# C++ Programming

**MUHAMMAD SHAHZAIB**

**DEP TASK 4**

**User Manual**

# Table of Contents

# Manual for Multi-Threaded Web Server

## 1. Introduction

The **Multi-Threaded Web Server** is a console-based C++ application designed to handle multiple client requests simultaneously. It listens for HTTP requests and serves static HTML files using the C++ Standard Library's threading capabilities. This web server provides a simple interface for handling concurrent connections, making it an ideal choice for users looking to host static websites or serve static content efficiently.

## 2. Features and Functions

### 1. Start the Server

**Functionality:** The server listens for incoming HTTP requests on a specified port and serves static HTML files from a designated directory.

**Steps:**

- The server is initialized on a specified port, usually **8080** by default.
- The server continuously listens for incoming client connections.
- For each client connection, a new thread is spawned to handle the request, allowing multiple clients to connect simultaneously.
- The server responds to HTTP GET requests by serving the requested file from the directory.
- If the requested file is not found, the server returns a 404 error page.

### 2. Handle Client Connections

**Functionality:** This function manages incoming client connections using multi-threading to ensure efficient request handling.

**Steps:**

- A client connects to the server using an HTTP request.
- The server accepts the connection and creates a new thread for handling the client's request.
- Each request is processed independently, allowing the server to manage multiple connections concurrently.

### 3. Serve Static Files

**Functionality:** This function serves static HTML and other supported files to clients based on the HTTP request.

**Steps:**

- The server reads the HTTP GET request to determine the requested file.
- The server searches for the requested file in the specified directory.

- If the file is found, the server sends the file content to the client with the appropriate MIME type.
- If the file is not found, a 404 error page is returned to the client.

## 4. MIME Type Detection

**Functionality:** This function determines the correct MIME type of the requested file based on its extension.

**Steps:**

- The server checks the file extension of the requested file.
- The server determines the MIME type based on common file extensions, such as html, css, js, png, jpg, jpeg, gif, and others.
- The server returns the file with the correct MIME type, ensuring the client can interpret the file correctly.

## 5. Error Handling

**Functionality:** This function handles errors gracefully, ensuring a robust and stable server operation.

**Steps:**

- The server checks for errors during socket creation, binding, and listening.
- If any error occurs, a descriptive error message is displayed.
- If a requested file is not found, a 404 error page is returned to the client.

## 6. Exit the Server

**Functionality:** The server can be stopped manually by terminating the process.

**Steps:**

- To stop the server, manually terminate the server application using your terminal or command prompt.
- Closing the terminal or command prompt running the server will also stop the server.

# 3. Instructions to Run the Program on a Windows System

- A C++ compiler (e.g., GCC, MSVC)
- A terminal or command prompt

**Steps to Compile and Run the Program**

- o Use the cd command to navigate to the directory where the source code file (MultiThreadedWebServer.cpp) is located.

- Compile the Program, Use a C++ compiler to compile the source code. For example, if using GCC:
- g++ -o MultiThreadedWebServer MultiThreadedWebServer.cpp -pthread o    Open a web browser and navigate to http://localhost:8080 to access the server. o    You can also test the server using tools like curl or Postman for different file requests.

# 4. How the Code Works
## 1. Initialization:

- When the program starts, it creates a socket and binds it to the specified port (8080).
- The server listens for incoming connections, preparing to handle client requests.

## 2. User Interaction:

- The server does not require direct user interaction after starting. It operates in the background, handling requests automatically.

## 3. File Handling:

- The server reads files from the current directory.
- HTML, CSS, JavaScript, image files, and others can be served based on their extensions.

## 4. Multi-Threading:

- For each incoming connection, a new thread is created to handle the request.
- This approach ensures multiple clients can connect and interact with the server simultaneously without blocking other requests.

## 5. Error Handling:

- Errors are handled gracefully, with descriptive error messages displayed for socket, binding, or file handling issues.
- A 404 error page is returned if a requested file is not found.

## 6. Exiting the Program:

- The server continues running until manually terminated by the user.
- Use Ctrl + C or close the terminal window to stop the server.

# 5. Example Usage
## Accessing a Static HTML File

1. Start the server by running the compiled program.
2. Open a web browser and navigate to http://localhost:8080/index.html.
3. The server will serve the index.html file from the current directory.

4. If the file does not exist, a 404 error page will be displayed.

   g++ -o MultiThreadedWebServer MultiThreadedWebServer.cpp -pthread

   ./MultiThreadedWebServer

```
Connection from 127.0.0.1:12345
Request received:
GET /index.html HTTP/1.1
Host: localhost:8080
...
Response sent for file: /index.html
```

```
Connection from 127.0.0.1:12347
Request received:
GET /nonexistent.html HTTP/1.1
Host: localhost:8080
...
Error opening 404 file
Response sent for file: /404.html
```

## 6. Troubleshooting

**Error: "Error creating socket"**

- o Check if your system allows socket creation. Ensure no other application uses the specified port (8080).

**Error: "Error binding socket"**

- o Verify that the specified port is not in use. Change the port number if necessary.

**404 Error Page**

- o Ensure the requested file exists in the current directory. Check the file name and extension.

**Server Not Responding**

- o Check if the server is running and listening on the correct port. Restart the server if needed.

**CODE:**

```cpp
#include <iostream>
#include <thread>
#include <string>
#include <fstream>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <dirent.h>
#include <sstream>
using namespace
std;
class TcpServer
{ public:
    TcpServer(int port) {          serverSocket =
socket(AF_INET, SOCK_STREAM, 0);          if
(serverSocket < 0) {          cerr << "Error
creating socket" << endl;          exit(1);
        }
        serverAddress.sin_family = AF_INET;
serverAddress.sin_port = htons(port);
serverAddress.sin_addr.s_addr = INADDR_ANY;
        if (bind(serverSocket, (struct
sockaddr*)&serverAddress, sizeof(serverAddress)) < 0) {
cerr << "Error binding socket" << endl;          exit(1);
        }
    }
    void startListen() {          if
(listen(serverSocket, 3) < 0) {
          cerr << "Error listening for connections" << endl;
exit(1);
        }

        cout << "Server listening on port 8080..." << endl;
        while (true) {          struct sockaddr_in
clientAddress;          socklen_t clientAddressLength =
sizeof(clientAddress);
```

```cpp
            int clientSocket = accept(serverSocket, (struct
sockaddr*)&clientAddress, &clientAddressLength);
            if (clientSocket < 0) {                 cerr <<
"Error accepting connection" << endl;
continue;
        }                   thread t(&TcpServer::handleRequestThread, this,
clientSocket);               t.detach();
        }
    }
private:
    void handleRequestThread(int clientSocket) {
        string request =
readRequest(clientSocket);
        string file =
parseRequest(request);
        sendResponse(clientSocket,
file);

close(clientSocket);
    }       string readRequest(int
clientSocket) {           char
buffer[1024];         string request;
        while (true) {              int bytesRead =
read(clientSocket, buffer, 1024);              if (bytesRead
< 0) {
            cerr << "Error reading from socket" << endl;
return "";
        }
            request += string(buffer, bytesRead);
if (request.find("\r\n\r\n") != string::npos) {
break;
        }
      }

    return request;
    }       string parseRequest(string
request) {
```

```cpp
        size_t pos = request.find("GET
");        if (pos == string::npos) {
return "404.html";
        }
        pos += 4;           size_t endPos =
request.find(" ", pos);            if (endPos ==
string::npos) {            return "404.html";
        }
        string file = request.substr(pos, endPos -
pos);        if (file == "/") {           file =
"/index.html";
        }
        return
file;
    }       string getContentType(const string&
file) {        size_t dotPos =
file.find_last_of(".");        if (dotPos ==
string::npos) {        return
"text/plain";
        }
        string extension = file.substr(dotPos + 1);        if (extension
== "html") return "text/html";        if (extension == "css") return
"text/css";        if (extension == "js") return
"application/javascript";        if (extension == "png") return
"image/png";        if (extension == "jpg" || extension == "jpeg") return
"image/jpeg";        if (extension == "gif") return "image/gif";


        return "text/plain";
    }
    void sendResponse(int clientSocket, string file)
{        ifstream fileStream("." + file);        if
(!fileStream.is_open()) {
fileStream.open("404.html");            if
(!fileStream.is_open()) {
            cerr << "Error opening file" << endl;
return;
        }
        }
```

```cpp
        string
fileContent((istreambuf_iterator<char>(fileStream)),
istreambuf_iterator<char>());
        string contentType =
getContentType(file);
        stringstream response;          response << "HTTP/1.1 200
OK\r\n";          response << "Content-Type: " << contentType << "\r\n";
response << "Content-Length: " << fileContent.length() << "\r\n";
response << "\r\n";          response << fileContent;
        write(clientSocket, response.str().c_str(),
response.str().length());      }        int serverSocket;      struct
sockaddr_in serverAddress; };
int main() {
    TcpServer server(8080);
server.startListen();
return 0;
}
```