

Assignment: Build a Multi-threaded HTTP Server Using Socket Programming

Objective

Design and implement a multi-threaded HTTP server from scratch using low-level socket programming. This assignment will deepen your understanding of the HTTP protocol, concurrent programming, and network security.

Overview

You will create an HTTP server that can handle multiple concurrent clients using threading, serve static files and binary content for GET requests, process JSON data for POST requests, and implement various HTTP protocol features including connection persistence and host validation.

Requirements

1. Server Configuration

- The server should run on `localhost` (127.0.0.1) by default
- Default port should be 8080
- The server should accept command-line arguments to optionally specify:
 - Port number (first argument)
 - Host address (second argument)
 - Maximum thread pool size (third argument, default: 10)
- Example: `./server 8000 0.0.0.0 20`

2. Socket Implementation

- Use TCP sockets for communication
- The server should bind to the specified host and port
- Listen for incoming connections (queue size of at least 50)
- Properly manage socket lifecycle for both persistent and non-persistent connections

3. Multi-threading & Concurrency

- Thread Pool Implementation:

- Implement a thread pool with a configurable maximum size (default: 10 threads)
- New client connections should be assigned to available threads
- If all threads are busy, new connections should wait in a queue
- Implement proper synchronization using mutexes/locks
- **Connection Queue:**
 - Maintain a queue for pending connections when thread pool is saturated
 - Log when clients are queued and when they are served
- **Thread Safety:**
 - Ensure all shared resources are properly synchronized
 - Avoid race conditions and deadlocks

4. HTTP Request Handling

- Parse incoming HTTP requests to extract:
 - Request method (GET, POST)
 - Request path
 - HTTP version
 - All headers (store in a dictionary/map structure)
- Support both GET and POST methods
- Return 405 "Method Not Allowed" for other methods (PUT, DELETE, etc.)
- Handle requests up to 8192 bytes in size
- Properly parse and validate HTTP request format

5. GET Request Implementation

A. HTML File Serving:

- Serve HTML files from the `resources` directory
- When root path `/` is requested, serve `index.html` by default
- Set Content-Type as `text/html; charset=utf-8`
- Example: `/page.html` → `resources/page.html`

B. Binary File Transfer (Images and Text):

- Support downloading of image files (PNG, JPEG) and text files (TXT)
- These files should be sent as binary data using `application/octet-stream`
- **Implementation Requirements:**
 - Read files in binary mode
 - Send the entire file content as binary stream
 - Set appropriate headers for file download
 - Include `Content-Disposition` header to trigger download in browsers

Content-Type Handling:

- `.html` → `text/html; charset=utf-8` (render in browser)
- `.txt` → `application/octet-stream` (download as file)
- `.png` → `application/octet-stream` (download as file)
- `.jpg/.jpeg` → `application/octet-stream` (download as file)
- Return 415 "Unsupported Media Type" for other file types

Binary Transfer Response Format:

```
http
HTTP/1.1 200 OK
Content-Type: application/octet-stream
Content-Length: [file size in bytes]
Content-Disposition: attachment; filename="[filename]"
Date: [current date in RFC 7231 format]
Server: Multi-threaded HTTP Server
Connection: [keep-alive or close]
```

[binary file data]

6. POST Request Implementation

- **JSON Processing:**
 - Only accept `application/json` Content-Type
 - Parse and validate JSON from request body
 - Return 400 "Bad Request" for invalid JSON
 - Return 415 "Unsupported Media Type" for non-JSON content
- **File Creation:**
 - Create a new file in `resources/uploads/` directory
 - Filename format: `upload_[timestamp]_[random_id].json`
 - Example: `upload_20240315_123456_a7b9.json`
 - Write the received JSON data to the file
 - Return 201 "Created" with the file path in response body
- **Response Format:**

```
json
{
  "status": "success",
  "message": "File created successfully",
  "filepath": "/uploads/upload_20240315_123456_a7b9.json"
}
```

7. Security Requirements

- **Path Traversal Protection:**
 - Implement strict path validation to prevent directory traversal attacks
 - Canonicalize paths and ensure they stay within `resources` directory
 - Block requests containing `..`, `./`, or absolute paths
 - Return 403 "Forbidden" for any unauthorized path access attempts
 - Examples of blocked requests:
 - `../../etc/passwd`
 - `../../../../sensitive.txt`
 - `//etc/hosts`
- **Host Header Validation:**
 - Check the `Host` header in all requests
 - Only accept requests where Host matches the server's address
 - Valid examples: `localhost:8080`, `127.0.0.1:8080`
 - Return 400 "Bad Request" for missing Host header
 - Return 403 "Forbidden" for mismatched Host header
 - Log security violations for monitoring

8. Connection Management

- **Keep-Alive Support:**
 - Check the `Connection` header in requests
 - If `Connection: keep-alive`, maintain the connection for subsequent requests
 - If `Connection: close`, close the connection after response
 - Default behavior (HTTP/1.1): Keep connection alive
 - Default behavior (HTTP/1.0): Close connection
- **Connection Timeout:**
 - Implement a timeout for persistent connections (30 seconds)
 - Close idle connections after timeout
 - Include `Keep-Alive: timeout=30, max=100` header in responses
- **Connection Limits:**
 - Maximum 100 requests per persistent connection
 - Close connection after reaching the limit

9. HTTP Response Format

Successful HTML Response (200 OK):

```

http
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: [size in bytes]
Date: [current date in RFC 7231 format]
```

Server: Multi-threaded HTTP Server
Connection: [keep-alive or close]
Keep-Alive: timeout=30, max=100

[HTML content]

Successful Binary File Response (200 OK):

http
HTTP/1.1 200 OK
Content-Type: application/octet-stream
Content-Length: [file size in bytes]
Content-Disposition: attachment; filename="image.png"
Date: [current date in RFC 7231 format]
Server: Multi-threaded HTTP Server
Connection: [keep-alive or close]

[binary data]

Successful POST Response (201 Created):

http
HTTP/1.1 201 Created
Content-Type: application/json
Content-Length: [size in bytes]
Date: [current date in RFC 7231 format]
Server: Multi-threaded HTTP Server
Connection: [keep-alive or close]

[JSON response body]

Error Responses:

- 400 Bad Request: Malformed request or missing Host header
- 403 Forbidden: Unauthorized path access or Host mismatch
- 404 Not Found: Requested resource doesn't exist
- 405 Method Not Allowed: Non-GET/POST methods
- 415 Unsupported Media Type: Wrong Content-Type or file type
- 500 Internal Server Error: Server-side errors
- 503 Service Unavailable: Thread pool exhausted (with Retry-After header)

10. Logging Requirements

Implement comprehensive logging with timestamps:

Server Startup:

[2024-03-15 10:30:00] HTTP Server started on http://127.0.0.1:8080

[2024-03-15 10:30:00] Thread pool size: 10

[2024-03-15 10:30:00] Serving files from 'resources' directory

[2024-03-15 10:30:00] Press Ctrl+C to stop the server

File Transfer Logging:

[2024-03-15 10:30:15] [Thread-1] Connection from 127.0.0.1:54321

[2024-03-15 10:30:15] [Thread-1] Request: GET /image.png HTTP/1.1

[2024-03-15 10:30:15] [Thread-1] Host validation: localhost:8080 ✓

[2024-03-15 10:30:15] [Thread-1] Sending binary file: image.png (45678 bytes)

[2024-03-15 10:30:15] [Thread-1] Response: 200 OK (45678 bytes transferred)

[2024-03-15 10:30:15] [Thread-1] Connection: keep-alive

Thread Pool Status:

[2024-03-15 10:35:00] Thread pool status: 8/10 active

[2024-03-15 10:35:30] Warning: Thread pool saturated, queuing connection

[2024-03-15 10:35:35] Connection dequeued, assigned to Thread-3

Testing Your Server

1. Directory Structure

Create the following structure:










```
project/
├── server.py (or server.c/server.java)
├── resources/
│   ├── index.html
│   ├── about.html
│   ├── contact.html
│   ├── sample.txt
│   ├── logo.png
│   └── photo.jpg
└── uploads/ (directory for POST uploads)
```

2. Test Files Preparation





- Create at least 3 HTML files (index.html, about.html, contact.html)
- Add at least 2 PNG images (various sizes, including one > 1MB)
- Add at least 2 JPEG images
- Add at least 2 text files with sample content

3. Test Scenarios





Basic Functionality:

-  GET / → Serves `resources/index.html` (displayed in browser)
-  GET `/about.html` → Serves HTML file
-  GET `/logo.png` → Downloads PNG file as binary
-  GET `/photo.jpg` → Downloads JPEG file as binary
-  GET `/sample.txt` → Downloads text file as binary
-  POST `/upload` with JSON → Creates file in uploads directory
-  GET `/nonexistent.png` → Returns 404
-  PUT `/index.html` → Returns 405
-  POST `/upload` with non-JSON → Returns 415




Binary Transfer Tests:

-  Downloaded PNG file matches original (checksum verification)
-  Downloaded JPEG file matches original
-  Large image files (>1MB) transfer completely
-  Binary data integrity maintained (no corruption)

Security Tests:

-  GET `../etc/passwd` → Returns 403
-  GET `../../../../config` → Returns 403
-  Request with `Host: evil.com` → Returns 403
-  Request without Host header → Returns 400

Concurrency Tests:

-  Handle 5 simultaneous file downloads
-  Queue connections when thread pool is full
-  Multiple clients downloading large files simultaneously

Deliverables

1. **Source Code:**
 - Well-commented server implementation
 - Binary file handling implementation
 - Thread pool management
 - Proper error handling
2. **Test Files:**
 - At least 3 HTML files in resources directory
 - At least 2 PNG images (one >1MB)
 - At least 2 JPEG images
 - At least 2 text files
 - Sample JSON files for POST testing
3. **Documentation:**
 - README with build and run instructions
 - Description of binary transfer implementation
 - Thread pool architecture explanation
 - Security measures implemented
 - Known limitations

Note: Create a GitHub repository and your code there. Share the link of the Github repository for submission

Important Notes

- **Binary Reading:** Ensure files are opened and read in binary mode to preserve data integrity
- **Buffer Management:** Use appropriate buffer sizes for efficient binary transfer (e.g., 4KB or 8KB chunks)
- **Error Handling:** Properly handle errors during file reading and socket writing

Submission Deadline

10th October, 2025
