***Project:* Socket Programming – HTTP Server and Proxy Serve**

In this project, you are required to implement a multi-threaded HTTP server that returns a document determined by the requested URI. You are also requested to implement a proxy server with some specific properties.  
You are also required to use ApacheBench program for testing your servers.

***Objectives of http server (all done)***

• Your server should be capable of providing concurrency via multi-threading.

• Your server program should take single argument which specifies the port number.

• Your server should return an HTML document according to the requested URI.

• Your server must send back an HTTP response line

• Your server should send back an HTTP response line that indicates errors

• Your server should print out information about every message received and every message sent.

• Your server should work when connected via a web browser

***Objectives of proxy server (all done)***

• Your proxy server will not be able to cache HTTP objects. **(extra 15 points) – done (video added)**

• Proxy Server should use port 8888. **(you can give any port number as an argument)**

• Your proxy server only directs the requests to your web server. **(extra 3 points) - done**

• In this project, proxy server has a restriction of URL length 9999.

• If the Web Server is not running currently, your proxy server would return a “Not Found” error message with status code 404.

• The proxy server should work when connected via a browser after configuring the proxy settings of your browser.

***Implementing Multi-Threaded HTTP Server***

***Socket implementation***

*• Implementing socket connection is pretty straight forward, you can see the commented code below:*

• /\* creating socket \*/

•

• sock\_desc = socket(AF\_INET, SOCK\_STREAM, 0);

• /\* server is socket struct, filling its attiributes \*/

•

• server.sin\_family = AF\_INET;

• server.sin\_addr.s\_addr = INADDR\_ANY;

• server.sin\_port = htons(PORT\_NO);

• /\* bind socket with its id \*/

•

• if (bind(sock\_desc, (struct sockaddr \*)&server, sizeof(server)) < 0)

• {

• printf("Binding failed\n");

• return 1;

• }

• /\* listens requests \*/

•

• listen(sock\_desc, 20);

•

***Multithread implementation***

*• For each connection request from client, new thread with given arguments runs tcp\_handler function which handles request & response cycle between client and the server.*

while (

(new\_socket = accept(sock\_desc, (struct sockaddr\*)&client, (socklen\_t \*)&c))

)

// Accept the connection.

{

printf("Connection accepted \n");

new\_sock = malloc(1);

\*new\_sock = new\_socket;

/\* thread create with argument of socket \*/

if (pthread\_create(&tid, NULL, (void \*)tcp\_handler, new\_sock))

{

printf("Thread couldn't created \n");

}

/\* no thread will finish code before job is done \*/

pthread\_join(tid, NULL);

}

***tcp\_handler function***

void \*tcp\_handler(void \*sock\_desc, void \*arg)

***socket id***

int sock = \*((int \*)sock\_desc);

***get request***

request = recv(sock, client\_reply, BUFFER\_SIZE, 0);

***this function gets the number from URL***

***ex: GET /1500 /…. .. .. … 1500***

int size = GET\_SIZE(client\_reply);

***if size < 1 (This means there is no integer inside http request)***

if(size < 1)

{

char \*message = "HTTP/1.0 400 Bad Request\r\nContent-Type: text/html\r\n\r\n<!doctype html><html><body> URL must be a number </body></html>";

write(sock, message, strlen(message));

return NULL;

}

***Put control ( if request is PUT then “Not implemented” error will be sent)***

if (PUT\_CONTROL(client\_reply) > 0)

***Post control ( if request is PUT then “Not implemented” error will be sent)***

if (POST\_CONTROL(client\_reply) > 0)

***100 – 20000 control ( if requested URL is not sized between this to, then error will be sent)***

if (size < 100 || 20000 < size)

***Constructing the http Response msg if every error handling segment passed***

***Send response to the client***

write(sock, temp, strlen(temp));

***free the allocated memories and sockets***

free(sock\_desc);

shutdown(sock, SHUT\_RDWR);

close(sock);

sock = -1;

return 0;

}

***Proxy server implementation***

• Proxy server implementation was more complex. Since in infinite loop there must be 3 connection at the same time. Also caching implemented this makes it more complex

**• Cache implementation**

**This node holds response msg, and a flag to indicate its full or not.**

**struct cache\_node**

{

char response\_msg[MAX\_SIZE];

int full;

};

**Create 20000 sized node array for 20000 different requests 1 to 20000**

**And initialize their flag to -1**

struct cache\_node \*st = malloc(20000 \* sizeof(struct cache\_node));

for (int i = 0; i < 20000; i++)

{

st[i].full = -1;

}

**Given as an argument to thread**

struct cache\_node \* st = info->st;

**Size is being index for our cache node array, if request is GET/ … /100 then we store response in 100. index of the array**

int size = GET\_SIZE(buffer);

**When request is come, if the nodes full flag in the index size show is < 0 then the request will go to the server, and response will be stored in the index shown by size.**

if (st[size].full < 0)

{

strcpy(st[size].response\_msg, buffer);

st[size].full = 1;

printf("REQUEST & RESPONSE CACHED \n");

printf("\nSERVER RESPONSE IS: \n\n %s", st[size].response\_msg);

}

**When request is come, this means response already stored in node array and no need to request it from server.**

if (st[size].full > 0)

{

write(info->client\_fd, st[size].response\_msg, sizeof(st[size].response\_msg));

printf("CACHED RESPONSE USED \n\n SERVER RESPONSE IS: %s \n\n", st[size].response\_msg);

return NULL;

}

**• Directing to any webserver implementation**

**For this we used C’s libraries, this functin converts a domain address to an ip address**

**ex: localhost to 127.0.0.1**

int hostname\_to\_ip(char \*hostname, char \*ip)

{

struct hostent \*he;

struct in\_addr \*\*addr\_list;

int i;

if ((he = gethostbyname(hostname)) == NULL)

{

// get the host info

printf("Couldn't make it");

return -1;

} addr\_list = (struct in\_addr \*\*)he->h\_addr\_list;

for (i = 0; addr\_list[i] != NULL; i++)

{

//Return the first one;

strcpy(ip, inet\_ntoa(\*addr\_list[i]));

return 0;

}

return 1;

}

**• Proxy implementation**

**This is argument node for our thread, which has cache arrays pointer, clients socket id, ip address, hostname (for print issues), and servers port**

struct thread\_arg

{

struct cache\_node \*st;

int client\_fd;

char ip[100];

char port[100];

char hostname[100];

};

**Inside main, we connecting proxy server with client**

proxy\_fd = socket(AF\_INET, SOCK\_STREAM, 0)

proxy\_sd.sin\_family = AF\_INET;

proxy\_sd.sin\_port = htons(atoi(proxy\_port));

proxy\_sd.sin\_addr.s\_addr = INADDR\_ANY;

bind(proxy\_fd, (struct sockaddr \*)&proxy\_sd, sizeof(proxy\_sd)

listen(proxy\_fd, SOMAXCONN)

**when a connection request is come**

client\_fd = accept(proxy\_fd, (struct sockaddr \*)&client, (socklen\_t \*)&c);

**we fill our thread argumans and invoke thread with function proxy\_handler**

if (client\_fd > 0)

{

struct thread\_arg \*item = malloc(sizeof(struct thread\_arg));

item->client\_fd = client\_fd;

item->st = st;

strcpy(item->hostname, hostname);

strcpy(item->ip, ip);

strcpy(item->port, server\_port);

pthread\_create(&tid, NULL, proxy\_handler, (void \*)item);

}

**proxy\_handler function**

void \*proxy\_handler(void \*arg)

**construct server socket connection between proxy (client – proxy already done)**

/\* server socket check \*/

if ((server\_fd = socket(AF\_INET, SOCK\_STREAM, 0)) < 0)

{

printf("Server socket could not created. \n");

return 0;

}

/\* set server socket variables \*/

memset(&server\_sd, 0, sizeof(server\_sd));

server\_sd.sin\_family = AF\_INET;

server\_sd.sin\_port = htons(atoi(info->port));

server\_sd.sin\_addr.s\_addr = inet\_addr(info->ip);

/\* connection check \*/

if ((connect(server\_fd, (struct sockaddr \*)&server\_sd, sizeof(server\_sd))) < 0)

{

printf("server connection not established");

return 0;

}

**URL construct**

**URL length check**

/\* URL length check \*/

if (strlen(URL) > 9999)

{

char \*message = "HTTP/1.0 400 Bad Request\r\nContent-Type: text/html\r\n\r\n<!doctype html><html><body> URL IS TOO LONG </body></html>";

write(info->client\_fd, message, sizeof(message));

return NULL;

}

**Server down check**

if (check\_read <= 0)

{

char \*message = "HTTP/1.0 400 Bad Request\r\nContent-Type: text/html\r\n\r\n<!doctype html><html><body> Server is down. </body></html>";

write(info->client\_fd, message, sizeof(message));

return NULL;

}

**Proxy send – request cycle**

**Get request from client**

check\_read = read(info->client\_fd, buffer, sizeof(buffer));

**Send it to the server**

write(server\_fd, buffer, sizeof(buffer));

**Receive response from server**

check\_read = read(server\_fd, buffer, sizeof(buffer));

**Send response to client**

write(info->client\_fd, buffer, sizeof(buffer));

**Multithread Server Run Time Screenshots**

**localhost:5000/10**

**localhost:5000/100**

**localhost:5000/20001**

**localhost:5000/a**

**Proxy Server Run Time Screenshots**

**Server 3000**

**Proxy 3000 8887**

**localhost:8888/50**

**localhost:8888/a**

**localhost:8888/150**

**Apache – Bench reports**

**ab -n 1000 -c 1 http://127.0.0.1:500/**

Since its working on localhost, its too fast to measure.

**ab -n 1000 -c 5 http://127.0.0.1:500/**

**ab -n 1000 -c 10 http://127.0.0.1:500/**

since concurrency level increase, delay increases.

**ab -n 1000 -c 1 -k http://127.0.0.1:500/**

**ab -n 1000 -c 5 -k http://127.0.0.1:500/**

**-k increases delay since every request tries too keep connection alive.**

**ab -n 1000 -c 10 -k http://127.0.0.1:500/**