Alexandria University
Faculty of Engineering
CSED 2025

Networks assignment 1 report

Submitted to

Eng. Mohamed Essam

Faculty of engineering Alex. University

Submitted by

Mkario Michel Azer
20011982

Islam Yasser Mahmoud
20010312

Faculty of engineering Alex. University

Table of Contents

Client:

Overview:

The code implements a multi-threaded client application designed to communicate with an HTTP server. The client can handle both GET and POST requests, supporting file transfers and server interactions. Threading is utilized to manage multiple connections simultaneously, providing responsive and efficient client-server interaction.

Major functions:

1. **endsWith and startsWith Functions:**
    - Purpose: Check if a given string ends or starts with a specified suffix or prefix.
    - Usage: Used for string manipulation in the code.
2. **HandlePostRequest Function:**
    - Purpose: Process a POST request command, extract the file path, read the content of the file, and append it to the command.
    - Usage: Called when a command is a POST request.
3. **receiveResponses Function:**
    - Purpose: Continuously receive responses from the server, extract file names and contents from the response, and save the files.
    - Usage: Executed in a separate thread to handle responses asynchronously.
4. **main Function:**
    - Purpose: The main entry point of the program.
    - Usage:
        - Initialize Winsock.
        - Create a client socket and connect to a server.
        - Print the initial server response.
        - Send a "Hello from client" message to the server.
        - Read commands from a file ("commands.txt").
        - Create a separate thread to handle responses from the server asynchronously.
        - Process each command, send it to the server, and wait for a response.
        - Sleep for a period to allow the response thread to finish processing.

Data structures:

Non-specific used.

Organization:

1. **Initialization:**
    - Initialize Winsock.
    - Create a client socket.
    - Set up server address and connect to the server.
2. **Server Communication:**
    - Receive and print the initial server response.
    - Send a "Hello from client" message to the server.
3. **Handling Commands:**
    - Open the "commands.txt" file.
    - Create a separate thread (**receiveResponses**) to handle server responses asynchronously.
    - Read each command from the file:
        - If the command is a POST request, call **HandlePostRequest** to append file content to the command.
        - Send the command to the server.
        - Sleep for a short duration between sending commands.
4. **Asynchronous Response Handling:**
    - In a separate thread (**receiveResponses**), continuously receive and process responses from the server.
    - If a response contains a "File-Name" header, extract the file name and content, save the file, and open it.
    - Continue processing responses until the connection is closed or an error occurs.
5. **Cleanup:**
    - Close the program.

Server:

Overview:

The code implements a basic multithreaded HTTP server capable of handling GET and POST requests. And communicates with multiple threads at a time.

Major functions:

1. **endsWith Function:**
    - This function checks if a given string ends with a specified suffix.
    - It is used for string comparison in various parts of the code.
2. **HandleGetRequest Function:**
    - This function handles GET requests from clients.
    - It attempts to open the specified file, and if successful, it sends an HTTP 200 OK response with the file content.
    - If the file doesn't exist, it sends an HTTP 404 Not Found response.
3. **HandlePostRequest Function:**
    - This function handles POST requests from clients.
    - It reads the data from the client, writes it to a file with the specified file path, and sends an HTTP 200 OK response with a success message.
4. **HandleTCPClient Function:**
    - This function is called for each connected client in a separate thread.
    - It sends an initial "Hello from Server" message to the client.
    - It reads data from the client in a loop, adjusts the receive timeout dynamically based on the number of active connections, and processes each line of data.
    - For each line, it tokenizes the input, and based on the command (GET or POST), it calls the corresponding handler function.
5. **main Function:**
    - Initializes Winsock and creates a server socket.
    - Binds the server socket to the server address and starts listening for incoming connections.
    - In an infinite loop, accepts incoming connections, creates a new thread for each connected client (**HandleTCPClient**), and continues listening for more connections.

Data structures:

Non-specific used.

Organization:

1. **Waiting for a Client:**
    - The program enters a loop where it continuously waits for a client to connect.
2. **Accepting Incoming Connection:**
    - The **accept** function is used to accept incoming connections. If successful, it returns a new socket (**clientSocket**) for communication with the client.
3. **Handling Successful Connection:**
    - If the **accept** operation is successful, the program prints a success message indicating that a client has connected.
4. **Creating a Thread for Client Handling:**
    - A new thread is created to handle the connected client. The **HandleTCPClient** function is executed in this thread, passing the **clientSocket** as an argument.
5. **Updating the Number of Active Connections:**
    - Before launching the thread, the number of active connections is incremented in a critical section to ensure thread safety.
6. **Continuing the Loop:**
    - The program continues to wait for more clients by going back to the beginning of the loop.
7. **Note:**
    - The loop runs indefinitely, and the program does not reach any code after this point due to the infinite loop structure. It's a server that continuously accepts and handles incoming client connections.

Code snippets

Client-side

```cpp
#include <iostream>
#include <winsock2.h>
#include <winsock.h>
#include <ws2tcpip.h>
#include <fstream>
#include <thread>
#include <bits/stdc++.h>
#include <unistd.h>

using namespace std;


// Function to check if a string starts with a specified prefix
bool startsWith(const string& str, const string& prefix) {
    if (str.length() < prefix.length()) return false;
    return str.compare( pos: 0,  n: prefix.length(),  str: prefix) == 0;
}

// Function to handle POST requests
void HandlePostRequest(string& command) {
    istringstream iss( str: command);
    string word;
    iss >> word;
    iss >> word;
    filesystem::path filePath( source: word);
    string fileName = filePath.filename().string();

    // Read the content of the file specified in the POST request
    ifstream inputFile( s: "../" + fileName);
    if(!inputFile.is_open()) {
        cout << "cannot open file" <<endl;
        exit( Code: -1);
    }
    string fileContent( beg: (istreambuf_iterator<char>( &: inputFile)),  end: istreambuf_iterator<char>());
    command += "\r\n" + fileContent + "\nendPOST";
}

// Function to receive responses from the server
void receiveResponses(SOCKET clientSocket, char buffer[]) {
    //char buffer[65536];
    int bytesRead;
    bytesRead = recv( s: clientSocket,  buf: buffer,  len: sizeof(buffer),  flags: 0);
    buffer[bytesRead] = '\0';
        cout << "bytesRead: "<<bytesRead << endl;
        cout << "whole response: \n" << buffer << endl;

        // Find the position of "File-Name": in the buffer
        size_t fileNamePos = string( s: buffer).find( s: "File-Name: ");
```

```
        // Check if the "File-Name": header is present in the response
        if (fileNamePos != string::npos) {
            // Extract the file name after the File-Name: header
            size_t startQuote = fileNamePos + strlen( Str: "File-Name: ");
            size_t endQuote = string( s: buffer).find( s: "\r\n", pos: startQuote);
            size_t nameSize = endQuote - startQuote;

            if (endQuote != string::npos) {
                string fileName = string( s: buffer).substr( pos: startQuote, n: endQuote - startQuote);

                size_t newStartQuote = endQuote + strlen( Str: "\r\n");
                size_t contentSize = bytesRead - newStartQuote - 4;
                cout << contentSize << endl;
                char temp[contentSize];

                // Shift the remaining characters to fill the gap
                for (size_t i = 0; i < contentSize ; i++) {
                    temp[i] = buffer[i + newStartQuote];
                }

                // Save content to file
                std::ofstream outputFile( s: "../" + fileName, mode: std::ios::binary);
```

```
                if (outputFile.is_open()) {
                    // Write the data to the file
                    outputFile.write( s: temp , n: contentSize);
                    outputFile.close();
                    cout << "Data saved to file: " << fileName << endl;
                } else {
                    cout << "Unable to open file: " << fileName << endl;
                }

                // Display the file
                std::wstring name( beg: fileName.begin(), end: fileName.end());

                // Use ShellExecute to open the image file with the default associated program
                ShellExecuteW( hwnd: 0, lpOperation: L"open", lpFile: name.c_str(), lpParameters: 0, lpDirectory: 0, nShowCmd: SW_SHOW);
            } else {
                cout << "Invalid response format: Missing closing quote for File-Name" << endl;
            }
        }
    }
```

```
int main() {
    // Initialize Winsock
    WSADATA wsa;
    if (WSAStartup( wVersionRequested: MAKEWORD(2, 2), lpWSAData: &wsa) != 0) {
        cout << "Failed to initialize Winsock" << endl;
        exit( Code: -1);
    }

    // Create a client socket
    SOCKET clientSocket;
    clientSocket = socket( af: AF_INET, type: SOCK_STREAM, protocol: 0);
    if (clientSocket == -1) {
        cout << "Failed to create client socket" << endl;
        exit( Code: -1);
    }

    // Set up the server address
    struct sockaddr_in serverAddress;
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_port = htons( hostshort: 80);
    inet_pton( Family: AF_INET, pStringBuf: "127.0.0.1", pAddr: &(serverAddress.sin_addr));

```

```
115     // Connect to the server
116     if (connect( s: clientSocket, name: (struct sockaddr*)&serverAddress, namelen: sizeof(struct sockaddr_in)) != 0) {
117         cout << "Failed to connect to server" << endl;
118         exit( Code: -1);
119     }
120
121     cout << "Connected to server successfully" << endl;
122
123     // Receive initial message from the server
124     char buffer[65536];
125     int bytesRead = recv( s: clientSocket, buf: buffer, len: sizeof(buffer), flags: 0);
126     buffer[bytesRead] = '\0';
127     cout << buffer << endl;
128
129     // Send a message to the server
130     const char *message = "Hello from client";
131     send( s: clientSocket, buf: message, len: strlen( Str: message), flags: 0);
132
133     // Open the commands file for reading
134     ifstream inputFile( s: "../commands.txt");
135     if(!inputFile.is_open()) {
136         cout << "cannot open file" << endl;
137         exit( Code: -1);
138     }
139
```

```
140     // Read commands from the file and send them to the server
141     string command;
142     while (getline( &: inputFile, &: command)) {
143         // If the command is a POST request, handle it
144         if (startsWith( str: command, prefix: "POST"))
145             HandlePostRequest( &: command);
146
147         // Append newline to the command
148         command = command + "\n";
149         const char* request = command.c_str();
150
151         // Send the command to the server
152         send( s: clientSocket, buf: request, len: command.length(), flags: 0);
153
154         cout << command << endl;
155         // receive the response from the server
156         receiveResponses(clientSocket, buffer);
157     }
158     return 0;
159 }
```

Server-side

```cpp
#include <iostream>
#include <winsock2.h>
#include <winsock.h>
#include <unistd.h>
#include <thread>
#include <bits/stdc++.h>

using namespace std;

int number_of_active_connections = 0;
CRITICAL_SECTION cs;
const int BASE_TIMEOUT = 20000; // 20 s

// Function to handle GET requests
void HandleGetRequest(SOCKET clientSocket, string fileName) {
    // Open the file in binary mode
    ifstream inputFile( s: "../" + fileName,  mode: ios::binary);

    if (!inputFile.is_open()) {
        // If the file does not exist, send a 404 Not Found response
        cout << fileName <<" not exists in server" <<endl;
        string Not_Found_Message= "HTTP/1.1 404 Not Found\r\n" + fileName + "\r\n\r\n";
        const char* response = Not_Found_Message.c_str();
        send( s: clientSocket,  buf: response,  len: Not_Found_Message.length(),  flags: 0);
    }
    else {
        // If the file exists, send a 200 OK response with file content
        string OK_Message= "HTTP/1.1 200 OK\r\n";
        string fileContent( beg: (istreambuf_iterator<char>( &: inputFile)),  end: istreambuf_iterator<char>());
        string temp = OK_Message + "File-Name: " + fileName + "\r\n" + fileContent + "\r\n\r\n";
        const char* response = temp.c_str();
        send( s: clientSocket,  buf: response,  len: temp.length(),  flags: 0);
        cout << fileContent.length() << endl;
    }
}

// Function to handle POST requests
void HandlePostRequest(SOCKET clientSocket, string filePath, istringstream& iss) {
    // Read data from the request until "endPOST" is encountered
    string data = "";
    string line;
    getline( &: iss,  &: line,  delim: '\n');
    while(line != "endPOST") {
        data += line + '\n';
        getline( &: iss,  &: line,  delim: '\n');
```

```
47
48          // Write the received data to a file
49          ofstream outputFile( s: "../" + filePath);
50          if (!outputFile.is_open()) {
51              cout << "Error opening file: " << filePath << endl;
52          }
53          // write data to file
54          outputFile << data;
55          outputFile.close();
56
57          // Send a 200 OK response indicating successful upload
58          string OK_Message= "HTTP/1.1 200 OK\r\n";
59          string temp = OK_Message + filePath + " uploaded successfully\r\n\r\n";
60          const char* response = temp.c_str();
61          send( s: clientSocket,  buf: response,  len: temp.length(),  flags: 0);
62      }
63
```

```
64  // Function to handle a TCP client
65  void HandleTCPClient(SOCKET clientSocket) {
66      // hand shaking
67      const char* message = "Hello from Server";
68      send( s: clientSocket,  buf: message,  len: strlen( Str: message),  flags: 0);
69
70      char buffer[4096];
71      int bytesRead = recv( s: clientSocket,  buf: buffer,  len: sizeof(buffer),  flags: 0);
72      buffer[bytesRead] = '\0';
73      cout << buffer << endl;
74
75      // Set timeout based on the number of active connections
76      int timeout = BASE_TIMEOUT / number_of_active_connections;
77      setsockopt( s: clientSocket,  level: SOL_SOCKET,  optname: SO_RCVTIMEO,  optval: (const char*)&timeout,  optlen: sizeof(int));
78
79      while((bytesRead = recv( s: clientSocket,  buf: buffer,  len: sizeof(buffer),  flags: 0)) > 0) {
80          buffer[bytesRead] = '\0';
81
82          // Update timeout based on the number of active connections
83          timeout = BASE_TIMEOUT / number_of_active_connections;
84          setsockopt( s: clientSocket,  level: SOL_SOCKET,  optname: SO_RCVTIMEO,  optval: (const char*)&timeout,  optlen: sizeof(int));
85
86          // stream to access buffer
87          istringstream iss( str: buffer);
88          string line;
89          while (getline( &: iss,  &: line,  delim: '\n')) {
90              // stream to split line by spaces
91              istringstream iss2( str: line);
```

```
            string arg;
            vector<string> args;

            while(iss2 >> arg)
                args.push_back(arg);

            // handle post request
            if(args[0] == "POST") HandlePostRequest(clientSocket, filePath: args[1], &: iss);
            // handle get request
            else HandleGetRequest(clientSocket, fileName: args[1]);
        }
    }
    // client will disconnect so decrement number of active connection
    cout << "Client disconnected" << endl;
    EnterCriticalSection( lpCriticalSection: &cs);
    number_of_active_connections--;
    LeaveCriticalSection( lpCriticalSection: &cs);
    closesocket( s: clientSocket);
}
```

```
int main() {
    // Initialize Winsock
    WSADATA wsa;
    if (WSAStartup( wVersionRequested: MAKEWORD(2, 2), lpWSAData: &wsa) != 0) {
        cout << "Failed to initialize Winsock" << endl;
        exit( Code: -1);
    }

    // Create a socket for the server
    SOCKET serverSocket;
    serverSocket = socket( af: AF_INET, type: SOCK_STREAM, protocol: 0);
    if (serverSocket == -1) {
        cout << "Failed to create server socket" << endl;
        exit( Code: -1);
    }

    // Set up the server address
    struct sockaddr_in serverAddress;
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_port = htons( hostshort: 80);
    serverAddress.sin_addr.s_addr = INADDR_ANY;
    memset( Dst: &serverAddress.sin_zero, Val: 0, Size: 8);
```

```
136     // Bind the server socket to the server address
137     if(bind( s: serverSocket,  name: (struct sockaddr*)&serverAddress,  namelen: sizeof(sockaddr)) != 0) {
138         cout << "Failed to bind server socket to server address" <<endl;
139         exit( Code: -1);
140     }
141
142     // Listen for incoming connections
143     if(listen( s: serverSocket,  backlog: 20) != 0) {
144         cout << "Server socket failed to Listen " << endl;
145         exit( Code: -1);
146     }
147
148     SOCKET clientSocket;
149     InitializeCriticalSection( lpCriticalSection: &cs);
150     while(true) {
151         cout << "Waiting for a client....." << endl;
152         clientSocket = accept( s: serverSocket,  addr: nullptr,  addrlen: nullptr);
153         if (clientSocket < 0) {
154             cout << "Failed to accept client " << endl;
155             continue;
156         }
157         cout << "A client connected successfully" << endl;
158         // create thread to handle client requests
159         thread( &: HandleTCPClient, clientSocket).detach();
```

```
161             // increment number of connections
162             EnterCriticalSection( lpCriticalSection: &cs);
163             number_of_active_connections++;
164             LeaveCriticalSection( lpCriticalSection: &cs);
165         }
166 }
167
```