University of Dhaka

Department of Computer Science and Engineering

CSE3111: Computer Networking Lab

Design and Implementation of
File Transfer Application
Using HTTP GET/POST Requests

Kabya Mithun Saha (16)
Muhaiminul Islam Ninad (43)

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

Submission Date: May 19, 2025

Contents

1 Introduction

File transfer is a fundamental networking operation that enables the movement of data between computer systems. In today's interconnected world, efficient and reliable file transfer mechanisms are essential for various applications, ranging from simple data sharing to complex distributed systems. The Hypertext Transfer Protocol (HTTP) has emerged as one of the most widely used protocols for file transfer due to its simplicity, flexibility, and compatibility with web infrastructure.

This lab report explores the implementation of a file transfer system using HTTP GET and POST requests. The system consists of a server component that can handle multiple client connections simultaneously and a client application that allows users to upload files to the server and download files from it. The implementation leverages the HTTP protocol's request-response model, where GET requests are used to download files from the server, and POST requests are used to upload files to the server.

Our implementation uses Java's built-in HTTP server capabilities to create a lightweight, yet powerful file transfer system. The lab focuses on understanding the core concepts of HTTP-based file transfer, including connection establishment, request processing, file transmission, and response handling. By implementing both client and server components, we gain practical insights into network programming and the inner workings of HTTP-based file transfers.

2 Objectives

The primary objectives of this lab assignment are:

1. To design and implement a robust file transfer system using HTTP GET and POST requests, enabling users to upload files to a server and download files from it.

2. To develop a server component capable of handling multiple client connections simultaneously, ensuring efficient and reliable file transfer operations.

3. To compare the HTTP-based file transfer approach with traditional socket programming techniques, analyzing the advantages, challenges, and practical applications of each method.

3 Design Details

The file transfer system consists of two main components: a server that handles file storage and retrieval, and a client that interacts with the server to upload and download files. Both components are implemented in Java, leveraging the language's built-in networking capabilities.

3.1 System Architecture

The system follows a client-server architecture, where multiple clients can connect to a single server to perform file operations. The communication between clients and servers is based on the HTTP protocol, which provides a standardized way of exchanging requests and responses.

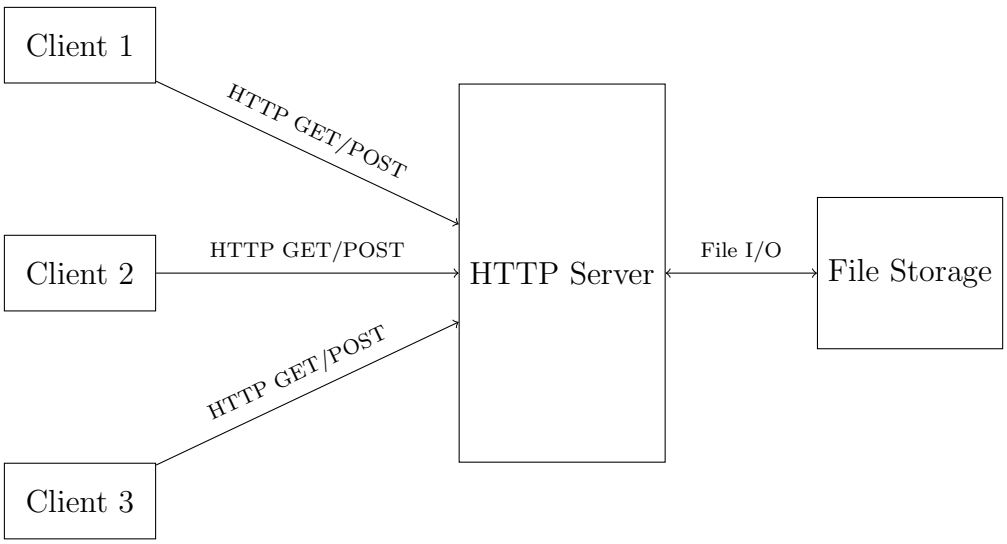

Figure 1: High-level architecture of the HTTP-based file transfer system

3.2 Server Design

The server component is implemented using Java's HttpServer class, which provides a simple framework for handling HTTP requests. The server listens on a specified port (8080 by default) and creates three contexts to handle different types of requests:

- **/download** - Handles GET requests for downloading files from the server
- **/upload** - Handles POST requests for uploading files to the server

- **/list** - Handles GET requests for listing available files on the server

Each context is associated with a handler that processes the specific type of request. The server uses a thread pool to handle multiple client connections simultaneously, ensuring that one slow client doesn't block others from being served.

3.3 Client Design

The client component provides a simple command-line interface that allows users to:

- Upload files to the server
- Download files from the server
- List available files on the server

The client uses Java's HttpURLConnection class to establish connections to the server and send HTTP requests. For file uploads, the client reads the file from the local file system and sends it as the body of a POST request. For file downloads, the client sends a GET request and saves the response body to the local file system.

3.4 Flow Chart of File Transfer Process

3.4.1 Upload Flow

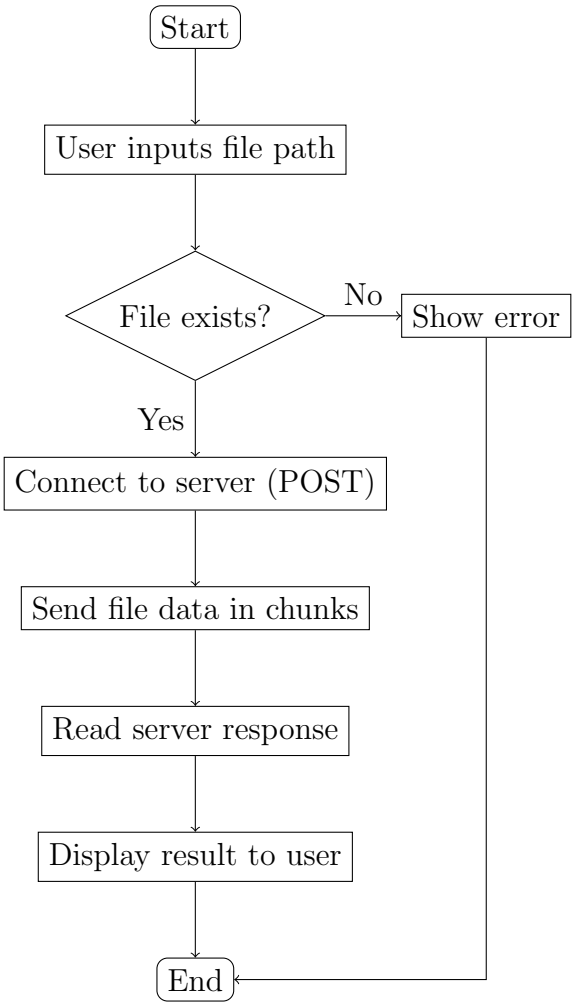

Figure 2: Flow chart for file upload process

3.4.2 Download Flow

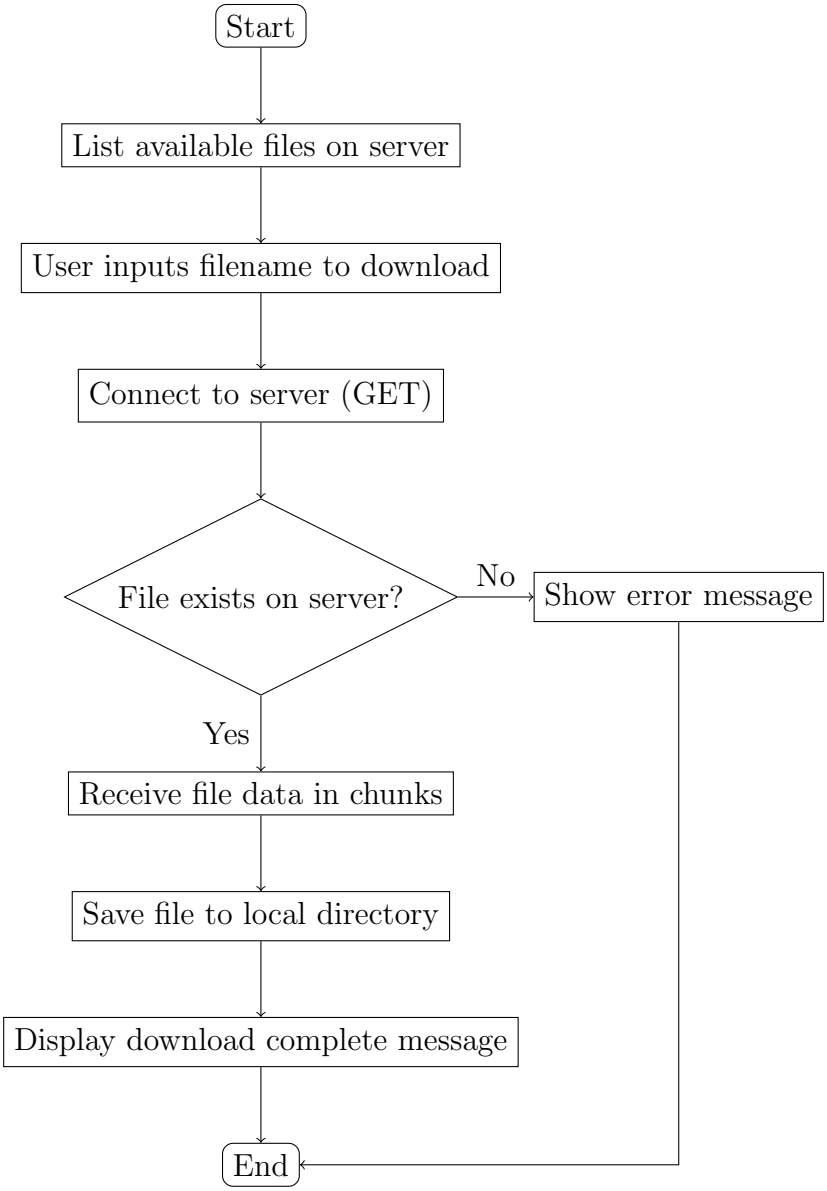

Figure 3: Flow chart for file download process

4 Implementation

This section presents the implementation details of the HTTP-based file transfer system. The implementation consists of two main components: the server (`SimpleHttpServer.java`) and the client (`HttpFileClient.java`).

4.1 Server Implementation

The server is implemented using Java's `HttpServer` class, which provides a simple way to create an HTTP server that can handle client requests. The server creates three contexts to handle different types of requests: file download, file upload, and file listing.

The core classes and methods in the server implementation are:

- `SimpleHttpServer` - The main class that sets up the HTTP server and creates the request handlers
- `DownloadHandler` - Handles GET requests for downloading files
- `UploadHandler` - Handles POST requests for uploading files
- `ListFilesHandler` - Handles GET requests for listing available files

Each handler implements the `HttpHandler` interface and overrides the `handle` method to process incoming requests. The handlers use Java's file I/O capabilities to read and write files as needed.

4.2 Client Implementation

The client is implemented using Java's `HttpURLConnection` class, which provides a simple way to establish HTTP connections and send requests to the server. The client provides a simple command-line interface that allows users to upload files, download files, and list available files on the server.

The core methods in the client implementation are:

- `main` - The entry point of the application that handles user input and calls appropriate methods
- `uploadFile` - Uploads a file to the server using a POST request
- `downloadFile` - Downloads a file from the server using a GET request
- `listFiles` - Lists available files on the server using a GET request

These methods use Java's file I/O capabilities to read and write files as needed, and they handle various error conditions that may arise during the file transfer process.

5 Result Analysis

This section presents the results of testing the HTTP-based file transfer system, including screenshots of the client and server in action.

5.1 Server Startup

When the server is started, it initializes the file directory and begins listening for incoming connections on port 8080. The server provides feedback about its status, including the port it's listening on and the directory it's using for file storage.

```
Server is running on port 8080
File directory:  ./files
Available files:  [example.txt, image.jpg]
```

Figure 4: Server startup console output

5.2 File Upload

When a client uploads a file to the server, both the client and server provide feedback about the progress and status of the upload operation.

5.2.1 Client-side Output

```
=== File Transfer Client ===
1.  Upload file
2.  Download file
3.  List available files
4.  Exit
Enter your choice:  1
Enter the path of the file to upload:
/path/to/document.pdf
Uploading file:  document.pdf
Uploading:  100% complete
Upload completed!
Server response:  File uploaded successfully:
document.pdf
```

Figure 5: Client console output during file upload

5.2.2 Server-side Output

```
Upload request for file:  document.pdf
File uploaded successfully:  document.pdf (1245678
bytes)
Available files:  [document.pdf, example.txt,
image.jpg]
```

Figure 6: Server console output during file upload

5.3 File Listing

When a client requests a list of available files, both the client and server provide feedback about the operation.

5.3.1 Client-side Output

```
=== File Transfer Client ===
1.  Upload file
2.  Download file
3.  List available files
4.  Exit
Enter your choice:  3
Fetching list of available files...
Available files:
- document.pdf
- example.txt
- image.jpg
```

Figure 7: Client console output during file listing

5.3.2 Server-side Output

```
Available files:  [document.pdf, example.txt,
image.jpg]
```

Figure 8: Server console output during file listing

5.4 File Download

When a client downloads a file from the server, both the client and server provide feedback about the progress and status of the download operation.

5.4.1 Client-side Output

```
=== File Transfer Client ===
1. Upload file
2. Download file
3. List available files
4. Exit
Enter your choice: 2
Fetching list of available files...
Available files:
- document.pdf
- example.txt
- image.jpg
Enter the name of the file to download:
document.pdf
Downloading file: document.pdf
Downloading: 100% complete
Download completed! File saved to:
./client/document.pdf
```

Figure 9: Client console output during file download

5.4.2 Server-side Output

```
Download request for file: document.pdf
File sent successfully: document.pdf
```

Figure 10: Server console output during file download

6 Discussion

In this lab, we implemented a file transfer system using HTTP GET and POST requests, which provided valuable insights into the workings of the HTTP protocol and its application in file transfer scenarios. This section

compares our HTTP-based approach with the traditional socket programming approach, highlights key learnings, and discusses challenges encountered during the implementation.

6.1 Comparison of HTTP vs. Socket Programming for File Transfer

6.1.1 Benefits of HTTP-based File Transfer

1. **Standardized Protocol**: HTTP is a well-established, standardized protocol with clear semantics for different operations. This standardization makes the implementation more straightforward and compatible with existing infrastructure.

2. **Built-in Error Handling**: HTTP includes built-in status codes (e.g., 200 OK, 404 Not Found) that simplify error handling and provide clear feedback to clients about the status of their requests.

3. **Higher-level Abstraction**: HTTP operates at a higher level of abstraction than raw sockets, which reduces the amount of low-level code required for tasks like connection management and data framing.

4. **Firewall Friendliness**: HTTP traffic typically uses port 80 or 443, which are commonly allowed through firewalls. This makes HTTP-based file transfer more reliable in restricted network environments.

5. **Statelessness**: The stateless nature of HTTP simplifies the server implementation, as each request-response pair is independent. This makes it easier to handle multiple clients and recover from errors.

6. **Easy Integration with Web Applications**: HTTP-based file transfer can be seamlessly integrated with web applications, enabling features like browser-based uploads and downloads.

7. **Concurrent Connections**: Modern HTTP servers are designed to handle multiple concurrent connections efficiently, which simplifies the implementation of a multi-client file transfer system.

6.1.2 Challenges and Limitations of HTTP-based File Transfer

1. **Overhead**: HTTP includes headers and other metadata that increase the overhead compared to raw socket transfers, potentially reducing efficiency for small files or frequent transfers.

2. **Connection Setup/Teardown**: For each file transfer, HTTP typically establishes a new connection, which adds latency compared to keeping a persistent socket connection open.

3. **Limited Bidirectional Communication**: Traditional HTTP (pre-WebSockets) is primarily designed for request-response patterns, making it less suitable for real-time bidirectional communication.

4. **Less Control over Low-level Details**: While the higher-level abstraction simplifies development, it also means less control over low-level network details, which might be important for some specialized applications.

6.2 Learning Outcomes

Through this lab assignment, we learned:

1. How to implement a basic HTTP server and client in Java for file transfer operations

2. The mechanics of HTTP GET and POST requests, including how to set headers, read request bodies, and write response bodies

3. How to handle binary file data in HTTP requests and responses

4. Techniques for monitoring and reporting file transfer progress to users

5. Error handling strategies for networking and file I/O operations

6. How to manage multiple concurrent client connections using thread pools

7. The importance of proper resource management, including closing streams and connections

6.3 Challenges Faced

During the implementation, we encountered several challenges:

1. **Handling Large Files**: Efficiently transferring large files required careful buffer management to avoid excessive memory usage and ensure smooth progress reporting.

2. **Character Encoding**: Ensuring proper encoding of filenames in URLs and responses, especially when dealing with special characters.

3. **Concurrent Access**: Managing concurrent access to shared resources, such as the file list and the filesystem.

4. **Error Propagation**: Ensuring that error messages were properly propagated from the server to the client and presented in a user-friendly manner.

5. **Progress Monitoring**: Implementing progress monitoring for file transfers, especially when the total file size was unknown in advance.

6. **Cross-platform Compatibility**: Ensuring that the file paths and separators worked correctly across different operating systems.

6.4 Future Improvements

Based on our experience, several improvements could be made to the system:

1. **Authentication and Authorization**: Add user authentication and file access control to secure the file transfer operations.

2. **HTTPS Support**: Implement HTTPS to encrypt the file transfers and protect sensitive data.

3. **Resumable Transfers**: Support for resuming interrupted file transfers, which would be particularly useful for large files.

4. **File Metadata**: Include file metadata (e.g., creation date, size, type) in the file listing.

5. **Directory Support**: Allow for hierarchical file organization with directories.

6. **File Overwrite Confirmation**: Add confirmations for file overwrite operations to prevent accidental data loss.

7. **Client-side GUI**: Develop a graphical user interface for the client to improve usability.

7 Conclusion

This lab assignment provided valuable hands-on experience with HTTP-based file transfer, highlighting both the advantages and challenges of this

approach compared to traditional socket programming. We successfully implemented a functional file transfer system that allows multiple clients to upload and download files concurrently.

The HTTP-based approach offers several advantages, including standardization, built-in error handling, and firewall friendliness, making it a good choice for many file transfer scenarios, especially those integrated with web applications. However, it also has limitations, such as higher overhead and less control over low-level details, which might make socket programming more suitable for certain specialized applications.

Overall, this lab deepened our understanding of network programming concepts and provided practical insights into the design and implementation of file transfer systems. The knowledge and skills gained from this experience will be valuable for future networking projects and real-world applications.