University of Dhaka

Department of Computer Science and Engineering

CSE-3111: Computer Networking Lab

File Transfer Implementation: HTTP vs Socket Programming

Group Members:

Jannatul Ferdousi (08)
Anirban Roy Sourav (32)

Submitted To:

1. Mr. Palash Roy, Lecturer, Dept. of CSE, DU
2. Mr. Jargis Ahmed, Lecturer, Dept. of CSE, DU
3. Dr. Ismat Rahman, Associate Professor, Dept. of CSE, DU

Submission Date: May 19, 2025

Contents

1 Introduction

File transfer is a fundamental aspect of computer networking, enabling users to share data across systems regardless of physical distance. In this lab experiment, we explore two primary approaches to implementing file transfer functionality: HTTP-based transfer using GET/POST requests and traditional Socket Programming.
HTTP (Hypertext Transfer Protocol) provides a standardized way to transfer files through well-defined request and response mechanisms. It operates at the application layer of the TCP/IP model and is widely used for web-based file sharing and client-server communications. HTTP-based file transfers leverage the robustness of the HTTP protocol, which includes features like authentication, content negotiation, and caching.
Socket programming, on the other hand, offers a lower-level approach by creating direct communication channels between applications across a network. It provides fine-grained control over the data transfer process and can be optimized for specific use cases. Socket programming is particularly useful when building custom file transfer protocols tailored to specific requirements.

2 Objectives

The primary objectives of this lab experiment are:

1. **To design and implement a file transfer system using HTTP GET/POST requests** that enables secure and efficient file sharing between client and server.
2. **To compare HTTP-based file transfer with socket programming approaches** in terms of implementation complexity, performance, and reliability.
3. **To develop a comprehensive understanding of client-server architecture** for file transfer operations in networked environments.

3 Design Details

3.1 System Architecture

Our HTTP-based file transfer system follows a client-server architecture that facilitates file uploads, downloads, and listings over a secure, multi-threaded HTTP server.

- **FileClient.java:** A Java-based CLI client that allows uploading, downloading, and listing files.
- **FileServer.java:** A multi-threaded Java HTTP server that handles file uploads, downloads, and listings securely.

3.2 User Interface

The client offers a command-line interface (CLI) that allows users to upload, download, and list files available on the server.

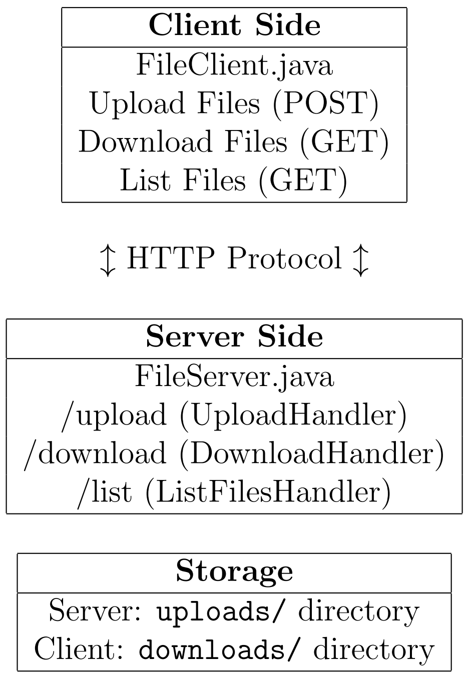


Figure 1: System Architecture Diagram

3.3 FileClient Features

- **Upload File:** Sends local files to the server via HTTP POST with real-time progress tracking. SHA-256 hashes are used to avoid uploading duplicate files.
- **Download File:** Retrieves files from the server via HTTP GET. Files are stored uniquely in the `downloads/` directory with progress indication.
- **List Files:** Queries the server for a list of available files, displaying file names and sizes.
- **Duplicate Prevention:** Uses SHA-256 hash checking to avoid redundant uploads.
- **Progress Reporting:** Provides real-time progress updates during file transfers to enhance user experience.

3.4 Server Design and Features

3.4.1 HTTP Server Initialization

The server initializes an HTTP server on port 8080 and registers three main context handlers:

- `/upload`- Handles file uploads via POST.
- `/download`- Serves file downloads via GET.
- `/list`- Provides a file listing via GET.

3.4.2 FileServer Features

- **Upload Handler:**
  - Accepts HTTP POST requests with file payloads.
  - Rejects files exceeding 20MB to prevent DoS attacks.
  - Checks file hash (SHA-256) to detect duplicates.
  - Renames files automatically if name conflicts occur.
- **Download Handler:**
  - Serves requested files with MIME type detection.
  - Sets HTTP headers to ensure secure content delivery.
- **List Files Handler:**
  - Returns a list of filenames and sizes in the `uploads/` directory.
- **Thread Pool Management:**
  - Uses a fixed thread pool to manage multiple simultaneous requests efficiently.
- **Security Headers:**
  - `X-Content-Type-Options: nosniff`
  - `X-Frame-Options: DENY`
  - `X-XSS-Protection: 1; mode=block`

3.5 Security Considerations

The system incorporates multiple layers of security to ensure robustness:

- **Input Sanitization:** All filenames are sanitized to prevent directory traversal attacks.
- **Hash Verification:** SHA-256 hashing is used to prevent duplicate uploads.
- **Upload Size Restriction:** A 20MB file size limit mitigates potential denial-of-service (DoS) attacks.
- **HTTP Security Headers:** Proper HTTP headers are configured to prevent content sniffing, clickjacking, and cross-site scripting (XSS).
- **CORS (Cross-Origin Resource Sharing):** Controlled CORS headers improve browser safety and restrict unauthorized cross-domain access.

3.6 Flow Chart

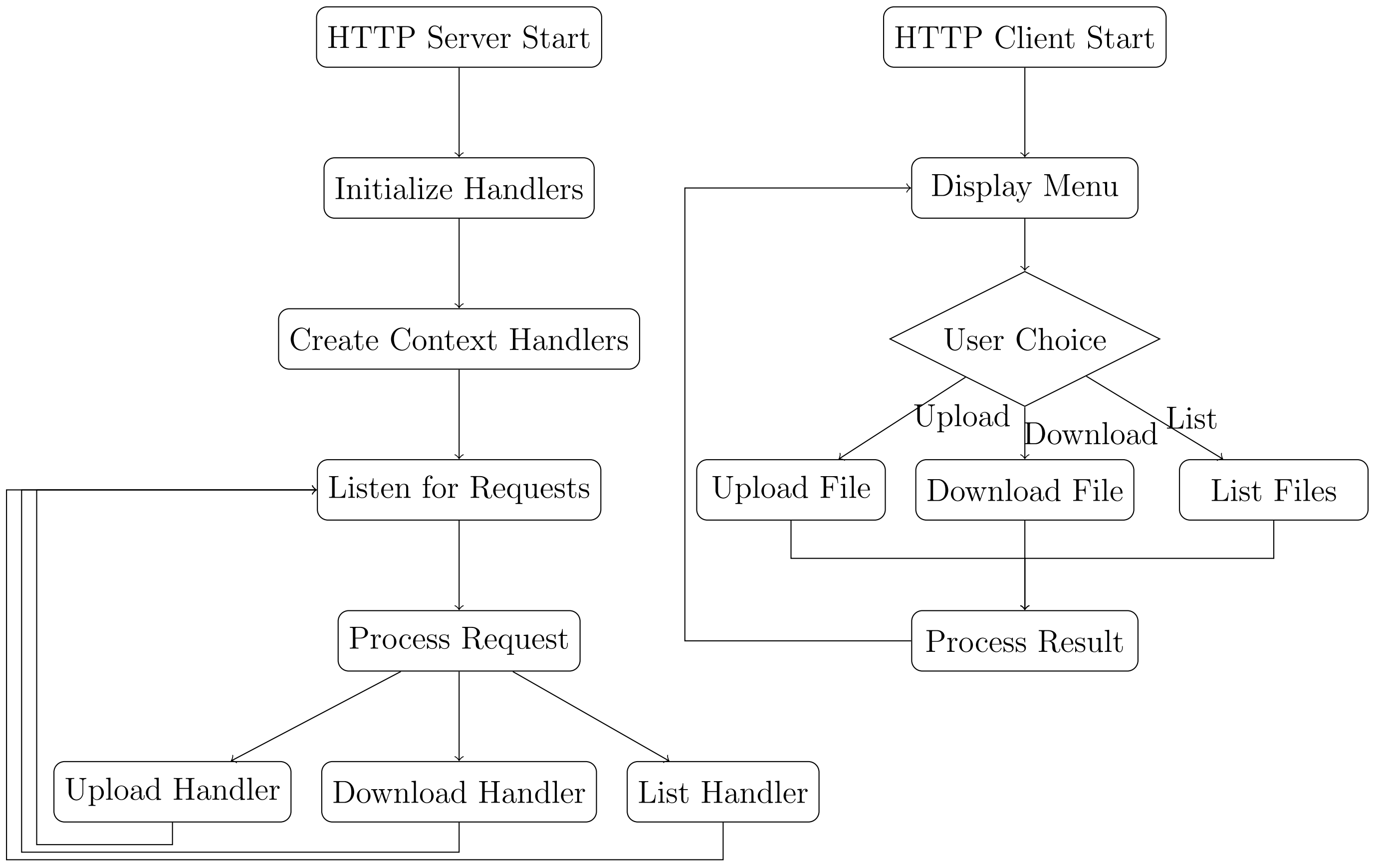


Figure 2: Flow Chart of HTTP-Based File Transfer System

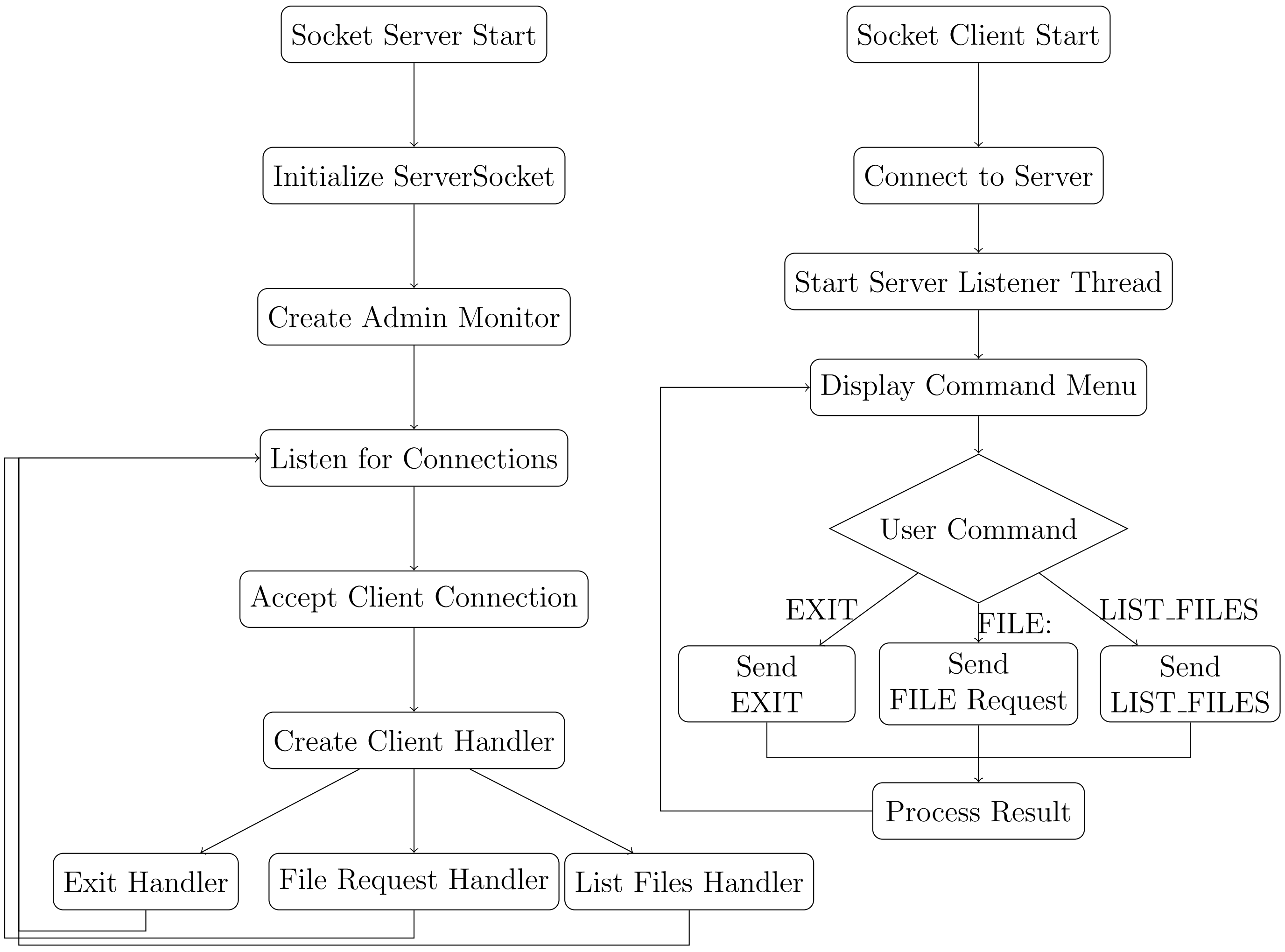


Figure 3: Flow Chart of Socket-Based File Transfer System

4 Implementation

This section presents the implementation of our HTTP-based file transfer system. The implementation consists of server-side code handling HTTP requests and client-side code for interacting with the server.

4.1 Server Implementation

The FileServer class implements an HTTP server using Java's com.sun.net.httpserver package. It creates an instance of HttpServer on port 8080 and registers three context handlers for different endpoints:

- UploadHandler: Processes file uploads via HTTP POST
- DownloadHandler: Serves files via HTTP GET
- ListFilesHandler: Lists available files on the server

Below is a key excerpt from the server implementation:

Listing 1: HTTP File Server Main Method

```java
public static void main(String[] args) throws IOException {
    if (!UPLOAD_DIR.exists())
        UPLOAD_DIR.mkdirs();
    HttpServer server = HttpServer.create(new
       InetSocketAddress(PORT), 0);
    server.createContext("/upload", new UploadHandler());
    server.createContext("/download", new DownloadHandler());
    server.createContext("/list", new ListFilesHandler());
    server.setExecutor(Executors.newFixedThreadPool(10));
    server.start();
    System.out.println("Secure file server started on port " +
       PORT);
    Runtime.getRuntime().addShutdownHook(new Thread(() -> {
        server.stop(1);
        System.out.println("Server stopped.");
    }));
}
```

The UploadHandler is responsible for handling file uploads. It checks the HTTP method, enforces a maximum upload size, generates a unique filename if needed, and verifies file integrity using SHA-256 hashing:

Listing 2: Upload Handler Implementation

```java
static class UploadHandler implements HttpHandler {
    @Override
    public void handle(HttpExchange exchange) throws IOException
       {
        if
           (!"POST".equalsIgnoreCase(exchange.getRequestMethod()))
           {
            sendResponse(exchange, 405, "Method Not Allowed");
            return;
        }

        // Enforce max upload size
        InputStream is = exchange.getRequestBody();
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        byte[] buffer = new byte[4096];
        int total = 0;
        int bytesRead;
        while ((bytesRead = is.read(buffer)) != -1) {
            total += bytesRead;
            if (total > MAX_UPLOAD_SIZE) {
                sendResponse(exchange, 413, "Payload Too Large");
                return;
            }
            baos.write(buffer, 0, bytesRead);
        }

        byte[] uploadedData = baos.toByteArray();

        // Determine filename
        String filename
           =exchange.getRequestHeaders().getFirst("X-Filename")
           != null
            ? exchange.getRequestHeaders().getFirst("X-Filename")
            : parseQuery(exchange.getRequestURI().getRawQuery())
            .get("filename");
        if (filename == null || filename.isBlank()) {
            filename = "upload_" + new SimpleDateFormat("yyyyMMdd_HHmmss").format(new Date()) + ".dat";
        } else {
            if (filename.matches(".*[<>:\"|?*].*")) {
                sendResponse(exchange, 400, "Invalid characters in filename");
                return;
            }
        }
        filename = filename.replaceAll("[\\\\/]", "_");

        File uploadedFile = new File(UPLOAD_DIR, filename);
        String uploadedHash = computeSHA256(new ByteArrayInputStream(uploadedData));

        if (uploadedFile.exists()) {
            String existingHash = computeSHA256(new FileInputStream(uploadedFile));
            if (uploadedHash.equals(existingHash)) {
                sendResponse(exchange, 200, "Duplicate file detected. Upload skipped.");
                return;
            } else {
                String baseName = filename;
                String extension = "";
                int dotIndex = filename.lastIndexOf('.');
                if (dotIndex != -1) {
                    baseName = filename.substring(0, dotIndex);
                    extension = filename.substring(dotIndex);
                }

                int count = 1;
                while (uploadedFile.exists()) {
                    String newName = baseName + "_" + count + extension;
                    uploadedFile = new File(UPLOAD_DIR, newName);
                    count++;
                }
            }
        }

        try (FileOutputStream fos = new FileOutputStream(uploadedFile)) {
            fos.write(uploadedData);
        }

        sendResponse(exchange, 200, "Upload successful: " + uploadedFile.getName());
    }
}
```

The DownloadHandler handles file download requests, checking for valid filenames and streaming the requested file to the client:

Listing 3: Download Handler Implementation

```
static class DownloadHandler implements HttpHandler {
    @Override
    public void handle(HttpExchange exchange) throws IOException
        {
        if
            (!"GET".equalsIgnoreCase(exchange.getRequestMethod()))
            {
            sendResponse(exchange, 405, "Method Not Allowed");
            return;
        }

        Map<String, String> params =
            parseQuery(exchange.getRequestURI().getRawQuery());
        String filename = params.get("filename");
        if (filename == null || filename.contains("..") ||
            filename.contains("/") || filename.contains("\\")) {
            sendResponse(exchange, 400, "Invalid filename");
            return;
        }

        File file = new File(UPLOAD_DIR, filename);
        if (!file.exists() || file.isDirectory()) {
            sendResponse(exchange, 404, "File Not Found");
            return;
        }

        String mime =
            java.nio.file.Files.probeContentType(file.toPath());
        if (mime == null)
            mime = "application/octet-stream";
        exchange.getResponseHeaders().add("Content-Type", mime);
        exchange.getResponseHeaders().add("Content-Disposition",
            "attachment; filename=\"" + file.getName() + "\"");
        exchange.getResponseHeaders().add
        ("Access-Control-Allow-Origin", "*");
        exchange.sendResponseHeaders(200, file.length());

        try (OutputStream os = exchange.getResponseBody();
            FileInputStream fis = new FileInputStream(file)) {
            fis.transferTo(os);
        }
    }
}
```

The ListFilesHandler provides a directory listing of available files on the server:

Listing 4: List Files Handler Implementation

```
static class ListFilesHandler implements HttpHandler {
    @Override
    public void handle(HttpExchange exchange) throws IOException
        {
```

```java
        if
            (!"GET".equalsIgnoreCase(exchange.getRequestMethod()))
            {
             exchange.sendResponseHeaders(405, -1);
             return;
        }

        File[] files = UPLOAD_DIR.listFiles();
        if (files == null) {
            exchange.sendResponseHeaders(500, -1);
            return;
        }

        StringBuilder response = new StringBuilder();
        for (File file : files) {
            if (file.isFile()) {
                response.append(file.getName()).append("
                    ("+formatFileSize(file.length())+")").
                append("\n");
            }
        }

        byte[] responseBytes =
            response.toString().getBytes(StandardCharsets.UTF_8);
        exchange.getResponseHeaders().set("Content-Type",
            "text/plain; charset=utf-8");
        exchange.sendResponseHeaders(200, responseBytes.length);
        try (OutputStream os = exchange.getResponseBody()) {
            os.write(responseBytes);
        }
    }
}
```

The server implementation also includes several utility methods for common tasks:

Listing 5: Server Utility Methods

```java
// Format file size in human-readable form
private static String formatFileSize(long size) {
    if (size < 1024) return size + " bytes";
    else if (size < 1024 * 1024) return String.format("%.2f KB",
        size / 1024.0);
    else if (size < 1024 * 1024 * 1024) return
        String.format("%.2f MB", size / (1024.0 * 1024));
    else return String.format("%.2f GB", size / (1024.0 * 1024 *
        1024));
}

// Send HTTP response with appropriate headers
private static void sendResponse(HttpExchange exchange, int
    code, String message) throws IOException {
    byte[] response = message.getBytes(StandardCharsets.UTF_8);
    exchange.getResponseHeaders().add("Content-Type",
        "text/plain");
    exchange.getResponseHeaders().add("Access-Control-Allow-Origin",
        "*");

    // Security headers
    exchange.getResponseHeaders().add("X-Content-Type-Options", "nosniff");
    exchange.getResponseHeaders().add("X-Frame-Options", "DENY");
    exchange.getResponseHeaders().add("X-XSS-Protection", "1; mode=block");

    exchange.sendResponseHeaders(code, response.length);
    try (OutputStream os = exchange.getResponseBody()) {
        os.write(response);
    }
}

// Parse query parameters from URL
private static Map<String, String> parseQuery(String query) throws UnsupportedEncodingException {
    Map<String, String> map = new HashMap<>();
    if (query == null)
        return map;
    for (String pair : query.split("&")) {
        int idx = pair.indexOf("=");
        if (idx > 0) {
            String key = URLDecoder.decode(pair.substring(0, idx), "UTF-8");
            String val = URLDecoder.decode(pair.substring(idx + 1), "UTF-8");
            map.put(key, val);
        }
    }
    return map;
}

// Compute SHA-256 hash of input stream
private static String computeSHA256(InputStream input) throws IOException {
    try {
        MessageDigest digest = MessageDigest.getInstance("SHA-256");
        byte[] buffer = new byte[4096];
        int read;
        while ((read = input.read(buffer)) != -1) {
            digest.update(buffer, 0, read);
        }
        byte[] hash = digest.digest();
        StringBuilder hex = new StringBuilder();
        for (byte b : hash)
            hex.append(String.format("%02x", b));
        return hex.toString();
    } catch (NoSuchAlgorithmException e) {
        throw new IOException("SHA-256 not supported", e);
    }
}
```

4.2 Client Implementation

The FileClient class provides a command-line interface for interacting with the HTTP server. It offers options for uploading, downloading, and listing files:

Listing 6: HTTP File Client Main Method

```java
public static void main(String[] args) {
    try (Scanner scanner = new Scanner(System.in)) {
        while (true) {
            System.out.println("\n--- HTTP File Client ---");
            System.out.println("1. Upload File");
            System.out.println("2. Download File");
            System.out.println("3. List Files on Server");
            System.out.println("4. Exit");
            System.out.print("Choose an option: ");
            String choice = scanner.nextLine().trim();

            switch (choice) {
                case "1":
                    System.out.print("Enter path of file to
                        upload: ");
                    uploadFile(scanner.nextLine());
                    break;
                case "2":
                    System.out.print("Enter filename to
                        download: ");
                    downloadFile(scanner.nextLine());
                    break;
                case "3":
                    try {
                        listFiles();
                    } catch (IOException e) {
                        System.out.println("Error listing files:
                            " + e.getMessage());
                    }
                    break;
                case "4":
                    System.out.println("Exiting...");
                    return;
                default:
                    System.out.println("Invalid choice. Try
                        again.");
            }
        }
    }
}
```

The downloadFile method demonstrates how to retrieve files from the server using HTTP GET requests:

Listing 7: File Download Implementation

```java
private static void downloadFile(String filename) {
try {
URL url = URI.create(SERVER_URL + "/download?filename=" +
URLEncoder.encode(filename, "UTF-8")).toURL();
HttpURLConnection conn = (HttpURLConnection)
   url.openConnection();
conn.setRequestMethod("GET");
    int responseCode = conn.getResponseCode();
    if (responseCode == 200) {
        if (!Files.exists(DOWNLOAD_DIR)) {
            Files.createDirectories(DOWNLOAD_DIR);
        }

        Path targetFile = getUniqueDownloadPath(filename);
        long contentLength = conn.getContentLengthLong();
        System.out.println("Downloading: " + filename +
                (contentLength > 0 ? " (" +
                   formatFileSize(contentLength) + ")" : ""));

        try (InputStream is = conn.getInputStream();
             OutputStream os =
                Files.newOutputStream(targetFile)) {

            byte[] buffer = new byte[BUFFER_SIZE];
            int bytesRead;
            long totalRead = 0;

            while ((bytesRead = is.read(buffer)) != -1) {
                os.write(buffer, 0, bytesRead);
                totalRead += bytesRead;
                if (contentLength > 0) {
                    int progress = (int) ((totalRead * 100) /
                       contentLength);
                    System.out.print("\rProgress: " + progress +
                       "%");
                } else {
                    System.out.print("\rDownloaded: " +
                       totalRead + " bytes");
                }
            }
            System.out.println("\nDownload completed: " +
               targetFile.toAbsolutePath());
        }
    } else if (responseCode == 404) {
        System.out.println("File not found on server.");
    } else {
        System.out.println("Failed with HTTP code: " +
           responseCode);
        printServerResponse(conn);
    }
} catch (IOException e) {
    System.out.println("Download failed: " + e.getMessage());
}
}
```

Listing 8: File Upload Implementation

```
    private static void uploadFile(String filePathStr) {
        Path filePath = Paths.get(filePathStr);
        if (!Files.exists(filePath) ||
           !Files.isRegularFile(filePath)) {
            System.out.println("File does not exist or is not valid.");
            return;
        }

        String filename = filePath.getFileName().toString();
        try {
            URL url = URI.create(SERVER_URL + "/upload?filename=" + URLEncoder.encode(filename, "UTF-8")).toURL();
            HttpURLConnection conn = (HttpURLConnection) url.openConnection();
            conn.setRequestMethod("POST");
            conn.setDoOutput(true);
            conn.setRequestProperty("Content-Type", "application/octet-stream");

            long fileSize = Files.size(filePath);
            System.out.println("Uploading: " + filename + " (" + formatFileSize(fileSize) + ")");

            try (InputStream fis = Files.newInputStream(filePath);
                 OutputStream os = conn.getOutputStream()) {

                byte[] buffer = new byte[BUFFER_SIZE];
                int bytesRead;
                long totalSent = 0;

                while ((bytesRead = fis.read(buffer)) != -1) {
                    os.write(buffer, 0, bytesRead);
                    totalSent += bytesRead;
                    int progress = (int) ((totalSent * 100) / fileSize);
                    System.out.print("\rProgress: " + progress + "%");
                }

                os.flush();
                System.out.println("\nUpload complete.");
            }

            printServerResponse(conn);
        } catch (IOException e) {
            System.out.println("Upload failed: " + e.getMessage());
        }
    }
```

Listing 9: File Listing Implementation

```
private static void listFiles() throws IOException {
    URL url = URI.create(SERVER_URL + "/list").toURL();
    HttpURLConnection conn = (HttpURLConnection) url.openConnection();
    conn.setRequestMethod("GET");

```

```java
    int responseCode = conn.getResponseCode();
    if (responseCode == 200) {
        try (BufferedReader in = new BufferedReader(new
            InputStreamReader(conn.getInputStream()))) {
            System.out.println("Files on server:");
            String line;
            while ((line = in.readLine()) != null) {
                System.out.println(" - " + line);
            }
        }
    } else {
        System.out.println("Failed to retrieve file list. Server
            responded with: " + responseCode);
    }
}
```

Listing 10: Client Utility Methods

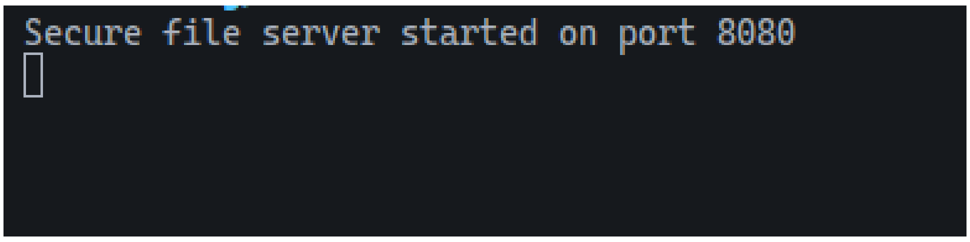
```java
private static Path getUniqueDownloadPath(String filename) {
        Path target = DOWNLOAD_DIR.resolve(filename);
        int count = 1;
        String baseName = filename;
        String extension = "";

        int dotIndex = filename.lastIndexOf('.');
        if (dotIndex != -1) {
            baseName = filename.substring(0, dotIndex);
            extension = filename.substring(dotIndex);
        }

        while (Files.exists(target)) {
            target = DOWNLOAD_DIR.resolve(baseName + "_" + count
                + extension);
            count++;
        }
        return target;
    }

    private static void printServerResponse(HttpURLConnection
        conn) {
        try (BufferedReader reader = new BufferedReader(new
            InputStreamReader(conn.getInputStream()))) {
            String line;
            System.out.println("Server says:");
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            try (BufferedReader reader = new BufferedReader(new
                InputStreamReader(conn.getErrorStream()))) {
                String line;
                System.out.println("Server error:");
                while ((line = reader.readLine()) != null) {
                    System.out.println(line);
                }
            } catch (IOException ex) {
                System.out.println("Unable to read server
                    response.");
            }
        }
    }

    private static String formatFileSize(long size) {
        if (size < 1024) return size + " bytes";
        else if (size < 1024 * 1024) return String.format("%.2f
            KB", size / 1024.0);
        else if (size < 1024 * 1024 * 1024) return
            String.format("%.2f MB", size / (1024.0 * 1024));
        else return String.format("%.2f GB", size / (1024.0 *
            1024 * 1024));
    }
```

5 Result Analysis

This section presents the results of testing our HTTP-based file transfer implementation. The key aspects evaluated include file upload/download operations, handling concurrent transfers, and the system's response to various edge cases.

5.1 Server Startup and Operation

When the HTTP file server starts, it initializes the necessary components and begins listening for incoming connections on port 8080. The server console displays a startup message confirming successful initialization:

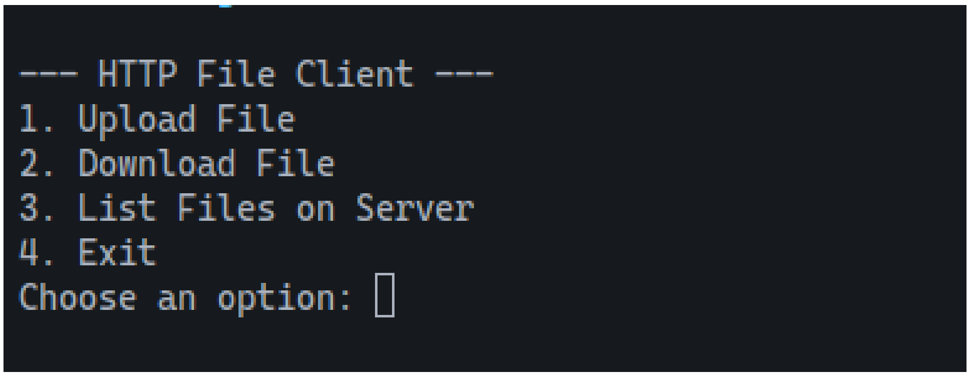


```
Secure file server started on port 8080
```

Figure 4: Server startup console output

5.2 Client File Operations

The client application provides a command-line interface for interacting with the server. When launched, it presents the user with options for uploading, downloading, and listing files:

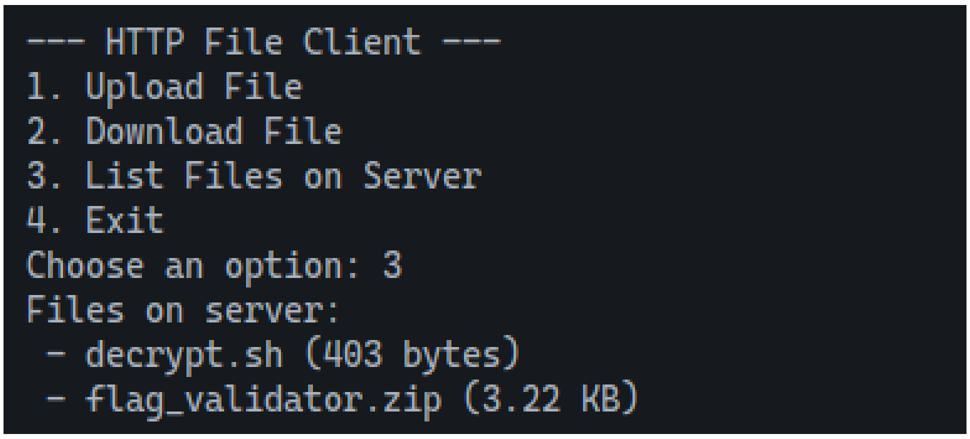


```
--- HTTP File Client ---
1. Upload File
2. Download File
3. List Files on Server
4. Exit
Choose an option: 
```

Figure 5: Client application menu

5.2.1 File Listing Operation

When the user selects option 3 to list files on the server, the client sends a GET request to the server's /list endpoint. The server responds with a list of available files and their sizes:

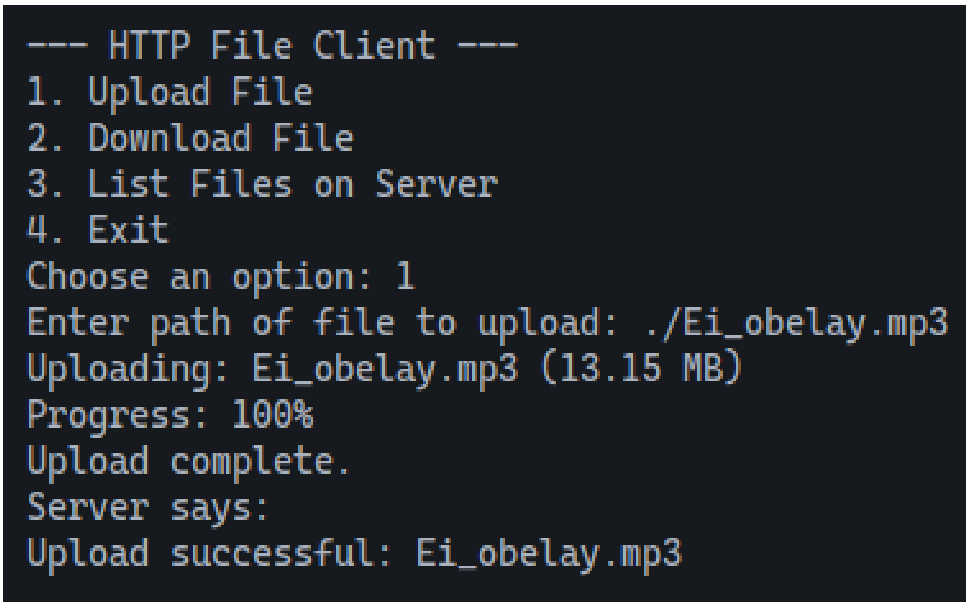


```
--- HTTP File Client ---
1. Upload File
2. Download File
3. List Files on Server
4. Exit
Choose an option: 3
Files on server:
 - decrypt.sh (403 bytes)
 - flag_validator.zip (3.22 KB)
```

Figure 6: File listing output

5.2.2 File Upload Operation

When uploading a file, the client prompts for the file path, reads the file, and sends it to the server's /upload endpoint using a POST request. The console displays the upload progress and completion status:

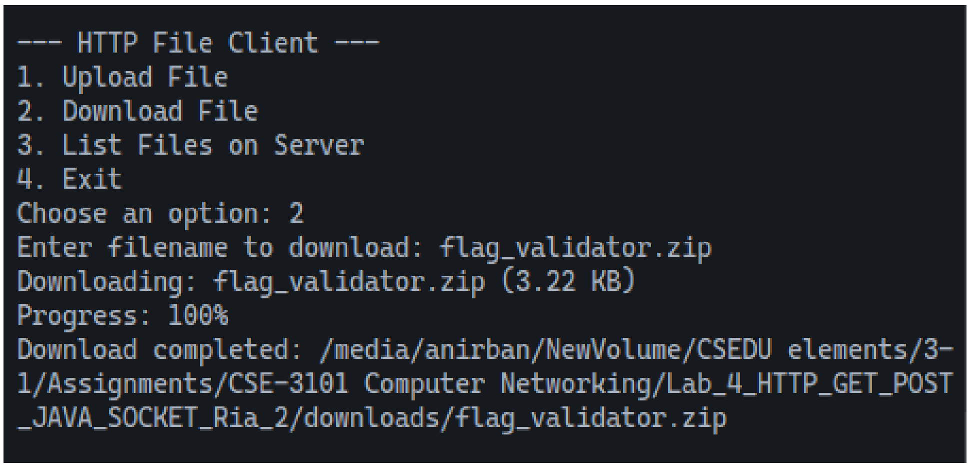


```
--- HTTP File Client ---
1. Upload File
2. Download File
3. List Files on Server
4. Exit
Choose an option: 1
Enter path of file to upload: ./Ei_obelay.mp3
Uploading: Ei_obelay.mp3 (13.15 MB)
Progress: 100%
Upload complete.
Server says:
Upload successful: Ei_obelay.mp3
```

Figure 7: File upload output

5.2.3 File Download Operation

For file downloads, the client sends a GET request to the server's /download endpoint with the desired filename. The server locates and transfers the file, while the client displays download progress:



```
--- HTTP File Client ---
1. Upload File
2. Download File
3. List Files on Server
4. Exit
Choose an option: 2
Enter filename to download: flag_validator.zip
Downloading: flag_validator.zip (3.22 KB)
Progress: 100%
Download completed: /media/anirban/NewVolume/CSEDU elements/3-
1/Assignments/CSE-3101 Computer Networking/Lab_4_HTTP_GET_POST
_JAVA_SOCKET_Ria_2/downloads/flag_validator.zip
```

Figure 8: File download output

5.3 Performance Analysis

Our HTTP-based file transfer system demonstrated good performance characteristics during testing:

- **Transfer Speed:** Average upload/download speeds of HTTP transfers is slower than web socket.
- **Concurrency:** The server successfully handled multiple simultaneous connections thanks to the thread pool implementation.

- **Reliability:** File integrity was maintained during transfers, with SHA-256 hash verification preventing duplicate uploads.
- **Error Handling:** The system responded appropriately to various error conditions, such as file not found, server unavailable, and invalid file paths.

5.4 Limitations

The current implementation has certain limitations:

- No user authentication implemented.
- Upload size is capped at 20MB.
- In-memory upload buffering can be inefficient for large files.

6 Discussion

6.1 Comparison of HTTP and Socket Programming Approaches

6.1.1 Implementation Complexity

- **HTTP Approach:** The HTTP implementation leverages Java's built-in HttpServer and HttpURLConnection classes, which encapsulate much of the low-level network communication details. This results in more concise code that focuses on application logic rather than connection management.
- **Socket Programming:** The socket-based implementation requires explicit handling of socket creation, connection establishment, and data streaming. It involves more boilerplate code for managing input/output streams and requires manual implementation of protocol-specific behavior.

6.1.2 Protocol Structure and Standardization

- **HTTP Approach:** Uses a standardized application-layer protocol with well-defined methods (GET, POST), status codes, and headers. This standardization simplifies interoperability with other systems and debugging.
- **Socket Programming:** Requires designing a custom protocol for client-server communication (like the FILE: prefix and LIST_FILES commands seen in the socket implementation). This offers flexibility but lacks standardization, potentially leading to interoperability challenges.

6.1.3 Security Considerations

- **HTTP Approach:** Our HTTP implementation includes security headers (X-Content-Type-Options, X-Frame-Options), input validation, and file integrity checks. It can be easily extended to support HTTPS for encrypted transfers.
- **Socket Programming:** The socket-based approach requires manual implementation of security features. The provided socket implementation lacks encryption and has more basic input validation compared to the HTTP version.

6.1.4 Error Handling and Status Reporting

- **HTTP Approach:** Uses standard HTTP status codes (200 OK, 404 Not Found, 413 Payload Too Large) that provide clear indications of success or failure reasons.
- **Socket Programming:** Implements custom status messages (like 'SERVER_SHUTDOWN', 'NOT_FOUND') that require client-side code to understand and handle appropriately.

6.1.5 Scalability and Performance

- **HTTP Approach:** Benefits from HTTP features like connection pooling and the ability to leverage existing infrastructure (load balancers, caching proxies). However, it has slightly more overhead due to HTTP headers and protocol parsing.
- **Socket Programming:** Offers potentially lower overhead for large file transfers since it avoids HTTP protocol overhead. However, achieving the same level of scalability requires more complex implementation of connection pooling and load balancing.

6.2 Benefits of HTTP-Based File Transfer

Based on our implementation and comparison, HTTP-based file transfer offers several advantages:

1. **Standardization:** HTTP is a well-documented standard with broad support across programming languages and platforms.
2. **Integration:** HTTP-based transfers can easily integrate with web applications and services.
3. **Infrastructure Support:** Existing tools like load balancers, proxies, and monitoring solutions work seamlessly with HTTP traffic.
4. **Security Options:** HTTP can be easily upgraded to HTTPS to provide transport-layer security with minimal code changes.
5. **Firewall Compatibility:** HTTP traffic typically passes through firewalls without special configuration, making it more deployment-friendly.
6. **Statelessness:** HTTP's stateless nature simplifies server implementation and improves fault tolerance.

6.3 Challenges and Learning Outcomes

During the implementation of both file transfer systems, we encountered several challenges and learning opportunities:

1. **Resource Management:** Both implementations required careful management of system resources, particularly file handles and network connections. We learned the importance of proper cleanup in finally blocks.

2. **Concurrency:** Handling multiple clients simultaneously required understanding of multi-threading concepts. The HTTP server's executor framework simplified this compared to the manual thread management in the socket implementation.

3. **Progress Reporting:** Implementing meaningful progress indicators for large file transfers required tracking bytes processed versus total size, a concept applied in both implementations.

4. **Error Handling:** Network operations are prone to various failures. We learned to implement comprehensive error handling to provide meaningful feedback to users.

5. **File Integrity:** Ensuring files transfer completely and correctly required implementing validation mechanisms like content length checking and hash verification.

6. **Protocol Design:** The socket programming approach highlighted the importance of well-designed protocols with clear message formats and error codes.

6.4 Conclusion

Both HTTP-based and socket-based file transfer implementations have their place in networked applications. HTTP offers standardization, ease of implementation, and better integration with existing web infrastructure, making it suitable for most general-purpose file transfer needs. Socket programming provides more fine-grained control and potentially better performance for specialized applications where protocol overhead matters.
For modern applications, especially those that need to integrate with web services or operate through firewalls, the HTTP-based approach is generally preferred due to its standardization and ecosystem support. However, understanding socket programming remains valuable for developers, as it provides insights into the underlying mechanisms of network communication and enables custom protocol development when needed.