Concurrent HTTP Proxy Server

CS425 - Computer Networks

Vaibhav Nagar(14785)

Email: vaibhavn@iitk.ac.in

August 31, 2016

Elementary features of Proxy Server

- Proxy server supports the GET method to serve clients requests and then connects to requested host and responds with host data.
- It can handle requests from both protocols HTTP/1.1 and HTTP/1.0.
- Multiple clients can connect and make request to it at the same time.
- Server perfectly provides appropriate Status-code and Response-Phrase values in response to errors or incorrect requests from client.
- Server is designed such that it can run continuously until an unrecoverable error occurs.
- The whole code is developed only in C language using its various libraries and string parsing library.

Design Choices

The code is written in imperative manner which uses a string parsing library.

- In contrast to Apache server which has fixed buffer size of 8KB, buffer size of my proxy server is fixed to 4KB for handling requests from client.
- Proxy server blocks after the connection from client is established until it gets double carriage return in the request made.
- Proxy server forks one new child process to handle a single GET request from the client.
- Proxy server uses the same 4KB buffer for storing the data from requested host and send to the client.
- Proxy server ensures two headers- Host and Connection: Close must be sent to the requested host.

Test Procedure

Testing has been done on the VM provided and as well as on the Ubuntu 14.04 machine. Mozilla Firefox is used as a client which is on the same machine and as well as on different machines in the same network. Moreover it has been tested using telnet and curl as well.
Server smoothly handled all the requests made by client during testing and provided the correct responses from requested host.

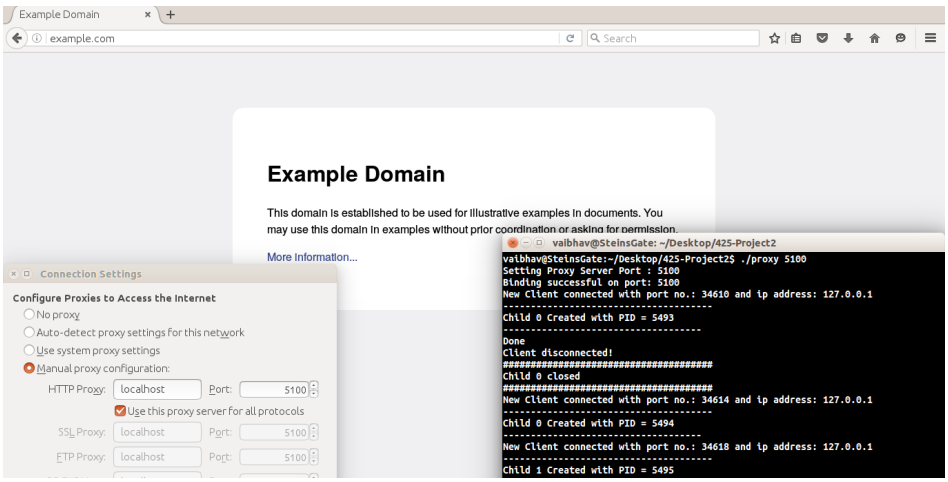

Figure 1: HTTP request from firefox to proxy server

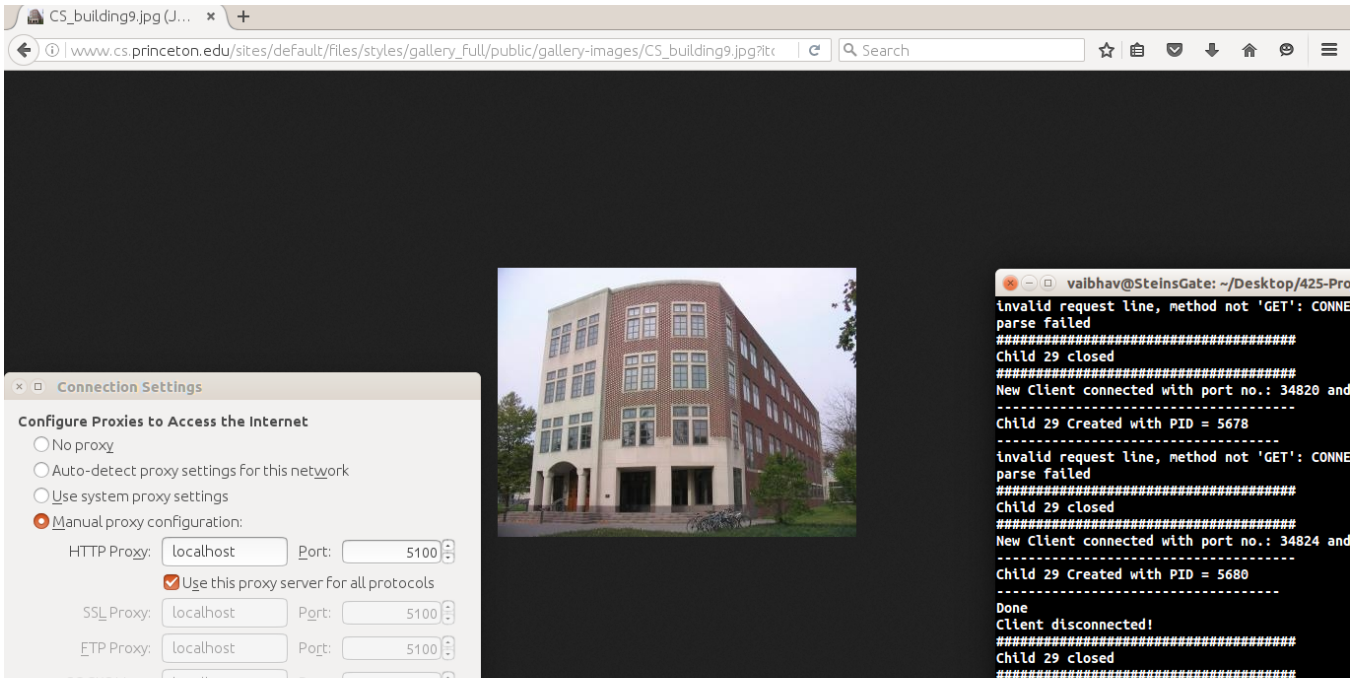

Figure 2: Image request from client

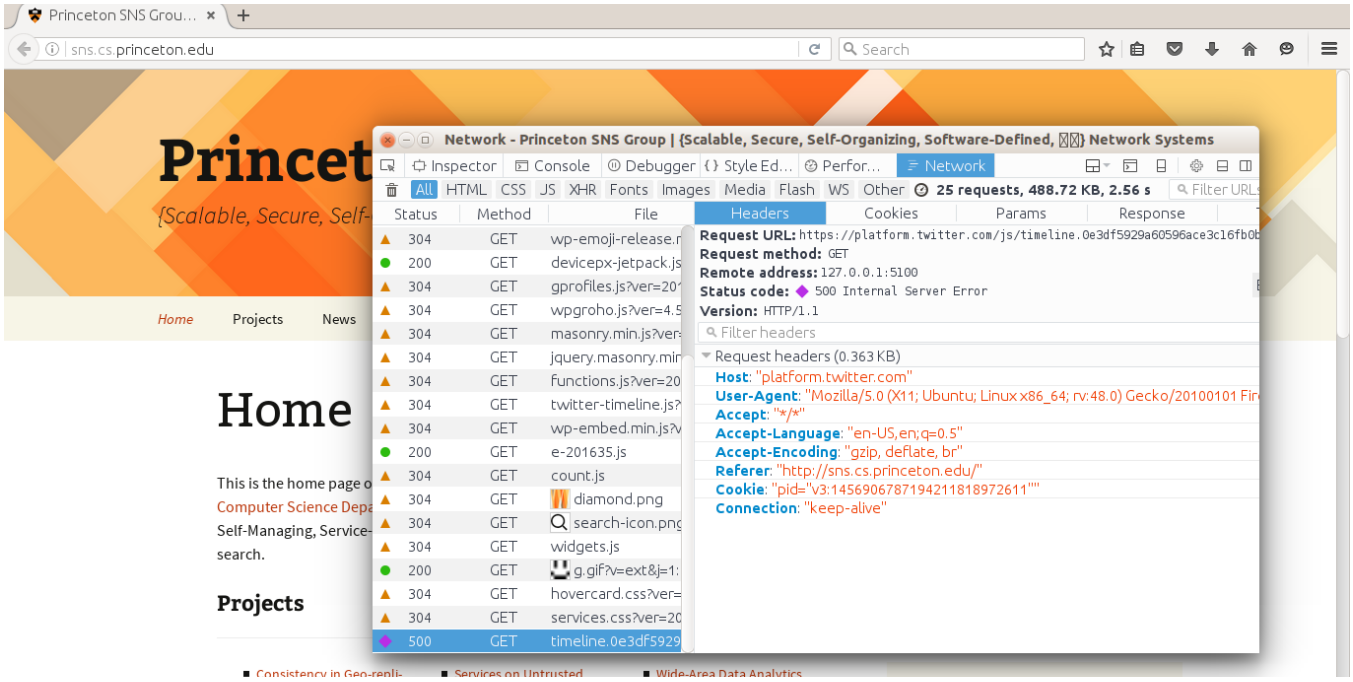

Figure 3: 500:Internal server error response for HTTPS request

Summary

Proxy Server successfully handles all GET requests made by client and create new child processes for each new request. Supports both protocols- HTTP 1.1 and 1.0. Server sends appropriate status code and response phrase message depending upon the type of error and request.

However, some test cases in python testing scripts which are checking the proxy response and direct response line by line due to which they were not able to passed. Line by line check is not reliable as some headers like Age, Date, Keep-alive header contains a max field whose values change with each new request.

Also I was facing some issue with http://example.com/ website which is taking too much time to get data from python script. Then after editing the script such that it makes only direct request to example.com, then I realized it was the issue of the script itself and not my proxy server.

In python_test.py script, my proxy server passed 3/4 or sometimes 4/4 test cases whereas, in python_test_conc.py script, my server passed 11/12 when I removed the website example.com from its urls array as it was taking too much time to respond.

Appendix

Source Code

```
1
2
3  #include "proxy_parse.h"
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <string.h>
8  #include <sys/types.h>
9  #include <sys/socket.h>
10 #include <netinet/in.h>
11 #include <netdb.h>
12 #include <arpa/inet.h>
13 #include <unistd.h>
14 #include <fcntl.h>
15 #include <time.h>
16 #include <sys/wait.h>
17 #include <errno.h>
18
19 #define MAX_BYTES 4096
20 #define MAX_CLIENTS 400
21
22 int port = 5100;                    // Default Port
23 int socketId;                      // Server Socket ID
24
25 pid_t client_PID[MAX_CLIENTS];              // PID of connected clients
26
27
28 int sendErrorMessage(int socket, int status_code)
29 {
30   char str[1024];
31   char currentTime[50];
32   time_t now = time(0);
33
34   struct tm data = *gmtime(&now);
35   strftime(currentTime,sizeof(currentTime),"%a, %d %b %Y %H:%M:%S %Z", &data);
36
37   switch(status_code)
38   {
39     case 400: snprintf(str, sizeof(str), "HTTP/1.1 400 Bad Request\r\nContent-
   Length: 95\r\nConnection: keep-alive\r\nContent-Type: text/html\r\nDate: %s\r\
   nServer: VaibhavN/14785\r\n\r\n<HTML><HEAD><TITLE>400 Bad Request</TITLE></HEAD
   >\n<BODY><H1>400 Bad Rqeuest</H1>\n</BODY></HTML>", currentTime);
40          printf("400 Bad Request\n");
41          send(socket, str, strlen(str), 0);
42          break;
43
44     case 403: snprintf(str, sizeof(str), "HTTP/1.1 403 Forbidden\r\nContent-Length
   : 112\r\nContent-Type: text/html\r\nConnection: keep-alive\r\nDate: %s\r\nServer
   : VaibhavN/14785\r\n\r\n<HTML><HEAD><TITLE>403 Forbidden</TITLE></HEAD>\n<BODY><
   H1>403 Forbidden</H1><br>Permission Denied\n</BODY></HTML>", currentTime);
```

```
45              printf("403 Forbidden\n");
46              send(socket, str, strlen(str), 0);
47              break;
48
49      case 404: snprintf(str, sizeof(str), "HTTP/1.1 404 Not Found\r\nContent-Length
    : 91\r\nContent-Type: text/html\r\nConnection: keep-alive\r\nDate: %s\r\nServer:
     VaibhavN/14785\r\n\r\n<HTML><HEAD><TITLE>404 Not Found</TITLE></HEAD>\n<BODY><
    H1>404 Not Found</H1>\n</BODY></HTML>", currentTime);
50              printf("404 Not Found\n");
51              send(socket, str, strlen(str), 0);
52              break;
53
54      case 500: snprintf(str, sizeof(str), "HTTP/1.1 500 Internal Server Error\r\
    nContent-Length: 115\r\nConnection: keep-alive\r\nContent-Type: text/html\r\
    nDate: %s\r\nServer: VaibhavN/14785\r\n\r\n<HTML><HEAD><TITLE>500 Internal
    Server Error</TITLE></HEAD>\n<BODY><H1>500 Internal Server Error</H1>\n</BODY></
    HTML>", currentTime);
55              //printf("500 Internal Server Error\n");
56              send(socket, str, strlen(str), 0);
57              break;
58
59      case 501: snprintf(str, sizeof(str), "HTTP/1.1 501 Not Implemented\r\nContent-
    Length: 103\r\nConnection: keep-alive\r\nContent-Type: text/html\r\nDate: %s\r\
    nServer: VaibhavN/14785\r\n\r\n<HTML><HEAD><TITLE>404 Not Implemented</TITLE></
    HEAD>\n<BODY><H1>501 Not Implemented</H1>\n</BODY></HTML>", currentTime);
60              printf("501 Not Implemented\n");
61              send(socket, str, strlen(str), 0);
62              break;
63
64      case 505: snprintf(str, sizeof(str), "HTTP/1.1 505 HTTP Version Not Supported\
    r\nContent-Length: 125\r\nConnection: keep-alive\r\nContent-Type: text/html\r\
    nDate: %s\r\nServer: VaibhavN/14785\r\n\r\n<HTML><HEAD><TITLE>505 HTTP Version
    Not Supported</TITLE></HEAD>\n<BODY><H1>505 HTTP Version Not Supported</H1>\n</
    BODY></HTML>", currentTime);
65              printf("505 HTTP Version Not Supported\n");
66              send(socket, str, strlen(str), 0);
67              break;
68
69      default:  return -1;
70
71    }
72
73    return 1;
74  }
75
76
77
78  int connectRemoteServer(char* host_addr, int port_num)
79  {
80    // Creating Socket for remote server---------------------------
81
82    int remoteSocket = socket(AF_INET, SOCK_STREAM, 0);
83
84    if( remoteSocket < 0)
```

```
85      {
86        printf("Error in Creating Socket.\n");
87        return -1;
88      }
89
90      // Get host by the name or ip address provided
91
92      struct hostent *host = gethostbyname(host_addr);
93      if(host == NULL)
94      {
95        fprintf(stderr, "No such host exists.\n");
96        return -1;
97      }
98
99      // inserts ip address and port number of host in struct `server_addr`
100     struct sockaddr_in server_addr;
101
102     bzero((char*)&server_addr, sizeof(server_addr));
103     server_addr.sin_family = AF_INET;
104     server_addr.sin_port = htons(port_num);
105
106     bcopy((char *)host->h_addr,(char *)&server_addr.sin_addr.s_addr,host->h_length);
107
108     // Connect to Remote server ----------------------------------------------------
109
110     if( connect(remoteSocket, (struct sockaddr*)&server_addr, (socklen_t)sizeof(
    server_addr)) < 0 )
111     {
112       fprintf(stderr, "Error in connecting !\n");
113       return -1;
114     }
115
116     return remoteSocket;
117   }
118
119
120   int handleGETrequest(int clientSocket, ParsedRequest *request, char *buf)
121   {
122     strcpy(buf, "GET ");
123     strcat(buf, request->path);
124     strcat(buf, " ");
125     strcat(buf, request->version);
126     strcat(buf, "\r\n");
127
128     size_t len = strlen(buf);
129
130     if (ParsedHeader_set(request, "Connection", "close") < 0){
131       printf("set header key not work\n");
132       //return -1;                    // If this happens Still try to send request
    without header
133     }
134
135     if(ParsedHeader_get(request, "Host") == NULL)
136     {
```

```
137       if(ParsedHeader_set(request, "Host", request->host) < 0){
138         printf("Set \"Host\" header key not working\n");
139       }
140     }
141 
142     if (ParsedRequest_unparse_headers(request, buf + len, (size_t)MAX_BYTES - len) <
      0) {
143       printf("unparse failed\n");
144       //return -1;                // If this happens Still try to send request
    without header
145     }
146 
147 
148     int server_port = 80;                 // Default Remote Server Port
149     if(request->port != NULL)
150       server_port = atoi(request->port);
151 
152     int remoteSocketID = connectRemoteServer(request->host, server_port);
153 
154     if(remoteSocketID < 0)
155       return -1;
156 
157     int bytes_send = send(remoteSocketID, buf, strlen(buf), 0);
158 
159     bzero(buf, MAX_BYTES);
160 
161     bytes_send = recv(remoteSocketID, buf, MAX_BYTES-1, 0);
162 
163     while(bytes_send > 0)
164     {
165       bytes_send = send(clientSocket, buf, bytes_send, 0);
166 
167       if(bytes_send < 0)
168       {
169         perror("Error in sending data to client socket.\n");
170         break;
171       }
172 
173       bzero(buf, MAX_BYTES);
174 
175       bytes_send = recv(remoteSocketID, buf, MAX_BYTES-1, 0);
176 
177 
178     }
179     printf("Done\n");
180 
181     bzero(buf, MAX_BYTES);
182 
183     close(remoteSocketID);
184 
185     return 0;
186 
187   }
188 
```

```
189
190 int checkHTTPversion(char *msg)
191 {
192   int version = -1;
193
194   if(strncmp(msg, "HTTP/1.1", 8) == 0)
195   {
196     version = 1;
197   }
198   else if(strncmp(msg, "HTTP/1.0", 8) == 0)
199   {
200     version = 1;                    // Handling this similar to version 1.1
201   }
202   else
203     version = -1;
204
205   return version;
206 }
207
208
209
210 int requestType(char *msg)
211 {
212   int type = -1;
213
214   if(strncmp(msg, "GET\0",4) == 0)
215     type = 1;
216   else if(strncmp(msg, "POST\0",5) == 0)
217     type = 2;
218   else if(strncmp(msg, "HEAD\0",5) == 0)
219     type = 3;
220   else
221     type = -1;
222
223   return type;
224 }
225
226
227
228 void respondClient(int socket)
229 {
230
231   int bytes_send, len;                          // Bytes Transferred
232
233
234   char *buffer = (char*)calloc(MAX_BYTES,sizeof(char));     // Creating buffer of
    4kb for a client
235
236   //bzero(buffer, MAX_BYTES);                     // Make buffer zero
237
238   bytes_send = recv(socket, buffer, MAX_BYTES, 0);         // Receive Request
239
240   while(bytes_send > 0)
241   {
```

```
242       len = strlen(buffer);
243       if(strstr(buffer, "\r\n\r\n") == NULL)
244       {
245         //printf("Carriage Return Not found!\n");
246         bytes_send = recv(socket, buffer + len, MAX_BYTES - len, 0);
247       }
248       else{
249         break;
250       }
251     }
252
253     if(bytes_send > 0)
254     {
255       //printf("%s\n",buffer);
256       len = strlen(buffer);
257
258       //Create a ParsedRequest to use. This ParsedRequest
259       //is dynamically allocated.
260       ParsedRequest *req = ParsedRequest_create();
261
262       if (ParsedRequest_parse(req, buffer, len) < 0)
263       {
264         sendErrorMessage(socket, 500);                    // 500 internal error
265           printf("parse failed\n");
266       }
267       else
268       {
269
270         bzero(buffer, MAX_BYTES);
271
272         int type = requestType(req->method);
273
274         if(type == 1)                        // GET Request
275         {
276           if( req->host && req->path && (checkHTTPversion(req->version) == 1) )
277           {
278             bytes_send = handleGETrequest(socket, req, buffer);   // Handle GET
    request
279             if(bytes_send == -1)
280             {
281               sendErrorMessage(socket, 500);
282             }
283           }
284
285           else
286             sendErrorMessage(socket, 500);           // 500 Internal Error
287
288         }
289         else if(type == 2)                     // POST Request
290         {
291           printf("POST: Not implemented\n");
292           sendErrorMessage(socket, 500);
293         }
294         else if(type == 3)                     // HEAD Request
```

```
295         {
296           printf("HEAD: Not implemented\n");
297           sendErrorMessage(socket, 500);
298         }
299         else                            // Unknown Method Request
300         {
301           printf("Unknown Method: Not implemented\n");
302           sendErrorMessage(socket, 500);
303         }
304 
305 
306       }
307 
308       ParsedRequest_destroy(req);
309 
310     }
311 
312     if( bytes_send < 0)
313     {
314       perror("Error in receiving from client.\n");
315     }
316     else if(bytes_send == 0)
317     {
318       printf("Client disconnected!\n");
319     }
320 
321     shutdown(socket, SHUT_RDWR);
322     close(socket);                          // Close socket
323     free(buffer);
324     return;
325   }
326 
327 
328 
329   int findAvailableChild(int i)
330   {
331     int j = i;
332     pid_t ret_pid;
333     int child_state;
334 
335     do
336     {
337       if(client_PID[j] == 0)
338         return j;
339       else
340       {
341         ret_pid = waitpid(client_PID[j], &child_state, WNOHANG);     // Finds status
    change of pid
342 
343         if(ret_pid == client_PID[j])                  // Child exited
344         {
345           client_PID[j] = 0;
346           return j;
347         }
```

```
348         else if(ret_pid == 0)                       // Child is still running
349         {
350           ;
351         }
352         else
353           perror("Error in waitpid call\n");
354       }
355       j = (j+1)%MAX_CLIENTS;
356     }
357     while(j != i);
358
359     return -1;
360   }
361
362
363
364   int main(int argc, char * argv[]) {
365
366     int newSocket, client_len;
367
368     struct sockaddr_in server_addr, client_addr;
369
370     bzero(client_PID, MAX_CLIENTS);
371
372     // Fetching Arguments
    ---------------------------------------------------------------------------
373     int params = 1;
374
375     if(argc == 2)
376     {
377       port = atoi(argv[params]);
378     }
379     else
380     {
381       printf("Wrong Arguments! Usage: %s <port-number>\n", argv[0]);
382       exit(1);
383     }
384
385     printf("Setting Proxy Server Port : %d\n", port);
386
387
388     // Creating socket
    ---------------------------------------------------------------------------
389
390     socketId = socket(AF_INET, SOCK_STREAM, 0);
391
392     if( socketId < 0)
393     {
394       perror("Error in Creating Socket.\n");
395       exit(1);
396     }
397
```

```
398     int reuse =1;
399     if (setsockopt(socketId, SOL_SOCKET, SO_REUSEADDR, (const char*)&reuse, sizeof(
        reuse)) < 0)
400         perror("setsockopt(SO_REUSEADDR) failed\n");
401
402     //
        ------------------------------------------------------------------------------
403
404     // Binding socket with given port number and server is set to connect with any
        ip address----------
405
406     bzero((char*)&server_addr, sizeof(server_addr));
407     server_addr.sin_family = AF_INET;
408     server_addr.sin_port = htons(port);
409     server_addr.sin_addr.s_addr = INADDR_ANY;
410
411     if( bind(socketId, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0 )
412     {
413       perror("Binding Error : Port may not be free. Try Using diffrent port number.\
        n");
414       exit(1);
415     }
416
417     printf("Binding successful on port: %d\n",port);
418
419     //
        ------------------------------------------------------------------------------
420
421     // Listening for connections and accept upto MAX_CLIENTS in queue
        ------------------------------------
422
423     int status = listen(socketId, MAX_CLIENTS);
424
425     if(status < 0 )
426     {
427       perror("Error in Listening !\n");
428       exit(1);
429     }
430
431     //
        ------------------------------------------------------------------------------
432
433     // Infinite Loop for accepting connections
        ----------------------------------------------------
434
435     int i=0;
436     int ret;
437
438     while(1)
439     {
440       //printf("Listening for a client to connect!\n");
```

```
441       bzero((char*)&client_addr, sizeof(client_addr));              // Clears struct
     client_addr
442       client_len = sizeof(client_addr);
443
444       newSocket = accept(socketId, (struct sockaddr*)&client_addr,(socklen_t*)&
    client_len); // Accepts connection
445       if(newSocket < 0)
446       {
447         fprintf(stderr, "Error in Accepting connection !\n");
448         exit(1);
449       }
450
451
452       // Getting IP address and port number of client
453
454       struct sockaddr_in* client_pt = (struct sockaddr_in*)&client_addr;
455       struct in_addr ip_addr = client_pt->sin_addr;
456       char str[INET_ADDRSTRLEN];                    // INET_ADDRSTRLEN: Default ip
    address size
457       inet_ntop( AF_INET, &ip_addr, str, INET_ADDRSTRLEN );
458       printf("New Client connected with port no.: %d and ip address: %s \n",ntohs(
    client_addr.sin_port), str);
459
460
461       //
    ------------------------------------------------------------------------------------------------------------------------
462       // Forks new client
463
464       i = findAvailableChild(i);
465
466       if(i>= 0 && i < MAX_CLIENTS)
467       {
468         ret = fork();
469
470         if(ret == 0)                    // Create child process
471         {
472           respondClient(newSocket);
473           exit(0);                    // Child exits
474         }
475         else
476         {
477           printf("----------------------------------------\nChild %d Created with PID
    = %d\n----------------------------------------\n", i,ret);
478           client_PID[i] = ret;
479
480         }
481       }
482       else
483       {
484         i = 0;
485         close(newSocket);
486         printf("No more Client can connect!\n");
487       }
```

```
488
489
490         // And goes back to listen again for another client
491     }
492
493     close(socketId);                    // Close socket
494
495     return 0;
496 }
497
498
```

Listing 1: Concurrent HTTP Proxy Server

PS: The code really looks better than this in sublime text editor on full screen.