Name: Sherif Mohamed Mostafa ID:20

“Programming assignment 1”

“Introduction to socket programming in C/C++”

Requirements:

In brief, the required is to reinvent the wheel by implementing a basic form (restricted subset) of the HTTP protocol, and using it to implement a web server and a web client that communicate using that protocol.

This includes:

- Implementing a multithreaded web server: the web server should accept incoming connection requests. It should then look for the GET request and pick out the name of the requested file then sends it to the client if it was found. If the request is POST then it sends OK message and waits for the uploaded file from the client. The server should also print the received request along with any optional lines till the blank line. The server responds with an OK message and the required file data (if file was found in case of a GET request) or a Not Found message if the file wasn’t found. The server is supposed to handle TXT, HTML and IMAGES. The server is run with the command ./my_server port_number. Either a multi-threaded or a multi-process could be used.
- Implementing an HTTP web client: the web client must read and parse a series of commands from input file. The commands syntax should be as follows, where file path is the path of the file on the server (including the file itself): client_get file-path host-name (port-number) client_post file-path host-name (port-number). Client shuts down when reaching end of file. Client should use reliable stream protocol SOCK_STREAM (corresponding to TCP)

and the internet domain protocol AF_INET (corresponding to IPv4). Client should run by the command ./my_client server_ip port_number.

- Adding simple HTTP/1.1 support to the web server: consisting of persistent connections and pipelining of client requests. Also, a heuristic is needed for the web server to know when to close a persistent connection after a timeout. This timeout needs to be configured in the server and ideally should be dynamic based on the number of other active connections the server is currently supporting. That is, if the server is idle, it can afford to leave the connection open for a relatively long period. If the server is busy, it may not be able to afford to have an idle connection sitting around (consuming kernel/thread resources) for very long.

Tools used:

- Programming language: C++.
- Compilation: used g++ and Makefile to compile.
- Text editor: Visual studio code.
- Operating system: used an Ubunto virtual machine (Using Oracle virtual box) on top of a windows 10 operating system. **Note that any error indicators appearing in the screen shots is because they were taken on windows for convenience.**

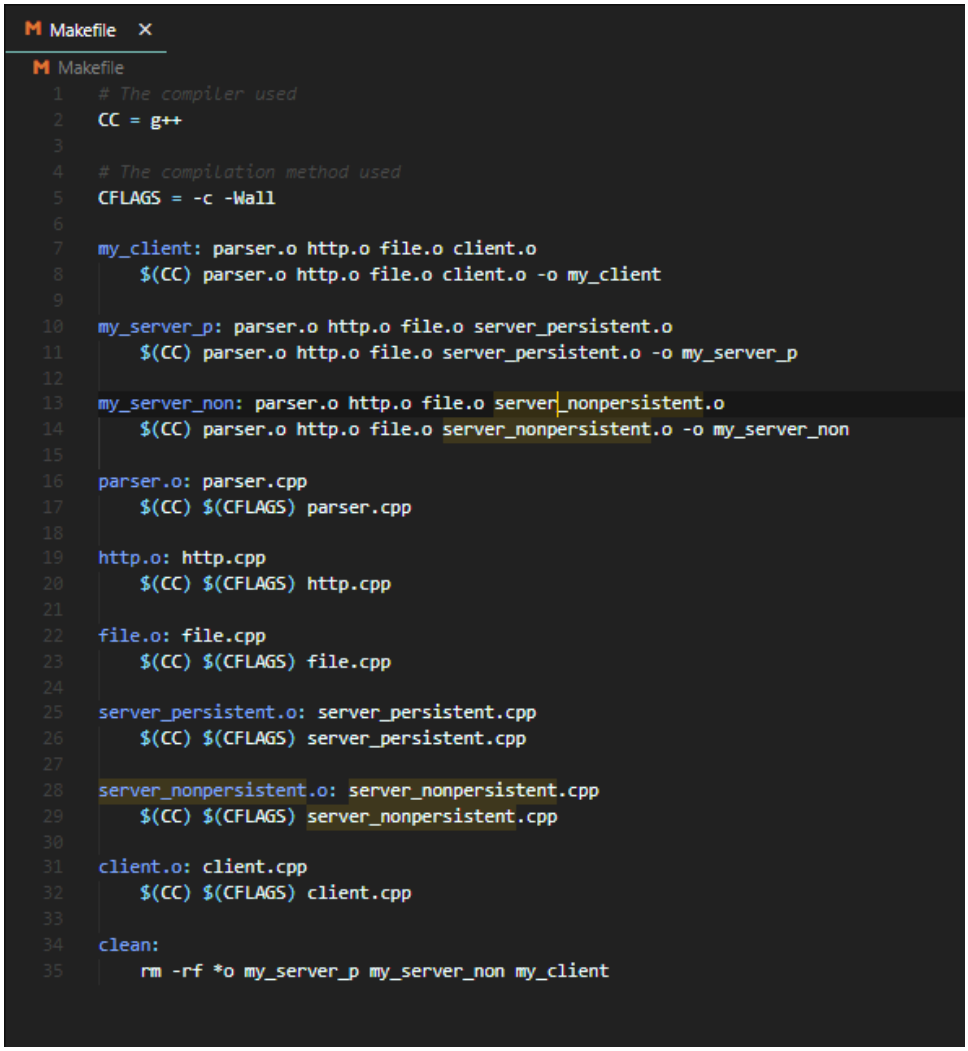
Overall organization:

Note that two sever versions were implemented one supports persistent connections and one that does not.

The program was organized into the following files:

Makefile:

- This file defines how the program is compiled and the generated executable files corresponding to the web server and the client.

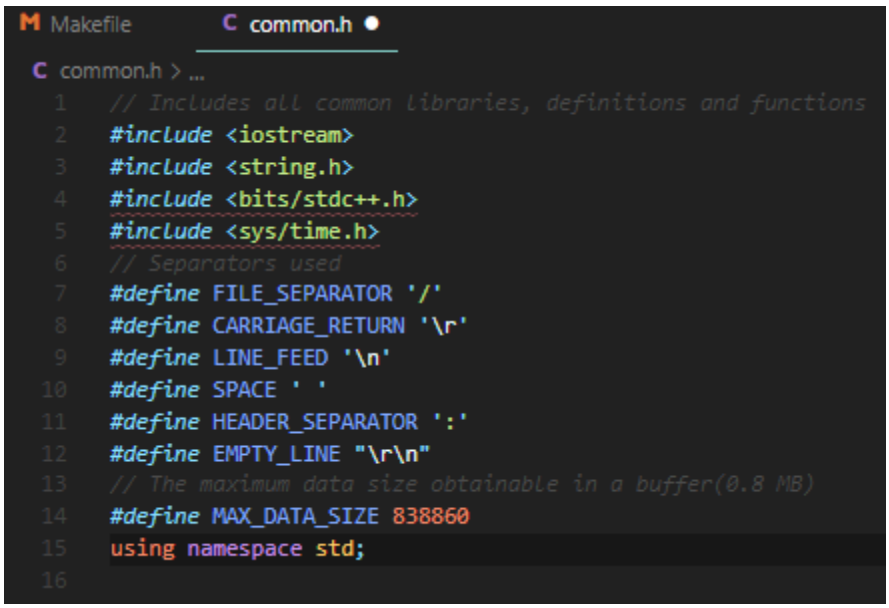


Header files:

These files define the functions and structures used, as well as any constants or library includes, they include:

Common.h:

- This file is used for the libraries commonly used to be included in it. It also defines some constants for file paths creation and HTTP messages creations. It defines a maximum size for buffered data.



HTTP.h:

- This header contains the libraries needed and definitions used for implementing HTTP. This includes a tailored parser library and all the includes in common.h. It contains constant definitions for parts of the HTTP message as the method, message code and status and HTTP versions.

```
// This header defines all the functions, inclusions and functions
// used in implementing the HTTP protocol.
// It also defines the important data types used
#include "parser.h"
// Definitions for request methods
#define GET_REQUEST "GET"
#define POST_REQUEST "POST"
// Definition for response messages
#define OK_MSG "OK"
#define OK_CODE "200"
#define NOT_FOUND_MSG "Not Found"
#define NOT_FOUND_CODE "404"
// Definitions for HTTP version
#define HTTP_0 "HTTP/1.0"
#define HTTP_1 "HTTP/1.1"
```

- It also includes type definitions for the following structures:
  - http_response: used for dealing with http response messages and keeps all the parts of the message to ease manipulation after parsing.

```
// The data structures required

// This structure is for keeping an HTTP response
// HTTP response format:
/*
    version Status_code Status_message
    header field name: value
    ..
    header field name: value

    entity body
*/
typedef struct
{
    // The http version used
    string version;
    // The status code
    string status_code;
    // The status message
    string status_message;
    // The headers as a map mapping header field name to its value
    map<string, string> headers;
    // The data provided in case of a response to a get request
    string entity_body;
} http_response;
```

- http_request: same as http_response but for http request messages.

```
// This structure is for keeping an HTTP request
// HTTP request format:
/*
    method URL version
    header field name: value
    ..
    header field name: value

    entity body
*/
typedef struct
{
    // The method in the request line
    // we support GET and POST requests
    string method;
    // The URL in the request line
    string url;
    // The version of the HTTP used (version http/1.1 for this assignment)
    string version;
    // A map mapping each header field to its value
    map<string, string> headers;
    // The data in the request body in case of response
    string entity_body;
} http_request;
```

- It has the following function declarations too:
  - Create_get_request : given the url for the web server (including the file path) and a map<string,string> of headers (header field name: value mapping), it creates the GET request by filling and returning an http_request struct.
  - Create_post_request: same as the preceding function but creates a POST request by filling and returning an http_request struct. It also requires the data sent to be added in the input.
  - Request_to_string: given an http_request struct it returns the string representation (text representation) of the request according to the HTTP message format:

Method url version\r\n
Header field name:value\r\n
……
\r\n
Entity body.

- String_to_request: given a request in string format (assumed to be in the correct previous format), this function parses the string input by the help of parser.h functionalities then it creates the corresponding http_request struct and returns it.

```
// The functions implemented for the requests
http_request create_get_request(string url, map<string, string> headers);
http_request create_post_request(string url, string data, map<string, string> headers);
string request_to_string(http_request request);
http_request string_to_request(string request_string);
```

- Create_ok_response: given the data that was asked for in the request (if this response is for a GET request, otherwise input data is an empty string “”) as well as a map of headers, it fills and returns the corresponding http_response struct for an OK response.
- Create_not_found_response: same as the preceding function but data isn’t required as input and the retuned http_response struct corresponds to a Not found response.
- string_to_response: parses the input string then fills and returns an http_response. This is done by the help of the functionalities provided in parser.h.
- response_to_string: given the http_response struct, it creates the response message as a string (text format) and returns it. This is done according to the HTTP response message format:

version status_code status_message\r\n

header field name:value\r\n

..

Entity body

```
// The functions implemented for responses
http_response create_ok_response(string data, map<string, string> headers);
http_response create_not_found_response(map<string, string> headers);
string response_to_string(http_response response);
http_response string_to_response(string response_string);
```

Parser.h:

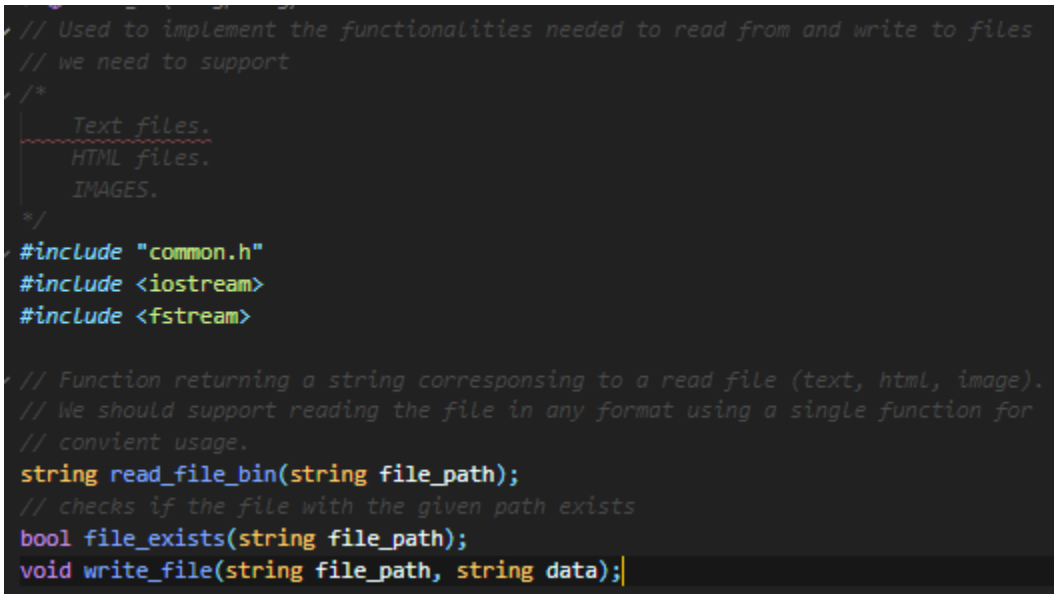
- This header includes the definitions for two functions used in parsing a string.
    - Split_to_lines: given a string it returns a vector of strings where each entry i in the vector corresponds to the ith line in the input string. The reason of implementing it as a separate function is that the lines in the http message are separated by “\r\n” not just “\n”.
    - Split_to_words: same as previous but splits the string according to a single character delimiter.

```
#include "common.h"
// This function takes a request string and splits it into components (line + headers + data)
vector<string> split_to_lines(string request);
// This function takes the string and splits it into words
vector<string> split_to_words(string line,char delim);
```

File.h:

- This header contains the functions needed for dealing with files, it uses ofstream, ifstream and ostringstream:
    - Read_file_bin: given a file path, it reads the file in binary format and returns a string. This function supports reading any file of a suitable length of any type (html, txt, image, pdf, C++…).

- File_exists: given the file path, this function checks whether the file exists or not.
- Write_file: given the file path and the data to be written in the file, it writes the file. If the file doesn't exist, it is created by default (to support POST requests).



```
// Used to implement the functionalities needed to read from and write to files
// we need to support
/*
    Text files.
    HTML files.
    IMAGES.
*/
#include "common.h"
#include <iostream>
#include <fstream>

// Function returning a string corresponsing to a read file (text, html, image).
// We should support reading the file in any format using a single function for
// convient usage.
string read_file_bin(string file_path);
// checks if the file with the given path exists
bool file_exists(string file_path);
void write_file(string file_path, string data);
```

Server.h:

- This header contains the libraries and function definitions needed to implement a simple HTTP web server. It also includes the needed constant definitions. These libraries support error logging, socket programming, multiprocessing as well as the tailored libraries http.h, file.h and common.h.

```c
// Defines the necessary functions and libraries needed for
// implementing the multithreaded server.
#include "common.h"
#include "http.h"
#include "file.h"

#include <unistd.h>
#include <errno.h>
// The includes needed for socket programming
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
// The includes needed for multiprocessing
#include <sys/wait.h>
#include <signal.h>
```

- The constants include a default port number 2000 (as those less than 1024 mainly require privileges) and a backlog value of 100 (how many queuing connections to handle at most).

```c
// chosen a random initial port larger than 1024 to prevent any required privelege constraints
#define DEFAULT_PORT "2000"
#define BACKLOG 100 // how many pending connections queue will hold
```

- The functions declared are:
    - Sigchild_handler: this function is assigned to sigaction struct to clear zombie processes without blocking.
    - Get_server_fd: given the port number, it returns a file descriptor for a socket with the machine IP and this port number for the server to listen on.
    - Reap_zombies: it sets the signal child handler implemented to reap the zombie processes.
    - Handle_connections: given the file descriptor for the socket on which the server is listening, it corresponds to an infinite loop in which the server accepts incoming connections and delegates work with the requests to a process.
    - Handle_request: given the http_request struct and client file descriptor, it handles the request by determining its method

and contents then sends the corresponding suitable response.

- Handle_spaces: given a url obtained with spaces ie: contains "%20" it returns a string replacing all "%20" with a space.

```
// Functions implemented

// A handler for children processes that just clears zombie processes without blocking.
void sigchild_handler(int s);
// Returns the socket file descriptor of the socket at which the server is listening
int get_server_fd(string port_number);
// Sets the signal child handler to reap zombie children
void reap_zombies();
// The main infinite loop where the server is handling the incoming connections and requests
void handle_connections(int listen_fd);
// functions that handles a received request
void handle_request(http_request request, int client_fd);
// handles spaces in url
string handle_spaces(http_request request);
```

Client.h:

- This header is used for implementing a simple HTTP client, it contains the libraries needed for socket programming, error handling and also http.h, file.h and common.h

```
// Defines the necessary functions and libraries needed for
// implementing the client.
#include "common.h"
#include "http.h"
#include "file.h"

#include <unistd.h>
#include <errno.h>
// The includes needed for socket programming
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
```

- It contains constants for defining the default server port to connect to (80), a path for the file containing the requests as well as constant definitions for request methods format in the requests file.

```
#define DEFAULT_PORT "80"
// defines the file at which the client requests are located
#define REQUESTS_PATH "client/commands"
#define CLIENT_GET "client_get"
#define CLIENT_POST "client_post"
```

- It contains the declarations for the following functions:
    - Connect_to_server: takes input the server IP and port number to which the client connects, connects to the server and returns the file descriptor for the socket at which it connected to the server if no error occurred and connection was accepted at the server. It returns -1 in case an error occurred.
    - Send_request: takes the file descriptor for the socket to which the client connected to with the server and an http_request struct, it converts the request to a string and sends it to the server.
    - Handle_response: takes input http_response and http_request that was sent corresponding to it, it handles the response by displaying whether the file was found or not in case of a response to a GET request and writes the data to the corresponding file if the response has data.

```
// Functions implemented

// returns the file descriptor resulting from this connection
int connect_to_server(string server_ip, string port_number);
http_response send_request(int server_fd,http_request request);
void handle_response(http_response response, http_request request);
```

Cpp files:

Here I note the Cpp files used for implementation and any important implementation details.

Parser.cpp

- Implements parser.h, mainly used to parse a string and return a vector of strings according to certain delimiters. Has a function tailored to obtain a request or a response string as a vector of lines where lines are separated by “\r\n” combination.

```
vector<string> split_to_lines(string request)
{
    int request_length = request.length();
    // We split by finding cr and lf
    // finding 2 cr lf after each other corresponds to the empty line
    int string_begining = 0;
    vector<string> result;
    for (int i = 1; i < request_length; i++)
    {
        // The case where a line feed and a carriage return was obtained
        if (request[i - 1] == CARRIAGE_RETURN && request[i] == LINE_FEED)
        {
            // The case where they represent the empty line
            if (string_begining == i - 1)
            {
                result.push_back(EMPTY_LINE);
                result.push_back(request.substr(i + 1, request_length - i));
                break;
            }
            // other wise push the line and advance the beggining
            result.push_back(request.substr(string_begining, i - string_begining - 1));
            string_begining = i + 1;
        }
    }
    return result;
}
```

File.cpp:

- Implements file.h, makes use of ostringstream to read a file in binary and then return it as a string to support reading any type of file.

```
string read_file_bin(string file_path)
{

    // oppening the file by using a file stream
    ifstream file(file_path, ios::binary);
    // using a string stream to obtain the read data as a string
    ostringstream ostrm;
    // reading the file in binary format
    ostrm << file.rdbuf();
    // converting to string format
    string content_as_string(ostrm.str());
    file.close();
    return content_as_string;
}

// Uses ifstream to check
bool file_exists(string file_path)
{
    ifstream file(file_path);
    string s;
    bool exists = file.good();
    file.close();
    return exists;
}

// Uses ifstream to write to the file
void write_file(string file_path, string data)
{
    ofstream file(file_path);
    file << data;
    file.close();
}
```

http.cpp

- Implements http.h, uses parser.h to parse a string and obtain the corresponding http_request or http_response.

```
http_request create_post_request(string url, string data, map<string, string> headers)
{
    // we just initialize the HTTP request struct and return it
    http_request request;
    request.url = url;
    request.method = POST_REQUEST;
    request.version = HTTP_1;
    request.headers = headers;
    request.entity_body = data;
    return request;
}

string request_to_string(http_request request)
{
    // Create the request in a single string as is defined in the .H file
    string res;
    // creating the request line
    string request_line = request.method + SPACE + request.url + SPACE + request.version + CARRIAGE_RETURN + LINE_FEED;
    // Creating the header lines
    string headers = "";
    // Iterating through the lines
    map<string, string>::iterator it;
    for (it = request.headers.begin(); it != request.headers.end(); it++)
    {
        headers += it->first + HEADER_SEPARATOR + it->second + CARRIAGE_RETURN + LINE_FEED;
    }
    // Creating the result string
    // Don't forget the empty line :)
    res = request_line + headers + EMPTY_LINE + request.entity_body;
    return res;
}
```

```
http_request string_to_request(string request_string)
{
    // splitting the request string to be line by line
    vector<string> request_by_line = split_to_lines(request_string);
    http_request result;
    // using the request line
    vector<string> request_line = split_to_words(request_by_line[0], SPACE);
    result.method = request_line[0];
    result.url = request_line[1];
    result.version = request_line[2];
    // Obtaining the headers
    int i = 1;
    while (request_by_line[i] != EMPTY_LINE)
    {
        vector<string> current_header = split_to_words(request_by_line[i], HEADER_SEPARATOR);
        result.headers[current_header[0]] = current_header[1];
        i++;
    }
    // if there is data keep it
    i++;
    if (i == (int)request_by_line.size() - 1)
    {
        result.entity_body = request_by_line[i];
    }
    return result;
}
```

Server_persistent.cpp

- Implements a persistent version of server.h.
- The main function obtains the server port number from the cmd or uses the default if it wasn't specified, it then obtains the file descriptor using get_server_fd, then calls reap_zombies and remains in infinite loop by calling hande_connections.

```
// The main function for running the server
// Should contain an infinite loop where the listener keeps
// listening to the incoming connection requests
int main(int argc, char **argv)
{
    // In case a port number was specified we initialize it
    // Server is called by ./my_server port_number
    if (argc > 1)
    {
        port_number = argv[1];
    }
    cout << "Server running on port " << port_number << endl;
    // file descriptor of the socket at which this server is listening
    int listen_fd = get_server_fd(port_number);
    if (listen_fd == -1)
    {
        fprintf(stderr, "error occured in obtainnig the server fd.\n");
        exit(1);
    }
    // initialize first the signal handler to reap te zombie processes
    reap_zombies();
    cout << "Server started listening on localhost at port number " << port_number << endl;
    cout << "waiting for connections...." << endl;
    handle_connections(listen_fd);
    return 0;
}
```

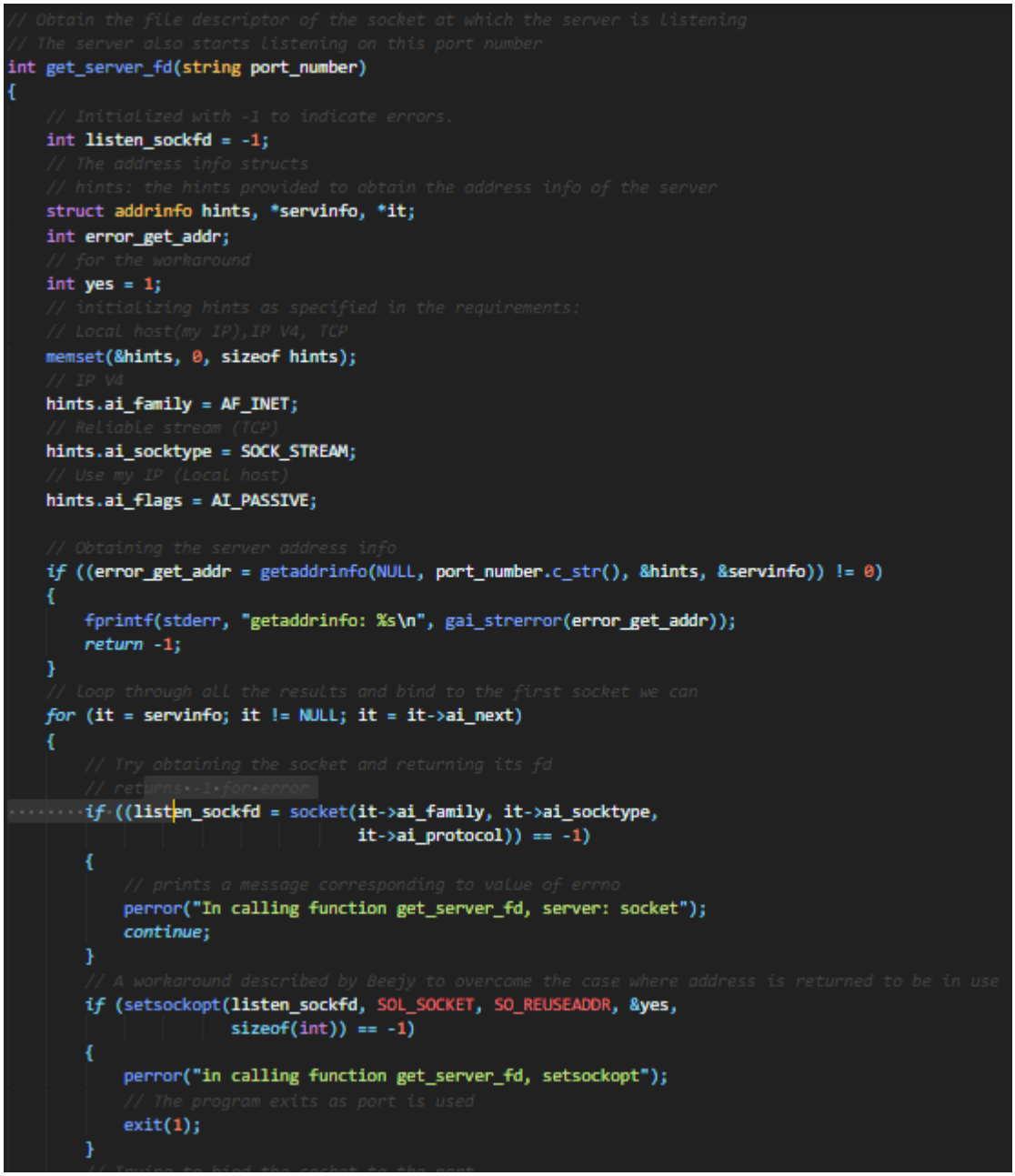
- Handling zombie processes is by reaping them without blocking, parent waits till all its child processes finish to prevent zombie processes.

```c
// A handler to handle zombie children
// n this way the parent wait for any child processes (pid = -1)
// and while there are zombie process (waitpid() return value is >0)
// it keep looping on calling wait.
void sigchild_handler(int s)
{
    (void)s; // quiet unused variable warning

    // waitpid() might overwrite errno, so we save and restore it:
    int saved_errno = errno;

    while (waitpid(-1, NULL, WNOHANG) > 0)
        ;

    errno = saved_errno;
}
```

```c
void reap_zombies()
{
    struct sigaction s;
    // reap all dead processes using the implemented sigchild handler
    s.sa_handler = sigchild_handler;
    sigemptyset(&s.sa_mask);
    s.sa_flags = SA_RESTART;
    if (sigaction(SIGCHLD, &s, NULL) == -1)
    {
        perror("when calling reap_zmbies, sigaction");
        exit(1);
    }
}
```

- To get the server fd:
    - Set a hint addrinfo struct to have AF_INET as ai_family (IPv4), SOCK_STREAM as ai_socktype (TCP) and AI_PASSIVE as ai_flags (to use the machine's IP (localhost)).
    - Obtain the server address info using the previous hints and the port number provided in function input.
    - The returned is a linked list of addrinfo, loop through them and try finding one that we can create its socket, the address (port number) isn't used and that we can bind the port to. If no address worked, exit and show error.
    - Start listening to this socket using the obtained file descriptor after having created and bind to it and return this

file descriptor to start handling incoming connections and requests from clients.



```
// Obtain the file descriptor of the socket at which the server is listening
// The server also starts listening on this port number
int get_server_fd(string port_number)
{
    // Initialized with -1 to indicate errors.
    int listen_sockfd = -1;
    // The address info structs
    // hints: the hints provided to obtain the address info of the server
    struct addrinfo hints, *servinfo, *it;
    int error_get_addr;
    // for the workaround
    int yes = 1;
    // initializing hints as specified in the requirements:
    // Local host(my IP),IP V4, TCP
    memset(&hints, 0, sizeof hints);
    // IP V4
    hints.ai_family = AF_INET;
    // Reliable stream (TCP)
    hints.ai_socktype = SOCK_STREAM;
    // Use my IP (Local host)
    hints.ai_flags = AI_PASSIVE;

    // Obtaining the server address info
    if ((error_get_addr = getaddrinfo(NULL, port_number.c_str(), &hints, &servinfo)) != 0)
    {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(error_get_addr));
        return -1;
    }
    // Loop through all the results and bind to the first socket we can
    for (it = servinfo; it != NULL; it = it->ai_next)
    {
        // Try obtaining the socket and returning its fd
        // returns -1 for error
        if ((listen_sockfd = socket(it->ai_family, it->ai_socktype,
                                    it->ai_protocol)) == -1)
        {
            // prints a message corresponding to value of errno
            perror("In calling function get_server_fd, server: socket");
            continue;
        }
        // A workaround described by Beejy to overcome the case where address is returned to be in use
        if (setsockopt(listen_sockfd, SOL_SOCKET, SO_REUSEADDR, &yes,
                       sizeof(int)) == -1)
        {
            perror("in calling function get_server_fd, setsockopt");
            // The program exits as port is used
            exit(1);
        }
        // Trying to bind the socket to the port
```

```
    // Trying to bind the socket to the port
    if (bind(listen_sockfd, it->ai_addr, it->ai_addrlen) == -1)
    {
        close(listen_sockfd);
        perror("in calling function get_server_fd, server: bind");
        // Try the next socket
        continue;
    }
    // If suitable socket found break
    break;
}
// Freeing the structure as it is not needed any more
freeaddrinfo(servinfo);
// If no suitable socket was found
if (it == NULL)
{
    fprintf(stderr, "server: failed to find a suitable socket to bind\n");
    exit(1);
}
// Try listening on this socket
if (listen(listen_sockfd, BACKLOG) == -1)
{
    perror("In calling function get_server_fd, listen");
    exit(1);
}
return listen_sockfd;
```

- To handle incoming connections:
    - Keep waiting to accept a new connection from a client then get its file descriptor and address information (done by the listener or parent process).
    - Make sure that the IP address provided in the client address info is of version 4 otherwise ignore this connection requests.
    - Create a child process and delegate handing the requests to it, close the clients file descriptor in the parent process as it doesn't need it. This offers pipelining functionality in the server as each client has its own dedicated process.
    - The child process keeps receiving the requests sent by the client, prints them and handles them. It exits and closes the connection when the client closes the connection. ***Timeouts weren't implemented as was required which is a shortcoming of this implementation, could be handled by use of select().***

- ***A multi-process approach was used because of the following:***
  - **Portability, unlike threads that depend on certain libraries.**
  - **Reliability, processes can be considered more reliable than threads.**
  - **It helps in increasing CPU utilization whereas multithreading would excel more in case a process could have several parallel worker threads doing it (as matrix multiplication for example).**
  - **Multi-processing can benefit from adding more CPUs.**
  - **Processes here don't need to share memory as each process is delegated to work with a client so no need for the shared address space property provided by multi-threading.**
- ***Problems of this approach:***
  - **Heavy creation time and context switching time compared to multi-threading (more overhead).**

```
// no timeouts at first then would be updated by use of select()
void handle_connections(int listen_fd)
{
    // The fd of the client trying to connect
    int client_fd;
    // keeps the client's address information
    struct sockaddr_storage client_addr;
    socklen_t sin_size = sizeof client_addr;

    while (1)
    {
        // Listener (parent) process would accept a connection if one exists
        // Then it would delegate the handling to a child process
        client_fd = accept(listen_fd, (struct sockaddr *)&client_addr, &sin_size);
        if (client_fd == -1)
        {
            perror("In handle_connections, accept");
            continue;
        }
        // To obtain the correct byte ordering (endian)
        // consider IPV4 only
        char s[INET_ADDRSTRLEN];
        if (client_addr.ss_family == AF_INET)
        {
            inet_ntop(AF_INET, &(((struct sockaddr_in *)&client_addr)->sin_addr),
                      s, sizeof s);
            cout << "Accepted connection from: " << s << endl;
        }
        else
        {
            close(client_fd);
            cout << "server supports only IPV4" << endl;
            continue;
        }
        // Creating a child process to delegate the work with the accepted connection to it
```

```
// Creating a child process to delegate the work with the accepted connection to t
if (!fork())
{
    // This is the child worker process
    // Receiving the client request to process it
    int num_received;
    char buf[MAX_DATA_SIZE];
    while ((num_received = recv(client_fd, buf, MAX_DATA_SIZE - 1, 0)) > 0)
    {
        // assuring that the buffer ends with a null character
        buf[num_received] = '\0';
        // parsing the received request to handle it
        // get and post requests are handled
        string request_string = string(buf);
        cout << "received request from " << s << " :" << endl
             << request_string << endl;
        // handling the request
        http_request request = string_to_request(request_string);
        handle_request(request, client_fd);
    }
    if (num_received == -1)
    {
        perror("recv");
        close(client_fd);
        exit(1);
    }
    else if (num_received == 0)
    {
        cout << "closing connection for socket " << client_fd << endl;
        close(client_fd);
        // Exit the child process for now
        exit(0);
    }
}
// parent doesn't need this for now connection is non-persistent
close(client_fd);
```

- To handle a request:
    - In case the request was a GET request:
        - Obtain the file path from the url.
        - If no file path found, consider it to be index.html (the default file).
        - If file doesn’t exist, create a not found response, convert it to a string and send it to the client using the client file descriptor provided as input to the function.
        - Other wise, create an OK response, read the file and send its content in the response entity body.
        - while the file isn’t sent completely, keep sending the remaining file content.

- In case the request was a POST request.
  - Create an OK response and convert it to a string.
  - Add a file with the specified file path in the post folder (folder in server location containing all the data obtained by a POST request).
  - Send the OK response to the client.

```
void handle_request(http_request request, int client_fd)
{
    request.url = handle_spaces(request);
    // get request
    // Send ok response and send the file data if found
    // else send a not found response
    if (request.method == GET_REQUEST)
    {
        string file_path = request.url.substr(1, request.url.length());
        if (file_path.length() == 0)
        {
            file_path = "index.html";
        }
        // may be required
        map<string, string> headers;
        // if file not found
        if (!file_exists(file_path))
        {
            http_response response = create_not_found_response(headers);
            string response_string = response_to_string(response);
            if (send(client_fd, response_string.c_str(), response_string.length(), 0) == -1)
            {
                perror("handle_request, send");
                return;
            }
        }
```

```
    // if file exists, send the data in the file
    else
    {
        // creating an OK response with no data still
        http_response response = create_ok_response("", headers);
        int empty_response_length = response_to_string(response).length();
        // reading the file to send it in chuncks
        string file_data = read_file_bin(file_path);
        response.entity_body = file_data;
        // sending the first packet
        string response_string = response_to_string(response);
        int num_bytes;
        if ((num_bytes = send(client_fd, response_string.c_str(), response_string.length(), 0)) == -1)
        {
            perror("handle_request, send");
            return;
        }
        // if the file wasn't completely sent, send the remaining of it
        int sent = num_bytes - empty_response_length;
        while (sent < (int)file_data.length())
        {
            if ((num_bytes = send(client_fd, file_data.substr(sent, file_data.length() - sent).c_str(), file_data.length() - sent, 0)) == -1)
            {
                perror("handle_request, send");
                return;
            }
            sent += num_bytes;
        }
```

```
// post request
// Just send ok response and write received data into the file corresponding to this client
if (request.method == POST_REQUEST)
{
    // may be required
    map<string, string> headers;
    // creating an OK response with no data
    http_response response = create_ok_response("", headers);
    string response_string = response_to_string(response);
    // writing the request entity body into a file
    string file_path = "post" + request.url;
    write_file(file_path, request.entity_body);
    if (send(client_fd, response_string.c_str(), response_string.length(), 0) == -1)
    {
        perror("handle_request, send");
        return;
    }
}
```

Server_nonpersistent.cpp

- Implements server.h, it was the initial server implementation. It opens a connection for each request. This is done by making the delegated process handle a single request and then close the connection. ***This was helpful in testing with the browser as I found that the browser tries to connect with each request.***

```
// Creating a child process to delegate the work with the accepted connection to it
if (!fork())
{ // this is the child worker process
    // child doesn't need the listener
    close(listen_fd);
    // Receiving the client request to process it
    int num_received;
    char buf[MAX_DATA_SIZE];
    if ((num_received = recv(client_fd, buf, MAX_DATA_SIZE - 1, 0)) == -1)
    {
        perror("recv");
        exit(1);
    }
    // assuring that the buffer ends with a null character
    buf[num_received] = '\0';
    // parsing the received request to handle it
    // get and post requests are handled
    string request_string = string(buf);
    cout << "received request from " << s << " :" << endl
         << request_string << endl;
    // handling the request
    http_request request = string_to_request(request_string);
    handle_request(request, client_fd);
    close(client_fd);
    // Exit the child process for now
    exit(0);
}
```

Client.cpp

- Implements client.h
    - The main function obtains the server IP as the first argument, it exits and displays an error message if the server IP wasn't specified. It also obtains the server port number as the second argument if specified (otherwise it is set to the default port number 80). Then it obtains the file descriptor of the socket to which it will send requests and receive responses by connecting to the server. It reads the request commands file and parses it to lines. It then sends each request to the server after building it correctly (I assumed a constant line to be sent in the POST requests as it wasn't clear where to get the data). After sending the request, it receives and handles the response. After finishing all the requests, the client closes the connection and shuts down.

```
// The main loop is here
int main(int argc, char **argv)
{

    if (argc == 1)
    {
        cout << "You must specify the server IP" << endl;
        exit(1);
    }
    string server_ip = argv[1];
    // The port to which we connect
    string port_number = DEFAULT_PORT;
    // In case a port number was specified we initialize it
    // client is called by ./my_client server_ip port_number
    if (argc > 2)
    {
        port_number = argv[2];
    }
    // Connecting to the server
    int server_fd = connect_to_server(server_ip, port_number);
    // obtaining all the requests from the file to send and receive with the server
    string requests_all = read_file_bin(REQUESTS_PATH);
    // sending each request and receiving the response as specified by the pseudo code
    vector<string> request_each = split_to_words(requests_all, '\n');

    // Empty headers as this client sends no headers
    map<string, string> headers;
```

```
// handle each request and then close the connection
for (int i = 0; i < (int)request_each.size(); i++)
{
    string curr_req = request_each[i];
    // splitting the line into parts to process it
    vector<string> request_comp = split_to_words(curr_req, SPACE);
    string request_type = request_comp[0];
    string request_url = request_comp[1];
    // building the request sending it and handling the response
    if (request_type == CLIENT_GET)
    {
        http_request request = create_get_request(request_url, headers);
        http_response response = send_request(server_fd, request);
        handle_response(response, request);
    }
    else if (request_type == CLIENT_POST)
    {
        http_request request = create_post_request(request_url, "Post body data", headers);
        http_response response = send_request(server_fd, request);
        handle_response(response, request);
    }
}
// finished all requests so close the connection
cout << "client shutting down" << endl;
close(server_fd);
return 0;
```

- To send a request:
    - Client creates a string corresponding to the http_request and sends it to the server.
    - It then receives the response and converts it to http_response then returns it to be used in handle_response.

```
// This function sends a request and then receives and returns a response
http_response send_request(int server_fd, http_request request)
{
    http_response response;
    // send the request
    string request_string = request_to_string(request);
    if (send(server_fd, request_string.c_str(), request_string.length(), 0) == -1)
    {
        perror("send_request, send");
        exit(1);
    }
    // Receive the response and return it
    char buf[MAX_DATA_SIZE];
    int num_received;
    if ((num_received = recv(server_fd, buf, MAX_DATA_SIZE - 1, 0)) > 0)
    {
        buf[num_received] = '\0';
        response = string_to_response(string(buf));
    }
    else if (num_received == -1)
    {
        perror("recv");
        close(server_fd);
        exit(1);
    }
    else if (num_received == 0)
    {
        cout << "connection was closed " << server_fd << endl;
        close(server_fd);
        // Exit the child process for now
        exit(0);
    }
    return response;
}
```

- Handling the response:
    - Print whether the file was found or not (in case of GET request).
    - In case the response has data, write it in the corresponding file (file path obtained from the request url) in a folder client/ in the client directory, which keeps all the files obtained by the client using GET requests as well as the commands file.

```
void handle_response(http_response response, http_request request)
{
    // handle not found
    if (response.status_code == NOT_FOUND_CODE)
    {
        cout << "File not found: " << request.url << endl;
    }
    else
    {
        // If response has data write it
        if (response.entity_body.length() > 0)
        {
            cout << "File found: " << request.url << endl;
            vector<string> splitted = split_to_words(request.url, '/');
            string file_path = "client/" + splitted[(int)splitted.size() - 1];
            // In case the index.html was requested
            if(splitted[(int)splitted.size() - 1] == "/")
                file_path += "index.html";
            write_file(file_path, response.entity_body);
        }
    }
}
```

- To connect to a server:
    - A similar approach is used as in get_server_fd in server_persistent but instead of bind we call connect, if a suitable addrinfo was found and so connection was successful, we return the corresponding file descriptor. Otherwise -1 is returned.

```
// This functions starts a TCP connection with the server
int connect_to_server(string server_ip, string port_number)
{
    // The socket file descriptor that we will return
    int server_fd;
    struct addrinfo hints, *server_info, *p;
    // To check if an error occured when obtaining the address info of the server
    int address_status;
    // Will be used to print the IP of server we connected to
    char s[INET_ADDRSTRLEN];
    // Setting the hints structure to zero (it will be used to help in
    // connecting to the server with correct protocols)
    memset(&hints, 0, sizeof hints);
    // IP V4
    hints.ai_family = AF_INET;
    // TCP
    hints.ai_socktype = SOCK_STREAM;

    if ((address_status = getaddrinfo(server_ip.c_str(), port_number.c_str(), &hints, &server_info)) != 0)
    {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(address_status));
        return 1;
    }

    // Loop through all the results and connect to the first we can
    for (p = server_info; p != NULL; p = p->ai_next)
    {
        if ((server_fd = socket(p->ai_family, p->ai_socktype,
                                p->ai_protocol)) == -1)
        {
            perror("connect_to_server, client: socket");
            continue;
        }
        // Try connecting to this socket
        if (connect(server_fd, p->ai_addr, p->ai_addrlen) == -1)
        {
            perror("connect_to_server, client: connect");
            close(server_fd);
            continue;
        }

        break;
    }
```

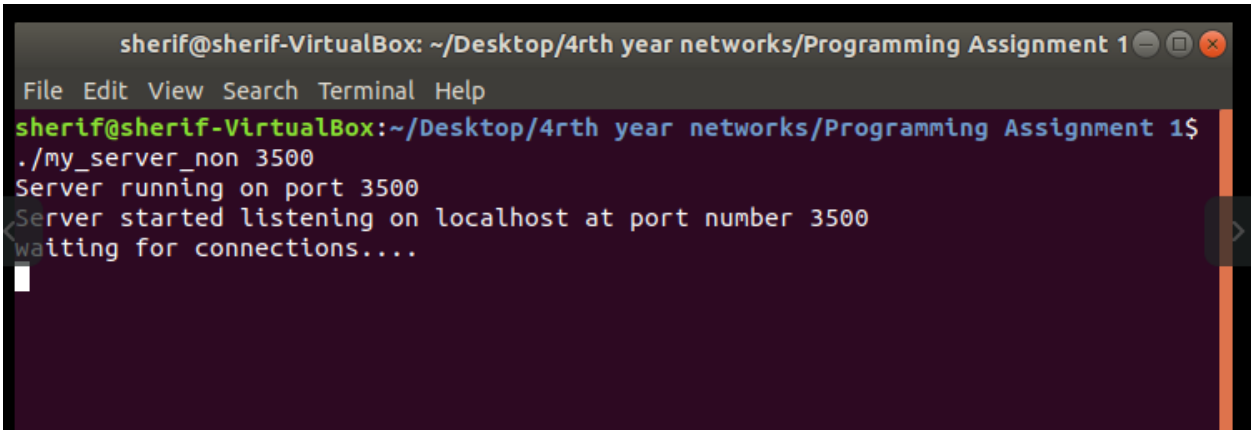
```
    // Check if we weren't able to connect to any socket
    if (p == NULL)
    {
        fprintf(stderr, "client: failed to connect\n");
        return 2;
    }

    inet_ntop(AF_INET, &(((struct sockaddr_in *)&p)->sin_addr),
              s, sizeof s);
    cout << "client connected to " << s << " at port " << port_number << endl;
    // all done with this structure
    freeaddrinfo(server_info);
    return server_fd;
```

Sample runs:

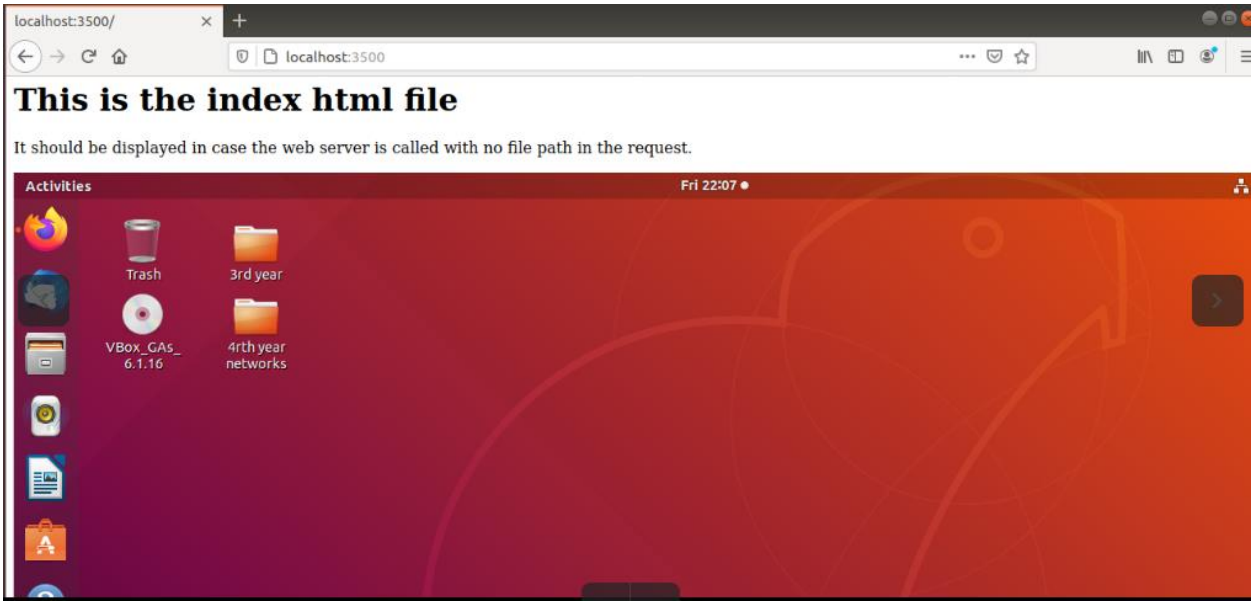
Bonus part:

Testing the web server with a real web browser **(Firefox)**:

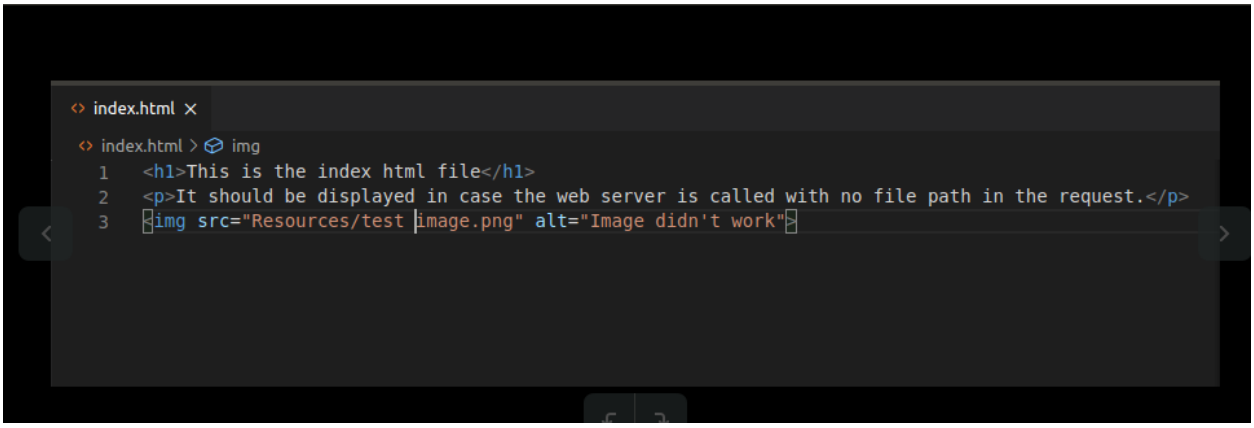
Running the non-persistent server:



Calling localhost/3500



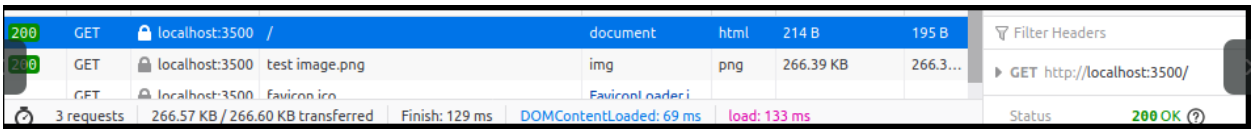
Corresponding index.html and image:



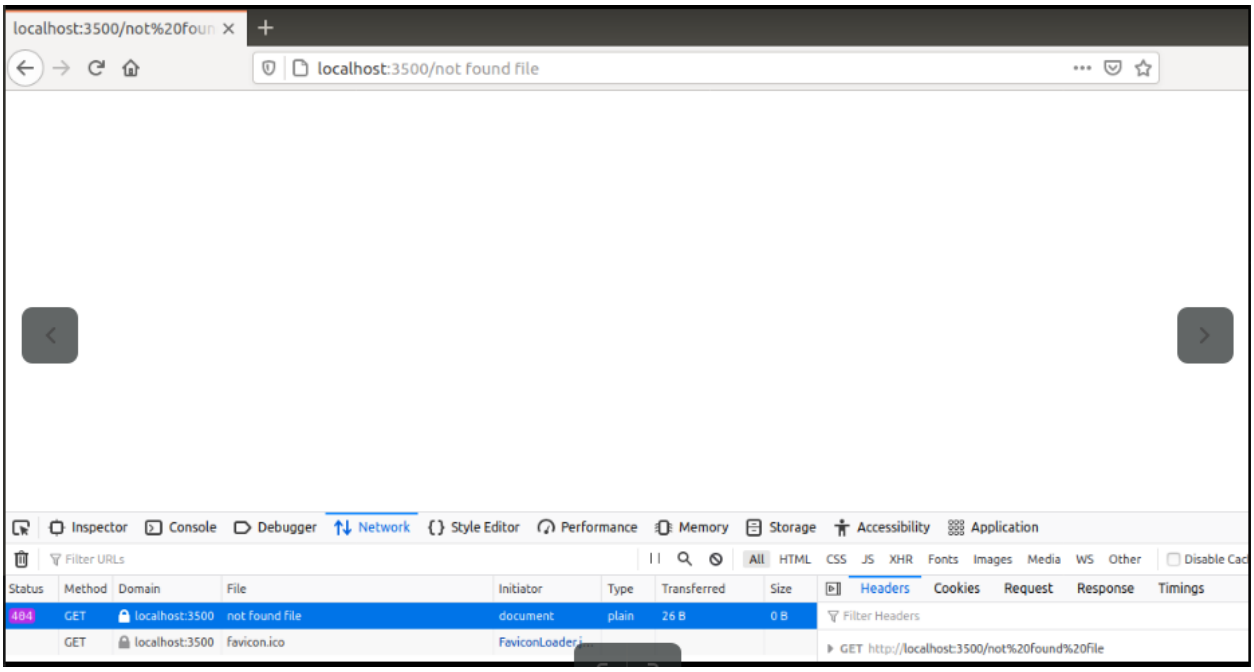
Corresponding requests received for the server:

```
Accepted connection from: 127.0.0.1
received request from 127.0.0.1 :
GET / HTTP/1.1
Host: localhost:3500
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:83.0) Gecko/20100101 Firefox/83.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1


Accepted connection from: 127.0.0.1
received request from 127.0.0.1 :
GET /Resources/test%20image.png HTTP/1.1
Host: localhost:3500
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:83.0) Gecko/20100101 Firefox/83.0
Accept: image/webp,*/*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Referer: http://localhost:3500/


Accepted connection from: 127.0.0.1
received request from 127.0.0.1 :
GET /favicon.ico HTTP/1.1
Host: localhost:3500
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:83.0) Gecko/20100101 Firefox/83.0
Accept: image/webp,*/*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
```

Corresponding response received by the browser:

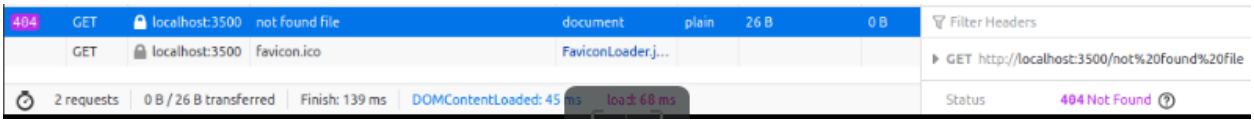


Calling a not found file:

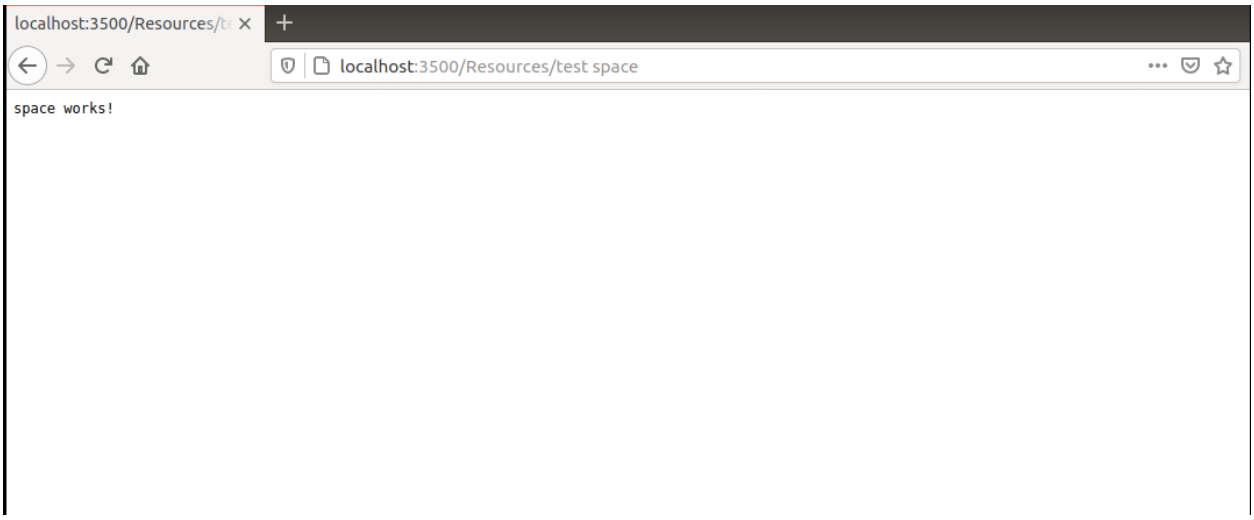


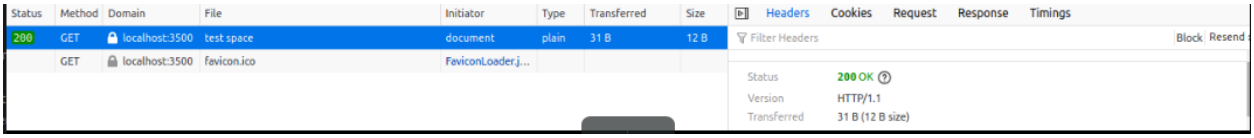
Received request:

Corresponding response status:

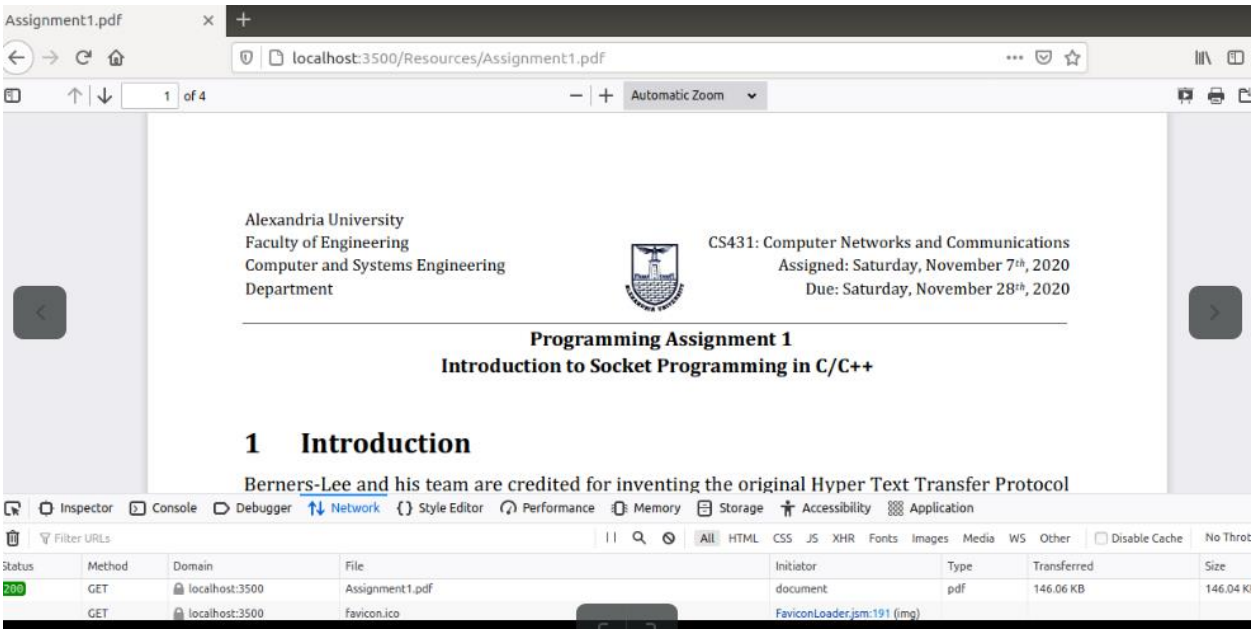


Calling a text file:



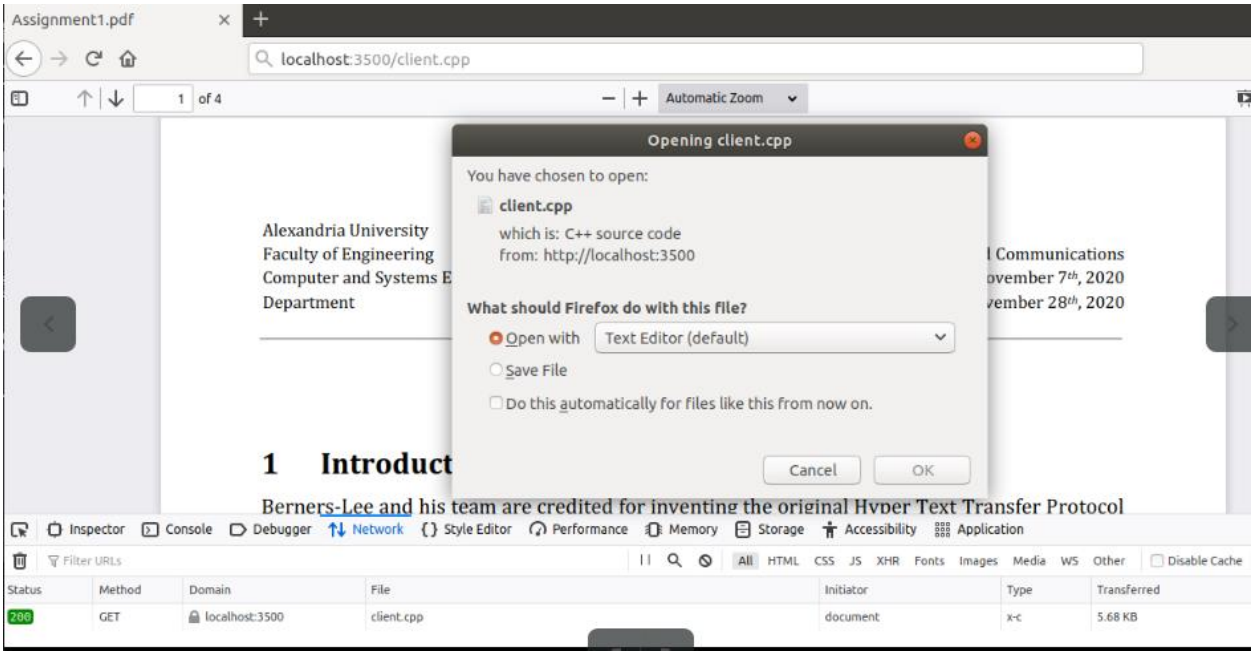


Calling a pdf file:




```
Accepted connection from: 127.0.0.1
received request from 127.0.0.1 :
GET /Resources/Assignment1.pdf HTTP/1.1
Host: localhost:3500
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:83.0) Gecko/20100101 Fire
fox/83.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0
.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
```

Calling a C++ file:

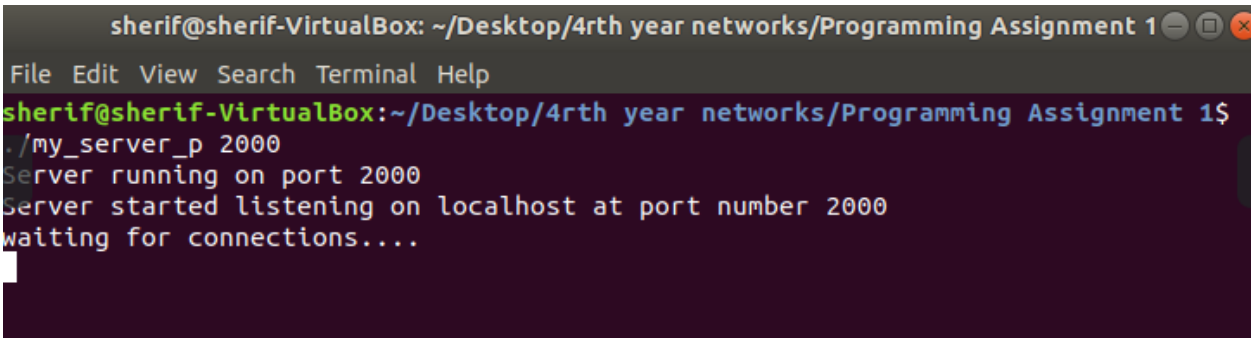


It even prompts to select the editor by which we want to open the received Cpp file.

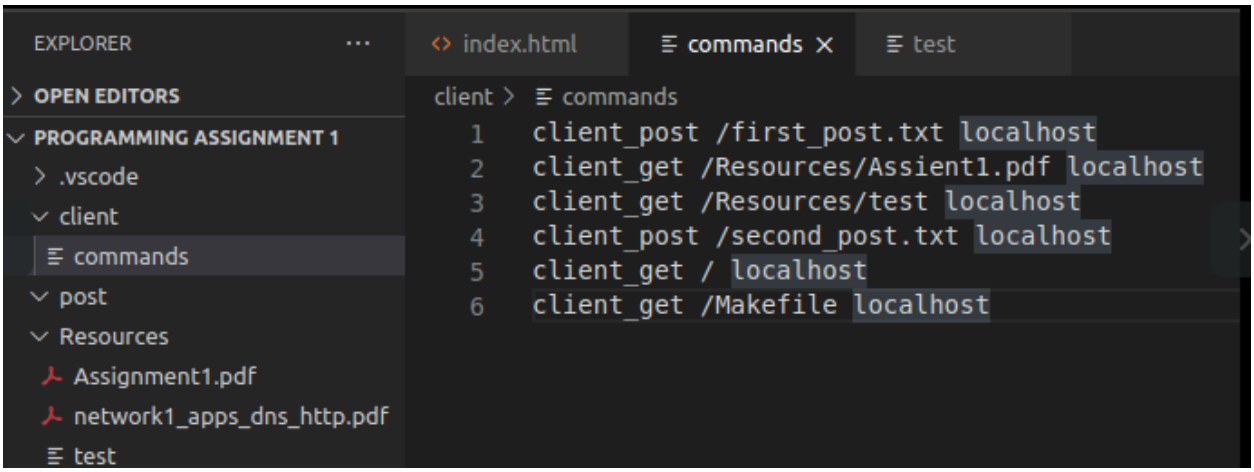
```
Accepted connection from: 127.0.0.1
received request from 127.0.0.1 :
GET /client.cpp HTTP/1.1
Host: localhost:3500
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:83.0) Gecko/20100101 Fire
fox/83.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0
.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
```

Testing the client with the persistent web server:

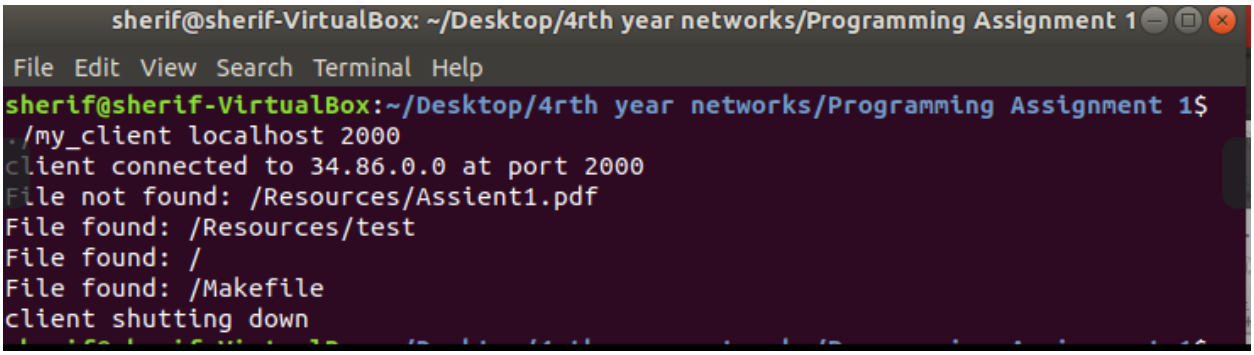
Running the persistent web server:



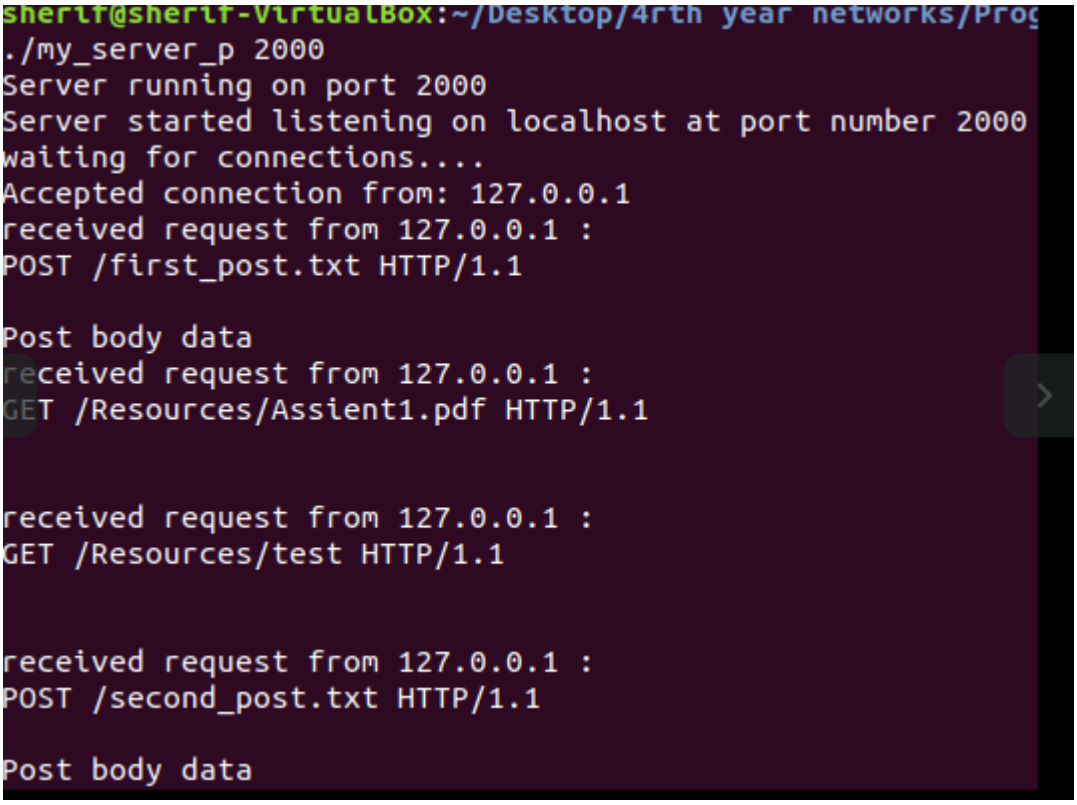
Client commands:



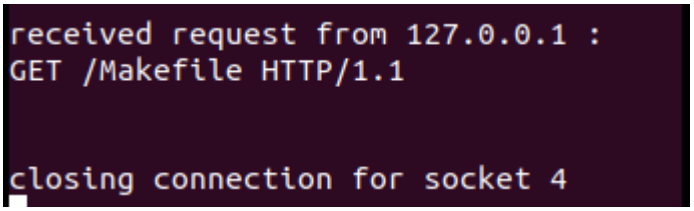
Running the client:



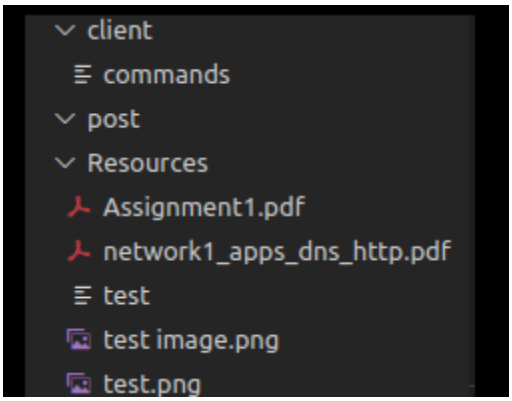
Requests received and displayed in the same connection by the server:



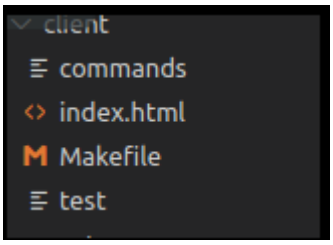
When the client closes connection, the server closes it too:



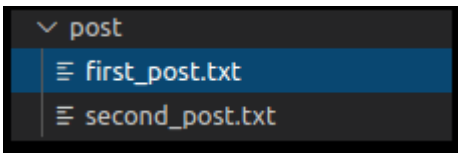
Folders before running:

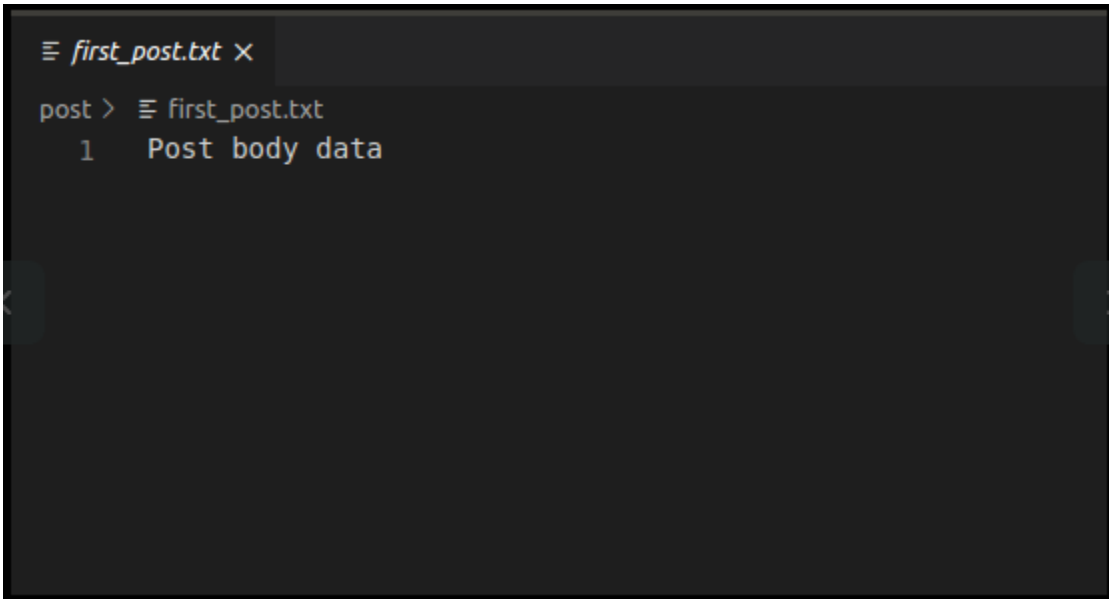


Files were obtained from GET requests:



Files were created by server for the post requests:





Assumptions and limitations:

- The commands file is in client folder in client directory and is called commands.
- The port number and server IP specified in the commands folder weren't used as according to the implementation and pseudo code followed, the client opens a connection once and then sends all the requests at once.
- The default server port number is 2000 and the default port number at which the client connects is 80.
- Multi-processing was used for the reasons stated in the description of the file server_persistent.cpp.
- Files obtained by GET requests are placed in the client folder in the client directory as was shown.
- Files created corresponding to post requests (if needed to be created) are placed in post folder in the server directory.
- *Timeouts weren't implemented and so it is assumed that the client closes connection for the persistent connection to close.* ***This is an identified limitation intended to be handled by the use of select() later.***

References:

https://www.geeksforgeeks.org/map-associative-containers-the-c-standard-template-library-stl/

http://www.cplusplus.com/forum/windows/115254/

http://www.martinbroadhurst.com/how-to-split-a-string-in-c.html

https://www.tutorialspoint.com/substring-in-cplusplus

https://www.tutorialspoint.com/http/http_responses.html

http://www.cplusplus.com/doc/tutorial/files/

https://stackoverflow.com/questions/17584784/read-a-binary-file-jpg-to-a-string-using-c

https://stackoverflow.com/questions/12774207/fastest-way-to-check-if-a-file-exist-using-standard-c-c11-c

https://www.w3schools.com/cpp/cpp_files.asp

https://fedemengo.github.io/blog/2018/02/SIGCHLD-handler.html

https://man7.org/linux/man-pages/man3/perror.3.html

https://www.gnu.org/software/libc/manual/html_node/Flags-for-Sigaction.html

https://stackoverflow.com/questions/14378957/detecting-a-timed-out-socket-from-a-set-of-sockets

https://stackoverflow.com/questions/19555121/how-to-get-current-timestamp-in-milliseconds-since-1970-just-the-way-java-gets