

1 Server

```
1  #include <stdio.h> // printf, perror, fopen, fread, feof, fclose
2  #include <string.h> // strlen
3  #include <stdlib.h> // exit
4  #include <unistd.h> // read, write, fork
5  #include <sys/socket.h> // socket, bind, listen, accept
6  #include <arpa/inet.h> // htons, sockaddr, sockaddr_in
7
8
9  // constants
10 #define PORT 31415
11 #define BUFFER_SIZE 1024
12
13
14
15 struct char_map {
16     char * key;
17     char * value;
18 };
19
20
21
22 int main() {
23
24     // local variables
25     int s, s_double; // sockets
26     char * command_line; // first line of request
27     struct char_map headers[100] = {{NULL, NULL}}; // headers
28     char header_buffer[BUFFER_SIZE] = {0}; // header buffer, here there will be all the
        info from the header
29     char response_buffer[BUFFER_SIZE] = {0}; // response buffer, will be used to temporarily
        store the response
30     char * method, * uri, * version; // parsed values from command_line
31     int i, yes = 1; // generic index
32
33     // define address
34     struct sockaddr_in server_address;
35     struct sockaddr_in client_address;
36
37
38
39
40
41     // socket
42     s = socket( AF_INET, SOCK_STREAM, 0);
43
44     // terminate if error
45     if( s == -1 ) {
46         perror("socket() failed");
47         return 1;
48     }
49
50
51     if ( -1 == setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) ) {
52         perror("setsockopt() failed");
```

```

53     return 1;
54 }
55
56 // define address
57 server_address.sin_family = AF_INET;
58 server_address.sin_port = htons(PORT);
59 server_address.sin_addr.s_addr = 0;
60
61 // bind
62 if( -1 == bind(s, (struct sockaddr *) &server_address, sizeof(struct sockaddr_in)) ) {
63     perror("bind() failed");
64     return 1;
65 }
66
67 // listen
68 if( -1 == listen(s, 5) ) {
69     perror("listen() failed");
70     return 1;
71 }
72
73
74 int sockaddr_size = sizeof(struct sockaddr);
75
76 while(1) {
77
78     // accept
79     s_double = accept(s, (struct sockaddr *) &client_address, &sockaddr_size);
80
81
82     // create sub-process
83     if (fork()) {
84         close(s_double);
85         continue;
86     }
87
88     // terminate if error
89     if( s_double == -1 ) {
90         perror("accept() failed");
91         return 1;
92     }
93
94
95     // parse the header
96     command_line = headers[0].key = header_buffer;
97     int lines = 0;
98
99     for(i = 0; read(s_double, header_buffer + i, 1); i++) {
100
101         // end of the line
102         if(header_buffer[i - 1] == '\r' && header_buffer[i] == '\n') {
103
104             // null-terminate
105             header_buffer[i - 1] = 0;
106
107             // check if it is the end
108             if( !headers[lines].key[0] )

```

```

109         break;
110
111         // create new line on the headers
112         lines++;
113         headers[lines].key = &header_buffer[i + 1];
114     }
115
116     if( header_buffer[i] == ':' && (headers[lines].value == NULL)) {
117
118         // start value
119         headers[lines].value = &header_buffer[i + 1] + 1;
120
121         // null-terminate
122         header_buffer[i] = 0;
123
124     }
125 }
126
127 // print headers
128 for(i = 0; i < lines; i++)
129     printf("%s ----> %s\n", headers[i].key, headers[i].value);
130
131
132 // parse method, uri, version
133 method = command_line;
134 for(i = 0; command_line[i] != ' '; i++);
135 command_line[i++] = 0;
136
137 uri = command_line + i;
138 for(; command_line[i] != ' '; i++);
139 command_line[i++] = 0;
140
141 version = command_line + i;
142 for(; command_line[i] != 0; i++);
143 command_line[i++] = 0;
144
145 // print values
146 printf("Method ----> %s\nURI ----> %s\nVersion ----> %s\n", method, uri, version);
147
148
149 // opens file
150 FILE * file = fopen(uri + 1, "rw");
151
152 if( file == NULL ) {
153
154     // create 404 response
155     sprintf(response_buffer, "HTTP/1.1 404 NOT FOUND\r\n\r\n<html><h1>File %s was not  
found.</h1></html>", uri);
156
157     // send response
158     if( -1 == write(s_double, response_buffer, strlen(response_buffer)) ){
159         perror("write() failed");
160         return 1;
161     }
162 } else {
163

```

```

164
165
166
167 // send accept header
168 sprintf(response_buffer, "HTTP/1.1 200 OK\r\n\r\n");
169
170 if( -1 == write(s_double, response_buffer, strlen(response_buffer)) ){
171     perror("write() failed");
172     return 1;
173 }
174
175 // read and send the file
176 while( !feof(file) ) {
177
178     // read 1Kb from the file
179     fread(response_buffer, 1, 1024, file);
180
181     // write the answer
182     if( -1 == write(s_double, response_buffer, strlen(response_buffer)) ){
183         perror("write() failed");
184         return 1;
185     }
186
187     for(i = 0; i < BUFFER_SIZE; i++) response_buffer[i] = 0;
188
189 }
190
191 fclose(file);
192
193 }
194
195
196 printf("\n\n\n");
197
198 // close socket and kill process
199 close(s_double);
200 exit(1);
201
202 }
203
204 return 0;
205
206 } // main

```

1.1 Content-Length

```
1  content_length = 0;
2
3  // get content length of the file
4  while( fgetc(file) != EOF )
5      content_length++;
6
7  printf("Content-Length: %d\n\n", content_length);
8
9
10 // send accept header
11 sprintf(response_buffer, "HTTP/1.1 200 OK\r\nContent-Length: %d\r\n\r\n", content_length);
12
13 if( -1 == write(s_double, response_buffer, strlen(response_buffer)) ){
14     perror("write() failed");
15     return 1;
16 }
17
18 // reset buffer
19 for(i = 0; i < BUFFER_SIZE; i++) response_buffer[i] = 0;
20
21
22
23 // pointer to the beginning
24 rewind(file);
25
26 // read and send the file
27 while( !feof(file) ) {
28
29     // read 1Kb from the file
30     fread(response_buffer, 1, 1024, file);
31
32     // write the answer
33     if( -1 == write(s_double, response_buffer, strlen(response_buffer)) ){
34         perror("write() failed");
35         return 1;
36     }
37
38     for(i = 0; i < BUFFER_SIZE; i++) response_buffer[i] = 0;
39
40 }
41
42 fclose(file);
```

1.2 Reflect

```
1 // process the '/reflect' request
2 if ( !strcmp(uri, "/reflect", strlen("/reflect")) ) {
3
4     // send accept header
5     sprintf(response_buffer, "HTTP/1.1 200 OK\r\n\r\n");
6
7     if( -1 == write(s_double, response_buffer, strlen(response_buffer)) ){
8         perror("write() failed");
9         return 1;
10    }
11
12    for(i = 0; i < BUFFER_SIZE; i++) response_buffer[i] = 0;
13
14
15
16    // send received request + CRLF
17    snprintf(response_buffer, BUFFER_SIZE, "%s %s %s\r\n", method, uri, version);
18
19    if( -1 == write(s_double, response_buffer, strlen(response_buffer)) ){
20        perror("write() failed");
21        return 1;
22    }
23
24    for(i = 0; i < BUFFER_SIZE; i++) response_buffer[i] = 0;
25
26
27    // send client IP and CRLF
28    char *client_ip_address = inet_ntoa(client_address.sin_addr); // extract ip address
29    A.B.C.D
30
31    snprintf(response_buffer, BUFFER_SIZE, "%s\r\n", client_ip_address);
32
33    if( -1 == write(s_double, response_buffer, strlen(response_buffer)) ){
34        perror("write() failed");
35        return 1;
36    }
37
38    for(i = 0; i < BUFFER_SIZE; i++) response_buffer[i] = 0;
39
40
41    // send port
42    snprintf(response_buffer, BUFFER_SIZE, "%d", ntohs(client_address.sin_port));
43
44    if( -1 == write(s_double, response_buffer, strlen(response_buffer)) ){
45        perror("write() failed");
46        return 1;
47    }
48
49    for(i = 0; i < BUFFER_SIZE; i++) response_buffer[i] = 0;
50
51    // close socket and kill process
52    close(s_double);
53    exit(1);
54 }
```

1.3 AUTH

```
1 // before accessing an existing file it is needed to AUTHENTICATE
2 if( !auth_value ) {
3
4     snprintf(response_buffer, BUFFER_SIZE, "HTTP/1.1 401 UNAUTHORIZED\r\nWWW-Authenticate:
5         Basic realm=\"Users\"\\r\\nConnection: close\\r\\n\\r\\n");
6
7     if( -1 == write(s_double, response_buffer, strlen(response_buffer)) ){
8         perror("write() failed");
9         return 1;
10    }
11
12    fclose(file);
13    close(s_double);
14
15    continue;
16 }
17
18
19
20 for(i = 0; i < BUFFER_SIZE; i++) response_buffer[i] = 0;
21
22 // extract base64_cred
23 for(i = 1; auth_value[i] != ' '; i++);
24 base64_cred = auth_value + i + 1;
25
26
27
28 snprintf(response_buffer, BUFFER_SIZE, "%s:%s", username, password);
29
30 if( strcmp( base64_cred, base64_encode(response_buffer, strlen(response_buffer)))){
31
32     printf("base64_cred = %s (%d)\\n", base64_cred, strlen(base64_cred));
33     printf("base64_corr = %s\\n", base64_encode(response_buffer, strlen(response_buffer)));
34
35     snprintf(response_buffer, BUFFER_SIZE, "HTTP/1.1 401 UNAUTHORIZED\r\nWWW-Authenticate:
36         Basic realm=\"Users\"\\r\\nConnection: close\\r\\n\\r\\n");
37
38     if( -1 == write(s_double, response_buffer, strlen(response_buffer)) ){
39         perror("write() failed");
40         return 1;
41    }
42
43    fclose(file);
44    close(s_double);
45
46    continue;
47 }
```

1.3.1 Base64

```

1 static const char base64_alphabet[] =
   "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";
2
3 char* base64_encode(const char* input_array, size_t input_size) {
4     // Allocate memory for the output string
5     char* output = (char*)malloc(((input_size + 2) / 3 * 4 + 1) * sizeof(char));
6     if (output == NULL) {
7         return NULL;
8     }
9
10    // Perform the Base64 encoding
11    size_t output_index = 0;
12    for (size_t i = 0; i < input_size; i += 3) {
13        // Encode the next 3 bytes
14        unsigned char byte1 = (i < input_size) ? input_array[i] : 0;
15        unsigned char byte2 = (i + 1 < input_size) ? input_array[i + 1] : 0;
16        unsigned char byte3 = (i + 2 < input_size) ? input_array[i + 2] : 0;
17
18        output[output_index++] = base64_alphabet[(byte1 >> 2) & 0x3F];
19        output[output_index++] = base64_alphabet[((byte1 & 0x3) << 4) | ((byte2 >> 4) & 0xF)];
20        output[output_index++] = base64_alphabet[((byte2 & 0xF) << 2) | ((byte3 >> 6) & 0x3)];
21        output[output_index++] = base64_alphabet[byte3 & 0x3F];
22    }
23
24    // Handle the case when the input size is not a multiple of 3
25    if (input_size % 3 == 1) {
26        output[output_index - 2] = '=';
27        output[output_index - 1] = '=';
28    } else if (input_size % 3 == 2) {
29        output[output_index - 1] = '=';
30    }
31
32    // Null-terminate the output string
33    output[output_index] = '\0';
34
35    return output;
36 }

```


1.4 Blacklist

```
1 // create link
2 sprintf(link, "%s%s", host, uri);
3
4 printf("%s\n", link);
5
6 // open blacklist
7 FILE * blacklist = fopen(BLACKLIST, "r");
8 char * blacklist_item;
9
10 // retrieve link from blacklist.txt if exists
11 while (fgets(blacklist_buffer, BUFFER_SIZE, blacklist)) {
12
13     // null terminate
14     blacklist_buffer[strlen(blacklist_buffer) - 1] = 0;
15
16     // if uri is in the blacklist
17     if ( !strcmp(blacklist_buffer, link, strlen(link)) ) {
18
19         if( referer ) {
20             printf("NOT NULL: %s\n", referer);
21             // parse the referer
22             for(i = 0; referer[i] != '/'; i++);
23             for(++i; referer[i] != ':'; i++);
24             for(++i; referer[i] != '/'; i++);
25
26             snprintf(response_buffer, BUFFER_SIZE, "HTTP/1.1 307 Temporary
                Redirect\r\nLocation: %s\r\nConnection: close\r\n\r\n", referer + i + 1);
27
28         } else {
29
30             printf("NULL: %s\n", referer);
31
32             snprintf(response_buffer, BUFFER_SIZE, "HTTP/1.1 403
                Forbidden\r\nConnection:close\r\n\r\n"
33                 "<html>"
34                 "<h1>You are not allowed to access in this
                page because it is blacklisted</h1>"
35                 "</html>");
36
37         }
38
39
40         if( -1 == write(s_double, response_buffer, strlen(response_buffer)) ){
41             perror("write() failed");
42             return 1;
43         }
44
45         for(i = 0; i < BUFFER_SIZE; i++) response_buffer[i] = 0;
46
47         // close socket and kill process
48         close(s_double);
49         exit(1);
50     }
51 }
```

```
52 |  
53 |     for(i = 0; i < BUFFER_SIZE; i++) blacklist_buffer[i] = 0;  
54 |  
55 | }
```

1.5 Cookie

```
1 // retrieve cookie value
2 for(i = 0; i < lines; i++)
3     if( !strcmp(headers[i].key, "Cookie") )
4         client_cookie_string = headers[i].value;
5
6 // if is not null
7 if( client_cookie_string ) {
8
9     // extract name and value of the Cookie
10    client_cookie_name = client_cookie_string;
11    for(i = 0; client_cookie_string[i] != '='; i++);
12    client_cookie_string[i++] = 0;
13
14    if(client_cookie_name)
15        client_cookie_value = atoi(client_cookie_string + i);
16 }
17
18 // if the client goes in the contact.html AND (does not have cookie OR the cookie name is incorrect OR
19 // the cookie value is incorrect)
20 if( !strcmp(uri, "/contact.html") && ( !client_cookie_name || strcmp(client_cookie_name,
21    COOKIE_NAME) || client_cookie_value != 1) ) {
22
23    snprintf(response_buffer, BUFFER_SIZE, "HTTP/1.1 403
24        Forbidden\r\nConnection:close\r\n\r\n<html><h1>You need to access <a
25        href=\"/index.html\">/index.html</a> before entering this page.</h1></html>");
26
27    write(s_double, response_buffer, strlen(response_buffer));
28
29    for(i = 0; i < BUFFER_SIZE; i++) response_buffer[i] = 0;
30
31    // close everything
32    fclose(file);
33    close(s_double);
34    exit(1);
35 }
36
37 // if client is in index.html AND (does not have cookie OR the cookie name is incorrect OR the cookie
38 // value is incorrect)
39 if( !strcmp(uri, "/index.html") && (!client_cookie_name || strcmp(client_cookie_name,
40    COOKIE_NAME) || client_cookie_value != 1) ) {
41
42    sprintf(response_buffer, "HTTP/1.1 200 OK\r\nSet-Cookie:%s=%d\r\n\r\n", COOKIE_NAME, 1);
43
44 } else if (!strcmp(uri, "/contact.html") && !strcmp(client_cookie_name, COOKIE_NAME) &&
45    client_cookie_value == 1) {
46
47    sprintf(response_buffer, "HTTP/1.1 200 OK\r\nSet-Cookie:%s=%d\r\n\r\n", COOKIE_NAME, 0);
48
49 } else {
50
51    sprintf(response_buffer, "HTTP/1.1 200 OK\r\n\r\n");
52
53 }
```

```
31 |  
32 | if( -1 == write(s_double, response_buffer, strlen(response_buffer)) ){  
33 |     perror("write() failed");  
34 |     return 1;  
35 | }  
36 |  
37 | for(i = 0; i < BUFFER_SIZE; i++) response_buffer[i] = 0;
```

1.6 Chunked

```
1 // create response header
2 sprintf(buffer, "HTTP/1.1 200 OK\r\nTransfer-Encoding: chunked\r\n\r\n");
3
4 // write response
5 if( write(s_double, buffer, strlen(buffer)) == -1 ) {
6     perror("write() failed");
7     return 1;
8 }
9
10 char chunk_size[20];
11
12 while( !feof(file) ){
13
14     // read at most 1Kb from the file
15     fread(buffer, 1, 1024, file);
16
17     // get first line of the response
18     sprintf(chunk_size, "%x\r\n", strlen(buffer));
19
20     // write the first line
21     if( write(s_double, chunk_size, strlen(chunk_size)) == -1 ) {
22         perror("write() failed");
23         return 1;
24     }
25
26     // write the chunk
27     if( write(s_double, buffer, strlen(buffer)) == -1 ) {
28         perror("write() failed");
29         return 1;
30     }
31
32     // end of the chunk
33     if( write(s_double, CRLF, strlen(CRLF)) == -1 ) {
34         perror("write() failed");
35         return 1;
36     }
37 }
38
39 // last chunk
40 sprintf(buffer, "0\r\n");
41
42 // write last chunk
43 if( write(s_double, buffer, strlen(buffer)) == -1 ) {
44     perror("write() failed");
45     return 1;
46 }
47 }
```

1.7 ETag

```
1 // get entity tag value by summing the ascii values of each character in the file
2 unsigned long e_tag_value = 0;
3 char character[1] = {0};
4
5 while( !feof(file) ) {
6     // read 1 character
7     fread(character, 1, 1, file);
8
9     // sums value
10    e_tag_value = e_tag_value + (unsigned long) character[0];
11 }
12
13 if( e_tag_request && (e_tag_request == e_tag_value) ) {
14
15     printf("\n\n");
16
17     // create header
18     sprintf(response_buffer, "HTTP/1.1 304 Not Modified\r\nETag: \"%d\"\r\nConnection:
19         close\r\n\r\n", e_tag_value);
20
21     if( -1 == write(s_double, response_buffer, strlen(response_buffer)) ){
22         perror("write() failed");
23         return 1;
24     }
25
26     // close file, socket and kill process
27     fclose(file);
28     close(s_double);
29     exit(1);
30 }
31
32 // send accept header
33 sprintf(response_buffer, "HTTP/1.1 200 OK\r\nETag: \"%d\"\r\n\r\n", e_tag_value);
34
35 if( -1 == write(s_double, response_buffer, strlen(response_buffer)) ){
36     perror("write() failed");
37     return 1;
38 }
39
40 // pointer back
41 rewind(file);
```

2 Client

```
1 #include <stdio.h>
2 #include <sys/socket.h>           // socket
3 #include <errno.h>                // errno
4 #include <arpa/inet.h>            // htons
5 #include <unistd.h>               // write
6 #include <string.h>               // strlen, strcmp
7 #include <stdlib.h>               // atoi
8
9
10 #define RESPONSE_SIZE 100 * 1024
11
12
13 char hbuf[10000];
14
15 struct headers{
16     char * n;
17     char * v;
18 } h[100];
19
20
21
22 int main(){
23
24     // local variables
25     struct sockaddr_in server_addr; // server address
26     int s;                          // socket
27     int t;                          // temporary
28     unsigned char * p;              // ip address pointer
29     int i, j;
30     char * statusline;
31
32
33     // create socket
34     s = socket( AF_INET, SOCK_STREAM, 0 );
35     // printf("Socket: %d\n", s);
36
37     if( s == -1){
38         printf("ERRNO = %d (%d)\n", errno, EAFNOSUPPORT);
39         perror("Socket fallita\n");
40         return 1;
41     }
42
43
44     /* Setup for request */
45
46     // set server addr
47     server_addr.sin_family = AF_INET;
48     server_addr.sin_port = htons(80);
49
50     // IPv4 server
51     p = (unsigned char *) &server_addr.sin_addr.s_addr;
52     p[0] = 142;   p[1] = 250;   p[2] = 187;   p[3] = 196;
53
54
```

```

55 // connect server
56 if(-1 == connect(s, (struct sockaddr *) &server_addr, sizeof(struct sockaddr_in))) {
57     perror("Connessione fallita\n");
58     return 1;
59 }
60
61
62 // send request
63 char * request = "GET / HTTP/1.1\r\n\r\n";
64 write(s, request, strlen(request));
65
66
67
68 statusline = h[0].n = hbuf;
69 j = 0;
70
71 // read headers
72 for( i = 0; read(s, hbuf + i, 1); i++ ){
73
74     // end of line
75     if( hbuf[i - 1] == '\r' && hbuf[i] == '\n'){
76
77         hbuf[i - 1] = 0;
78
79         if( !( h[j].n[0] ) )
80             break;
81
82         h[++j].n = &hbuf[i + 1];
83     }
84
85     // end of name
86     if( (hbuf[i] == ':') && (h[j].v == NULL) ){
87
88         h[j].v = &hbuf[i + 1];
89         hbuf[i] = 0;
90     }
91 }
92
93 // print headers
94 for(i = 0; i < j; i++)
95     printf("%s ----> %s\n", h[i].n, h[i].v);
96 printf("\n\n");
97
98 char response[RESPONSE_SIZE];
99
100 for ( i = 0; t = read(s, response + i, RESPONSE_SIZE - 1 - i); i += t ) {}
101
102 // null-terminate response
103 response[i] = 0;
104
105 printf("%s\n\n", response);
106
107
108 return 0;
109
110 } // main

```


2.1 Chunked

```
1  int content_length;
2  for(i = 0; i < j; i++)
3      if( !strcmp( h[i].n , "Content-Length" ))
4          content_length = atoi(h[i].v);
5
6  char response[2000000];
7  if ( !content_length ){
8
9      // read and print response
10     for ( i = 0; t = read(s, response + i, content_length - i); i += t ) {}
11
12     response[i] = 0;
13     printf("%s\n\n", response);
14
15     return 0; // end
16 }
17
18 // prepare reading chunks
19 long chunk_size;
20 char chunk_buffer[8];
21
22 // will contain all the read bytes
23 j = 0;
24
25 do { // exit when chunk_size == 0
26
27     // chunk_buffer and hex2dec concersion
28     for(chunk_size = 0, i = 0;
29         read(s, chunk_buffer + i, 1) && !(chunk_buffer[i - 1] == '\r' && chunk_buffer[i] ==
30             '\n');
31         i++) {
32
33         // to lower case
34         if( chunk_buffer[i] >= 'A' && chunk_buffer[i] <= 'F')
35             chunk_buffer[i] = chunk_buffer[i] - ('a' - 'A');
36
37         // conversion from letter to dec
38         if( chunk_buffer[i] >= 'a' && chunk_buffer[i] <= 'f')
39             chunk_size = chunk_size * 16 + chunk_buffer[i] - 'a' + 10;
40
41         // convert numbers too
42         if( chunk_buffer[i] >= '0' && chunk_buffer[i] <= '9')
43             chunk_size = chunk_size * 16 + chunk_buffer[i] - '0';
44
45     }
46
47     // read chunk and display inside response
48     for( i = 0; // iterator
49         t = read(s, response + j, chunk_size - i); // from response + j, add 'chunk-size - i- bytes
50         i += t, j += t); // increment i and totla bytes j
51
52     // read last 2 chars (CRLF)
53 }
```

```
54     read(s, chunk_buffer, 2);
55
56
57 } while( chunk_size );
58
59
60 // null-terminate
61 response[j] = 0;
62 printf("%s\n\n", response);
```

2.2 Caching

```
1  // substitute '/' with '_'
2  for( i = 0; i < strlen(file_name); i++)
3      if( file_name[i] == '/' )
4          file_name[i] = '_';
5
6  // save file cached file path
7  snprintf(file_path, 1024, "%s%s", CAHCE_PATH, file_name);
8
9  printf("Cache file path: %s\n", file_path);
10
11
12  // opens file
13  FILE * cache_file = fopen(file_path, "r");
14
15  if ( !cache_file ) {
16
17      printf("File '%s' does NOT exist.\n", file_path);
18
19      flag = 1;
20
21  } else {
22
23      printf("File '%s' EXISTS.\n", file_path);
24
25      // gets date
26      fgets(cache_date_string, 200, cache_file);
27      printf("File Date: %s\n", cache_date_string);
28
29      // converts date
30      struct tm cache_date = get_tm_date(cache_date_string);
31
32      // get current time
33      time_t now = time(NULL);
34      struct tm *current_time = localtime(&now);
35
36      printf("NOW: %d\n", mktime(current_time));
37      printf("CACHE: %d\n", mktime(&cache_date));
38
39      if (difftime(mktime(current_time), mktime(&cache_date)) > 1){
40          printf("EXPIRED.\n");
41          flag = 1;
42      }
43  }
44
45
46  /* enters if there is no cache or if the cache is expired */
47  if (flag) {
48
49      // send request
50      char * request = "GET / HTTP/1.0\r\n\r\n";
51      write(s, request, strlen(request));
52
53      // read and ignore header
54  }
```

```

55     for( i = 0; read(s, hbuf + i, 1); i++ ){
56
57         // end of line
58         if( hbuf[i - 1] == '\r' && hbuf[i] == '\n'){
59
60             hbuf[i - 1] = 0;
61
62             if( !( h[j].n[0] ) )
63                 break;
64
65             j++;
66             h[j].n = &hbuf[i + 1];
67         }
68
69         // end of name
70         if( (hbuf[i] == ':') && (h[j].v == NULL) ){
71
72             h[j].v = &hbuf[i + 1] + 1;
73             hbuf[i] = 0;
74         }
75     }
76
77
78     // opens the: create or erase everything
79     cache_file = fopen(file_path, "w");
80
81     // write date in the file
82     char date_str[100];
83     strftime(date_str, sizeof(date_str), "%a, %d %b %Y %H:%M:%S %Z", &real_expires_date);
84     fprintf(cache_file, "%s\n", date_str);
85
86     // add CRLF
87     fwrite(&CRLF, strlen(CRLF), 1, cache_file);
88
89     // write response body in the file
90     for ( i = 0; t = read(s, response + i, RESPONSE_SIZE - 1 - i); i += t );
91     fprintf(cache_file, "%s", response);
92     printf("\n\n\n\n%s\n\n", response);
93
94     fclose(cache_file);
95
96     return 0;
97 } // new request needed
98
99
100 printf("NOT EXPIRED.\n\n\n\n");
101
102
103
104 while ( !feof(cache_file) ) {
105
106     // reads 1KB
107     fread(response, 1024, 1, cache_file);
108     printf("%s\n", response);
109
110     // resets

```

```

111     for( i=0; i < 1024; i++) response[i] = 0;
112
113 }

```

2.2.1 Date parsing

```

1  struct tm get_tm_date(char * date_string) {
2
3  char * date_buffer = date_string;
4  struct tm date = {0};
5  int i;
6
7  // skip name of day
8  for(i = 0; date_string[i] != ','; i++);
9  date_string[i] = 0;
10 date.tm_wday = day2sunday(date_buffer);
11 date_string[++i] = 0;
12
13
14 // extract day
15 date_buffer = date_buffer + i + 1;
16 for(++i; date_string[i] != ' '; i++);
17 date_string[i++] = 0;
18
19 date.tm_mday = atoi(date_buffer);
20
21
22 // extract month
23 date_buffer = date_string + i;
24 for(; date_string[i] != ' '; i++);
25 date_string[i++] = 0;
26
27 date.tm_mon = month2int(date_buffer) - 1;
28
29 // [ . . . ]
30
31 // set the remaining fields
32 date.tm_isdst = -1; // Let mktime determine if DST is in effect
33
34 char date_str[100];
35 strftime(date_str, sizeof(date_str), "%a, %d %b %Y %H:%M:%S %Z", &date);
36 printf("Converted Date: %s\n", date_str);
37
38 return date;
39 }

```

3 Proxy

```
1  #include <stdio.h> // printf, perror, fopen, fread, feof, fclose
2  #include <string.h> // strlen
3  #include <stdlib.h> // exit
4  #include <unistd.h> // read, write, fork
5  #include <sys/socket.h> // socket, bind, listen, accept
6  #include <arpa/inet.h> // htons, sockaddr, sockaddr_in
7  #include <netdb.h> // gethostbyname
8
9
10 // constants
11 #define PORT 58141
12 #define BUFFER_SIZE 1024
13
14
15
16 struct char_map {
17     char * key;
18     char * value;
19 };
20
21
22
23 int main() {
24
25     // local variables
26     int i, t; // generic index, generic variable
27     int s, s_double, s_remote; // sockets
28     char * command_line; // first line of request
29     struct char_map headers[100] = {{NULL, NULL}}; // headers
30     char header_buffer[BUFFER_SIZE] = {0}; // header buffer, here there will be all the info
31     // from the header
32     char request_buffer[BUFFER_SIZE] = {0}; // request buffer, will be used to store and send
33     // the request
34     char response_buffer[BUFFER_SIZE] = {0}; // response buffer, will be used to temporarily
35     // store the response
36     char * method, * uri, * version; // parsed values from command_line
37     char * scheme, * host, * filename, * port; // parsed values from GET or CONNECT
38
39     // define address
40     struct sockaddr_in local_address;
41     struct sockaddr_in remote_address;
42     struct sockaddr_in server_address;
43
44
45     // socket
46     s = socket( AF_INET, SOCK_STREAM, 0);
47
48     // terminate if error
49     if( s == -1 ) {
50         perror("socket() failed");
51         return 1;
```

```

52     }
53
54
55     // define address
56     local_address.sin_family    = AF_INET;
57     local_address.sin_port      = htons(PORT);
58     local_address.sin_addr.s_addr = 0;
59
60     if ( -1 == setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) ) {
61         perror("setsockopt() failed");
62         return 1;
63     }
64
65     // bind
66     if( -1 == bind(s, (struct sockaddr *) &local_address, sizeof(struct sockaddr_in)) ) {
67         perror("bind() failed");
68         return 1;
69     }
70
71     // listen
72     if( -1 == listen(s, 10) ) {
73         perror("listen() failed");
74         return 1;
75     }
76
77     // initialize remote (client) address
78     remote_address.sin_family    = AF_INET;
79     remote_address.sin_port      = htons(0);
80     remote_address.sin_addr.s_addr = 0;
81
82
83     int sockaddr_size = sizeof(struct sockaddr);
84
85     while(1) {
86
87         // accept
88         s_double = accept(s, (struct sockaddr *) &remote_address, &sockaddr_size);
89
90
91         // create sub-process
92         if (fork()) {
93             close(s_double);
94             continue;
95         }
96
97         // terminate if error
98         if( s_double == -1 ) {
99             perror("accept() failed");
100             return 1;
101         }
102
103
104         // parse the header
105         command_line = headers[0].key = header_buffer;
106         int lines = 0;
107

```

```

108     for(i = 0; read(s_double, header_buffer + i, 1); i++) {
109
110         // end of the line
111         if(header_buffer[i - 1] == '\r' && header_buffer[i] == '\n') {
112
113             // null-terminate
114             header_buffer[i - 1] = 0;
115
116             // check if it is the end
117             if( !headers[lines].key[0] )
118                 break;
119
120             // create new line on the headers
121             lines++;
122             headers[lines].key = &header_buffer[i + 1];
123         }
124
125         if( header_buffer[i] == ':' && (headers[lines].value == NULL)) {
126
127             // start value
128             headers[lines].value = &header_buffer[i + 1] + 1;
129
130             // null-terminate
131             header_buffer[i] = 0;
132
133         }
134     }
135
136     // print headers
137     for(i = 0; i < lines; i++)
138         printf("%s ----> %s\n", headers[i].key, headers[i].value);
139
140
141     // parse method, uri, version
142     method = command_line;
143     for(i = 0; command_line[i] != ' '; i++);
144     command_line[i++] = 0;
145
146     uri = command_line + i;
147     for(; command_line[i] != ' '; i++);
148     command_line[i++] = 0;
149
150     version = command_line + i;
151     for(; command_line[i] != 0; i++);
152     command_line[i++] = 0;
153
154     // print values
155     printf("Method ----> %s\nURI ----> %s\nVersion ----> %s\n\n\n\n", method, uri,
156           version);
157
158
159     if( !strcmp(method, "GET") ) { // GET http://www.example.com/dir/file
160
161         scheme = uri;
162

```



```

163 // parse the URI address, by getting the host and the resource
164 for(i = 0; uri[i] != ':' && uri[i]; i++)
165
166 if (uri[i] == ':') // null terminate
167     uri[i++] = 0;
168 else { // check correctness
169     printf("Parsing error (expected ':').\n");
170     exit(1);
171 }
172
173 if (uri[i] != '/' || uri[i + 1] != '/') {
174     printf("Parsing error (expected '/').\n");
175     exit(1);
176 }
177
178 i = i + 2;
179
180 // save host
181 host = uri + (++i);
182
183 // find position where host finishes
184 for(; uri[i] && uri[i] != '/'; i++);
185
186
187 if (uri[i] == '/') // null terminate
188     uri[i++] = 0;
189 else { // check correctness
190     printf("Parsing error (expected '/').\n");
191     exit(1);
192 }
193
194 // initialize filename
195 filename = uri + i;
196
197
198 // resolve host name
199 printf("GET host=%s\n", host);
200 struct hostent * remote = gethostbyname(host);
201
202 // create socket to connect to the remote
203 s_remote = socket( AF_INET, SOCK_STREAM, 0);
204
205 // terminate if error
206 if( s_remote == -1 ) {
207     perror("socket() failed");
208     return 1;
209 }
210
211 // set up remote server address
212 server_address.sin_family = AF_INET;
213 server_address.sin_port = htons(80);
214 server_address.sin_addr.s_addr = *((unsigned int*)(remote->h_addr));
215
216
217 // connect to the remote server
218 if( -1 == connect( s_remote, (struct sockaddr *) &server_address, sizeof(struct

```

```

219         sockaddr_in))) {
220             perror("connect() failed");
221             return 1;
222         }
223
224         // create request
225         snprintf(request_buffer, BUFFER_SIZE, "GET %s
226             HTTP/1.1\r\nHost:%s\r\nConnection:close\r\n\r\n", filename, host);
227
228         // write request
229         write(s_remote, request_buffer, strlen(request_buffer));
230
231         // reset buffer
232         for(i = 0; i < BUFFER_SIZE; i++) request_buffer[i] = 0;
233
234         // receive response
235         while( t = read(s_remote, response_buffer, BUFFER_SIZE))
236             write(s_double, response_buffer, t);
237
238         // close socket
239         close(s_remote);
240
241     } else if( !strcmp(method, "CONNECT") ) { // CONNECT www.example.com:443 HTTP/1.1
242
243         // parse host and port
244         host = uri;
245
246         // end of host
247         for(i = 0; uri[i] != ':'; i++);
248
249         // null-terminate
250         uri[i++] = 0;
251
252         // set port
253         port = uri + i;
254
255         // resolve host name
256         printf("CONNECT host=%s\n", host);
257         struct hostent * remote = gethostbyname(host);
258
259         // terminate if error
260         if (remote == NULL) {
261             printf("gethostbyname() failed.\n");
262             return 1;
263         }
264
265         // create socket to connect to the remote
266         s_remote = socket( AF_INET, SOCK_STREAM, 0);
267
268         // terminate if error
269         if( s_remote == -1 ) {
270             perror("socket() failed");
271             return 1;
272         }
273
274         // setup remote address

```

```

273     server_address.sin_family    = AF_INET;
274     server_address.sin_port      = htons( (unsigned short) atoi(port) );
275     server_address.sin_addr.s_addr = * ( unsigned int* ) remote -> h_addr;
276
277     // connect to the remote server
278     if( -1 == connect( s_remote, (struct sockaddr *) &server_address, sizeof(struct
        sockaddr_in))) {
279         perror("connect() failed");
280         return 1;
281     }
282
283     // create request
284     snprintf(request_buffer, BUFFER_SIZE, "HTTP/1.1 200 Established\r\n\r\n");
285
286     // write request
287     write(s, request_buffer, strlen(request_buffer));
288
289     // reset buffer
290     for(i = 0; i < BUFFER_SIZE; i++) request_buffer[i] = 0;
291
292     // s_remote is the socket to the server
293     if( fork() ) { // parent
294
295         // read response from server and forwards it to the client
296         for(i = 0; t = read(s_remote, response_buffer + i, BUFFER_SIZE - i); i+=t) {
297
298             // write response
299             write(s_double, response_buffer, strlen(response_buffer));
300
301             // reset buffer
302             for(i = 0; i < BUFFER_SIZE; i++) response_buffer[i] = 0;
303         }
304
305     } else { // child
306
307         // receive response from client and forwards it to the server
308         for(i = 0; t = read(s_double, response_buffer + i, BUFFER_SIZE - i); i+=t) {
309
310             // write response
311             write(s_remote, response_buffer, strlen(response_buffer));
312
313             // reset buffer
314             for(i = 0; i < BUFFER_SIZE; i++) response_buffer[i] = 0;
315         }
316
317         close(s_remote);
318         exit(1);
319
320     }
321
322 } else {
323     // create response
324     sprintf(response_buffer, "HTTP/1.1 501 Not Implemented\r\n\r\n");
325
326     // send
327     write(s_double, response_buffer, strlen(response_buffer));

```

```
328     }
329
330     // close socket and kill process
331     close(s_double);
332     exit(1);
333
334 }
335
336 return 0;
337
338 } // main
```