Project 4: HTTP Pseudo-Streaming Server

Design Document

– Oorjit Chowdhary (oorjitc@uw.edu), EE 419/565 - Spring 2025

Server Design, Model, and Components

The server is implemented as a multi-threaded, persistent HTTP/1.1-compliant file server using low-level TCP sockets. Upon startup, the server scans the `content/` directory to build a metadata index containing the path, size, and MIME type for all available files. Once initialized, the server listens on a specified TCP port and accepts client connections using a dedicated listener socket. For each incoming connection, a new thread is spawned using Python's `threading` module to handle the client's persistent session. This design enables concurrent request handling without blocking.

Request processing begins with parsing the HTTP request line and headers. The server supports the `GET` method and determines the appropriate response based on the request URI and headers. If the requested file is found and not restricted, a `200 OK` response is generated for files under 5MB. For video or large media files exceeding 5MB, the server uses the HTTP `Range` header to return `206 Partial Content` responses, limiting each chunk to 5MB to support pseudo-streaming. If no range is specified in the request, the server proactively responds with the first 5MB using `206 Partial Content` to ensure compatibility with browser streaming behavior.

The server enforces access control by returning `403 Forbidden` for any request targeting the `confidential/` directory and returns a `404 Not Found` response for missing files. All responses include proper HTTP headers such as `Date`, `Last-Modified`, `Content-Type`, `Content-Length`, `Connection`, and `Accept-Ranges`. The response generator is responsible for reading the appropriate byte ranges from disk and assembling the final response in RFC 2616-compliant format, including support for persistent connections.

Handling Multiple Clients

Each connection is handled by a dedicated thread using Python's `threading` module. Threads process requests concurrently without blocking other clients. Persistent connections are supported, allowing each thread to serve multiple sequential requests over the same socket.

Estimated concurrency:

- Light usage (<50 clients): stable.
- Stress tested up to 200 concurrent clients using ApacheBench (`ab`) with `-k -c 200 -n 200`: ~95% completion within 500ms.
- Beyond 500 clients: thread scaling limits begin to show, so additional optimizations (e.g., thread pool, asyncio) may be needed.

Libraries Used

All libraries used are part of Python's standard library:

- `socket` – TCP socket communication
- `os`, `sys` – File system and argument parsing
- `datetime` – RFC-compliant timestamp generation
- `threading` – Concurrent client support

Extra Capabilities

Supports over 10 standard MIME content types via file extension mapping.