1 Server

```c
#include <stdio.h> // printf, perror, fopen, fread, feof, fclose
#include <string.h> // strlen
#include <stdlib.h> // exit
#include <unistd.h> // read, write, fork
#include <sys/socket.h> // socket, bind, listen, accept
#include <arpa/inet.h> // htons, sockaddr, sockaddr_in


// constants
#define PORT 31415
#define BUFFER_SIZE 1024



struct char_map {
    char * key;
    char * value;
};



int main() {

    // local variables
    int s, s_double;                               // sockets
    char * command_line;                           // first line of request
    struct char_map headers[100] = {{NULL, NULL}}; // headers
    char header_buffer[BUFFER_SIZE] = {0};         // header buffer, here there will be all the info from the header
    char response_buffer[BUFFER_SIZE] = {0};       // response buffer, will be used to temporarily store the response
    char * method, * uri, * version;               // parsed values from command_line
    int i, yes = 1;                                        // generic index

    // define address
    struct sockaddr_in server_address;
    struct sockaddr_in client_address;





    // socket
    s = socket( AF_INET, SOCK_STREAM, 0);

    // terminate if error
    if( s == -1 ) {
        perror("socket() failed");
        return 1;
    }


    if ( -1 == setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) ) {
        perror("setsockopt() failed");
        return 1;
    }

    // define address
    server_address.sin_family   = AF_INET;
    server_address.sin_port     = htons(PORT);
    server_address.sin_addr.s_addr = 0;

    // bind
    if( -1 == bind(s, (struct sockaddr *) &server_address, sizeof(struct sockaddr_in)) ) {
        perror("bind() failed");
        return 1;
    }

    // listen
    if( -1 == listen(s, 5) ) {
        perror("listen() failed");
        return 1;
    }


    int sockaddr_size = sizeof(struct sockaddr);

    while(1) {

        // accept
        s_double = accept(s, (struct sockaddr *) &client_address, &sockaddr_size);


        // create sub-process
        if (fork()) {
            close(s_double);
            continue;
        }

        // terminate if error
        if( s_double == -1 ) {
            perror("accept() failed");
            return 1;
        }


        // parse the header
        command_line = headers[0].key = header_buffer;
        int lines = 0;

        for(i = 0; read(s_double, header_buffer + i, 1); i++) {

            // end of the line
            if(header_buffer[i - 1] == '\r' && header_buffer[i] == '\n') {

                // null-terminate
                header_buffer[i - 1] = 0;

                // check if it is the end
                if( !headers[lines].key[0] )
```

```
                    break;

                // create new line on the headers
                lines++;
                headers[lines].key = &header_buffer[i + 1];
            }

            if( header_buffer[i] == ':' && (headers[lines].value == NULL)) {

                // start value
                headers[lines].value = &header_buffer[i + 1] + 1;

                // null-terminate
                header_buffer[i] = 0;

            }
        }

        // print headers
        for(i = 0; i < lines; i++)
            printf("%s ----> %s\n", headers[i].key, headers[i].value);


        // parse method, uri, version
        method = command_line;
        for(i = 0; command_line[i] != ' '; i++);
        command_line[i++] = 0;

        uri = command_line + i;
        for(; command_line[i] != ' '; i++);
        command_line[i++] = 0;

        version = command_line + i;
        for(; command_line[i] != 0; i++);
        command_line[i++] = 0;

        // print values
        printf("Method ----> %s\nURI ----> %s\nVersion ----> %s\n", method, uri, version);


        // opens file
        FILE * file = fopen(uri + 1, "rw");

        if( file == NULL ) {

            // create 404 response
            sprintf(response_buffer, "HTTP/1.1 404 NOT FOUND\r\n\r\n<html><h1>File %s was not found.</h1></html>", uri);

            // send response
            if( -1 == write(s_double, response_buffer, strlen(response_buffer)) ){
                perror("write() failed");
                return 1;
            }

        } else {
```

```c



            // send accept header
            sprintf(response_buffer, "HTTP/1.1 200 OK\r\n\r\n");

            if( -1 == write(s_double, response_buffer, strlen(response_buffer)) ){
                perror("write() failed");
                return 1;
            }

            // read and send the file
            while( !feof(file) ) {

                // read 1Kb from the file
                fread(response_buffer, 1, 1024, file);

                // write the answer
                if( -1 == write(s_double, response_buffer, strlen(response_buffer)) ){
                    perror("write() failed");
                    return 1;
                }

                for(i = 0; i < BUFFER_SIZE; i++) response_buffer[i] = 0;

            }

            fclose(file);

        }


        printf("\n\n\n");

        // close socket and kill process
        close(s_double);
        exit(1);

    }

    return 0;

} // main
```

1.1 Content-Length

```c
    content_length = 0;

    // get content length of the file
    while( fgetc(file) != EOF )
        content_length++;

    printf("Content-Length: %d\n\n\n", content_length);


    // send accept header
    sprintf(response_buffer, "HTTP/1.1 200 OK\r\nContent-Length: %d\r\n\r\n", content_length);

    if( -1 == write(s_double, response_buffer, strlen(response_buffer)) ){
        perror("write() failed");
        return 1;
    }

    // reset buffer
    for(i = 0; i < BUFFER_SIZE; i++) response_buffer[i] = 0;



    // pointer to the beginning
    rewind(file);

    // read and send the file
    while( !feof(file) ) {

        // read 1Kb from the file
        fread(response_buffer, 1, 1024, file);

        // write the answer
        if( -1 == write(s_double, response_buffer, strlen(response_buffer)) ){
            perror("write() failed");
            return 1;
        }

        for(i = 0; i < BUFFER_SIZE; i++) response_buffer[i] = 0;

    }

    fclose(file);
```

1.2 Reflect

```c
    // process the ‘/reflect‘ request
    if ( !strncmp(uri, "/reflect", strlen("/reflect")) ) {

        // send accept header
        sprintf(response_buffer, "HTTP/1.1 200 OK\r\n\r\n");

        if( -1 == write(s_double, response_buffer, strlen(response_buffer)) ){
            perror("write() failed");
            return 1;
        }

        for(i = 0; i < BUFFER_SIZE; i++) response_buffer[i] = 0;



        // send received request + CRLF
        snprintf(response_buffer, BUFFER_SIZE, "%s %s %s\r\n", method, uri, version);

        if( -1 == write(s_double, response_buffer, strlen(response_buffer)) ){
            perror("write() failed");
            return 1;
        }

        for(i = 0; i < BUFFER_SIZE; i++) response_buffer[i] = 0;


        // send client IP and CRLF
        char *client_ip_address = inet_ntoa(client_address.sin_addr); // extract ip address
            A.B.C.D

        snprintf(response_buffer, BUFFER_SIZE, "%s\r\n", client_ip_address);

        if( -1 == write(s_double, response_buffer, strlen(response_buffer)) ){
            perror("write() failed");
            return 1;
        }

        for(i = 0; i < BUFFER_SIZE; i++) response_buffer[i] = 0;


        // send port
        snprintf(response_buffer, BUFFER_SIZE, "%d", ntohs(client_address.sin_port));

        if( -1 == write(s_double, response_buffer, strlen(response_buffer)) ){
            perror("write() failed");
            return 1;
        }

        for(i = 0; i < BUFFER_SIZE; i++) response_buffer[i] = 0;

        // close socket and kill process
        close(s_double);
        exit(1);
    }
```

1.3 AUTH

```
// before accessing an existing file it is needed to AUTHENTICATE
if( !auth_value ) {

    snprintf(response_buffer, BUFFER_SIZE, "HTTP/1.1 401 UNAUTHORIZED\r\nWWW-Authenticate: Basic realm=\"Users\"\r\nConnection: close\r\n\r\n");

    if( -1 == write(s_double, response_buffer, strlen(response_buffer)) ){
        perror("write() failed");
        return 1;
    }


    fclose(file);
    close(s_double);

    continue;

}


for(i = 0; i < BUFFER_SIZE; i++) response_buffer[i] = 0;

// extract base64_cred
for(i = 1; auth_value[i] != ' '; i++);
base64_cred = auth_value + i + 1;



snprintf(response_buffer, BUFFER_SIZE, "%s:%s", username, password);

if( strcmp( base64_cred, base64_encode(response_buffer, strlen(response_buffer)))){

    printf("base64_cred = %s (%d)\n", base64_cred, strlen(base64_cred));
    printf("base64_corr = %s\n", base64_encode(response_buffer, strlen(response_buffer)));

    snprintf(response_buffer, BUFFER_SIZE, "HTTP/1.1 401 UNAUTHORIZED\r\nWWW-Authenticate: Basic realm=\"Users\"\r\nConnection: close\r\n\r\n");

    if( -1 == write(s_double, response_buffer, strlen(response_buffer)) ){
        perror("write() failed");
        return 1;
    }


    fclose(file);
    close(s_double);

    continue;
}
```

1.3.1 Base64

```c
static const char base64_alphabet[] =
    "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";

char* base64_encode(const char* input_array, size_t input_size) {
    // Allocate memory for the output string
    char* output = (char*)malloc(((input_size + 2) / 3 * 4 + 1) * sizeof(char));
    if (output == NULL) {
        return NULL;
    }

    // Perform the Base64 encoding
    size_t output_index = 0;
    for (size_t i = 0; i < input_size; i += 3) {
        // Encode the next 3 bytes
        unsigned char byte1 = (i < input_size) ? input_array[i] : 0;
        unsigned char byte2 = (i + 1 < input_size) ? input_array[i + 1] : 0;
        unsigned char byte3 = (i + 2 < input_size) ? input_array[i + 2] : 0;

        output[output_index++] = base64_alphabet[(byte1 >> 2) & 0x3F];
        output[output_index++] = base64_alphabet[((byte1 & 0x3) << 4) | ((byte2 >> 4) & 0xF)];
        output[output_index++] = base64_alphabet[((byte2 & 0xF) << 2) | ((byte3 >> 6) & 0x3)];
        output[output_index++] = base64_alphabet[byte3 & 0x3F];
    }

    // Handle the case when the input size is not a multiple of 3
    if (input_size % 3 == 1) {
        output[output_index - 2] = '=';
        output[output_index - 1] = '=';
    } else if (input_size % 3 == 2) {
        output[output_index - 1] = '=';
    }

    // Null-terminate the output string
    output[output_index] = '\0';

    return output;
}
```

1.4 Blacklist

```c
// create link
sprintf(link, "%s%s", host, uri);

printf("%s\n", link);

// open blacklist
FILE * blacklist = fopen(BLACKLIST, "r");
char * blacklist_item;

// retrive link from blacklist.txt if exists
while (fgets(blacklist_buffer, BUFFER_SIZE, blacklist)) {

    // null terminate
    blacklist_buffer[strlen(blacklist_buffer) - 1] = 0;

    // if uri is in the blacklist
    if ( !strncmp(blacklist_buffer, link, strlen(link)) ) {

        if( referer ) {
            printf("NOT NULL: %s\n", referer);
            // parse the referer
            for(i = 0; referer[i] != '/'; i++);
            for(++i; referer[i] != ':'; i++);
            for(++i; referer[i] != '/'; i++);

            snprintf(response_buffer, BUFFER_SIZE, "HTTP/1.1 307 Temporary Redirect\r\nLocation: %s\r\nConnection: close\r\n\r\n", referer + i + 1);

        } else {

            printf("NULL: %s\n", referer);

            snprintf(response_buffer, BUFFER_SIZE, "HTTP/1.1 403 Forbidden\r\nConnection:close\r\n\r\n"
                                               "<html>"
                                               "<h1>You are not allowed to access in this page because it is blacklisted</h1>"
                                               "</html>");

        }


        if( -1 == write(s_double, response_buffer, strlen(response_buffer)) ){
            perror("write() failed");
            return 1;
        }

        for(i = 0; i < BUFFER_SIZE; i++) response_buffer[i] = 0;

        // close socket and kill process
        close(s_double);
        exit(1);

    }

    for(i = 0; i < BUFFER_SIZE; i++) blacklist_buffer[i] = 0;

}
```

1.5 Cookie

```c
// retrieve cookie value
for(i = 0; i < lines; i++)
    if( !strcmp(headers[i].key, "Cookie") )
        client_cookie_string = headers[i].value;

// if is not null
if( client_cookie_string ) {

    // extract name and value of the Cookie
    client_cookie_name = client_cookie_string;
    for(i = 0; client_cookie_string[i] != '='; i++);
    client_cookie_string[i++] = 0;

    if(client_cookie_name)
        client_cookie_value = atoi(client_cookie_string + i);
}
```

```c
// if the client goes in the contact.html AND (does not have cookie OR the cookie name is incorrect OR the cookie value is incorrect)
if( !strcmp(uri, "/contact.html") && ( !client_cookie_name || strcmp(client_cookie_name, COOKIE_NAME) || client_cookie_value != 1)) {

    snprintf(response_buffer, BUFFER_SIZE, "HTTP/1.1 403 Forbidden\r\nConnection:close\r\n\r\n<html><h1>You need to access <a href=\"/index.html\">/index.html</a> before entering this page.</h1></html>");

    write(s_double, response_buffer, strlen(response_buffer));

    for(i = 0; i < BUFFER_SIZE; i++) response_buffer[i] = 0;

    // close everything
    fclose(file);
    close(s_double);
    exit(1);
}


// if client is in index.html AND (does not have cookie OR the cookie name is incorrect OR the cookie value is incorrect)
if( !strcmp(uri, "/index.html") && (!client_cookie_name || strcmp(client_cookie_name, COOKIE_NAME) || client_cookie_value != 1)) {

    sprintf(response_buffer, "HTTP/1.1 200 OK\r\nSet-Cookie:%s=%d\r\n\r\n", COOKIE_NAME, 1);

} else if (!strcmp(uri, "/contact.html") && !strcmp(client_cookie_name, COOKIE_NAME) && client_cookie_value == 1) {

    sprintf(response_buffer, "HTTP/1.1 200 OK\r\nSet-Cookie:%s=%d\r\n\r\n", COOKIE_NAME, 0);

} else {

    sprintf(response_buffer, "HTTP/1.1 200 OK\r\n\r\n");

}
```

```c

if( -1 == write(s_double, response_buffer, strlen(response_buffer)) ){
    perror("write() failed");
    return 1;
}

for(i = 0; i < BUFFER_SIZE; i++) response_buffer[i] = 0;
```

1.6 Chunked

```c
// create response header
sprintf(buffer, "HTTP/1.1 200 OK\r\nTransfer-Encoding: chunked\r\n\r\n");

// write response
if( write(s_double, buffer, strlen(buffer)) == -1 ) {
    perror("write() failed");
    return 1;
}

char chunk_size[20];

while( !feof(file) ){

    // read at most 1Kb from the file
    fread(buffer, 1, 1024, file);

    // get first line of the response
    sprintf(chunk_size, "%x\r\n", strlen(buffer));

    // write the first line
    if( write(s_double, chunk_size, strlen(chunk_size)) == -1 ) {
        perror("write() failed");
        return 1;
    }

    // write the chunk
    if( write(s_double, buffer, strlen(buffer)) == -1 ) {
        perror("write() failed");
        return 1;
    }

    // end of the chunk
    if( write(s_double, CRLF, strlen(CRLF)) == -1 ) {
        perror("write() failed");
        return 1;
    }

}

// last chunk
sprintf(buffer, "0\r\n");

// write last chunk
if( write(s_double, buffer, strlen(buffer)) == -1 ) {
        perror("write() failed");
        return 1;
}
```

1.7 ETag

```
// get entity tag value by summing the ascii values of each caracter in the file
unsigned long e_tag_value = 0;
char character[1] = {0};

while( !feof(file) ) {
    // read 1 character
    fread(character, 1, 1, file);

    // sums value
    e_tag_value = e_tag_value + (unsigned long) character[0];
}

if( e_tag_request && (e_tag_request == e_tag_value) ) {

    printf("\n\n");

    // create header
    sprintf(response_buffer, "HTTP/1.1 304 Not Modified\r\nETag: \"%d\"\r\nConnection:
        close\r\n\r\n", e_tag_value);

    if( -1 == write(s_double, response_buffer, strlen(response_buffer)) ){
        perror("write() failed");
        return 1;
    }

    // close file, socket and kill process
    fclose(file);
    close(s_double);
    exit(1);
}

// send accept header
sprintf(response_buffer, "HTTP/1.1 200 OK\r\nETag: \"%d\"\r\n\r\n", e_tag_value);

if( -1 == write(s_double, response_buffer, strlen(response_buffer)) ){
    perror("write() failed");
    return 1;
}

// pointer back
rewind(file);
```

2 Client

```c
#include <stdio.h>
#include <sys/socket.h>         // socket
#include <errno.h>              // errno
#include <arpa/inet.h>          // htons
#include <unistd.h>             // write
#include <string.h>             // strlen, strcmp
#include <stdlib.h>             // atoi


#define RESPONSE_SIZE 100 * 1024


char hbuf[10000];

struct headers{
    char * n;
    char * v;
} h[100];



int main(){

    // local varibles
    struct sockaddr_in server_addr; // server address
    int s;                          // socket
    int t;                          // temporary
    unsigned char * p;              // ip address piointer
    int i, j;
    char * statusline;


    // create socket
    s = socket( AF_INET, SOCK_STREAM, 0 );
    // printf("Socket: %d\n", s);

    if( s == -1){
        printf("ERRNO = %d (%d)\n", errno, EAFNOSUPPORT);
        perror("Socket fallita\n");
        return 1;
    }


    /* Setup for request */

    // set server addr
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(80);

    // IPv4 server
    p = (unsigned char *) &server_addr.sin_addr.s_addr;
    p[0] = 142;   p[1] = 250;   p[2] = 187;    p[3] = 196;


    // connect server
    if(-1 == connect(s, (struct sockaddr *) &server_addr, sizeof(struct sockaddr_in))){
        perror("Connessione fallita\n");
        return 1;
    }


    // send request
    char * request = "GET / HTTP/1.1\r\n\r\n";
    write(s, request, strlen(request));



    statusline = h[0].n = hbuf;
    j = 0;

    // read headers
    for( i = 0; read(s, hbuf + i, 1); i++ ){

        // end of line
        if( hbuf[i - 1] == ’\r’ && hbuf[i] == ’\n’){

            hbuf[i - 1] = 0;

            if( !( h[j].n[0] ) )
                break;

            h[++j].n = &hbuf[i + 1];
        }

        // end of name
        if( (hbuf[i] == ’:’) && (h[j].v == NULL) ){

            h[j].v = &hbuf[i + 1];
            hbuf[i] = 0;
        }
    }

    // print headers
    for(i = 0; i < j; i++)
        printf("%s ----> %s\n", h[i].n, h[i].v);
    printf("\n\n");

    char response[RESPONSE_SIZE];

    for ( i = 0; t = read(s, response + i, RESPONSE_SIZE - 1 - i); i += t ) {}

    // null-terminate response
    response[i] = 0;

    printf("%s\n\n", response);


    return 0;

} // main
```

2.1 Chuncked

```c
int content_length;
for(i = 0; i < j; i++)
    if( !strcmp( h[i].n , "Content-Length" ))
        content_length = atoi(h[i].v);

char response[2000000];
if ( !content_length ){

    // read and print response
    for ( i = 0; t = read(s, response + i, content_length - i); i += t ) {}

    response[i] = 0;
    printf("%s\n\n", response);

    return 0; // end
}


// prepare reading chuncks
long chunk_size;
char chunk_buffer[8];

// will contain all the read bytes
j = 0;

do {   // exit when chunk_size == 0

    // chunck_buffer and hex2dex concersion
    for(chunk_size = 0, i = 0;
        read(s, chunk_buffer + i, 1) && !(chunk_buffer[i - 1] == ’\r’ && chunk_buffer[i] == ’\n’);
        i++) {

        // to lower case
        if( chunk_buffer[i] >= ’A’ && chunk_buffer[i] <= ’F’)
            chunk_buffer[i] = chunk_buffer[i] - (’a’ - ’A’);


        // conversion from letter to dec
        if( chunk_buffer[i] >= ’a’ && chunk_buffer[i] <= ’f’)
            chunk_size = chunk_size * 16 + chunk_buffer[i] - ’a’ + 10;

        // convert numbers too
        if( chunk_buffer[i] >= ’0’ && chunk_buffer[i] <= ’9’)
            chunk_size = chunk_size * 16 + chunk_buffer[i] - ’0’;

    }

    // read chunk and display inside response
    for( i = 0;                                  // iterator
        t = read(s, response + j, chunk_size - i); // from response + j, add ’chunck-size - i- bytes
        i += t, j += t);                         // increment i and totla bytes j

    // read last 2 chars (CRLF)
    read(s, chunk_buffer, 2);


} while( chunk_size );


// null-terminate
response[j] = 0;
printf("%s\n\n", response);
```

2.2 Caching

```c
// substitute '/' with '_'
for( i = 0; i < strlen(file_name); i++)
    if( file_name[i] == '/' )
        file_name[i] = '_';

// save file cached file path
snprintf(file_path, 1024, "%s%s", CAHCE_PATH, file_name);

printf("Cache file path: %s\n", file_path);


// opens file
FILE * cache_file = fopen(file_path, "r");

if ( !cache_file ) {

    printf("File '%s' does NOT exist.\n", file_path);

    flag = 1;

} else {

    printf("File '%s' EXISTS.\n", file_path);

    // gets date
    fgets(cache_date_string, 200, cache_file);
    printf("File Date: %s\n", cache_date_string);

    // converts date
    struct tm cache_date = get_tm_date(cache_date_string);

    // get current time
    time_t now = time(NULL);
    struct tm *current_time = localtime(&now);

    printf("NOW:  %d\n", mktime(current_time));
    printf("CACHE: %d\n", mktime(&cache_date));

    if (difftime(mktime(current_time), mktime(&cache_date)) > 1){
        printf("EXPIRED.\n");
        flag = 1;
    }

}


/* enters if there is no cache or if the cache is expired */
if (flag) {

    // send request
    char * request = "GET / HTTP/1.0\r\n\r\n";
    write(s, request, strlen(request));

    // read and ignore header
    for( i = 0; read(s, hbuf + i, 1); i++ ){

        // end of line
        if( hbuf[i - 1] == ’\r’ && hbuf[i] == ’\n’){

            hbuf[i - 1] = 0;

            if( !( h[j].n[0] ) )
                break;

            j++;
            h[j].n = &hbuf[i + 1];
        }

        // end of name
        if( (hbuf[i] == ’:’) && (h[j].v == NULL) ){

            h[j].v = &hbuf[i + 1] + 1;
            hbuf[i] = 0;
        }
    }


    // opens the: create or erase everything
    cache_file = fopen(file_path, "w");

    // write date in the file
    char date_str[100];
    strftime(date_str, sizeof(date_str), "%a, %d %b %Y %H:%M:%S %Z", &real_expires_date);
    fprintf(cache_file, "%s\n", date_str);

    // add CRLF
    fwrite(&CRLF, strlen(CRLF), 1, cache_file);

    // write response body in the file
    for ( i = 0; t = read(s, response + i, RESPONSE_SIZE - 1 - i); i += t );
    fprintf(cache_file, "%s", response);
    printf("\n\n\n\n\n%s\n\n", response);

    fclose(cache_file);

    return 0;

} // new request needed


printf("NOT EXPIRED.\n\n\n\n");


while ( !feof(cache_file) ) {

    // reads 1KB
    fread(response, 1024, 1, cache_file);
    printf("%s\n", response);

    // resets
    for( i=0; i < 1024; i++) response[i] = 0;

}
```

2.2.1 Date parsing

```
    struct tm get_tm_date(char * date_string) {

    char * date_buffer = date_string;
    struct tm date = {0};
    int i;

    // skip name of day
    for(i = 0; date_string[i] != ’,’; i++);
    date_string[i] = 0;
    date.tm_wday = day2sunday(date_buffer);
    date_string[++i] = 0;


    // extract day
    date_buffer = date_buffer + i + 1;
    for(++i; date_string[i] != ’ ’; i++);
    date_string[i++] = 0;

    date.tm_mday = atoi(date_buffer);


    // extract month
    date_buffer = date_string + i;
    for(; date_string[i] != ’ ’; i++);
    date_string[i++] = 0;

    date.tm_mon = month2int(date_buffer) - 1;

    // [ . . . ]

    // set the remaining fields
    date.tm_isdst = -1; // Let mktime determine if DST is in effect

    char date_str[100];
    strftime(date_str, sizeof(date_str), "%a, %d %b %Y %H:%M:%S %Z", &date);
    printf("Converted Date: %s\n", date_str);

    return date;
}
```

3 Proxy

```c
#include <stdio.h> // printf, perror, fopen, fread, feof, fclose
#include <string.h> // strlen
#include <stdlib.h> // exit
#include <unistd.h> // read, write, fork
#include <sys/socket.h> // socket, bind, listen, accept
#include <arpa/inet.h> // htons, sockaddr, sockaddr_in
#include <netdb.h> // gethostbyname


// constants
#define PORT 58141
#define BUFFER_SIZE 1024



struct char_map {
    char * key;
    char * value;
};



int main() {

    // local variables
    int i, t;                                   // generic index, generic variable
    int s, s_double, s_remote;                  // sockets
    char * command_line;                        // first line of request
    struct char_map headers[100] = {{NULL, NULL}}; // headers
    char header_buffer[BUFFER_SIZE] = {0};      // header buffer, here there will be all the info from the header
    char request_buffer[BUFFER_SIZE] = {0};     // request buffer, will be used to store and send the request
    char response_buffer[BUFFER_SIZE] = {0};    // response buffer, will be used to temporarily store the response
    char * method, * uri, * version;            // parsed values from command_line
    char * scheme, * host, * filename, * port;  // parsed values from GET or CONNECT

    // define address
    struct sockaddr_in local_address;
    struct sockaddr_in remote_address;
    struct sockaddr_in server_address;





    // socket
    s = socket( AF_INET, SOCK_STREAM, 0);

    // terminate if error
    if( s == -1 ) {
        perror("socket() failed");
        return 1;
    }


    // define address
    local_address.sin_family   = AF_INET;
    local_address.sin_port     = htons(PORT);
    local_address.sin_addr.s_addr = 0;

    if ( -1 == setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) ) {
        perror("setsockopt() failed");
        return 1;
    }

    // bind
    if( -1 == bind(s, (struct sockaddr *) &local_address, sizeof(struct sockaddr_in)) ) {
        perror("bind() failed");
        return 1;
    }

    // listen
    if( -1 == listen(s, 10) ) {
        perror("listen() failed");
        return 1;
    }

    // initialize remote (client) address
    remote_address.sin_family    = AF_INET;
    remote_address.sin_port      = htons(0);
    remote_address.sin_addr.s_addr = 0;


    int sockaddr_size = sizeof(struct sockaddr);

    while(1) {

        // accept
        s_double = accept(s, (struct sockaddr *) &remote_address, &sockaddr_size);


        // create sub-process
        if (fork()) {
            close(s_double);
            continue;
        }

        // terminate if error
        if( s_double == -1 ) {
            perror("accept() failed");
            return 1;
        }


        // parse the header
        command_line = headers[0].key = header_buffer;
        int lines = 0;

```

```c
        for(i = 0; read(s_double, header_buffer + i, 1); i++) {

            // end of the line
            if(header_buffer[i - 1] == '\r' && header_buffer[i] == '\n') {

                // null-terminate
                header_buffer[i - 1] = 0;

                // check if it is the end
                if( !headers[lines].key[0] )
                    break;

                // create new line on the headers
                lines++;
                headers[lines].key = &header_buffer[i + 1];
            }

            if( header_buffer[i] == ':' && (headers[lines].value == NULL)) {

                // start value
                headers[lines].value = &header_buffer[i + 1] + 1;

                // null-terminate
                header_buffer[i] = 0;

            }
        }

        // print headers
        for(i = 0; i < lines; i++)
            printf("%s ----> %s\n", headers[i].key, headers[i].value);


        // parse method, uri, version
        method = command_line;
        for(i = 0; command_line[i] != ' '; i++);
        command_line[i++] = 0;

        uri = command_line + i;
        for(; command_line[i] != ' '; i++);
        command_line[i++] = 0;

        version = command_line + i;
        for(; command_line[i] != 0; i++);
        command_line[i++] = 0;

        // print values
        printf("Method ----> %s\nURI ----> %s\nVersion ----> %s\n\n\n\n", method, uri, version);



        if( !strcmp(method, "GET") ) { // GET http://www.example.com/dir/file

            scheme = uri;

           // parse the URI addredd, by getting the host and the resource
           for(i = 0; uri[i] != ':' && uri[i]; i++)

           if (uri[i] == ':')    // null terminate
               uri[i++] = 0;
           else {                // check correctness
               printf("Parsing error (expected ':').\n");
               exit(1);
           }

           if (uri[i] != '/' || uri[i + 1] != '/') {
               printf("Parsing error (expected '//').\n");
               exit(1);
           }

           i = i + 2;

           // save host
           host = uri + (++i);

           // find position where host finishes
           for(; uri[i] && uri[i] != '/'; i++);


           if (uri[i] == '/')    // null terminate
               uri[i++] = 0;
           else {                // check correctness
               printf("Parsing error (expected '/').\n");
               exit(1);
           }

           // initialize filename
           filename = uri + i;


           // resolve host name
           printf("GET host=%s\n", host);
           struct hostent * remote = gethostbyname(host);

           // create socket to connect to the remote
           s_remote = socket( AF_INET, SOCK_STREAM, 0);

           // terminate if error
           if( s_remote == -1 ) {
               perror("socket() failed");
               return 1;
           }

           // set up remote server address
           server_address.sin_family    = AF_INET;
           server_address.sin_port      = htons(80);
           server_address.sin_addr.s_addr = *(unsigned int*)(remote->h_addr);


           // connect to the remote server
           if( -1 == connect( s_remote, (struct sockaddr *) &server_address, sizeof(struct
               sockaddr_in))) {
               perror("connect() failed");
               return 1;
           }

           // create request
           snprintf(request_buffer, BUFFER_SIZE, "GET /%s HTTP/1.1\r\nHost:%s\r\nConnection:close\r\n\r\n", filename, host);

           // write request
           write(s_remote, request_buffer, strlen(request_buffer));

           // reset buffer
           for(i = 0; i < BUFFER_SIZE; i++) request_buffer[i] = 0;

           // receive response
           while( t = read(s_remote, response_buffer, BUFFER_SIZE))
               write(s_double, response_buffer, t);

           // close socket
           close(s_remote);

       } else if( !strcmp(method, "CONNECT") ) { // CONNECT www.example.com:443 HTTP/1.1

           // parse host and port
           host = uri;

           // end of host
           for(i = 0; uri[i] != ':'; i++);

           // null-terminate
           uri[i++] = 0;

           // set port
           port = uri + i;

           // resolve host name
           printf("CONNECT host=%s\n", host);
           struct hostent * remote = gethostbyname(host);

           // terminate if error
           if (remote == NULL) {
               printf("gethostbyname() failed.\n");
               return 1;
           }

           // create socket to connect to the remote
           s_remote = socket( AF_INET, SOCK_STREAM, 0);

           // terminate if error
           if( s_remote == -1 ) {
               perror("socket() failed");
               return 1;
           }

           // setup remote address
           server_address.sin_family    = AF_INET;
           server_address.sin_port      = htons( (unsigned short) atoi(port) );
           server_address.sin_addr.s_addr = * ( unsigned int* ) remote -> h_addr;

           // connect to the remote server
           if( -1 == connect( s_remote, (struct sockaddr *) &server_address, sizeof(struct sockaddr_in))) {
               perror("connect() failed");
               return 1;
           }

           // create request
           snprintf(request_buffer, BUFFER_SIZE, "HTTP/1.1 200 Established\r\n\r\n");

           // write request
           write(s, request_buffer, strlen(request_buffer));

           // reset buffer
           for(i = 0; i < BUFFER_SIZE; i++) request_buffer[i] = 0;

           // s_remote is the socket to the server
           if( fork() ) { // parent

               // read response from server and forwards it to the client
               for(i = 0; t = read(s_remote, response_buffer + i, BUFFER_SIZE - i); i+=t) {

                   // write response
                   write(s_double, response_buffer, strlen(response_buffer));

                   // reset buffer
                   for(i = 0; i < BUFFER_SIZE; i++) response_buffer[i] = 0;
               }

           } else { // child

               // receive response from client and forwards it to the server
               for(i = 0; t = read(s_double, response_buffer + i, BUFFER_SIZE - i); i+=t) {

                   // write response
                   write(s_remote, response_buffer, strlen(response_buffer));

                   // reset buffer
                   for(i = 0; i < BUFFER_SIZE; i++) response_buffer[i] = 0;
               }

               close(s_remote);
               exit(1);

           }

       } else {
           // create response
           sprintf(response_buffer, "HTTP/1.1 501 Not Implemented\r\n\r\n");

           // send
           write(s_double, response_buffer, strlen(response_buffer));
       }

       // close socket and kill process
       close(s_double);
       exit(1);

   }

   return 0;

} // main
```