

Project 1:

Simple HTTP Client and Server

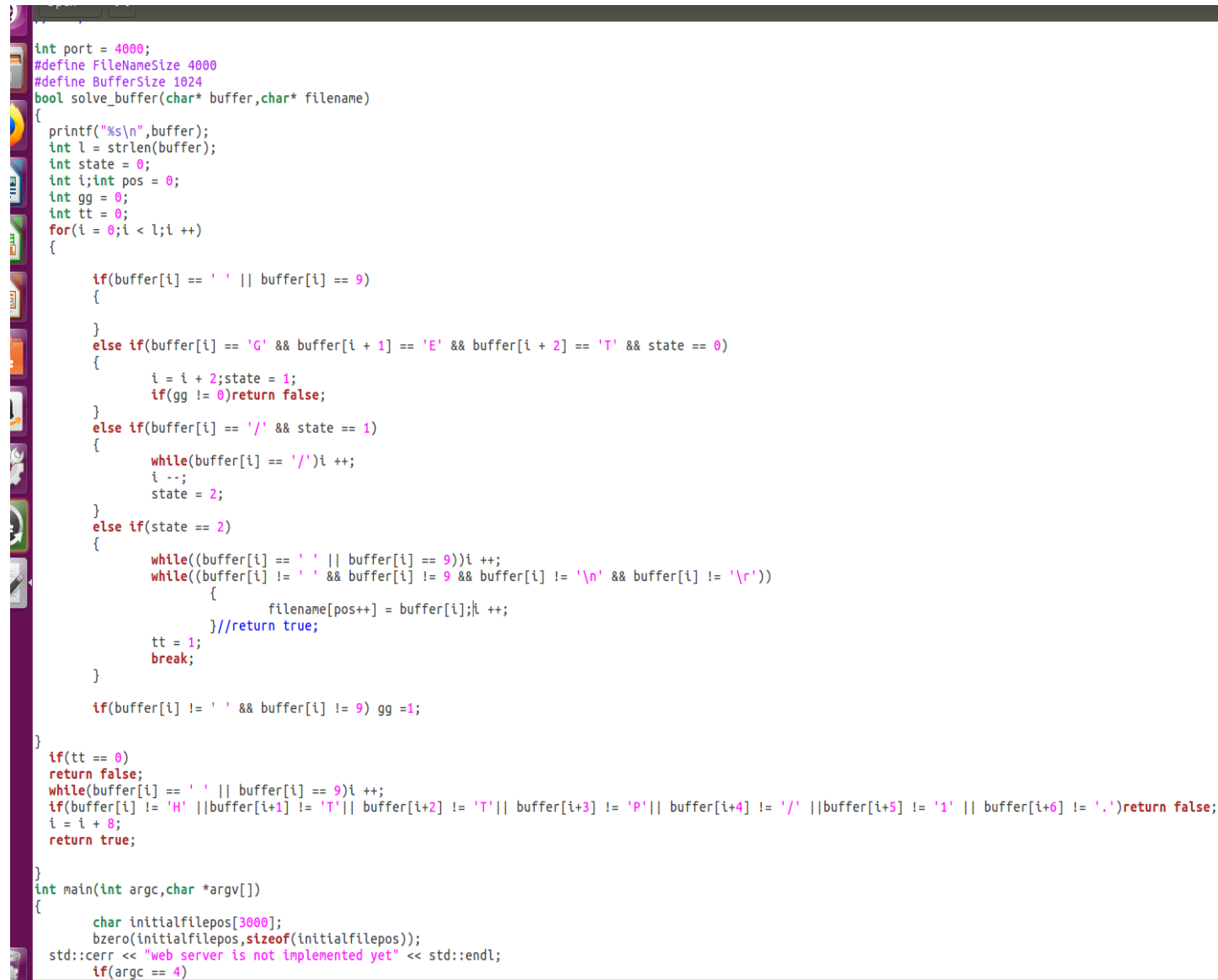
Submitted by

SANKALP PANGHAL (Student Id – 17317444)

YUE CHANG (Student Id -)

Q.the high level design of your server and client

DESIGN FOR WEB-SERVER

A screenshot of a code editor showing C++ code for a web server. The code defines constants for port (4000), file name size (4000), and buffer size (1024). It includes a function 'solve_buffer' that processes a buffer and filename, and a 'main' function that sets up initial file positions and checks for command-line arguments. The code uses various control structures like if, while, for, and return statements to handle different states and buffer contents.

```
int port = 4000;
#define FileNameSize 4000
#define BufferSize 1024
bool solve_buffer(char* buffer,char* filename)
{
    printf("%s\n",buffer);
    int l = strlen(buffer);
    int state = 0;
    int i;int pos = 0;
    int gg = 0;
    int tt = 0;
    for(i = 0;i < l;i++)
    {
        if(buffer[i] == ' ' || buffer[i] == 9)
        {
        }
        else if(buffer[i] == 'G' && buffer[i + 1] == 'E' && buffer[i + 2] == 'T' && state == 0)
        {
            i = i + 2;state = 1;
            if(gg != 0)return false;
        }
        else if(buffer[i] == '/' && state == 1)
        {
            while(buffer[i] == '/')i++;
            i--;
            state = 2;
        }
        else if(state == 2)
        {
            while((buffer[i] == ' ' || buffer[i] == 9))i++;
            while((buffer[i] != ' ' && buffer[i] != 9 && buffer[i] != '\n' && buffer[i] != '\r'))
            {
                filename[pos++] = buffer[i];i++;
            }
            tt = 1;
            break;
        }

        if(buffer[i] != ' ' && buffer[i] != 9) gg =1;
    }
    if(tt == 0)
    return false;
    while(buffer[i] == ' ' || buffer[i] == 9)i++;
    if(buffer[i] != 'H' ||buffer[i+1] != 'T' || buffer[i+2] != 'T' || buffer[i+3] != 'P' || buffer[i+4] != '/' ||buffer[i+5] != '1' || buffer[i+6] != '.')return false;
    i = i + 8;
    return true;
}
int main(int argc,char *argv[])
{
    char initialfilepos[3000];
    bzero(initialfilepos,sizeof(initialfilepos));
    std::cerr << "web server is not implemented yet" << std::endl;
    if(argc == 4)
```

ABOVE IMAGE IS OVERALL web-server CODE SCREENSHOT

➔ (SEE IMAGE 1.1 BELOW)

Web-server expects 3 arguments, and works as expected in the project work, but if no argument provided then it assumes domain as localhost, port as 4000 and current working directory as default.

So as execution in main starts, it checks if `argc == 4` i.e. if arguments are input by user or not, if yes then inside if we resolve the arguments for later use in code such as getting port number to int type from character array and other arguments respectively

```

int main(int argc, char *argv[])
{
    char initialfilepos[3000];
    bzero(initialfilepos, sizeof(initialfilepos));
    std::cerr << "web server is not implemented yet" << std::endl;
    if(argc == 4)
    {
        printf("argc:%d\n", argc);
        int maxx = 0;
        for(int i = 0; i < strlen(argv[2]); i++)
        {
            maxx *= 10;
            maxx += (argv[2][i] - '0');
        }
        port = maxx;

        for(int i = 0; i < strlen(argv[3]); i++)
        {
            initialfilepos[i] = argv[3][i];
        }
        if(initialfilepos[strlen(initialfilepos)-1] != '/')
        {
            initialfilepos[strlen(initialfilepos)] = '/';
        }
    }

    // do your stuff here! or not if you don't want to!!!!
    struct sockaddr_in ServerAddr;
    bzero(&ServerAddr, sizeof(ServerAddr)); // initialize the server address
    ServerAddr.sin_family = AF_INET;
    ServerAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    ServerAddr.sin_port = htons(port);
    int ServerSocket = socket(AF_INET, SOCK_STREAM, 0);

    if (bind(ServerSocket, (struct sockaddr*)&ServerAddr, sizeof(ServerAddr)) == -1)
    {
        printf("Server Bind Port: %d Error!\n", port);
        exit(1);
    }

    if (listen(ServerSocket, 5) == -1)
    {
        printf("Server Listen Error!\n");
        exit(1);
    }

    printf("Server is ready to run!\n");
}

```

IMAGE 1.1

- ➔ We define **struct sockaddr_in ServerAddr** to store information such as localhost, 127.0.0.1 and port using `hton()` (host to network- int to binary function).
A new socket **int ServerSocket = socket(AF_INET, SOCK_STREAM, 0);** is created for binding with these **ServerAddr** structure
- ➔ There is an **if** statement to check for any binding error, so if any binding error, server prints **Server Bind Port: PORTNO Error! And program exits with 1 status code**
- ➔ Then another **if** statement **if (listen(ServerSocket, 5) == -1)** is used to check and listening error.
- ➔ After all above steps, if no problem or exit status 1 encountered, **web-server** prints **Server is ready to run!**

```

printf("Server is ready to run!\n");
while(1)
{
    struct sockaddr_in ClientAddr;
    socklen_t length = sizeof(ClientAddr);

    int NewServerSocket = accept(ServerSocket, (struct sockaddr*)&ClientAddr, &length);
    if (NewServerSocket < 0)
    {
        printf("Server Accept Error!\n");
        break;
    }

    //if is father ...
    if(fork(>0)
    {
        close(NewServerSocket);} //sleep(20);}
    else
    {
        char information[INET_ADDRSTRLEN] = {'\0'};
        inet_ntop(ClientAddr.sin_family, &ClientAddr.sin_addr, information, sizeof(information));

        printf("Accept a connection from: %s : %d\n", information, ntohs(ClientAddr.sin_port));

        //if(fork() < 0)printf("ERROR\n");
        //if is son ...
        char FileName[FileNameSize + 1];
        bzero(FileName, sizeof(FileName));
        char Buffer[BufferSize];
        bzero(Buffer, sizeof(Buffer));

        recv(NewServerSocket, Buffer, BufferSize, 0);

        if(strlen(Buffer) > FileNameSize)
        {
            printf("Too Long File Name!\n");
            exit(1);
        }

        // printf("%s\n", Buffer);

        if(!solve_buffer(Buffer, FileName)){
            char bb[3000] = "HTTP/1.0 400 Bad Request \r\n\r\n";
            send(NewServerSocket, bb, strlen(bb), 0);
            exit(0);
        }
        printf("%s\n", FileName);
        bool ct = 1;
        for(int i = 0; i < FileNameSize; i++)
        {
            if(FileName[i] == '\n' || FileName[i] == '\r')ct = 0;
            FileName[i] *= ct;
        }
        for(int i = 0; i < strlen(FileName); i++)
        {
            initialfilepos[strlen(initialfilepos)] = FileName[i];
        }
    }
}

```

IMAGE 1.2

(SEE ABOVE IMAGE 1.2)

- ➔ Infinite while loop is used **while(1)** which extends upto bottom of the code
- ➔ Now we create new Server Socket named **NewServerSocket = accept(ServerSocket, (struct sockaddr*)&ClientAddr, &length);** for which if is null we print server accept error, or else connection is accepted
- ➔ **Multi-threading is used --- Now we use fork() from <sys/types.h> for multiple client request handling, which if not used then NewServerSocket is left open and another client has to wait to contact the server.**

So if fork() return a positive PID then it means new thread created and we close(NewServerSocket)

- ➔ We create character array **information[INET_ADDRSTRLEN]** to hold information details of client and then **web-server** prints **Accept a connection from: information : PORTNO**
- ➔ We create two character arrays named **FileName** and **Buffer**.
Buffer is used to collect all incoming packets from client using **recv(NewServerSocket, Buffer, BufferSize, 0);**
While we pass **FileName** later to our **self defined** function **solve_buffer(char* buffer, char* filename)** so we can store **Object's Path Name from URL of GET request**
- ➔ Now we use **if(strlen(Buffer) > FileNameSize)** to print **Too Long File Name!** and exit with code **=1**
- ➔ **(SEE IMAGE BELOW 1.3)**
if(!solve_buffer(Buffer,FileName) is used to check for either bad GET request, BAD OR WRONG HTTP REQUEST (like misspelled GET or wrong http version ex smallcase http in GET REQUEST), in which case **solve_buffer()** function returns false and **web-server** prints **HTTP/1.0 400 Bad Request** and sends this as HTTP RESPONSE to the client, and exits with **0** status code.
As described before, this function also stores **Object's Path Name from URL of GET REQUEST** to **FileName**

```

bool solve_buffer(char* buffer, char* filename)
{
    printf("%s\n", buffer);
    int l = strlen(buffer);
    int state = 0;
    int i; int pos = 0;
    int gg = 0;
    int tt = 0;
    for(i = 0; i < l; i++)
    {
        if(buffer[i] == ' ' || buffer[i] == 9)
        {
        }
        else if(buffer[i] == 'G' && buffer[i + 1] == 'E' && buffer[i + 2] == 'T' && state == 0)
        {
            i = i + 2; state = 1;
            if(gg != 0) return false;
        }
        else if(buffer[i] == '/' && state == 1)
        {
            while(buffer[i] == '/') i++;
            i--;
            state = 2;
        }
        else if(state == 2)
        {
            while((buffer[i] == ' ' || buffer[i] == 9)) i++;
            while((buffer[i] != ' ' && buffer[i] != 9 && buffer[i] != '\n' && buffer[i] != '\r'))
            {
                filename[pos++] = buffer[i]; i++;
            } //return true;
            tt = 1;
            break;
        }

        if(buffer[i] != ' ' && buffer[i] != 9) gg = 1;
    }

    if(tt == 0)
    return false;
    while(buffer[i] == ' ' || buffer[i] == 9) i++;
    if(buffer[i] != 'H' || buffer[i+1] != 'T' || buffer[i+2] != 'P' || buffer[i+3] != 'P' || buffer[i+4] != '/' || buffer[i+5] != '1' || buffer[i+6] != '.') return false;
    i = i + 8;
    return true;
}

```

IMAGE 1.3 SELF_DEFINED FUNCTION solve_buffer() image

- ➔ Then rest of the code as displayed below, tries to read and transfer file from proper directory, in which case **if it fails to read** (using fopen() function), then it returns http response **HTTP/1.0 404 Not Found \r\n\r\n** or else **HTTP/1.0 200 OK \r\n\r\n** and **sends the file to client using fread() function**
- Finally **web-server** closes file pointer, prints **File Transfer Success, closes ServerSocket** and ends working.

```

}
for(int i = 0; i < strlen(fileName); i++)
{
    initialfilepos[strlen(initialfilepos)] = fileName[i];
}

FILE *fp = fopen(initialfilepos, "r");
if (fp == NULL)
{
    char bb[3000] = "HTTP/1.0 404 Not Found \r\n\r\n";
    send(NewServerSocket, bb, strlen(bb), 0);
    printf("File: %s Not Found!\n", initialfilepos);
    exit(0);
}
char bb[3000] = "HTTP/1.0 200 OK \r\n\r\n";
send(NewServerSocket, bb, strlen(bb), 0);
//
bzero(Buffer, sizeof(Buffer));
int length3 = 0;
while(1)
{
    length3 = fread(Buffer, sizeof(char), BufferSize, fp);
    if(length3 <= 0) break;
    if (send(NewServerSocket, Buffer, length3, 0) < 0)
    {
        printf("Send File:%s Error!\n", fileName);
        break;
    }
    // printf("%s\n", Buffer);
    bzero(Buffer, sizeof(Buffer));
}
fclose(fp);
//char bb[3000] = "HTTP/1.0 200 OK \r\n";
//send(NewServerSocket, bb, strlen(bb), 0);
printf("File %s Transfer Success!\n", fileName);
close(NewServerSocket);
break;
}
}close(ServerSocket);
return 0;

```


DESIGN FOR WEB-CLIENT

`./web-client URL1 URL2 URL3` is capable of sending multiple non-persistent connections to `localhost:4000/URLNO`

```
int main(int argc, char *argv[])
{
    // do your stuff here! or not if you don't want to.
    for(int iiii = 1; iiii < argc; iiii++)
    {
        bzero(arg, sizeof(arg));
        bzero(request, sizeof(request));
        for(int jjj = 0; jjj < strlen(argv[iiii]); jjj++)
            arg[jjj] = argv[iiii][jjj];

        pre_solve();

        printf("%d %s %s\n", port, request, fn);
        struct sockaddr_in ClientAddr;
        bzero(&ClientAddr, sizeof(ClientAddr)); //initialize the server address
        ClientAddr.sin_family = AF_INET;
        ClientAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
        ClientAddr.sin_port = htons(0);
        int ClientSocket = socket(AF_INET, SOCK_STREAM, 0);

        if (bind(ClientSocket, (struct sockaddr*)&ClientAddr, sizeof(ClientAddr)) == -1)
        {
            printf("Client Bind Port Error!\n");
            exit(1);
        }
        struct sockaddr_in ServerAddr;
        bzero(&ServerAddr, sizeof(ServerAddr)); //initialize the server address
        ServerAddr.sin_family = AF_INET;
        ServerAddr.sin_addr.s_addr = inet_addr("127.0.0.1");
        ServerAddr.sin_port = htons(port);
        socklen_t ServerAddrLength = sizeof(ServerAddr);
        if (connect(ClientSocket, (struct sockaddr*)&ServerAddr, ServerAddrLength) < 0)
        {
            printf("Can Not Connect To Server!\n");
            exit(1);
        }
        printf("Client Connected\n");

        //char FileName[FileNameSize + 1] = "spacer.GIF";
        //bzero(FileName, sizeof(FileName));
        //printf("Input File Name\n");
        //scanf("%s", FileName);
        char Buffer[BufferSize];
        bzero(Buffer, sizeof(Buffer));
        if(strlen(fn) > BufferSize)
        {
            printf("Too Long File Name!\n");
            exit(1);
        }
        strncpy(Buffer, request, strlen(request));
        // char Buffer2[2000] = "GT /index.html http/1.0";
        send(ClientSocket, Buffer, BufferSize, 0);
    }
}
```

Image 2.1

- We use global variable `char arg[3000]`, which holds value of URL1, URL2 upto total count of `argc` i.e. upto all arguments, in turns as `for(int iii = 1;iii < argc;iii ++)` loop completes (2nd line of code)

```
int port = 4000;
char arg[3000];
char request[3000];
char fn[3000];
#define FileNameSize 4000
#define BufferSize 1024
void pre_solve()
{
    request[0] = 'G';
    request[1] = 'E';
    request[2] = 'T';
    request[3] = ' ';
    port = 0;
    bzero(fn, sizeof(fn));
    if(arg[0] != 'h' || arg[1] != 't' || arg[2] != 't' || arg[3] != 'p' || arg[4] != ':' || arg[5] != '/') {}
    else
    {
        int i = 6;
        while(arg[i] != ':') i++;
        i++; port = 0;
        while(arg[i] >= '0' && arg[i] <= '9')
        {
            port *= 10;
            port += (arg[i] - '0'); i++;
        }
        int gg = 4;
        while(i < strlen(arg))
        {
            request[gg] = arg[i]; i++; gg++;
        }
        request[gg] = ' ';
        while(request[gg] != '/') gg--; gg++; int popo = 0;
        while(gg < strlen(request)) fn[popo++] = request[gg++];
        request[gg++] = ' ';
        request[gg++] = 'H';
        request[gg++] = 'T';
        request[gg++] = 'T';
        request[gg++] = 'P';
        request[gg++] = '/';
        request[gg++] = '1';
        request[gg++] = '.';
        request[gg++] = '0';
        request[gg++] = '\r';
        request[gg++] = '\n';
    }
}
```

IMAGE PRE-SOLVE

- Self-defined function `pre_solve()` is called which (SEE IMAGE ABOVE)
1. Generates request on global character array named `request`, i.e. `GET /fn HTTP/1.0`
 2. Where `fn` (nickname filename) holds value for **Object-file-path** from the URL
 3. It stores correct value of port as an integer from character array
`pre_solve()` also stores correct `fn` value for each loop run i.e. for different `arg`

(Refer to image 2.1 above)

- Then we create **ClientSocket** and **struct sockaddr_in ClientAddr**
And bind them, for which if any error, **web-client** prints **Client Bind Port Error!**
- We create **struct sockaddr_in ServerAddr** , which holds correct server address details like 127.0.0.1 and port mentioned in URL.
- Web-client then tries to connect with **connect(ClientSocket, (struct sockaddr*)&ServerAddr, ServerAddrLength)** ,for which if any error the program prints "**Can Not Connect To Server!\n**" and exits with 1 status code
Or else it prints **Client Connected**
- We then create a char array called **buffer[1024]** , copies string from **request** to **buffer** using **strncpy()** predefined function, and sends it into **ClientSocket**

```
//web-client from below open the file from SAME DIRECTORY WHERE IT STARTS
FILE *fp;

bzero(Buffer, sizeof(Buffer));
int l = recv(ClientSocket, Buffer, 20, 0); //buffersize

if(Buffer[9] == '2' && Buffer[10] == '0' && Buffer[11] == '0')
{
    fp = fopen(fn, "w+");
    int ii = 0;
    printf("buffer:%s\n", Buffer);
    while(Buffer[ii] != 'K') ii++; ii++; ii++;
    while(Buffer[ii] == '\r' || Buffer[ii] == '\n') ii++;
    char gg[2000];
    int kk = 0;
    for(; ii < strlen(Buffer); ii++) gg[kk++] = Buffer[ii];

    // fwrite(gg, sizeof(char), strlen(gg), fp);
    bzero(Buffer, sizeof(Buffer));

    while(1)
    {
        int l = recv(ClientSocket, Buffer, BufferSize, 0);
        // printf("\nOUTPUT BUFFER\n%s", Buffer);
        if(l == 0) break;

        fwrite(Buffer, sizeof(char), l, fp);
        bzero(Buffer, sizeof(Buffer));
    }
    bzero(Buffer, sizeof(Buffer));
    printf("Client Succeed\n");
    // fclose(fp);
} else { printf("%s\n", Buffer); }
fclose(fp);
close(ClientSocket);
```

Image 2.3

(See image 2.3 above)

- We create **FILE*** type pointer called **fp** , receive the data from **ClientSocket** , store them into **Buffer** and if the response to **Connect** is **200 status code**, web-client will download the file by using function **fp = fopen(fn,"w+")** and **fwrite(Buffer, sizeof(char), l, fp);**
- Character array **gg[]** will store the data from the server after the characters **\r\n\r\n** i.e. **until the end of HTTP** message
- Until **recv** function doesn't send 0 or NULL **or we can say** until file transfer has completed, web-client prints **Client Succeed** and closes **fclose(fp)** and **close(ClientSocket)**

Q The Problems your ran into and how you solved the problems

- Biggest issue was to handle multiple clients

Initially **accepting** socket on server was kept busy and kept clients waiting for very long time, so we used **fork()** call function, inside an infinite loop so that we can **close accepting socket** in which case **it is again ready to handle more clients**, as can be seen above from **web-server code**.

- HTTP requests and responses

We didn't used any classes or header files to encode or decode HTTP messages, which meant we took care of correct implementation and format of messages by self writing, using many if for loops and testing web-server with browser to see if it is being correctly implemented. Writing manually Correct and Proper format, And moreover corresponding error codes in correct http response format was challenging.

Note- In case of downloading a file, additional characters were getting added to the file, which was making the format of certain .zip files wrong, hence we modified web-client to handle first 20chars for HTTP message, rest was being writing to the file. Hence afterward file download was correctly implemented.

- When performing test cases, like sending large files and opening them simultaneously in the browser, we encountered problems such as – Sometimes browser was not able to connect to server, sometimes in large file case partial file was sent instead of full large file
By continuously modifying the necessary changes in code, we solved the problems

Q. additional instructions to build your project (if your project uses some other libraries

- No, our **web-client** and **web-server** just use **Makefile** to build these files, although below are certain header files included in the project

Web-server - #include <string>, #include <thread>, #include <iostream>, #include <sys/types.h>, #include <sys/socket.h>, #include <netinet/in.h>, #include <arpa/inet.h>, #include <string.h>, #include <stdio.h>, #include <errno.h>, #include <unistd.h>
Web-client - #include <string>, #include <thread>, #include <iostream>, #include <string>, #include <sys/types.h>, #include <sys/socket.h>, #include <netinet/in.h>, #include <arpa/inet.h>, #include <string.h>, #include <stdio.h>, #include <errno.h>, #include <unistd.h>

Q. how you tested your code and why

After Running **web-server**, we tested our web-server with modern client such as **mozilla web browser**. After web-browser was successfully able to display the web page to screen we got assured that there could be no problem in sending of index.html file.

More over we used **BurpSuite software** (<https://portswigger.net/burp>), which acted as proxy between our **mozilla** and **web-server**, so we tested different cases such as

1. http request message ending with only \r\n or only \n in which case, our **web-server** was successfully working and giving corresponding Response codes.

Hence our web-server was able to handle and read request even before whole message was read (such as by removing \r\n out of \r\n\r\n at the end of HTTP request message)

2. All HTTP request (**GET method only**) are being handled successfully even if there are no headers
3. Appropriate error codes such as **HTTP/1.0 404 NOT FOUND** is displayed on typing bad file path or bad file name

And

HTTP/1.0 400 BAD REQUEST is displayed on either anything else than GET or lowercase http.

We tested **web-client** with multiple URL as arguments on the web-server to see if file is being downloaded correctly and multiple requests are being handled simultaneously.

Q. Contribution of each team member

We both Sankalp (Student Id- 17317444 and Yue (Student Id -) have complete understanding of all the lessons that can be learnt from this project and shared countable required hours in the LG12 lab together for different test cases and modifications and can write code independently.

Having said that

Yue (Student Id -) started to work on the project days earlier than Sankalp (Student Id- 17317444) so essentially it is Yue's code that was kept modifying over the last two weeks, to counter that Sankalp compiled all this brief pdf report with explanations.