**ECEN 602**

**NETWORK SIMULATION ASSIGNMENT – 04**

**TEAM 17**

**Mohammad Faisal Khan**

**Amiya Ranjan Panda**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**README**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**TITLE : Implementation of a simple HTTP proxy server and HTTP command line client based on RFC 1945 with additional caching feature.**

**INTRODUCTION :**

**This code is a part of the Network simulation Assignment for ECEN 602 at Texas A&M University.**

**It has been sucessfully compiled, executed and tested on gcc compiler (part of standard LINUX).**

**The Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems.**

**HTTP functions as a request–response protocol in the client–server computing model.**

**A web browser, for example, may be the client and an application running on a computer hosting a website may be the server. The client**

**submits an HTTP request message to the server. The server, which provides resources such as HTML files and other content, or performs**

**other functions on behalf of the client, returns a response message to the client. The response contains completion status information**

**about the request and may also contain requested content in its message body. HTTP resources are identified and located on the network**

**by Uniform Resource Locators (URLs), using the Uniform Resource Identifiers (URI's) schemes http and https. URIs and hyperlinks in HTML**

**documents form interlinked hypertext documents.**

**HTTP client initiates a request by establishing a Transmission Control Protocol (TCP) connection to a particular port on a server**

**(typically port 80, occasionally port 8080; see List of TCP and UDP port numbers). An HTTP server listening on that port waits for a**

**client's request message. Upon receiving the request, the server sends back a status line, such as "HTTP/1.1 200 OK", and a message**

**of its own. The body of this message is typically the requested resource, although an error message or other information may also be**

**returned. In case of the web proxy server, it behaves like a server in the client-proxy interface and as a client in the**

**proxy-server interface.**

**In this implementation, When the proxy server receives a client request, it will first check its cached data in an attempt to serve**

**the request. If there is not a valid cache entry, however, (1) the request will be proxied to the intended destination, (2) the response**

**will be sent by the proxy to the client, and (3) the response will also be cached by the proxy for later use. Your proxy cache should**

**maintain at least 10 document entries in the cache. Entries should be replaced in a Least Recently Used (LRU) fashion.**

**Common Errors and Catches:**

**-If data is not input correctly on the command line as per the ordering given below, it throws a segmentation error.**

**-This is NOT to be assumed as an error.**

**-If data is missing from the command line, it throws segmentation error too.**

**-This is an iterative server, fork() is not used instead select() is used.**

**-The well-known socket for the HTTP server is port number 80 but the server uses a different one(asked in argument) and the connection is established in ephemeral port**

**as negotiated by the web server. This is done to avoid the creation of invariance in the proxy server processing.**

**-The url format is "www.abcd.com/path".**

**Usage:**

**1. 'make clean' to remove all previously created object files.**

**2. 'make' to compile the Server source code.**

**3. ./proxy <ip to bind> <port to bind> to run the proxy server first.**

**4. ./client <proxy address> <proxy port> <url to retrieve> to run the client.**

**steps :**

**1. make**

**2. ./proxy <ip to bind> <port to bind>**

**3. ./client <proxy address> <proxy port> <url to retrieve>**

**Package content:**

**1. proxy.c**

**2. client.c**

**3. Makefile**

**Tests:**

**1. A cache hit returns the saved data to the requester.**

**2. A request that is not in the cache is proxied, saved in the cache, and returned to the requester.**

**3. A cache miss with 10 items already in the cache is proxied, saved in the LRU location in cache, and the data is returned to the requester.**

**4. A stale Expires header in the cache is accessed, the cache entry is replaced with a fresh copy, and the fresh data is delivered to the requester.**

**5. A stale entry in the cache without an Expires header is determined based on the last Web server access time and last modification time, the stale**

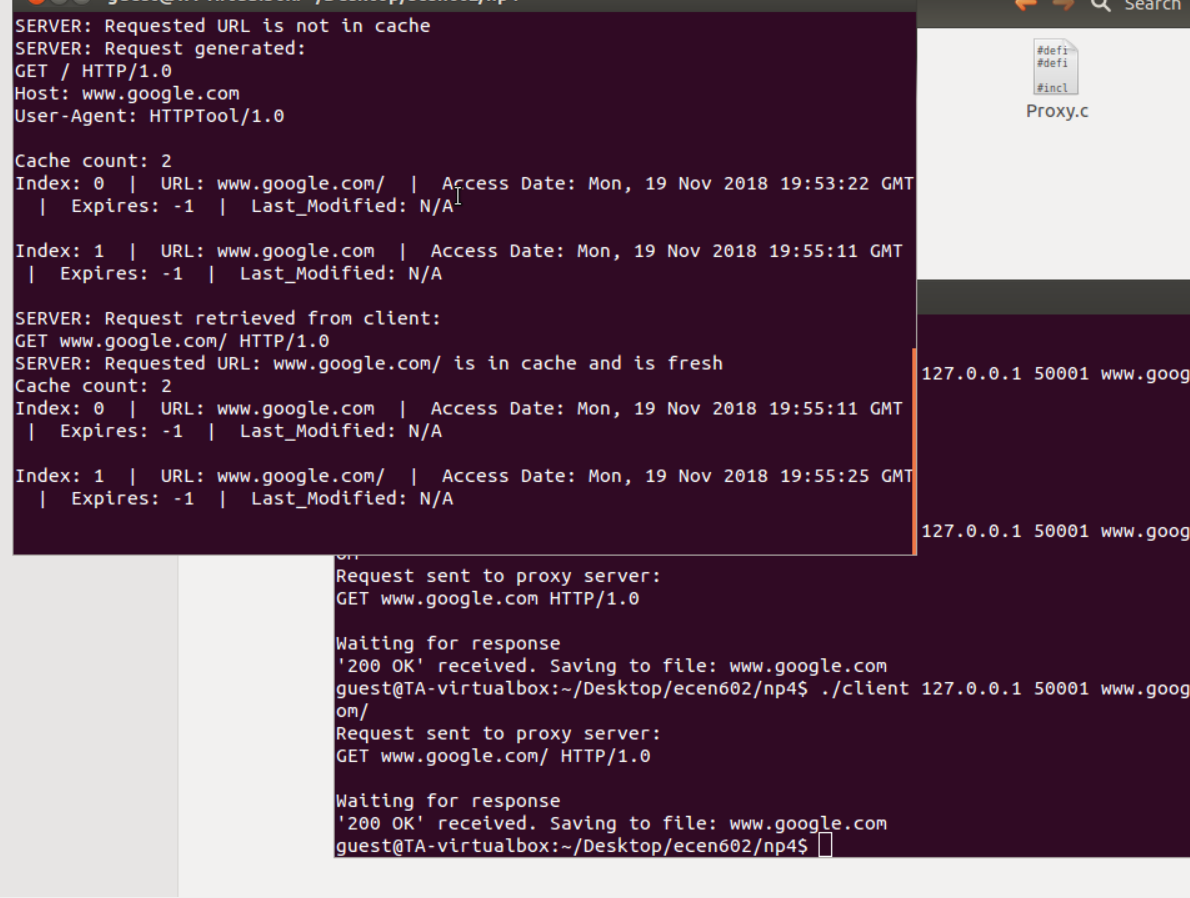
**cache entry is replaced with fresh data, and the fresh data is delivered to the requester.**

**6. A cache entry without an Expires header that has been previously accessed from the Web server in the last 24 hours and was last modified more than**

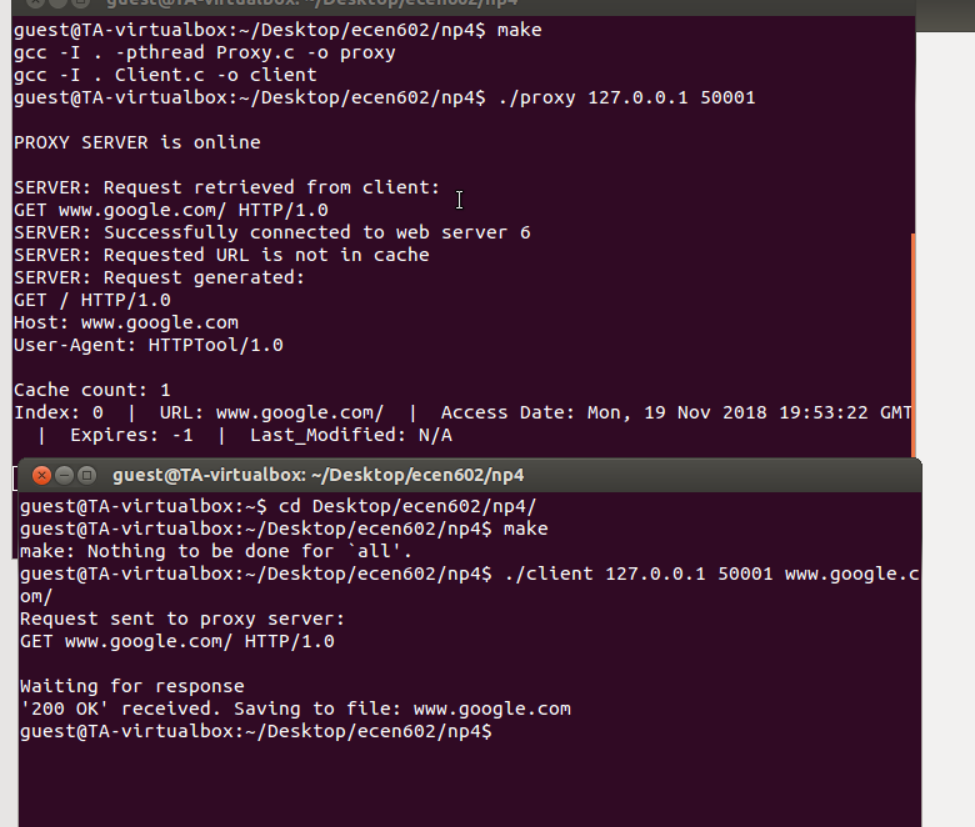
**one month ago is returned to the requester.**

**7. Three clients can simultaneously access the proxy server and get the correct data.**

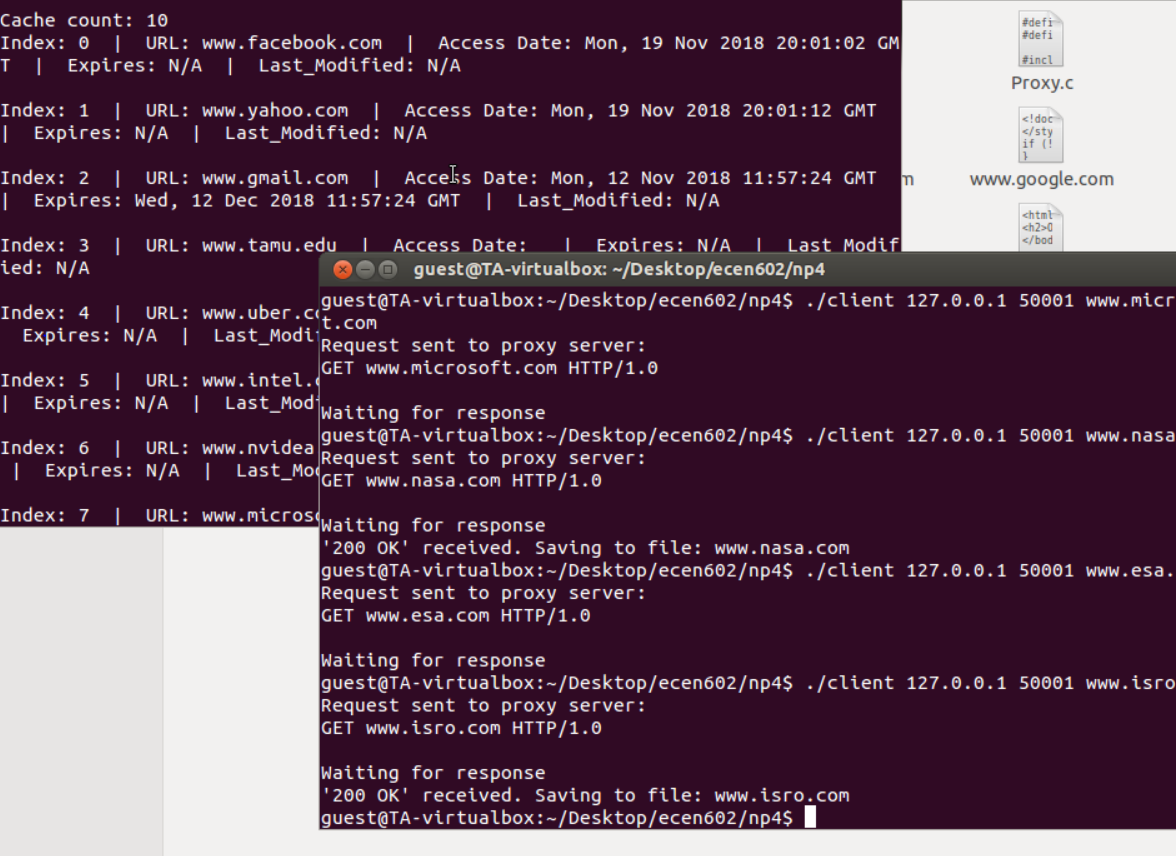
**A cache hit returns the saved data to the requester.**

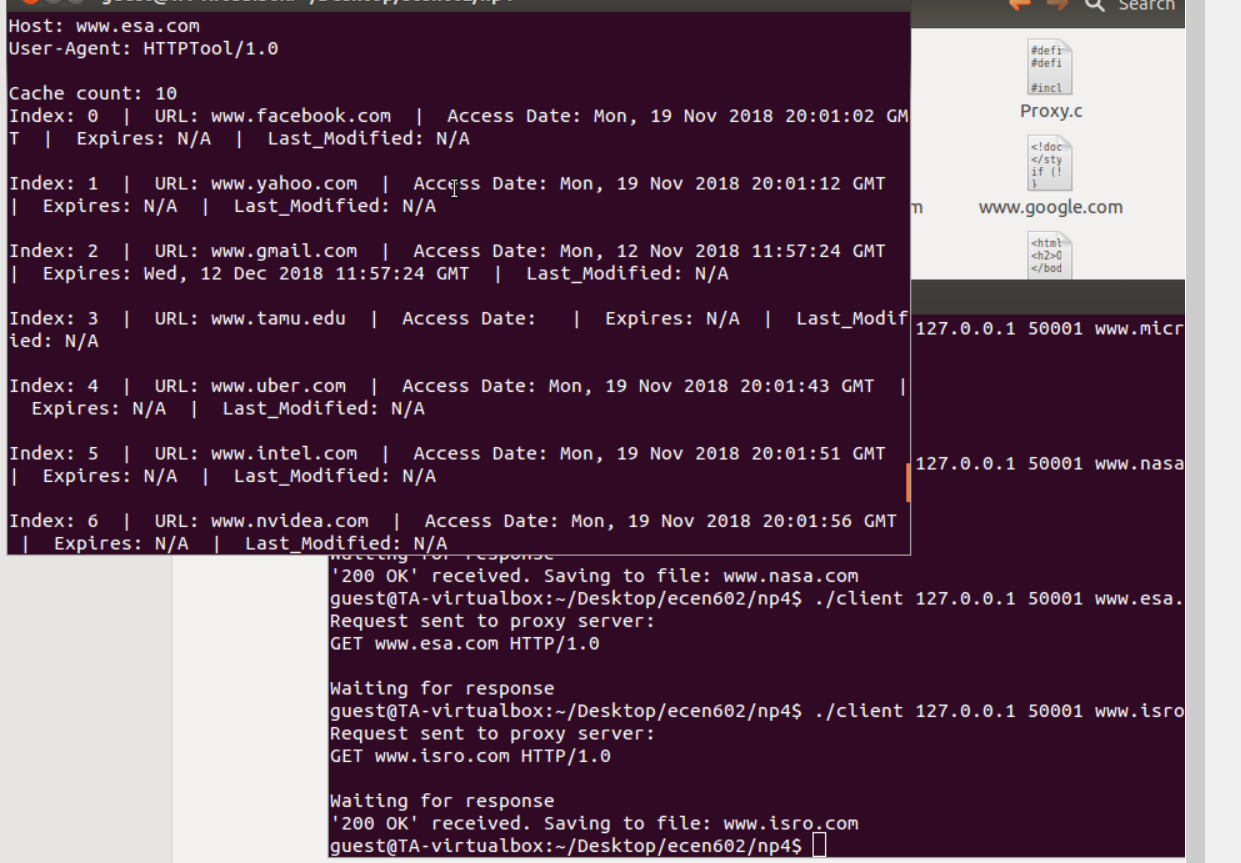
****

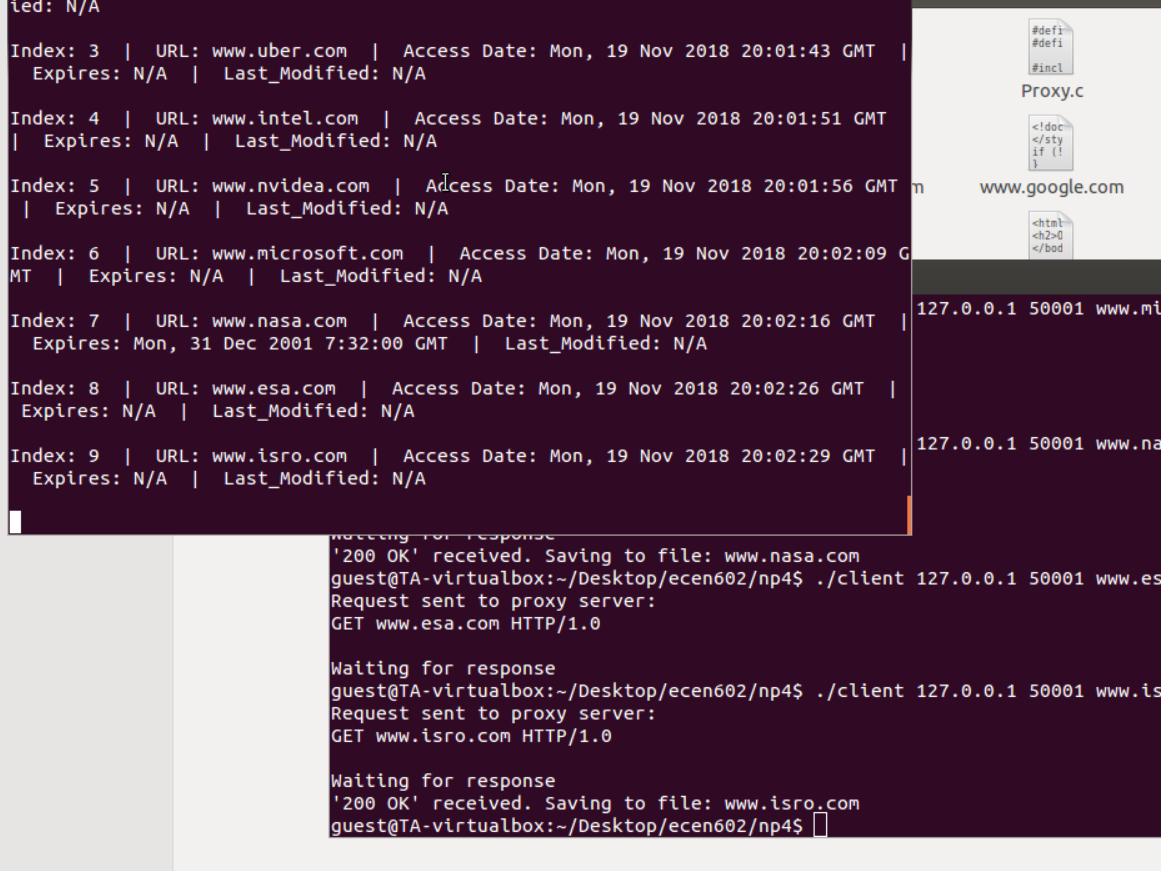
**A request that is not in the cache is proxied, saved in the cache, and returned to the requester.**

****

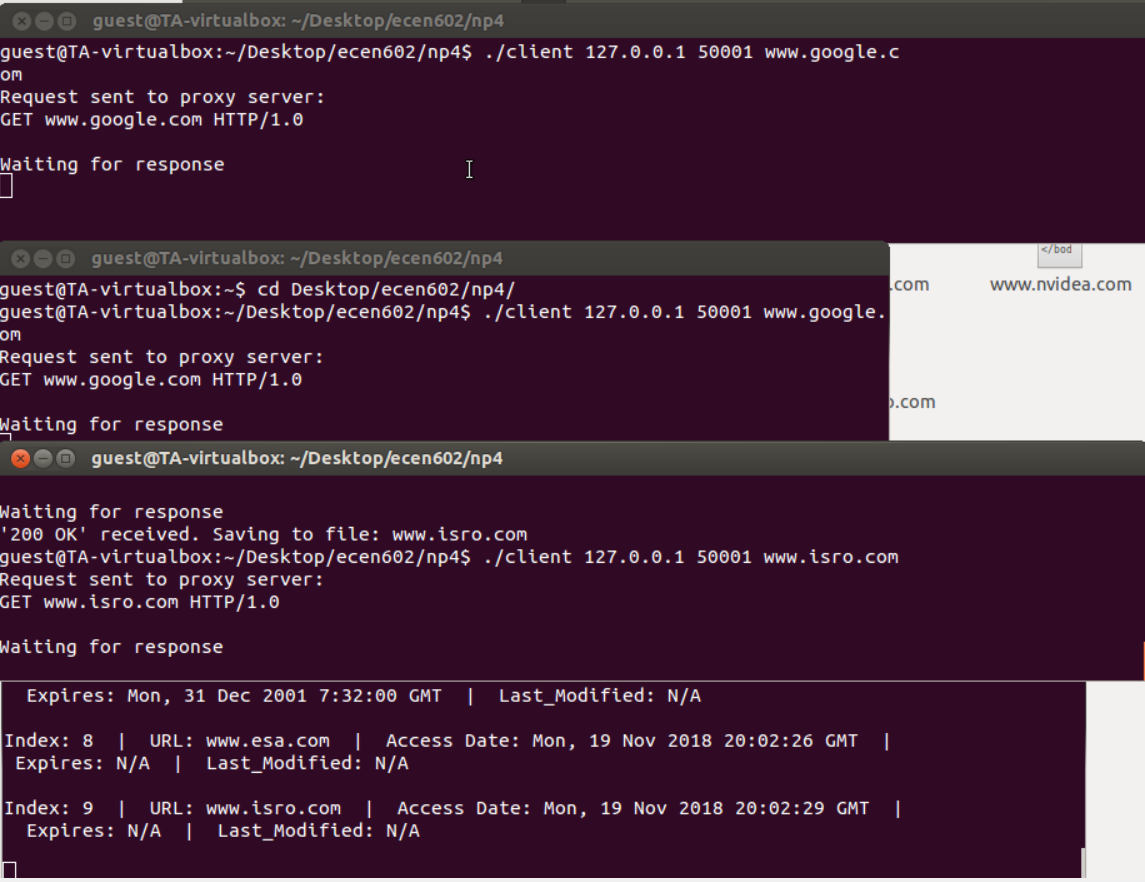
**A cache miss with 10 items already in the cache is proxied, saved in the LRU location in cache, and the data is returned to the requester.**

****

****

****

**Three clients can simultaneously access the proxy server and get the correct data**

****

**CODE :**

**PROXY :**

#define \_\_USE\_XOPEN 1

#define \_XOPEN\_SOURCE 700

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <errno.h>

#include <strings.h>

#include <string.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <sys/socket.h>

#include <netdb.h>

#include <arpa/inet.h>

#include <sys/wait.h>

#include <signal.h>

#include <dirent.h>

#include <pthread.h>

#include <time.h>

#define MAX\_CACHE\_ENTRY 10

#define MAX\_LEN 1024

struct Cache {

char URL[256];

char Last\_Modified[50];

char Access\_Date[50];

char Expires[50];

char \*body;

};

static const struct Cache Clear\_Entry;

int num\_cache\_entries = 0;

struct Cache Proxy\_Cache[MAX\_CACHE\_ENTRY];

int parse\_URL (char\* URL, char \*hostname, int \*port, char \*path) {

char \*token;

char \*host\_temp, \*path\_temp;

char \*tmp1, \*tmp2;

int num = 0;

char s[16];

if (strstr(URL,"http") != NULL){

token = strtok(URL, ":");

tmp1 = token + 7;

}

else{

tmp1 = URL;

}

tmp2 = malloc (64);

memcpy(tmp2, tmp1, 64);

if(strstr(tmp1, ":") != NULL){

host\_temp = strtok(tmp1, ":");

\*port = atoi(tmp1 + strlen(host\_temp) + 1);

sprintf(s, "%d", \*port);

path\_temp = tmp1 + strlen(host\_temp) + strlen(s) + 1;

}

else{

host\_temp = strtok(tmp1, "/");

\*port = 80;

path\_temp = tmp2 + strlen(host\_temp);

}

if (strcmp(path\_temp, "") == 0)

strcpy(path\_temp, "/");

memcpy(hostname, host\_temp, 64);

memcpy(path, path\_temp, 256);

return(0);

}

int parseHDR(const char\* hdr, char\* buf, char\* op) {

char \*st = strstr(buf, hdr);

if(!st) {

return 0;

}

char \*end = strstr(st, "\r\n");

st += strlen(hdr);

while(\*st == ' ')

++st;

while(\*(end - 1) == ' ')

--end;

strncpy(op, st, end - st);

op[end - st] = '\0';

return 1;

}

int err\_sys(const char\* x) // Error display source code

{

perror(x);

exit(1);

}

int Extract\_Read(int fd, char \*msg) { // Extracts message body from read socket

int total = 0;

char buffer[MAX\_LEN] = {0};

int cnt = 1;

int h;

while(cnt>0) {

memset(buffer, 0, sizeof(buffer));

cnt = read(fd, buffer, MAX\_LEN);

if (cnt == 0) break;

strcat(msg, buffer);

total = total + cnt;

if (buffer[cnt - 1] == EOF) {

strncpy(msg,msg,(strlen(msg)-1));

total--;

break;

}

}

return total;

}

int Update\_Cache(char \*URL, char \*buf, int flag, int x) {

int j=0;

int p=0;

if (flag == 1) { // New entry

if (num\_cache\_entries==MAX\_CACHE\_ENTRY){

Proxy\_Cache[0] = Clear\_Entry; // Popping LRU

for (j=0; j<MAX\_CACHE\_ENTRY; j++){

if (j+1!=MAX\_CACHE\_ENTRY)

Proxy\_Cache[j] = Proxy\_Cache[j+1];

else {

// Add new entry at the head (latest)

memset(&Proxy\_Cache[j], 0, sizeof(struct Cache));

memcpy(Proxy\_Cache[j].URL,URL,256);

Proxy\_Cache[j].body = (char \*) malloc(strlen(buf));

memcpy(Proxy\_Cache[j].body,buf,strlen(buf));

parseHDR("Expires:", buf, Proxy\_Cache[j].Expires);

parseHDR("Last-Modified:", buf, Proxy\_Cache[j].Last\_Modified);

parseHDR("Date:", buf, Proxy\_Cache[j].Access\_Date);

}

}

}

else { // If cache has not reached max allowed capacity (MAX\_CACHE\_ENTRY)

Proxy\_Cache[num\_cache\_entries] = Clear\_Entry;

memcpy(Proxy\_Cache[num\_cache\_entries].URL,URL,256);

parseHDR("Expires:", buf, Proxy\_Cache[num\_cache\_entries].Expires);

parseHDR("Last-Modified:", buf, Proxy\_Cache[num\_cache\_entries].Last\_Modified);

parseHDR("Date:", buf, Proxy\_Cache[num\_cache\_entries].Access\_Date);

Proxy\_Cache[num\_cache\_entries].body = (char \*) malloc(strlen(buf));

memcpy(Proxy\_Cache[num\_cache\_entries].body,buf,strlen(buf));

num\_cache\_entries++;

}

}

else { // Existing entry

struct Cache tmp;

memset(&tmp, 0, sizeof(struct Cache));

tmp = Proxy\_Cache[x];

for (j=x; j<num\_cache\_entries; j++){

if (j==num\_cache\_entries-1)

break;

Proxy\_Cache[j] = Proxy\_Cache[j+1];

}

Proxy\_Cache[num\_cache\_entries -1] = tmp;

struct tm tmp\_t;

time\_t nw = time(NULL);

tmp\_t = \*gmtime(&nw);

const char\* op\_tmp = "%a, %d %b %Y %H:%M:%S GMT";

strftime (Proxy\_Cache[num\_cache\_entries - 1].Access\_Date, 50, op\_tmp, &tmp\_t);

}

}

int Cache\_Display () {

int t = 0;

if (num\_cache\_entries == 0)

printf("Cache is unoccupied currently\n");

else {

printf("Cache count: %d\n", num\_cache\_entries);

for (t=0; t<num\_cache\_entries; t++) {

if (strcmp(Proxy\_Cache[t].Expires, "") != 0 && strcmp(Proxy\_Cache[t].Last\_Modified, "") != 0)

printf("Index: %d | URL: %s | Access Date: %s | Expires: %s | Last\_Modified: %s\n\n", t, Proxy\_Cache[t].URL, Proxy\_Cache[t].Access\_Date, Proxy\_Cache[t].Expires, Proxy\_Cache[t].Last\_Modified);

else if (strcmp(Proxy\_Cache[t].Expires, "") == 0 && strcmp(Proxy\_Cache[t].Last\_Modified, "") == 0)

printf("Index: %d | URL: %s | Access Date: %s | Expires: N/A | Last\_Modified: N/A\n\n", t, Proxy\_Cache[t].URL, Proxy\_Cache[t].Access\_Date);

else if (strcmp(Proxy\_Cache[t].Expires, "") == 0)

printf("Index: %d | URL: %s | Access Date: %s | Expires: N/A | Last\_Modified: %s\n\n", t, Proxy\_Cache[t].URL, Proxy\_Cache[t].Access\_Date, Proxy\_Cache[t].Last\_Modified);

else if (strcmp(Proxy\_Cache[t].Last\_Modified, "") == 0)

printf("Index: %d | URL: %s | Access Date: %s | Expires: %s | Last\_Modified: N/A\n\n", t, Proxy\_Cache[t].URL, Proxy\_Cache[t].Access\_Date, Proxy\_Cache[t].Expires);

}

}

return 0;

}

int Fresh (int cache\_ptr) {

struct tm tmp\_t;

time\_t nw = time(NULL);

tmp\_t = \*gmtime(&nw);

struct tm EXPIRES;

if (strcmp(Proxy\_Cache[cache\_ptr].Expires, "") != 0) {

strptime(Proxy\_Cache[cache\_ptr].Expires, "%a, %d %b %Y %H:%M:%S %Z", &EXPIRES);

time\_t EXP = mktime(&EXPIRES);

time\_t NOW = mktime(&tmp\_t);

if (difftime (NOW, EXP) < 0)

return 1;

else

return -1;

}

else

return -1;

}

int Cache\_Element(char \*URL) {

int b=0;

for (b=0; b<MAX\_CACHE\_ENTRY; b++) {

if (strcmp(Proxy\_Cache[b].URL, URL)==0) {

return b;

}

}

return -1;

}

int WebS\_Socket (char \*host) {

struct addrinfo dynamic\_addr, \*ai, \*p;

int ret\_val = 0;

int webs\_sockfd = 0;

memset(&dynamic\_addr, 0, sizeof dynamic\_addr);

dynamic\_addr.ai\_family = AF\_INET;

dynamic\_addr.ai\_socktype = SOCK\_STREAM;

if ((ret\_val = getaddrinfo(host, "http", &dynamic\_addr, &ai)) != 0) {

fprintf(stderr, "SERVER: %s\n", gai\_strerror(ret\_val));

exit(1);

}

for(p = ai; p != NULL; p = p->ai\_next) {

webs\_sockfd = socket(p->ai\_family, p->ai\_socktype, p->ai\_protocol);

if (webs\_sockfd >= 0 && (connect(webs\_sockfd, p->ai\_addr, p->ai\_addrlen) >= 0))

break;

}

if (p == NULL)

webs\_sockfd = -1;

freeaddrinfo(ai);

return webs\_sockfd;

}

int Proxy\_Server(int client\_fd) {

int webs\_sockfd;

char \*msg;

char forward\_client\_msg[MAX\_LEN] = {0};

int ret;

int cache\_el = 0;

//char resp[1024] = {0};

char \*resp = NULL;

//char to\_client[10240] = {0};

char \*to\_client = NULL;

//string Method;

//string Protocol;

char path[256];

char hostname[64];

int port = 80;

char URL[256] = {0};

char Method[8] = {0};

char Protocol[16] = {0};

char cond\_msg[256] = {0};

char url\_parse[256] = {0};

int check = 0;

//memset(&url, 0, sizeof url);

msg = (char \*) malloc (MAX\_LEN);

ret = read(client\_fd, msg, MAX\_LEN);

printf("SERVER: Request retrieved from client: \n%s", msg);

if (ret < 0)

err\_sys ("SERVER: Error in extracting message request from client");

sscanf(msg, "%s %s %s", Method, URL, Protocol);

//free (msg);

//printf("SERVER: URL extracted: %s\n", URL);

if ((cache\_el = Cache\_Element (URL)) != -1 && (Fresh (cache\_el) == 1)) {

//printf("Cache\_el: %d\n", cache\_el);

printf ("SERVER: Requested URL: %s is in cache and is fresh\n", URL);

Update\_Cache(URL, NULL, 0, cache\_el);

to\_client = (char \*) malloc(strlen(Proxy\_Cache[cache\_el].body));

memcpy(to\_client, Proxy\_Cache[cache\_el].body, strlen(Proxy\_Cache[cache\_el].body));

}

else { // Either URL is not cached or it is stale

memset(hostname, 0, 64);

memset(path, 0, 256);

memcpy(&url\_parse[0], &URL[0], 256);

parse\_URL (url\_parse, hostname, &port, path);

if ((webs\_sockfd = WebS\_Socket (hostname)) == -1)

err\_sys ("SERVER: Error in connecting with web server");

printf ("SERVER: Successfully connected to web server %d\n", webs\_sockfd);

if (cache\_el != -1) { // If cache entry exists but has expired

printf ("SERVER: Requested URL: %s is in cache but is expired\n", URL);

//split\_URL (URL, split\_url);

if (strcmp(Proxy\_Cache[cache\_el].Expires, "") != 0 && strcmp(Proxy\_Cache[cache\_el].Last\_Modified, "") != 0)

snprintf(cond\_msg, MAX\_LEN, "%s %s %s\r\nHost: %s\r\nUser-Agent: HTTPTool/1.0\r\nIf-Modified\_Since: %s\r\n\r\n", Method, path, Protocol, hostname, Proxy\_Cache[cache\_el].Expires);

else if (strcmp(Proxy\_Cache[cache\_el].Expires, "") == 0 && strcmp(Proxy\_Cache[cache\_el].Last\_Modified, "") == 0)

snprintf(cond\_msg, MAX\_LEN, "%s %s %s\r\nHost: %s\r\nUser-Agent: HTTPTool/1.0\r\nIf-Modified\_Since: %s\r\n\r\n", Method, path, Protocol, hostname, Proxy\_Cache[cache\_el].Access\_Date);

else if (strcmp(Proxy\_Cache[cache\_el].Expires, "") == 0)

snprintf(cond\_msg, MAX\_LEN, "%s %s %s\r\nHost: %s\r\nUser-Agent: HTTPTool/1.0\r\nIf-Modified\_Since: %s\r\n\r\n", Method, path, Protocol, hostname, Proxy\_Cache[cache\_el].Last\_Modified);

else if (strcmp(Proxy\_Cache[cache\_el].Last\_Modified, "") == 0)

snprintf(cond\_msg, MAX\_LEN, "%s %s %s\r\nHost: %s\r\nUser-Agent: HTTPTool/1.0\r\nIf-Modified\_Since: %s\r\n\r\n", Method, path, Protocol, hostname, Proxy\_Cache[cache\_el].Expires);

printf("Conditional GET Generated: \n%s", cond\_msg);

write(webs\_sockfd, cond\_msg, MAX\_LEN);

//resp = malloc (10240); // FIXME: May be needed to increase allocation

//memset(resp, 0, 1024);

resp = (char \*) malloc (100000);

check = Extract\_Read(webs\_sockfd, resp);

//printf("Checking: %d\n", check);

//Extract\_Read(webs\_sockfd, resp);

to\_client = (char \*) malloc(strlen(resp));

if (strstr(resp, "304 Not Modified") != NULL) {

printf("'304 Not Modified' received. Sending file in cache\n");

memcpy(to\_client, Proxy\_Cache[cache\_el].body, strlen(Proxy\_Cache[cache\_el].body));

Update\_Cache(URL, NULL, 0, cache\_el);

}

else {

printf("SERVER: File was modified\n");

memcpy(to\_client, resp, strlen(resp));

Update\_Cache(URL, NULL, 0, cache\_el); // move to head (LRU) of the queue

Proxy\_Cache[--num\_cache\_entries] = Clear\_Entry; // Popping LRU

Update\_Cache(URL, resp, 1, 0); // treat like a new entry as it was modified

}

}

else { // document is not cached

printf ("SERVER: Requested URL is not in cache\n");

memset(forward\_client\_msg, 0, MAX\_LEN);

snprintf(forward\_client\_msg, MAX\_LEN, "%s %s %s\r\nHost: %s\r\nUser-Agent: HTTPTool/1.0\r\n\r\n", Method, path, Protocol, hostname);

printf("SERVER: Request generated: \n%s", forward\_client\_msg);

write(webs\_sockfd, forward\_client\_msg, MAX\_LEN);

resp = (char \*) malloc (100000);

check = Extract\_Read(webs\_sockfd, resp);

to\_client = (char \*) malloc(strlen(resp));

memcpy(to\_client, resp, strlen(resp));

Update\_Cache(URL, resp, 1, 0);

}

}

Cache\_Display();

write(client\_fd, to\_client, strlen(to\_client) + 1);

}

int main (int argc, char \*argv[])

{

int sockfd, comm\_fd, bind\_fd, listen\_fd;

int port\_number ;

struct sockaddr\_storage remoteaddr;

socklen\_t addrlen;

if (argc != 3){

err\_sys ("USAGE: ./proxy <Server IP Address> <Port\_Number>");

return 0;

}

port\_number = atoi(argv[2]);

struct sockaddr\_in servaddr;

sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

if (sockfd < 0)

err\_sys ("ERR: Socket Error");

memset( &servaddr, 0 , sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

servaddr.sin\_addr.s\_addr = inet\_addr(argv[1]);

servaddr.sin\_port = htons(port\_number);

bind\_fd = bind(sockfd, (struct sockaddr \*)&servaddr, sizeof(servaddr));

if (bind\_fd < 0)

err\_sys ("ERR: Bind Error");

listen\_fd = listen(sockfd, 10);

if (listen\_fd < 0)

err\_sys ("ERR: Listen Error");

memset(Proxy\_Cache,0,MAX\_CACHE\_ENTRY\*sizeof(struct Cache));

addrlen = sizeof remoteaddr;

pthread\_t x;

printf("\nPROXY SERVER is online\n\n");

while(1)

{

comm\_fd = accept(sockfd, (struct sockaddr\*)&remoteaddr,&addrlen);

pthread\_create(&x, NULL, (void \*)(&Proxy\_Server), (void \*)(intptr\_t)comm\_fd);

}

}

CLIENT :

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netdb.h>

#include <unistd.h>

#include <arpa/inet.h>

#include <errno.h>

#include <dirent.h>

#include <pthread.h>

#include <signal.h>

#include <sys/wait.h>

int parse\_URL (char\* URL, char \*hostname, int \*port, char \*path) {

char \*token;

char \*host\_temp, \*path\_temp;

char \*tmp1, \*tmp2;

int num = 0;

char s[16];

if (strstr(URL,"http") != NULL){

token = strtok(URL, ":");

tmp1 = token + 7;

}

else{

tmp1 = URL;

}

tmp2 = malloc (64);

memcpy(tmp2, tmp1, 64);

if(strstr(tmp1, ":") != NULL){

host\_temp = strtok(tmp1, ":");

\*port = atoi(tmp1 + strlen(host\_temp) + 1);

sprintf(s, "%d", \*port);

path\_temp = tmp1 + strlen(host\_temp) + strlen(s) + 1;

}

else{

host\_temp = strtok(tmp1, "/");

\*port = 80;

path\_temp = tmp2 + strlen(host\_temp);

}

if (strcmp(path\_temp, "") == 0)

strcpy(path\_temp, "/");

memcpy(hostname, host\_temp, 64);

memcpy(path, path\_temp, 256);

return(0);

}

int err\_sys(const char\* x) // Error display source code

{

perror(x);

exit(1);

}

int main(int argc,char \*argv[])

{

int sockfd, inet\_a2n\_ret, conn\_ret, n;

char buff[100000] = {0};

int sendret = 0;

int recvret = 0;

unsigned int port\_number ;

char \*p, \*ptr;

char req[100];

char path[256] = {0};

char hostname[64] = {0};

int port = 80;

char URL[256] = {0};

if (argc != 4){

err\_sys ("USAGE: ./client <Server\_IP\_Address> <Port\_Number> <URL>");

exit(1);

}

port\_number = atoi(argv[2]);

struct sockaddr\_in servaddr;

bzero(&servaddr,sizeof servaddr);

servaddr.sin\_family=AF\_INET;

servaddr.sin\_port=htons(port\_number);

inet\_a2n\_ret = inet\_aton(argv[1], (struct in\_addr \*)&servaddr.sin\_addr.s\_addr); // FIXME: Maybe a problem

if (inet\_a2n\_ret <= 0)

err\_sys ("ERR: inet\_aton error");

sockfd=socket(AF\_INET,SOCK\_STREAM,0);

if (sockfd < 0)

err\_sys ("ERR: Socket Error");

conn\_ret = connect(sockfd,(struct sockaddr \*)&servaddr,sizeof(servaddr));

if (conn\_ret < 0)

err\_sys ("ERR: Connect Error");

memset(req, 0, 100);

sprintf(req, "GET %s HTTP/1.0\r\n", argv[3]);

printf("Request sent to proxy server: \n%s\n", req);

sendret = send(sockfd, req, strlen(req), 0);

if (sendret == -1) {

err\_sys("CLIENT: Send");

exit(2);

}

memset(buff, 0, 100000);

parse\_URL(argv[3], hostname, &port, path);

FILE \*fp;

fp=fopen(hostname, "w");

printf("Waiting for response\n");

recvret = recv(sockfd, buff, 100000, 0);

if (recvret <= 0) {

err\_sys("CLIENT: Recv");

fclose(fp);

close(sockfd);

return 1;

}

if((strstr(buff, "200")) != NULL)

printf("'200 OK' received. Saving to file: %s\n", hostname);

else if ((strstr(buff, "400") != NULL))

printf("'400 Bad Request' received. Saving to file: %s\n", hostname);

else if ((strstr(buff, "404") != NULL))

printf("'404 Page Not Found' received. Saving to file: %s\n", hostname);

ptr = strstr(buff, "\r\n\r\n");

fwrite(ptr+4, 1, strlen(ptr)-4, fp);

fclose(fp);

close(sockfd);

return 0;

}

MAKEFILE :

# 'make all' for compiling all code in package

all: proxy client

# 'make server' for compiling Server.c

proxy: Proxy.c

gcc -I . -pthread Proxy.c -o proxy

# 'make client' for compiling Client.c

client: Client.c

gcc -I . Client.c -o client

# 'make clean' for discarding all previously created object files

clean:

$(RM) client server