Concurrent HTTP Server

CS425 - Computer Networks

Vaibhav Nagar(14785)

Email: vaibhavn@iitk.ac.in

August 19, 2016

Elementary features of Server

- Server supports the GET method to retrieve files from it.
- 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 makes persistent connections with clients upto certain extent i.e. once the connection is established between server and client, server then starts handling requests from it using same connection and process until client closes the connection.
- Server is designed such that it can run continuously until an unrecoverable error occurs.
- Server has functionalities of setting port and root directory from command line.
- The whole code is developed only in C language using its various libraries.

Additional Features

Date and Server fields in the Response message Header

Server sends the current date and time in the same format as defined by the "RFC 7231 Date/Time Formats" in the Date field along with the name of the server in Server field of response message header.
Current date and time is fetched using time and gmtime functions defined in time.h library and the correct format is produced by the strftime function.

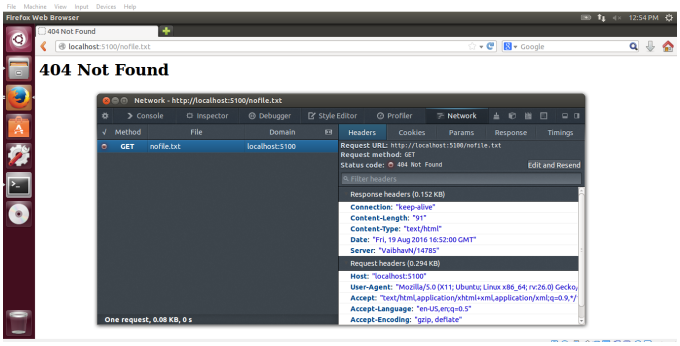

Figure 1: Listing all Hyperlinked files in directory images/

HyperLinked Directory

Server is capable of sending the files and folders in hyper-linked format when requested resource is directory. All the files are listed when the client sends the GET request with the path of a directory in its URI.
To achieve this objective dirent.h library is used, which provides opendir, readdir and closedir functions to get all the files present in the directory requested. Then the each file name is sent inside the HTML code as a message to client so that it can be listed in the hyperlink format.

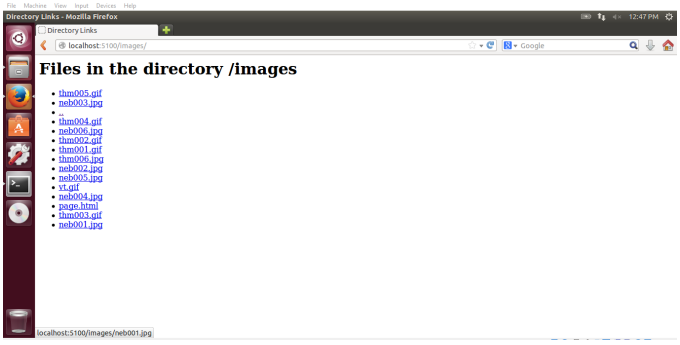

Figure 2: Listing all Hyperlinked files in directory images/

Test Procedure

Testing has been done on the VM provided and as well as on the Ubuntu 14.04 machine. Browsers like Mozilla Firefox, Chrome, Internet Explorer and Microsoft Edge are used as a clients in which some of them are on same machine (as a localhost) and some on different machines in the same network.
Server smoothly handled all the requests made by client during testing and provided the correct responses as expected.

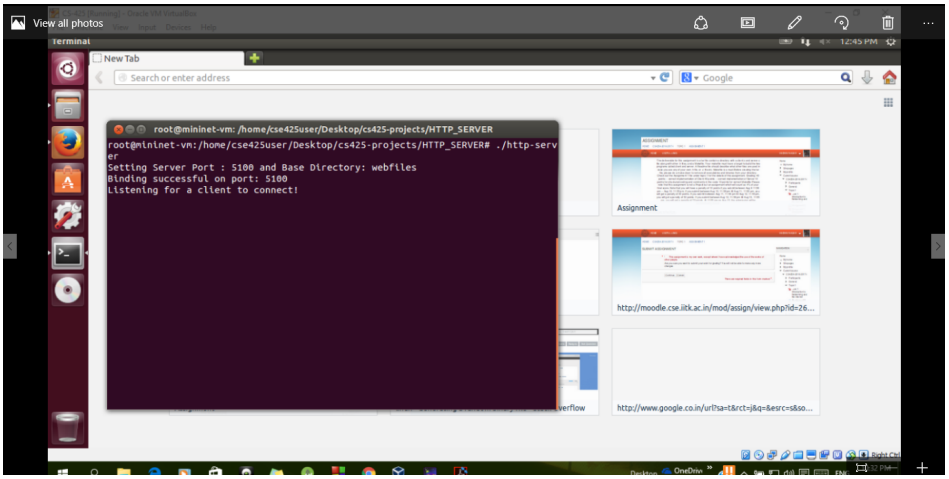

Figure 3: Server listening on default port: 5100

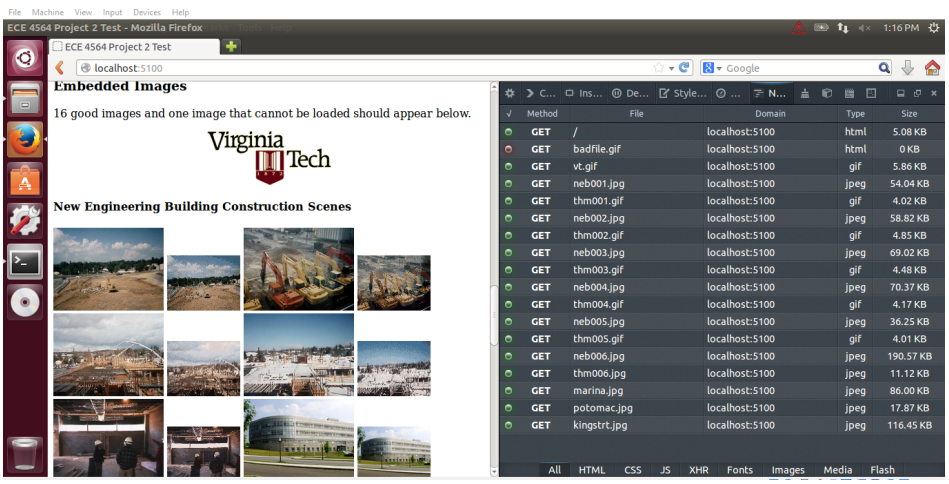

Figure 4: Showing all GET requests made by client(Firefox)

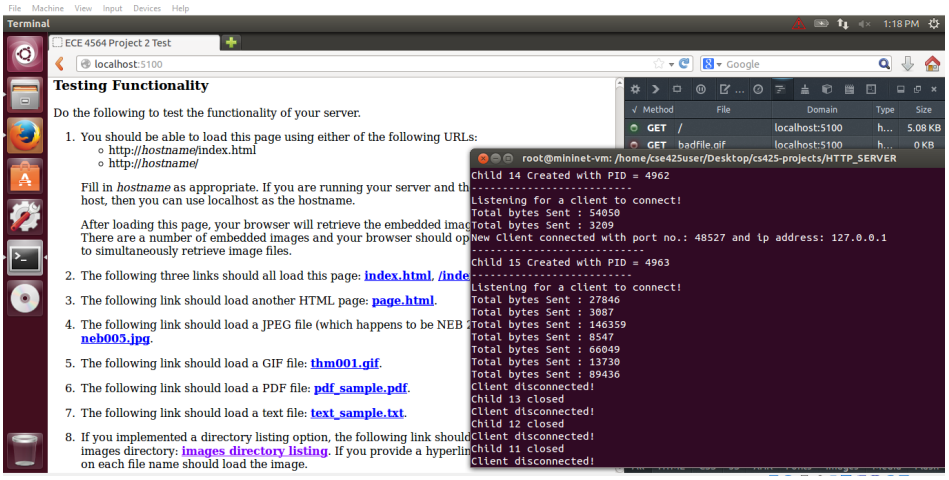

Figure 5: Child created for client and Bytes sent by server for each request

Summary

Server successfully handles all GET requests made by client and create new child processes for each client. Lists all hyperlinked files in directory as requested. 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.

The only problem occurs is in achieving persistent connections. Server is capable of maintaining the persistency, but only by using 5-6 child processes for a client when several GET requests are made at the same time by the same client. But if GET requests are fewer then it maintains persistent connection with the client using only one child process.

Appendix

Source Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/sendfile.h>
#include <time.h>
#include <sys/wait.h>
#include <dirent.h>
#include <errno.h>

#define MAX_BYTES 4096
#define MAX_CLIENTS 1000

int port = 5100;                    // Default Port
int socketId;                       // Server Socket ID
char *base_directory;                   // Base directory of server

pid_t client_PID[MAX_CLIENTS];              // PID of connected clients


int sendErrorMessage(int socket, int status_code)
{
char str[1024];
char currentTime[50];
time_t now = time(0);

struct tm data = *gmtime(&now);
strftime(currentTime,sizeof(currentTime),"%a, %d %b %Y %H:%M:%S %Z", &data);

switch(status_code)
{
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);
printf("400 Bad Request\n");
send(socket, str, strlen(str), 0);
break;

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  char* getContentType(char *path)
78  {
79  char *dot = strrchr(path, '.');          // return the address of last '.' found
     in string
80  char * extension;
81
82  if(!dot || dot == path)
83  extension = "";
```

```
84  else
85  extension = dot + 1;
86
87  if(strncmp(extension, "html", 4) == 0 || strncmp(extension, "htm", 3) == 0)
88  return "text/html";
89  else if(strncmp(extension, "txt", 3) == 0)
90  return "text/plain";
91  else if(strncmp(extension, "jpeg", 4) == 0 || strncmp(extension, "jpg", 3) == 0)
92  return "image/jpeg";
93  else if(strncmp(extension, "gif", 3) == 0)
94  return "image/gif";
95  else if(strncmp(extension, "pdf", 3) == 0)
96  return "Application/pdf";
97  else
98  return "application/octet-stream";
99
100 }
101
102
103 int sendHeaderMessage(int socket, char *head, char *media, int file_size)
104 {
105 char keep_alive[]      = "\r\nConnection: keep-alive";
106 char content_type[]    = "\r\nContent-Type: ";
107 char content_length[] = "\r\nContent-Length: ";
108 char date[]        = "\r\nDate: ";
109 char server_name[]     = "\r\nServer: VaibhavN/14785";
110 char new_line[]      = "\r\n\r\n";
111
112 char cLength[20];
113 snprintf(cLength,sizeof(cLength), "%d",file_size);     // Content Length: convert
     int to string
114
115 char currentTime[50];
116 time_t now = time(0);
117
118 struct tm data = *gmtime(&now);
119 strftime(currentTime,sizeof(currentTime),"%a, %d %b %Y %H:%M:%S %Z", &data);  //
     Get current time
120
121 char *header = (char*)calloc(strlen(head) + strlen(keep_alive) + strlen(
     content_type) + strlen(media) + strlen(content_length) + strlen(cLength) +
     strlen(date) + strlen(currentTime) + strlen(server_name) + strlen(new_line) +
     20, sizeof(char));
122
123
124 strcpy(header, head);
125 strcat(header, content_type);
126 strcat(header, media);
127 strcat(header, content_length);
128 strcat(header, cLength);
129 strcat(header, keep_alive);
130 strcat(header, date);
131 strcat(header, currentTime);
132 strcat(header, server_name);
```

```
133  strcat(header, new_line);
134
135  int bytes_send = send(socket, header, strlen(header), 0);
136
137  free(header);
138
139  return bytes_send;
140  }
141
142
143  int sendFile(int socket, int fd, char *path)
144  {
145
146  struct stat st;
147  fstat(fd, &st);
148  int file_size = st.st_size;                    // Get file size
149
150  char *mediaType = getContentType(path);           // Get media type of content
151
152  int bytes_send = sendHeaderMessage(socket, "HTTP/1.1 200 OK", mediaType, file_size
     );
153
154
155  if(bytes_send > 0)                         // Header Message sent successfully
156  {
157  bytes_send = sendfile(socket, fd, NULL, file_size);   // send file data
158
159  while(bytes_send < file_size)              // If sent data less tham file size
160  {
161  bytes_send = sendfile(socket, fd, NULL, file_size); // Send again
162
163  printf("\n\nSending File Again\n\n");
164  if(bytes_send <= 0)                    // Connection break;
165  {
166  bytes_send = sendErrorMessage(socket, 500);   // Unexpected server error
167  return bytes_send;
168  }
169  }
170  }
171  else
172  {
173  bytes_send = sendErrorMessage(socket, 500);       // Unexpected server error
174  return bytes_send;
175  }
176
177  printf("Total bytes Sent : %d\n", bytes_send);
178
179  return bytes_send;
180  }
181
182
183  int sendDirectory(int socket, char *path, char *dir_path)
184  {
185  DIR *dir;
```

```
186  struct dirent *entry;
187
188  char buffer[MAX_BYTES];
189
190  dir = opendir(path);                          // Open directory
191
192  int bytes_send;
193
194  int contentLength = 0;
195
196  if(strncmp(&dir_path[strlen(dir_path) -1], "/", 1) == 0)     // Removes Last
      forward slash
197  strcpy(&dir_path[strlen(dir_path) -1], "\0");
198
199  if(dir != NULL)
200  {
201  //------------------------------Calulate length of message to be send
      ------------------------------
202  while((entry = readdir(dir)) != NULL)
203  {
204  if(strcmp(entry->d_name, ".") == 0) continue;
205  contentLength += strlen(dir_path) + 2*strlen(entry->d_name) + 25; // Calculated
206  }
207  contentLength += 110 + strlen(dir_path);
208  closedir(dir);
209  //
      ------------------------------------------------------------------------------------------------------------------------------


210
211  dir = opendir(path);
212  bytes_send = sendHeaderMessage(socket, "HTTP/1.1 200 OK", "text/html",
      contentLength);
213
214  if(bytes_send > 0)                            // Header message sent successfully
215  {
216
217  snprintf(buffer,sizeof(buffer),"<HTML><HEAD><TITLE>Directory Links</TITLE></HEAD><
      BODY><H1>Files in the directory %s</H1><ul>", dir_path);
218
219  bytes_send = send(socket, buffer, strlen(buffer), 0);
220
221  if(bytes_send > 0)
222  {
223  while((entry = readdir(dir)) != NULL)
224  {
225  if(strcmp(entry->d_name, ".") == 0) continue;
226
227  bzero(buffer,MAX_BYTES);
228
229  snprintf(buffer,sizeof(buffer), "<li><a href=\"%s/%s\">%s</a></li>",dir_path,
      entry->d_name, entry->d_name);
230
231  bytes_send = send(socket, buffer, strlen(buffer), 0);    // Send files one by one
232
```

```
233   if(bytes_send <= 0)                        // Connection is broken
234   break;
235   }
236   }
237   else
238   {
239   bytes_send = sendErrorMessage(socket, 500);           // Unexpected Error
240   return bytes_send;
241   }
242
243   bzero(buffer,MAX_BYTES);
244
245   snprintf(buffer,sizeof(buffer), "</ul></BODY></HTML>");
246   bytes_send = send(socket, buffer, strlen(buffer), 0);
247
248   closedir(dir);                              // Close dir
249
250   return bytes_send;
251   }
252   else
253   {
254   closedir(dir);
255   bytes_send = sendErrorMessage(socket, 500);             // Unexpected server error
256   return bytes_send;
257   }
258   }
259   else
260   {
261   if( errno == EACCES)                          //  Check errno value
262   {
263   perror("Permission Denied\n");
264   bytes_send = sendErrorMessage(socket, 403);
265   return bytes_send;
266   }
267   else
268   {
269   perror("Directory Not Found\n");
270   bytes_send = sendErrorMessage(socket, 404);           // Directory Not Found
271   return bytes_send;
272   }
273   }
274
275   }
276
277
278   int checkHTTPversion(char *msg)
279   {
280   int version = -1;
281
282   if(strncmp(msg, "HTTP/1.1", 8) == 0)
283   {
284   version = 1;
285   }
```

```
286  else if(strncmp(msg, "HTTP/1.0", 8) == 0)       // Server can also handle 1.0
      requests in the same way as it does to handle 1.1 requests
287  {
288  version = 1;                    // Hence setting same version as 1.1
289  }
290  else
291  version = -1;
292
293  return version;
294  }
295
296
297  int requestType(char *msg)
298  {
299  int type = -1;
300
301  if(strncmp(msg, "GET\0",4) == 0)
302  type = 1;
303  else if(strncmp(msg, "POST\0",5) == 0)
304  type = 2;
305  else if(strncmp(msg, "HEAD\0",5) == 0)
306  type = 3;
307  else
308  type = -1;
309
310  return type;
311  }
312
313
314  int handleGETrequest(int socket, char *msg)
315  {
316  char file_path[500];
317  char dir_path[500];
318  bzero(dir_path,sizeof(dir_path));
319  bzero(file_path,sizeof(file_path));
320
321  int fd;                        // File descriptor
322
323  int bytes_send;
324
325  if(strlen(msg) == 0 || strncmp(msg, "/", 1) !=0)  // Error
326  {
327  printf("message Error!");
328  sendErrorMessage(socket, 400);          // 400 Bad Request
329  return 1;
330  }
331
332  if(strlen(msg) == 1)                // Default file open index.html
333  {
334  strcpy(file_path, base_directory);
335  strcat(file_path, "/index.html");
336  }
337  else
338  {
```

```
339 strcpy(file_path, base_directory);        // concatenate requested file name in
    base_directory
340 strcat(file_path, msg);
341 strcpy(dir_path, msg);
342 }
343
344 struct stat s;
345 if( (stat(file_path, &s) == 0 && S_ISDIR(s.st_mode)) )    // Given File Path is a
    directory
346 {
347 printf("Send directory links\n");
348 bytes_send = sendDirectory(socket, file_path, dir_path);    // Send directory
    links
349
350 return bytes_send;
351 }
352
353 fd = open(file_path, O_RDONLY);             // Otherwise open requested file
354
355 if(fd == -1)
356 {
357 if( errno == EACCES)
358 {
359 perror("Permission Denied\n");
360 sendErrorMessage(socket, 403);          // Permission Denied
361 return 1;
362 }
363 else
364 {
365 perror("File does not exist\n");
366 sendErrorMessage(socket, 404);          // File not found
367 return 1;
368 }
369 }
370
371 bytes_send = sendFile(socket, fd, file_path);     // Send file content
372
373 close(fd);                         // Close file
374
375 return bytes_send;
376
377 }
378
379
380
381 void respondClient(int socket)
382 {
383
384 int bytes_send;                          // Bytes Transferred
385
386 char buffer[MAX_BYTES];                      // Creating buffer of 4kb for a client
387
388 bzero(buffer, MAX_BYTES);                    // Make buffer zero
389
```

```
390   bytes_send = recv(socket, buffer, MAX_BYTES, 0);        // Receive File Name
391
392   while(bytes_send > 0)
393   {
394   //printf("%s\n",buffer);
395   char *message[3];
396
397   if(strlen(buffer) > 0)
398   {
399   message[0] = strtok(buffer, " \t\n");          // stores Request Method
400
401   int type = requestType(message[0]);
402   if(type == 1)                        // GET Request
403   {
404
405   message[1] = strtok(NULL, " \t\n");          // stores request file path
406   message[2] = strtok(NULL, " \t\n");          // stores HTTP version
407
408   if(strlen(message[2]) && checkHTTPversion(message[2]) == 1)
409   bytes_send = handleGETrequest(socket, message[1]);     // Handle GET request
410
411   else
412   sendErrorMessage(socket, 505);           // Incorrect HTTP version
413
414   }
415   else if(type == 2)                      // POST Request
416   {
417   printf("POST: Not implemented");
418   sendErrorMessage(socket, 501);
419   }
420   else if(type == 3)                      // HEAD Request
421   {
422   printf("HEAD: Not implemented");
423   sendErrorMessage(socket, 501);
424   }
425   else                            // Unknown Method Request
426   {
427   printf("Unknown Method: Not implemented");
428   sendErrorMessage(socket, 501);
429   }
430   }
431   else
432   {
433   printf("ERROR\n");
434   sendErrorMessage(socket, 400);               // 400 Bad Request
435   }
436
437   bzero(buffer, MAX_BYTES);
438   bytes_send = recv(socket, buffer, sizeof(buffer), 0);   // Recieve Next Request
       from Cliemt
439
440   }
441
442   if( bytes_send < 0)
```

```
443  {
444  perror("Error in receiving from client.\n");
445  }
446  else if(bytes_send == 0)
447  {
448  printf("Client disconnected!\n");
449  }
450
451  close(socket);                          // Close socket
452
453  return;
454  }
455
456
457  int findAvailableChild(int i)
458  {
459  int j = i;
460  pid_t ret_pid;
461  int child_state;
462
463  do
464  {
465  if(client_PID[j] == 0)
466  return j;
467  else
468  {
469  ret_pid = waitpid(client_PID[j], &child_state, WNOHANG);    // Finds status change
       of pid
470
471  if(ret_pid == client_PID[j])                  // Child exited
472  {
473  client_PID[j] = 0;
474  return j;
475  }
476  else if(ret_pid == 0)                       // Child is still running
477  {
478  ;
479  }
480  else
481  perror("Error in waitpid call\n");
482  }
483  j = (j+1)%MAX_CLIENTS;
484  }
485  while(j != i);
486
487  return -1;
488  }
489
490
491  int main(int argc, char *argv[])
492  {
493  int newSocket, client_len;
494
495  struct sockaddr_in server_addr, client_addr;
```

```
496
497 base_directory = (char*)malloc(45*sizeof(char));
498 char *temp_directory;
499
500 strcpy(base_directory, "webfiles"); // Need to be changed accordingly
501
502 bzero(client_PID, MAX_CLIENTS);
503
504 // Fetching Arguments
    ---------------------------------------------------------------------------
505 int params = 1;
506
507 for(; params < argc; params++)
508 {
509 if(strcmp(argv[params], "-p") == 0)
510 {
511 params++;
512
513 if(params < argc)
514 {
515 port = atoi(argv[params]);
516 continue;
517 }
518 else
519 {
520 printf("Wrong Arguments! Usage: %s [-p PortNumber] [-b BaseDirectory]\n", argv[0])
    ;
521 exit(1);
522 }
523 }
524 else if(strcmp(argv[params], "-b") == 0)
525 {
526 params++;
527
528 if(params < argc)
529 {
530 struct stat s;
531 if( !(stat(argv[params], &s) == 0 && S_ISDIR(s.st_mode)))
532 {
533 printf("Error: No such directory exist!\n");
534 exit(1);
535 }
536
537 temp_directory = argv[params];
538
539 int k = strlen(temp_directory) - 1;
540
541 if(strncmp(&temp_directory[k], "/", 1) == 0)          // Removing / from the last
542 strcpy(&temp_directory[k], "\0");
543
544 char *temp = (char*)realloc(base_directory, sizeof(char)*strlen(temp_directory));
545 base_directory = temp;
546 strcpy(base_directory, temp_directory);
```

```c
continue;
}
else
{
printf("Wrong Arguments! Usage: %s [-p PortNumber] [-b BaseDirectory]\n", argv[0]);
exit(1);
}
}
else
{
printf("Wrong Arguments! Usage: %s [-p PortNumber] [-b BaseDirectory]\n", argv[0]);
exit(1);
}
}

printf("Setting Server Port : %d and Base Directory: %s\n", port, base_directory);


// Creating socket ------------------------------------------------------------------------


socketId = socket(AF_INET, SOCK_STREAM, 0);

if( socketId < 0)
{
perror("Error in Creating Socket.\n");
exit(1);
}

int reuse =1;
if (setsockopt(socketId, SOL_SOCKET, SO_REUSEADDR, (const char*)&reuse, sizeof(reuse)) < 0)
perror("setsockopt(SO_REUSEPORT) failed");

//----------------------------------------------------------------------------------------------


// Binding socket with given port number and server is set to connect with any ip address--------------

bzero((char*)&server_addr, sizeof(server_addr));
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(port);
server_addr.sin_addr.s_addr = INADDR_ANY;

if( bind(socketId, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0 )
{
perror("Binding Error : Port may not be free. Try Using diffrent port number.\n");
exit(1);
```

```
593   }
594
595   printf("Binding successful on port: %d\n",port);
596
597   //------------------------------------------------------------------------------------------------------------------------
598
599   // Listening for connections and accept upto MAX_CLIENTS in queue
      ------------------------------------
600
601   int status = listen(socketId, MAX_CLIENTS);
602
603   if(status < 0 )
604   {
605   perror("Error in Listening !\n");
606   exit(1);
607   }
608
609   //------------------------------------------------------------------------------------------------------------------------
610
611   // Infinite Loop for accepting connections
      ------------------------------------------------------------
612
613   int i=0;
614   int ret;
615
616   while(1)
617   {
618   printf("Listening for a client to connect!\n");
619   bzero((char*)&client_addr, sizeof(client_addr));              // Clears struct
       client_addr
620   client_len = sizeof(client_addr);
621
622   newSocket = accept(socketId, (struct sockaddr*)&client_addr, &client_len);  //
       Accepts connection
623   if(newSocket < 0)
624   {
625   fprintf(stderr, "Error in Accepting connection !\n");
626   exit(1);
627   }
628
629
630   // Getting IP address and port number of client
631
632   struct sockaddr_in* client_pt = (struct sockaddr_in*)&client_addr;
633   struct in_addr ip_addr = client_pt->sin_addr;
634   char str[INET_ADDRSTRLEN];                          // INET_ADDRSTRLEN: Deafult ip
       address size
635   inet_ntop( AF_INET, &ip_addr, str, INET_ADDRSTRLEN );
636   printf("New Client connected with port no.: %d and ip address: %s \n",ntohs(
       client_addr.sin_port), str);
```

```
637
638
639  //
     ------------------------------------------------------------------------------------------
640  // Forks new client
641
642  i = findAvailableChild(i);
643
644  if(i>= 0 && i < MAX_CLIENTS)
645  {
646  ret = fork();
647
648  if(ret == 0)                    // Create child process
649  {
650  respondClient(newSocket);
651  printf("Child %d closed\n", i);
652  exit(0);                      // Child exits
653  }
654  else
655  {
656  printf("------------------------------\nChild %d Created with PID = %d\n
     ------------------------------\n", i,ret);
657  client_PID[i] = ret;
658
659  }
660  }
661  else
662  {
663  i = 0;
664  close(newSocket);
665  printf("No more Client can connect!\n");
666  }
667
668
669  // And goes back to listen again for another client
670  }
671
672  close(socketId);                    // Close socket
673  return 0;
674  }
675
```

Listing 1: Concurrent HTTP Server

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