



## Computer & Systems Engineering Department

Networks

### Assignment 1 report

<u>Name</u>	<u>ID</u>
Adel Mahmoud Mohamed Abdelrahman	20010769
Abdulrahman Aladdin Essayed	20010824

## *The client side:*

- The overall organization:

- The code for the client is broken into files and for each file there's a header that has the signatures of the functions in it and those headers are as follows:


```
#ifndef ERROR_HANDLING_H
#define ERROR_HANDLING_H

// function to handle the errors that happens due to user mistakes
// it notifies the user where that happened and then it exits the program
void DieWithUserMessage(const char *msg, const char *detail);

// function to handle the errors that happens due to errors in system calls
// it notifies the user where that happened and then it exits the program
void DieWithSystemMessage(const char *msg);

#endif // ERROR_HANDLING_H
```

```


#ifndef SEND_REQUEST_H
#define SEND_REQUEST_H


// the prototype of the send_get_request function
void send_get_request(int socket, const char *file_path);

// the prototype of the send_post_request function
void send_post_request(int socket, const char *file_path);

#endif

```

```


1  #ifndef HANDLE_RESPONSE_H
2  #define HANDLE_RESPONSE_H
3
4  //function to handle the response of the get request
5  void handle_get_response(char *response, char *file_name, int
    sock, size_t responseLength);
6
7  //function to handle the response of the Post request
8  void handle_post_response(char *response, char *file_name);
9
10 #endif // HANDLE_RESPONSE_H

```

And for each header file there's a corresponding **file.c** that defines those functions and we'll come to some of them later.

- And there's the file called **client.c** which contains the main function, and this is the file where the connection between the server and the client is made and then the communication begins.
- And there's a **makefile** to run the client-side code easily by simply:
  - Type **make** in the terminal this will compile all the .c files into object files
  - And then the main object file is called **my\_client** and it's of course executable so we shall execute it by typing **./my\_client** but we must also add at least the ipv4 address of the server and the optional Port

number which is 8080 as default if not provided in argv[] and here's the make file.

```
1 my_client
  : client.o error_handling.o send_request.o handle_response.o
2   gcc client.o error_handling.o send_request.o handle_respons
  e.o -o my_client
3
4 client.o: client.c
5   gcc -c client.c
6
7 error_handling.o: error_handling.c error_handling.h
8   gcc -c error_handling.c
9
10 send_request.o: send_request.c send_request.h
11   gcc -c send_request.c
12
13 handle_response.o: handle_response.c handle_response.h
14   gcc -c handle_response.c
15
16 clean:
17   rm *.o my_client
```

- And to run the program also we have to specify the desired operations in a text file called commands.txt in the following format for GET and POST.

- **The major functions:**

```
GET index.html
GET index.png
POST index.txt
GET index.jpeg
```

- this function is defined in client.c file and it forms the sockaddr\_in struct of the server (note that the address of the server must be provided in args and if the port is not, then it's 8080 by default)

```
1 struct sockaddr_in get_server_address(in_port_t portNumber, char
   *serverIPv4)
2 {
3     struct sockaddr_in server_address;
4     memset(&server_address, 0, sizeof(server_address));
5     // Zero out structure
6
7     // Configure server address
8     server_address.sin_family = AF_INET;
9     // to define the IPv4 family for the address
10    server_address.sin_port = htons(portNumber);
11    // to convert our port "8080" from the custom host byte order t
12    o the standardized network byte order (big-endian)
13
14    // Convert IP address to binary form and set it in the server ad
15    dress sin_addr attribute
16    // and this format is also in network byte order
17    int rtnVal = inet_pton(AF_INET, serverIPv4, &server_address.
18    sin_addr);
19    if (rtnVal == 0)
20    {
21        DieWithUserMessage("inet_pton() failed",
22        "invalid address string");
23    }
24    else if (rtnVal < 0)
25    {
26        DieWithSystemMessage("inet_pton() failed");
27    }
28
29    // if we got here then we've created the server address successf
30    ully
31    return server_address;
32 }
```

```

1 // fuction to send the Post request
2 void send_post_request(int socket, const char *filename)
3 {
4     // Determine content type based on file extension
5     const char *content_type;
6     if (strstr(filename, ".html") || strstr(filename, ".htm"))
7     {
8         content_type = "text/html";
9     }
10    else if (strstr(filename, ".txt"))
11    {
12        content_type = "text/plain";
13    }
14    else
15    {
16        content_type = "image/jpeg";
17    }
18
19    // Open the file
20    FILE *file = fopen(filename, "rb");
21    if (file == NULL)
22    {
23        perror("Error opening file, or maybe the file doesn't exist");
24        exit(1);
25    }
26
27    // Determine the file size
28    fseek(file, 0, SEEK_END);
29    long file_size = ftell(file);
30    fseek(file, 0, SEEK_SET);
31
32    // Construct and send the HTTP POST request headers
33    char headers[BUFFER_SIZE];
34    snprintf(headers, sizeof(headers),
35             "POST %s HTTP/1.1\r\n"
36             "Content-Type: %s\r\n"
37             "Content-Length: %ld\r\n"
38             "\r\n",
39             filename, content_type, file_size);
40
41    // the first send(the one that contains the header and the request line)
42    ssize_t bytes_sent = send(socket, headers, strlen(headers), 0);
43
44    if (bytes_sent == -1)
45    {
46        DieWithSystemMessage("send() failed");
47    }
48
49    if (bytes_sent != strlen(headers))
50    {
51        DieWithUserMessage("send()", "sent unexpected number of bytes");
52    }
53
54    // Send the file content in chunks of 1024 bytes
55    char buffer[BUFFER_SIZE];
56    size_t bytes_read;
57
58    while ((bytes_read = fread(buffer, 1, sizeof(buffer), file)) > 0)
59    {
60        ssize_t bytes_sent = send(socket, buffer, bytes_read, 0);
61        if (bytes_sent == -1)
62        {
63            perror("Error sending file content");
64            break;
65        }
66    }
67
68    // Close the file
69    fclose(file);
70 }
71

```

- Note that in the previous function `send_post_request()` we note that we specify in the header the length of the content to let the server know in advance what's the total size of the file that I will send so he can loop till he receives the whole file and the buffer size is set to 1024 bytes.
- And the same thing is done when handling the get response, first, the server sends me the header which contains the content length of the file then I loop till the whole file is received and here's a snap of that:

```

1  while (totalBytesRcvd < content_length)
2  {
3      char buffer[BUFFER_SIZE];
4      size_t numBytes = recv(sock, buffer, BUFFER_SIZE, 0);
5      if (numBytes < 0)
6          DieWithSystemMessage("recv() failed");
7
8      else if (numBytes == 0)
9          DieWithUserMessage("recv()", "connection closed prematurely");
10
11     totalBytesRcvd += numBytes;
12     fwrite(buffer, 1, numBytes, fp);
13 }
14 // here means that we've successfully recieved the whole file
15 // now we close our file
16 fclose(fp);

```

## • Data structures used:

- holding the address of the server in a struct of type `sockaddr_in`

```

1  struct sockaddr_in server_address;

```

- A buffer to hold the data sent and received of size `BUFFER_SIZE` which is defined constant of 1024 bytes

```

1  char buffer[BUFFER_SIZE] = {0};

```

## Server Side

## Overall organization

- The code is broken into .c and .h files, and there is also a makefile to compile the server code.
  - o Header files (.h) contain signature of the functions, struct, and global variables.
  - o C files (.c) contain the actual implementation of the functions
  - o Make file contains files to be compiled and the compilation options like Wall and Wextra.
  - o The main file is server.c, that is the file containing the main function of the program.

## Major functions

- handleClient(), called after establishing the connection successfully to run in a separate thread, it is responsible for receiving client's request then delegate the responsibility to another function to response to that request.
- handlePOST(), called by the handleClient() if the received request is POST, it is responsible for extracting the file name, receive the data and then write it on the disk.
- handleGET(), called by the handleClient() if the received request is GET, it is responsible for extracting the filename, then if it exists it sends end, else it responds with status code 404 not found.
- decrement\_active\_clients(), that decrements the number of active clients when a connection is closed or timed out.
- update\_timeout(), that updates the timeout when the number of active connections changes.

## Data Structure Used

- Struct params, it capsulates the client socket number, timeout pointer pointing on the global timeout of the system, and pointer to the active clients number, it is used to pass the socket data to its handling thread.
- Struct sockaddr in, holds the local address structure of the server.
- Struct timeval, holds the timeout.

## Bonus

The server performs well with real browsers

