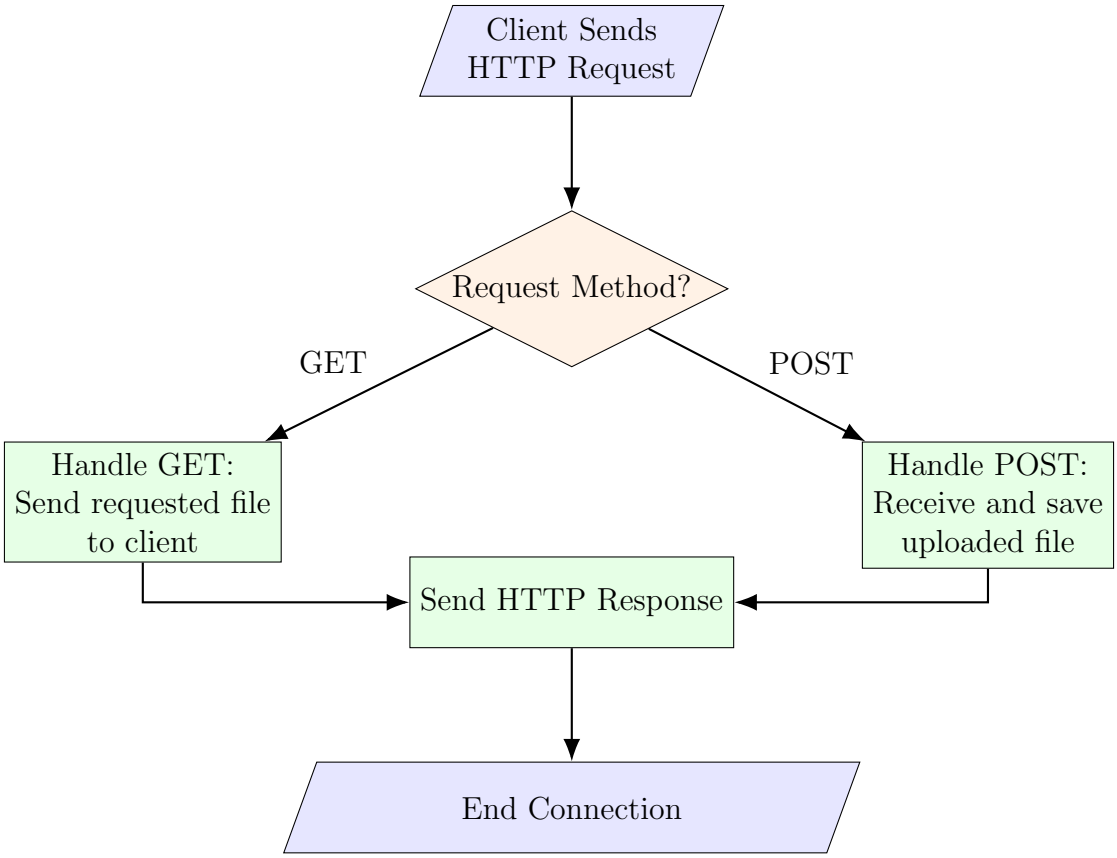

Figure 2: Flowchart of File Transfer using HTTP GET and POST

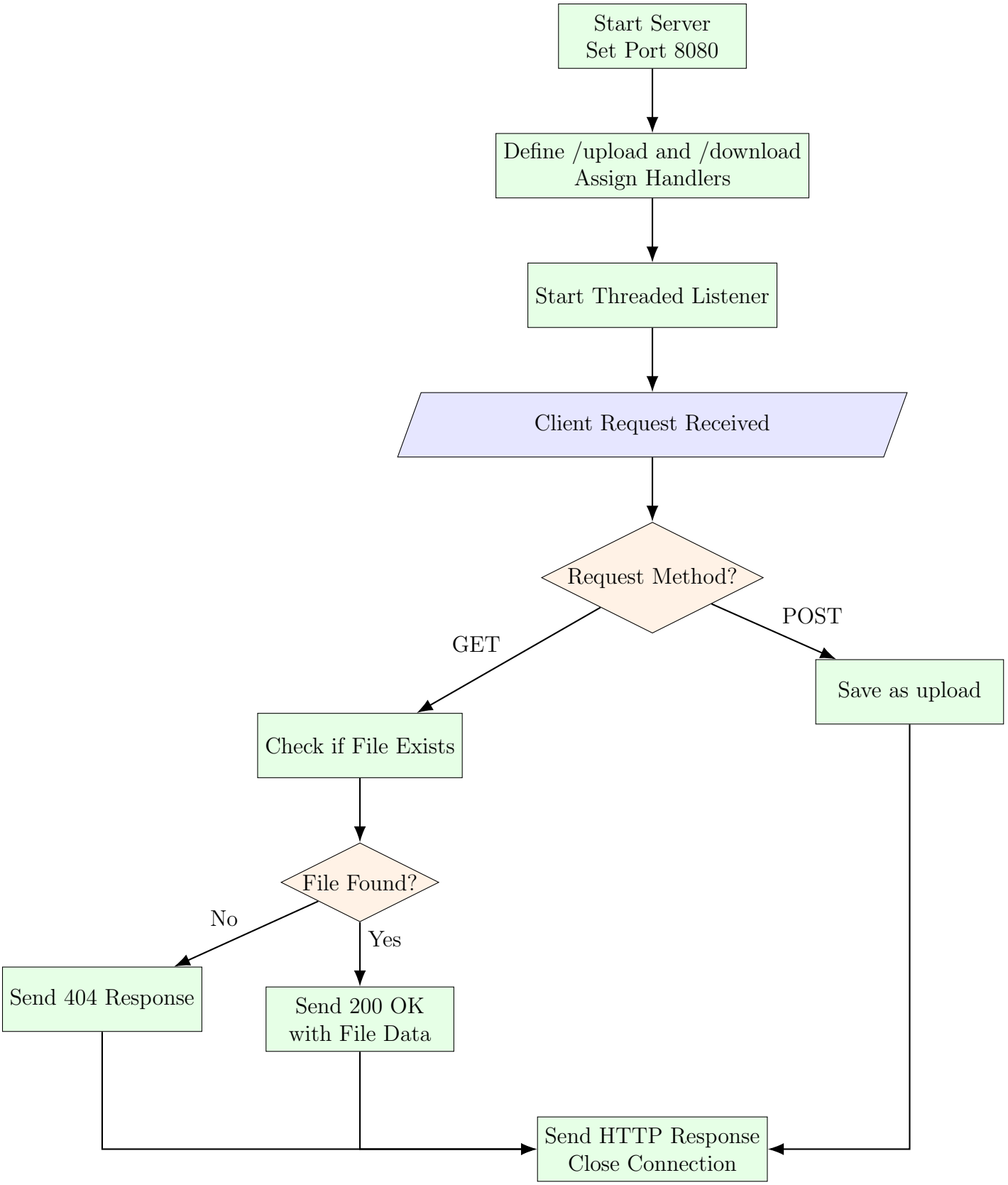

Figure 3: Flowchart of Server-Side Algorithm for HTTP File Transfer

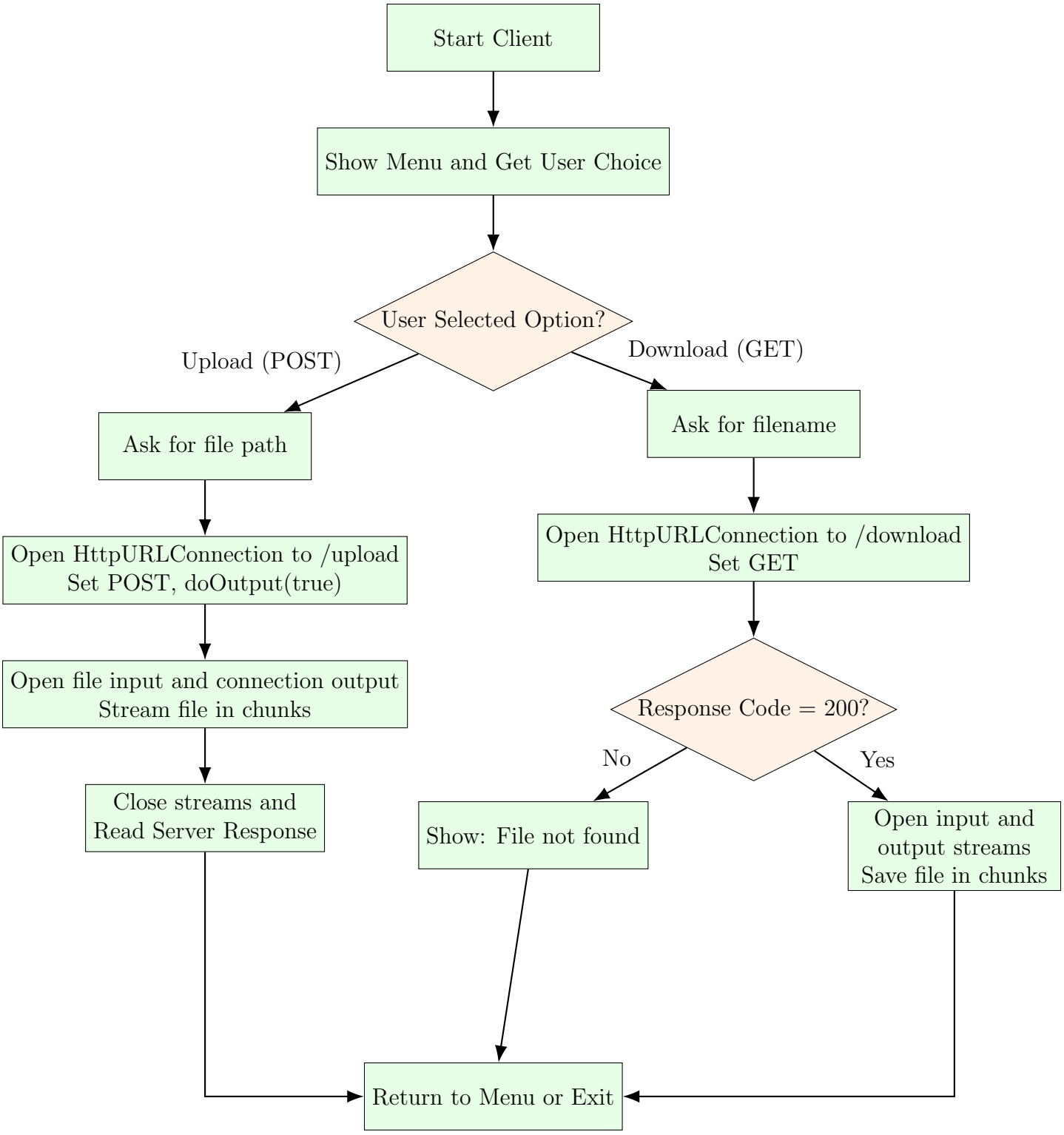

Figure 4: Flowchart of Client-Side Algorithm for HTTP File Upload and Download

5 Implementation

5.1 Server Side Implementation

The server is responsible for handling client requests for uploading and downloading files using the HTTP protocol over raw sockets. It listens for connections and processes both GET and POST requests. The steps for the server implementation are as follows:

1. Create a `ServerSocket` to listen on port `8080`.
2. Accept incoming client connections using a loop.
3. For each client connection, spawn a new thread to handle the request concurrently.
4. Read the HTTP request line using a `BufferedReader`.
5. Parse the HTTP method (GET or POST) and the request path (e.g., `/download?filename=test.txt`
6. For a GET request:
   - Extract the filename from the query string.
   - Check if the file exists in the server directory (`HTTP_server_files`).
   - If found, send HTTP 200 OK response with headers (`Content-Type`, `Content-Length`, `Content-Disposition`) and stream the file content.
   - If not found, send an HTTP 404 Not Found response.
7. For a POST request:
   - Generate a unique filename using a timestamp (e.g., `upload_16843523123`).
   - Read the file data from the client's input stream in chunks.
   - Save the file to the server directory.
   - Send an HTTP 200 OK response with a confirmation message.
8. If the method is not GET or POST, respond with HTTP 405 Method Not Allowed.
9. After responding to the request, close the client connection.

```java
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;
import java.nio.file.*;

public class HTTPServer {
    private static final int PORT = 8080;
    private static final String FILE_DIR = "HTTP_server_files";

    public static void main(String[] args) {
        new File(FILE_DIR).mkdirs();
        try (ServerSocket sock = new ServerSocket(PORT)) {
            System.out.println("HTTP Server running on port: " + PORT);
            while (true) {
                Socket client = sock.accept();
                new Thread(() -> handleClient(client)).start();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private static void handleClient(Socket client) {
        try (
            BufferedReader reader = new BufferedReader(new
               InputStreamReader(client.getInputStream()));
            OutputStream out = client.getOutputStream()
        ) {
            String requestLine = reader.readLine();
            if (requestLine == null || requestLine.isEmpty()) return;

            System.out.println("Received: " + requestLine);
            String[] parts = requestLine.split(" ");
            String method = parts[0];
            String path = parts[1];

            if (method.equals("GET") &&
               path.startsWith("/download?filename=")) {
                String filename =
                   path.substring("/download?filename=".length());
                File file = new File(FILE_DIR, filename);

                if (file.exists() && file.isFile()) {
                    byte[] data = Files.readAllBytes(file.toPath());
                    out.write(("HTTP/1.1 200 OK\r\n" +
                            "Content-Type: application/octet-stream\r\n" +
                            "Content-Length: " + data.length + "\r\n" +
                            "Content-Disposition: attachment;
                               filename=\"" + file.getName() +
                               "\"\r\n\r\n").getBytes());
                    out.write(data);
                } else {
                    send404(out);
                }
```

```
50
51              } else if (method.equals("POST") && path.equals("/upload")) {
52                  String uploadedFilename = "upload_" +
                        System.currentTimeMillis();
53                  File file = new File(FILE_DIR, uploadedFilename);
54
55                  InputStream in = client.getInputStream();
56                  try (FileOutputStream fos = new FileOutputStream(file)) {
57                      byte[] buffer = new byte[8192];
58                      int len;
59                      while ((len = in.read(buffer)) != -1) {
60                          fos.write(buffer, 0, len);
61                          if (len < buffer.length) break;
62                      }
63                  }
64
65                  String response = "HTTP/1.1 200 OK\r\nContent-Type:
                        text/plain\r\n\r\nFile uploaded as " +
                        uploadedFilename;
66                  out.write(response.getBytes());
67
68              } else {
69                  send404(out);
70              }
71
72          } catch (IOException e) {
73              e.printStackTrace();
74          }
75      }
76
77      private static void send404(OutputStream out) throws IOException {
78          String msg = "HTTP/1.1 404 Not Found\r\nContent-Type:
                text/plain\r\n\r\nFile not found";
79          out.write(msg.getBytes());
80      }
81  }
```

Listing 1: Server Side

5.2 Client Side Implementation

The client provides an interface for the user to upload files to the server and download files from the server using HTTP POST and GET requests respectively. The implementation steps for the client are as follows:

1. Display a menu to the user with options to:
    - Upload a file (HTTP POST)
    - Download a file (HTTP GET)
    - Exit the program

2. For uploading a file:
    - Ask the user to input the name of the file located in the `Clientsdownloads` directory.
    - Construct a URL object pointing to `http://localhost:8080/upload`.
    - Open a `HttpURLConnection` and configure it with method `POST` and `doOutput(true)`.
    - Open a `FileInputStream` to read the file and an `OutputStream` from the connection to write the file to the server.
    - Send the file in chunks, close all streams, and display the server's confirmation response.
3. For downloading a file:
    - Ask the user to enter the name of the file to be downloaded from the server.
    - Construct a URL object in the format `http://localhost:8080/download?filename=<filenam`
    - Open a `HttpURLConnection` and set the method to `GET`.
    - If the server responds with HTTP 200 OK, read the input stream and write the file to the `Clientsdownloads` directory using a `FileOutputStream`.
    - If the server responds with 404, display an appropriate "File not found" message.
4. After each operation, return to the menu and wait for the next user choice.

```

import java.io.*;
import java.net.HttpURLConnection;
import java.net.URI;
import java.net.URL;
import java.util.Scanner;

class Clientside {
    public static final String SERVER_URL = "http://localhost:8080";
    public static final String DOWNLOAD_DIR = "Clientsdownloads";

    public void uploadFile(String filePath) {
        File file = new File(DOWNLOAD_DIR, filePath); // File is expected
            in Clientsdownloads/
        System.out.println("Looking for: " + file.getAbsolutePath());

        if (!file.exists() || !file.isFile()) {
            System.out.println("File doesn't exist.");
            return;
        }

        try {
            URI uri = URI.create(SERVER_URL + "/upload");
            URL url = uri.toURL();
            HttpURLConnection connection = (HttpURLConnection)
                url.openConnection();
```

```
25              connection.setDoOutput(true);
26              connection.setRequestMethod("POST");
27              connection.setRequestProperty("Content-Type",
                    "application/octet-stream");
28              connection.setFixedLengthStreamingMode(file.length());
29
30              try (
31                  OutputStream os = connection.getOutputStream();
32                  FileInputStream fis = new FileInputStream(file)
33              ) {
34                  byte[] buffer = new byte[8192];
35                  int bytesRead;
36                  while ((bytesRead = fis.read(buffer)) != -1) {
37                      os.write(buffer, 0, bytesRead);
38                  }
39              }
40
41              int responseCode = connection.getResponseCode();
42              if (responseCode == HttpURLConnection.HTTP_OK) {
43                  try (BufferedReader br = new BufferedReader(new
                        InputStreamReader(connection.getInputStream()))) {
44                      System.out.println("Upload successful. Server
                            response: " + br.readLine());
45                  }
46              } else {
47                  System.out.println("Upload failed. HTTP " + responseCode);
48              }
49
50          } catch (IOException e) {
51              System.out.println("Error during upload: " + e.getMessage());
52          }
53      }
54
55      public void downloadFile(String filename) {
56          try {
57              URI uri = new URI("http", "localhost:8080", "/download",
                    "filename=" + filename, null);
58              URL url = uri.toURL();
59              HttpURLConnection connection = (HttpURLConnection)
                    url.openConnection();
60              connection.setRequestMethod("GET");
61
62              int responseCode = connection.getResponseCode();
63              if (responseCode == 404) {
64                  System.out.println("Error 404: File not found on
                        server.");
65                  return;
66              } else if (responseCode != 200) {
67                  System.out.println("Download failed. HTTP " +
                        responseCode);
68                  return;
69              }
70
71              new File(DOWNLOAD_DIR).mkdirs(); // Ensure Clientsdownloads
```

```
                exists
            File outFile = new File(DOWNLOAD_DIR, filename);

            try (
                InputStream is = connection.getInputStream();
                FileOutputStream fos = new FileOutputStream(outFile)
            ) {
                byte[] buffer = new byte[8192];
                int bytesRead;
                while ((bytesRead = is.read(buffer)) != -1) {
                    fos.write(buffer, 0, bytesRead);
                }
            }

            System.out.println("Downloaded to: " +
                outFile.getAbsolutePath());

        } catch (Exception e) {
            System.out.println("Error during download: " +
                e.getMessage());
        }
    }
}

public class HTTPClient {
    Clientside cs = new Clientside();
    public static final String DOWNLOAD_DIR = "Clientsdownloads";

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        HTTPClient client = new HTTPClient();
        new File(DOWNLOAD_DIR).mkdirs(); // Ensure download folder exists

        while (true) {
            System.out.println("\n1. Upload File (POST)");
            System.out.println("2. Download File (GET)");
            System.out.println("3. Exit");
            System.out.print("> ");
            String choice = scanner.nextLine().trim();

            switch (choice) {
                case "1":
                    System.out.print("Enter file name to upload from
                        Clientsdownloads/: ");
                    client.cs.uploadFile(scanner.nextLine().trim());
                    break;
                case "2":
                    System.out.print("Enter filename to download: ");
                    client.cs.downloadFile(scanner.nextLine().trim());
                    break;
                case "3":
                    System.out.println("Exiting...");
                    return;
                default:
```

```
                        System.out.println("Invalid choice.");
                }
            }
        }
}
```

Listing 2: Client Side.

6 Result Analysis

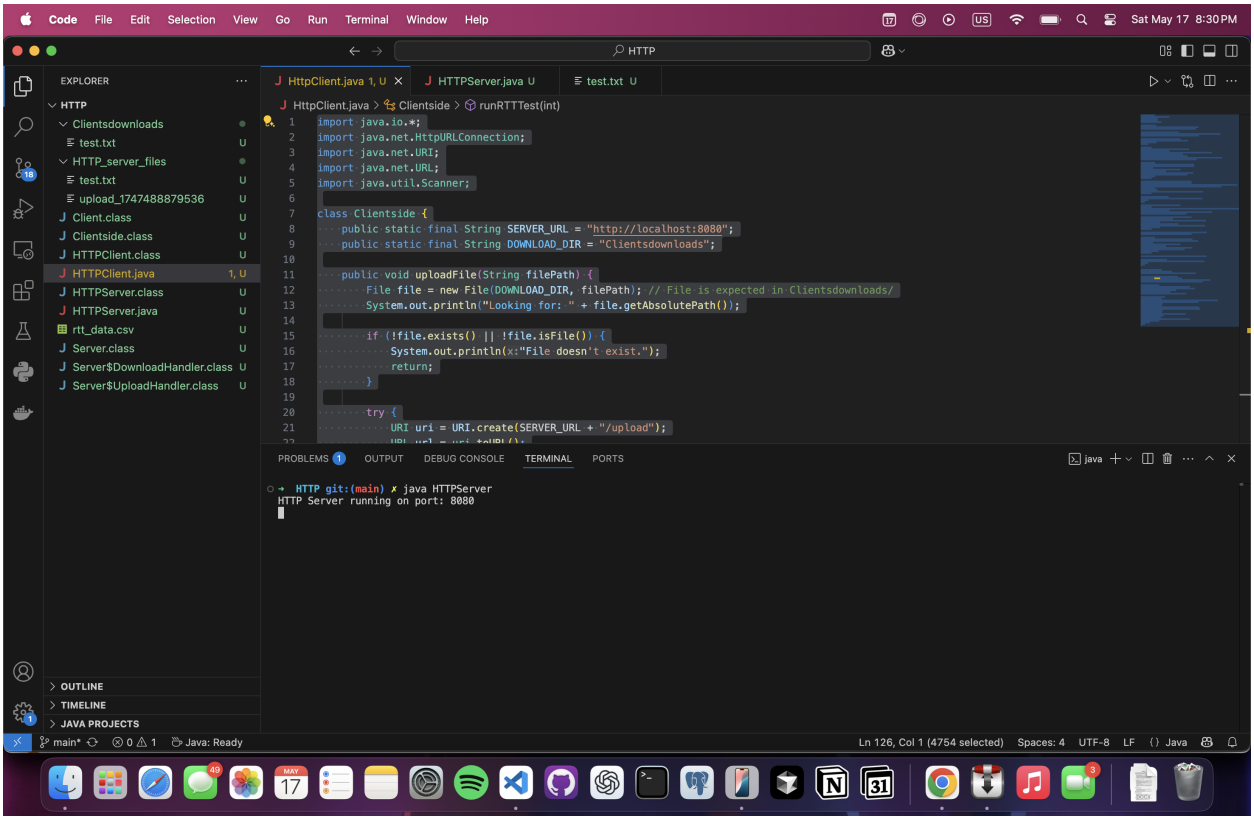

Figure 5: Server Running on Port 8080.

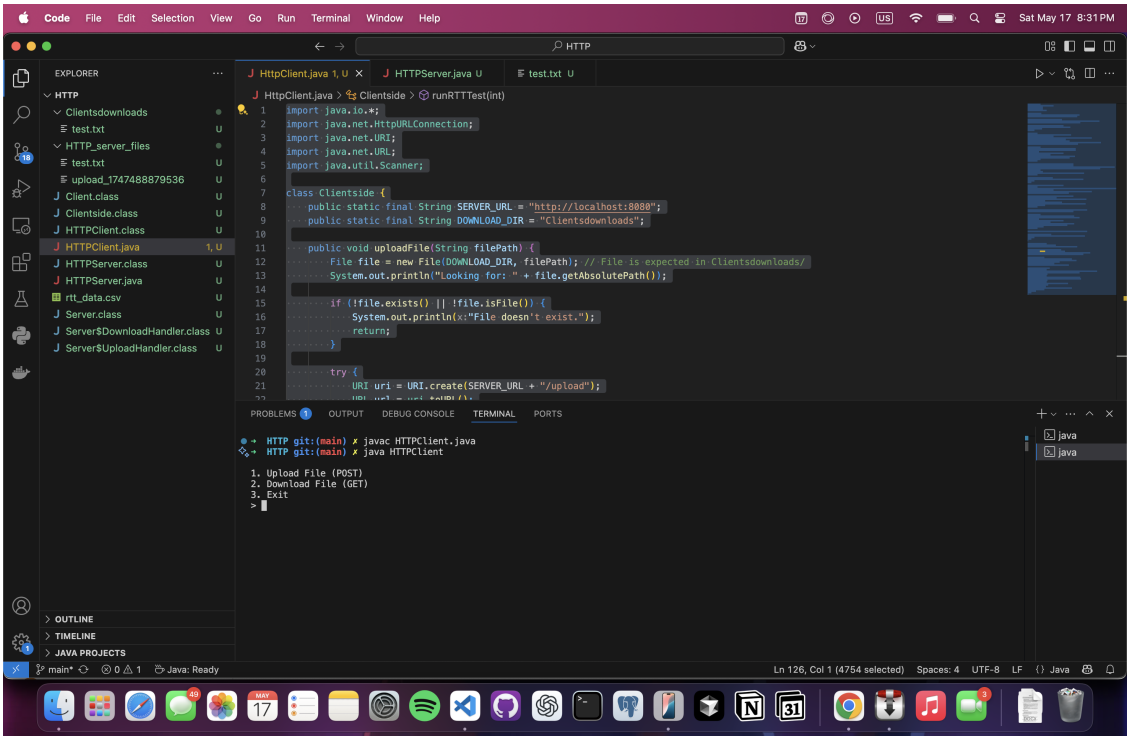

Figure 6: Client connected and can send multiple messages.

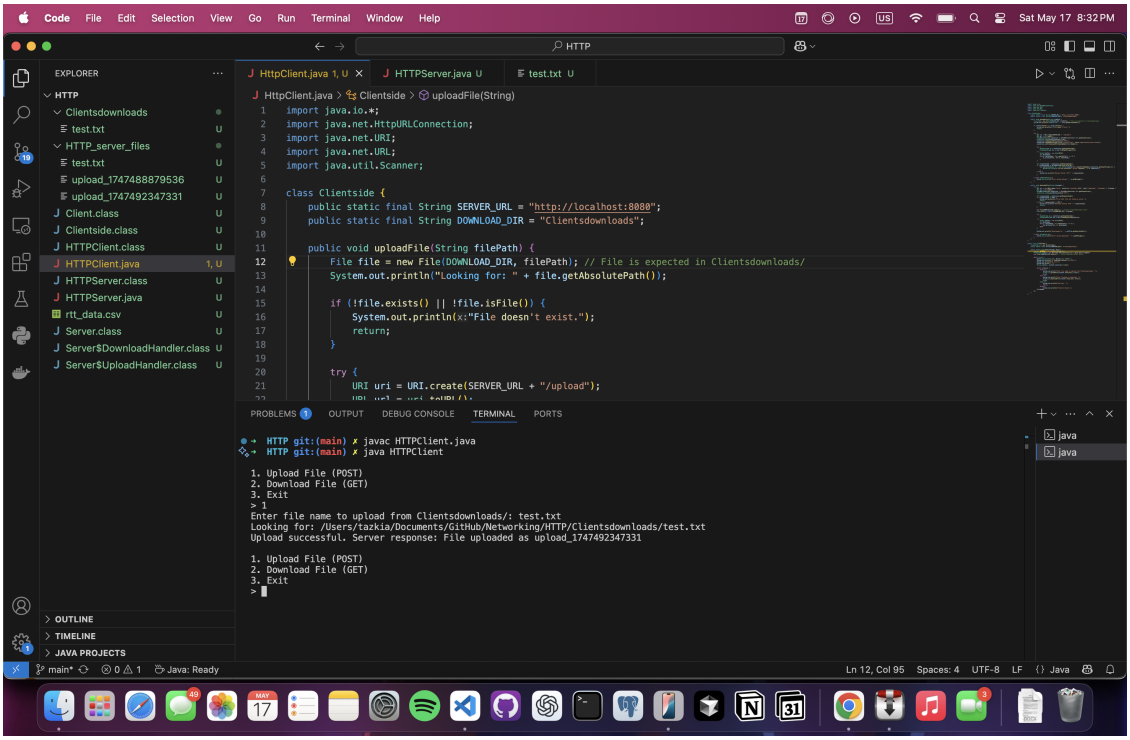

Figure 7: Client connected and can upload files to server(POST).

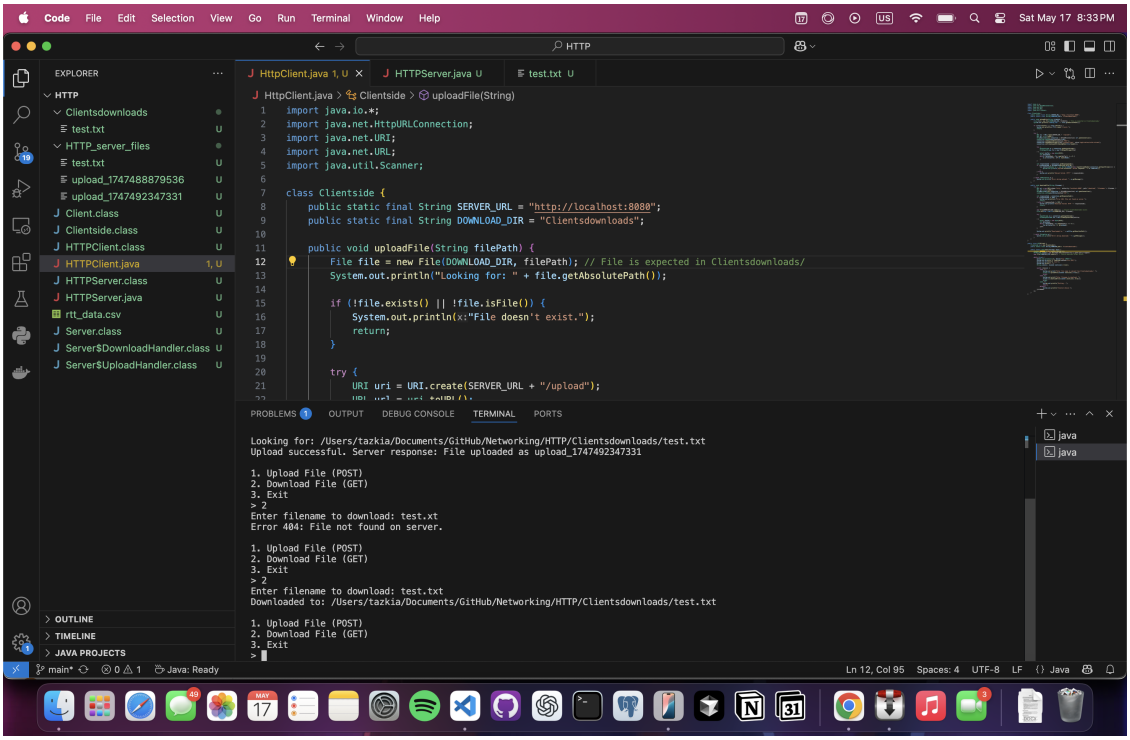

Figure 8: Client connected and can download files to server(GET)

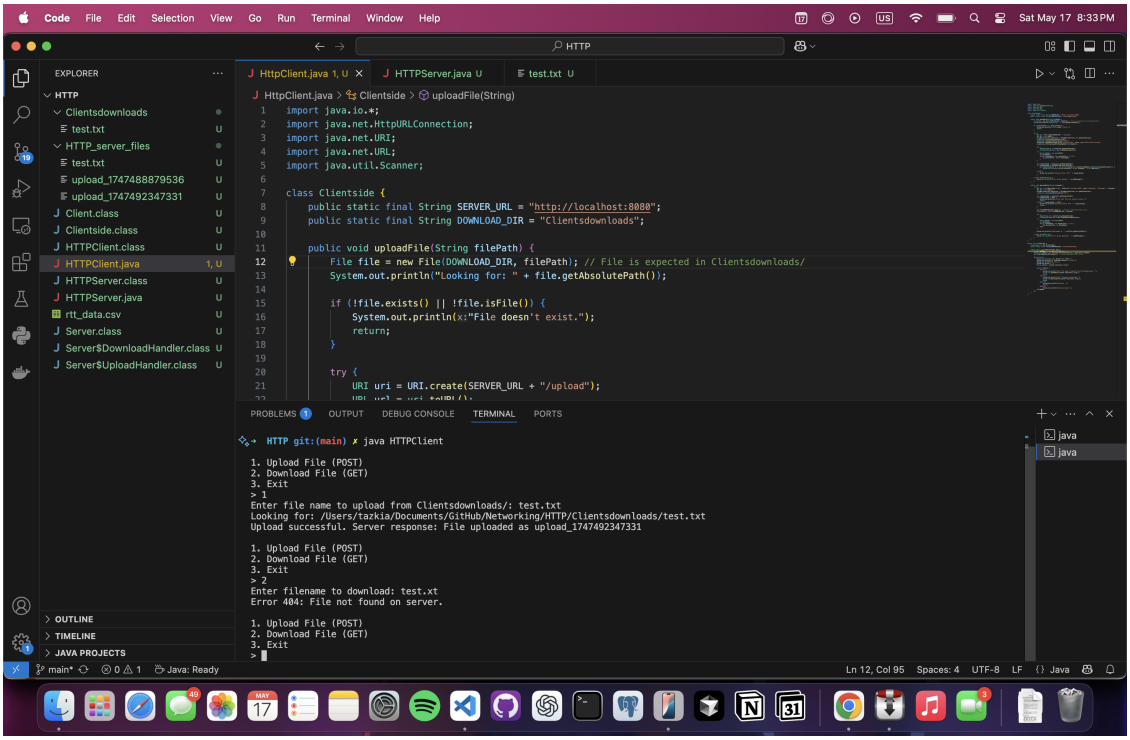

Figure 9: Error 404 when the file is not in server side

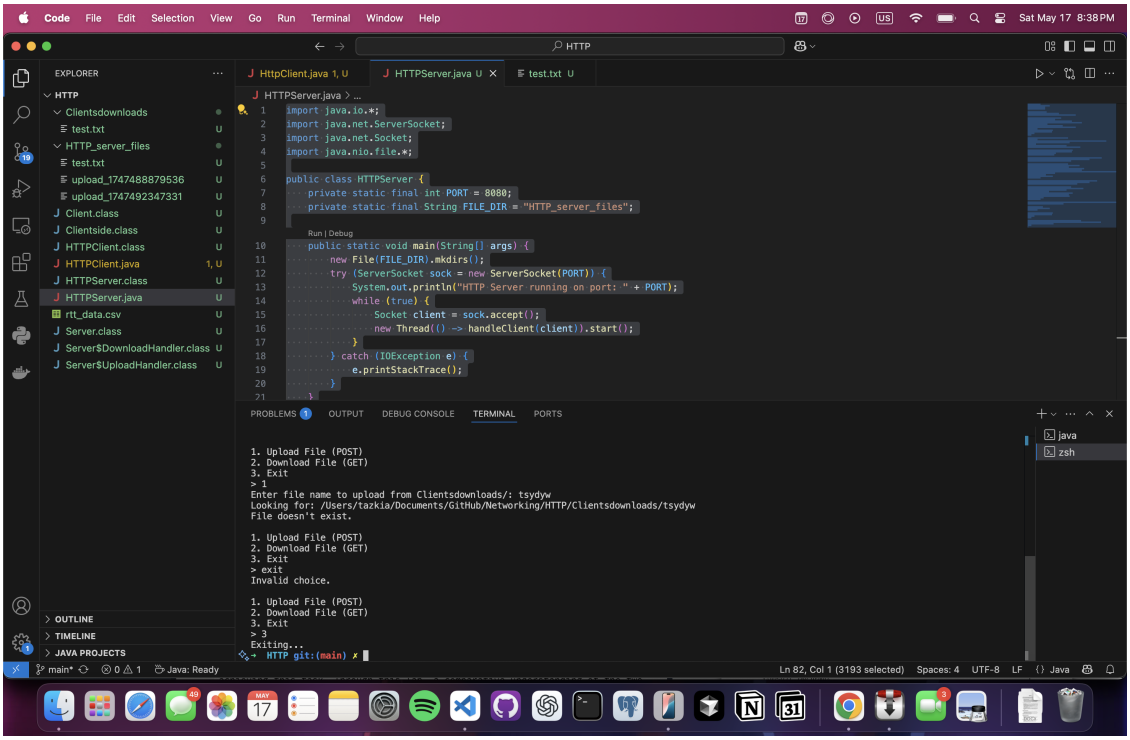

Figure 10: "exit" closes the connection for client.

7 Discussion

File transfer is a fundamental operation in networked applications, and both Socket Programming and HTTP-based requests offer distinct approaches to achieving this task. Through this lab, a comparative understanding of the two methods was developed.

Socket programming provides low-level access to network communication, allowing developers to define the protocol, manage data streams, and directly control the transmission process. This approach is powerful and flexible but often complex and error-prone. It requires manual handling of connection establishment, data framing, error checking, and termination. While suitable for custom protocols or real-time systems, it is less accessible for general-purpose file transfer applications.

In contrast, HTTP GET and POST methods offer a high-level abstraction built on top of the TCP/IP stack. HTTP simplifies the communication process through a stateless, request-response model that is widely supported and easy to integrate across web platforms. By using HTTP:

- File upload and download become platform-independent.
- Security features (such as HTTPS) are easily implementable.
- Development time is reduced due to existing libraries and tools.
- Maintenance and debugging are simplified due to standardized behavior.

From this lab, the key learning outcome was understanding how HTTP enables structured and interoperable file transfer without the overhead of managing low-level socket details. Implementing both GET and POST methods helped reinforce the concepts of client-server communication and the stateless nature of HTTP.

- **Handling multipart/form-data:** Properly formatting POST requests for file uploads required a clear understanding of multipart encoding and appropriate header configurations.
- **File path and permission errors:** Ensuring correct server-side paths and managing read/write permissions was essential to avoid runtime errors during upload and download operations.
- **Debugging HTTP requests and responses:** Interpreting status codes, response headers, and payloads was initially difficult without using browser developer tools or debugging utilities like Postman.
- **Maintaining consistency across platforms:** Testing behavior across different browsers and HTTP clients revealed minor inconsistencies, which required adjustments in request handling logic.

8 Conclusion

The lab successfully demonstrated the implementation of file transfer using HTTP GET and POST requests within a client-server model. Through this exercise, a practical understanding of how HTTP facilitates structured communication for file upload and download was developed. Compared to low-level socket programming, HTTP offers a more accessible and standardized approach, especially suited for web-based applications.

The experience provided valuable insights into HTTP protocol behavior, request/response structure, and server-side file handling. Despite facing several technical challenges, the hands-on implementation helped reinforce theoretical concepts and highlighted the importance of protocol-level decisions in application design. Overall, this lab emphasized the practicality and effectiveness of using HTTP for reliable file transfer in modern networked environments.