**Programming Assignment 4**

Chenjie Luo (UIN: 324007289)

Yu-Wen Chen(UIN: 227009499)

1. **Description**

The project is divided into two parts: server and client. In this assignment Chenjie Luo is responsible for server side as well as LRU cache while Yu-Wen Chen is responsible is responsible for client side and expire feature. We verified functionalities of two parts in different test cases together.

For the server’s architecture, when target file is executed, users will type in two inputs which are: server’s IP address as well as port number. After creating the socket and bind successfully, we will listen to all sockets in FD\_SET. When a request is received, we will firstly check if the url is cached in our LRU cache and not expire yet. If that is the case, we will directly respond to the client using the data in cache. Otherwise, we will need to send request to the target web server and request from it. The read buffer size I designed for url is 1024 bytes from client and the buffer to transmit html data to 102400 bytes. After we received it, we added it into cache and respond to the client as well. To realize this functionality, I designed and implemented our LRU cache. Firstly, I designed a data structure called Node to store entities in the LRU cache. It consists of key (url), data (html data), expireat, next (next Node) and last\_modified. For the LRU cache, it basically removed least recently used data from cache. Therefore, a linked-list-like structure could achieve this functionality. But since we will need to realize the functionality to move certain node to the tail of linked list when it was accessed, I used unordered\_map to implement the cache. For each element in unordered\_map, it mapped key(url) to its previous node and then this could help me access the address of current node which are to be moved to the end of linked list. I designed implement LRUcache as an object. It contains several variables including current size of cache, capacity, Header (header node of the linked list), Tail (tail node of the linked list) and several member functions including get(), push() print\_status() and moveToTail().

For the client side, the user needs to type server’s IP address, server’s port number and the requested URL as inputs. After receiving the URL, the client parses the URL into three portions: host name, data path and file name. Then, the client uses the input address and port number to connect to the server, sends the requested URL to the server and then waits for the server to provide the file from the requested URL. After receiving the file, the client first parses the first line of the file to get the response code and print it out, and then write the file into local directory with the parsed file name.

1. **Instruction to run our code**

1. After downloading the file from github, type “make” in the command line to generate

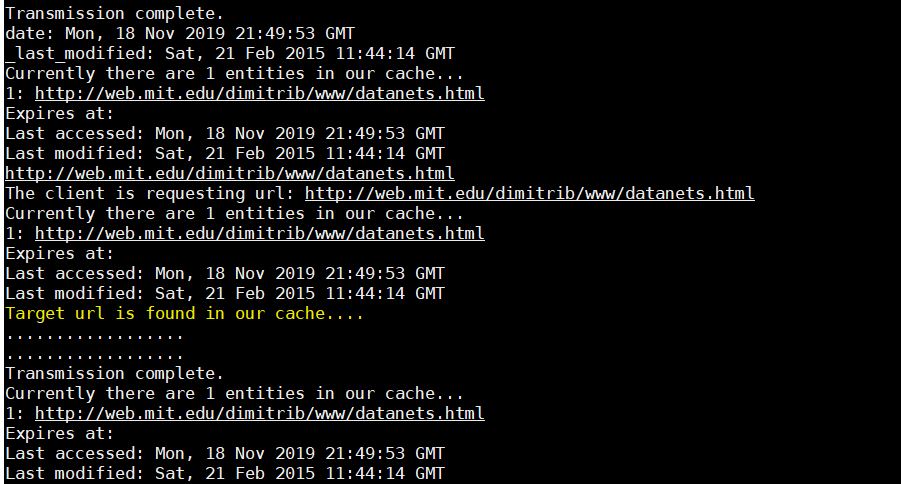
the execution file: server and client

2. Type “./server [server IP address] [port number]” to execute the server, and then type ./client [server IP address] [port number] [requested URL] to send request to server

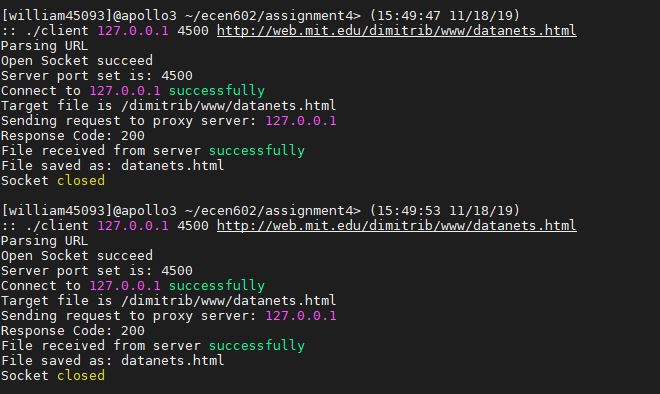
3. Now, users can use the proxy server and client to get the file from the requested URL

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

Server

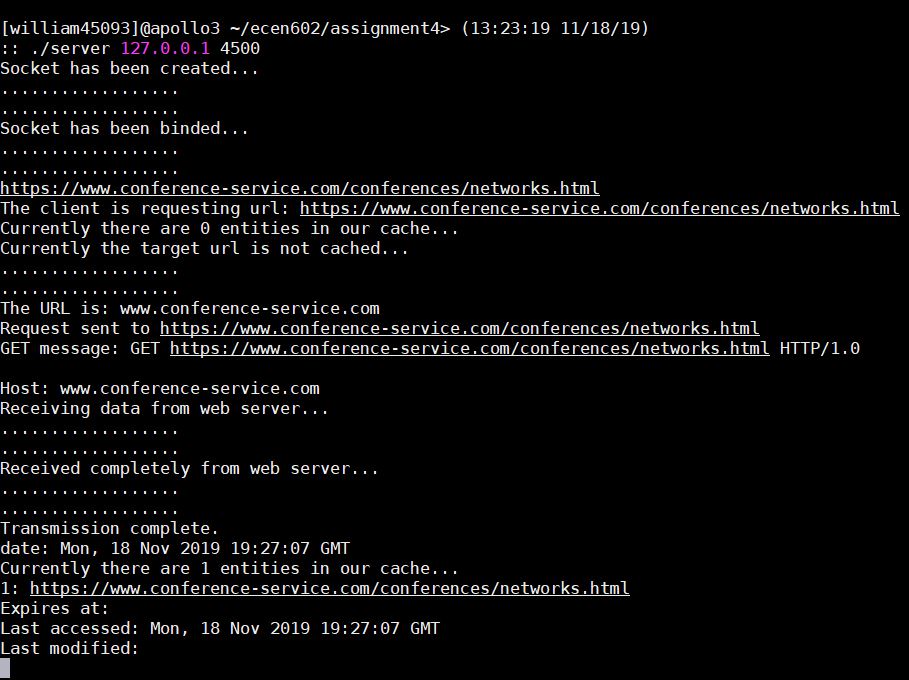


Client

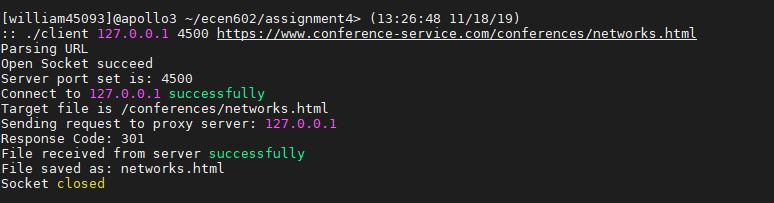


1. A request that is not in the cache is proxied, saved in the cache, and returned to the requester

Server



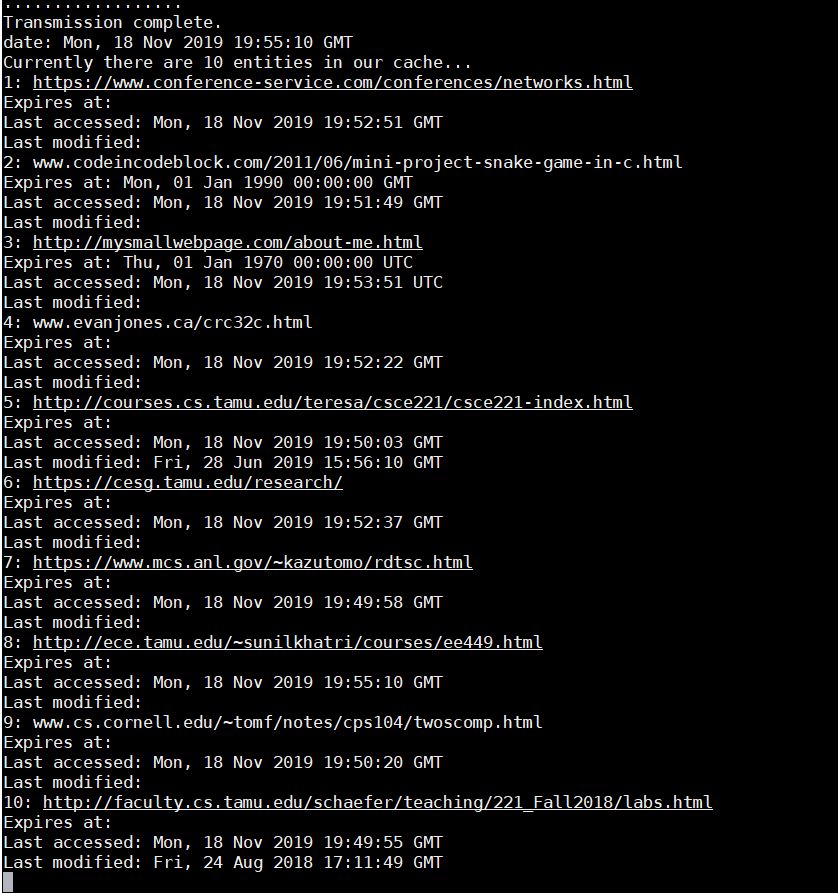
Client



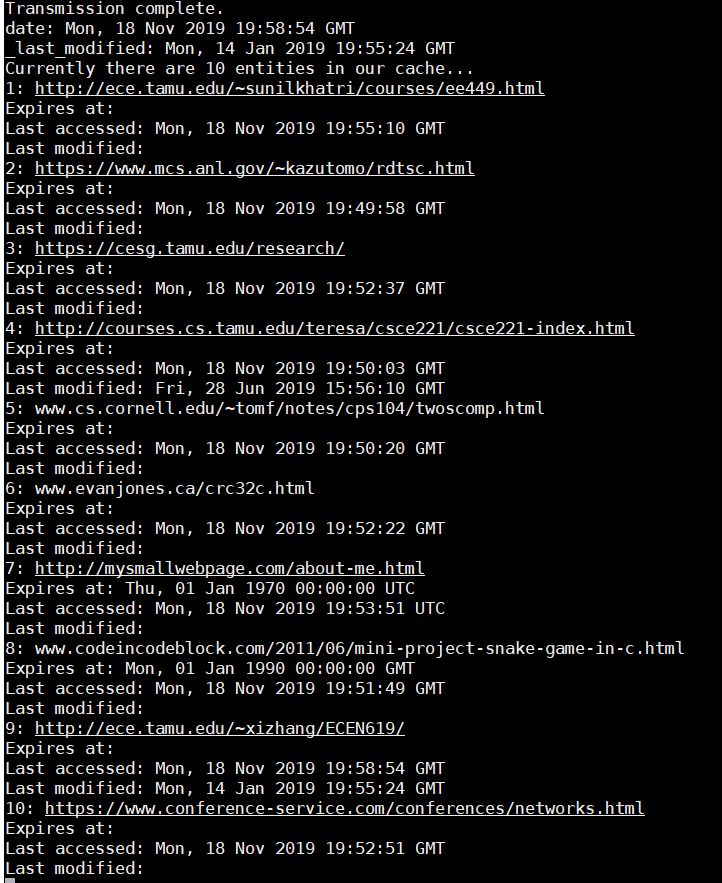
1. A cache miss with 10 items already in the cache is proxied, saved in the LRU location in cache, and the data is returned to the requester

Server

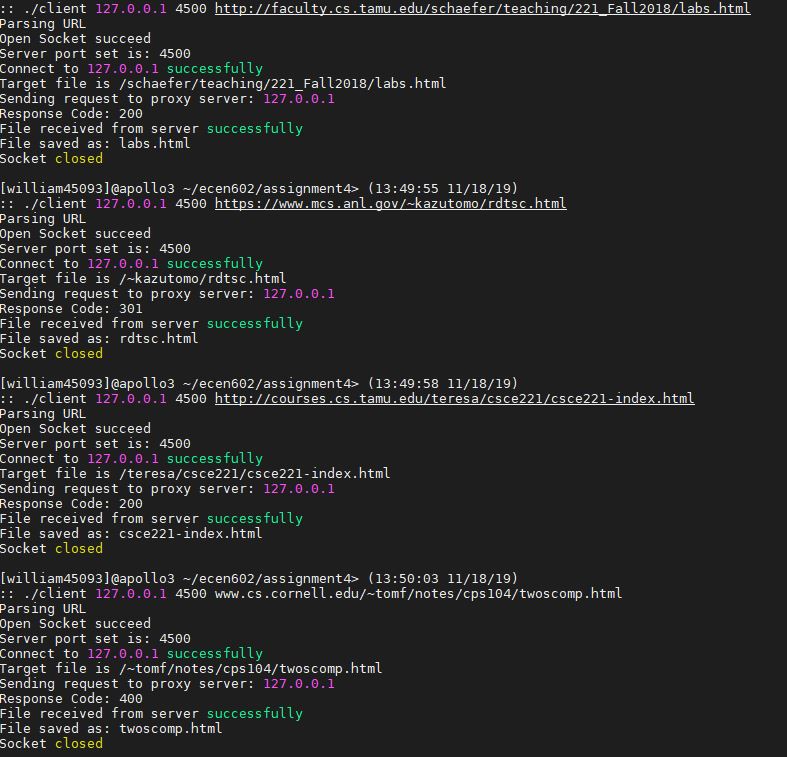
Full cache

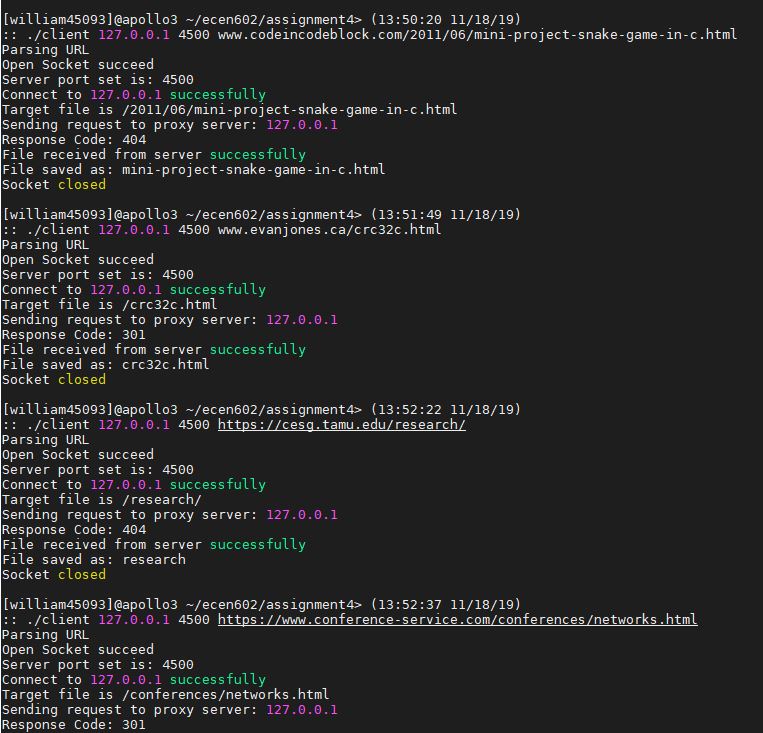


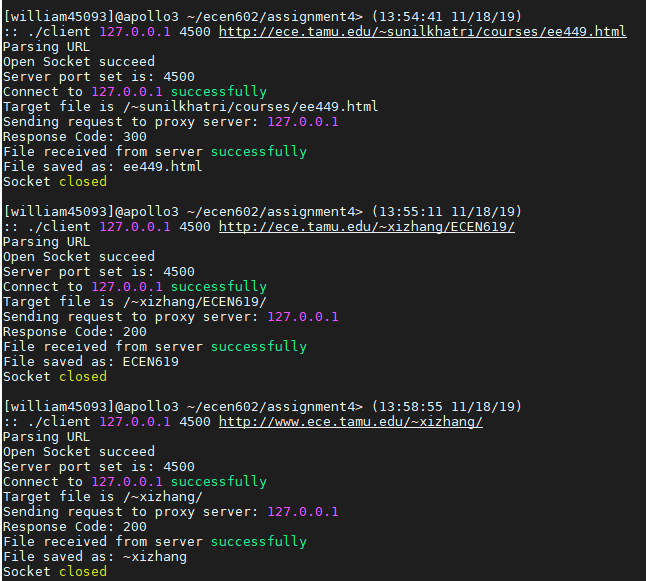
LRU cache: http://faculty.cs.tamu.edu/schaefer/teaching/221\_Fall2018/labs.html is replaced



Client - first input cache is: http://faculty.cs.tamu.edu/schaefer/teaching/221\_Fall2018/labs.html

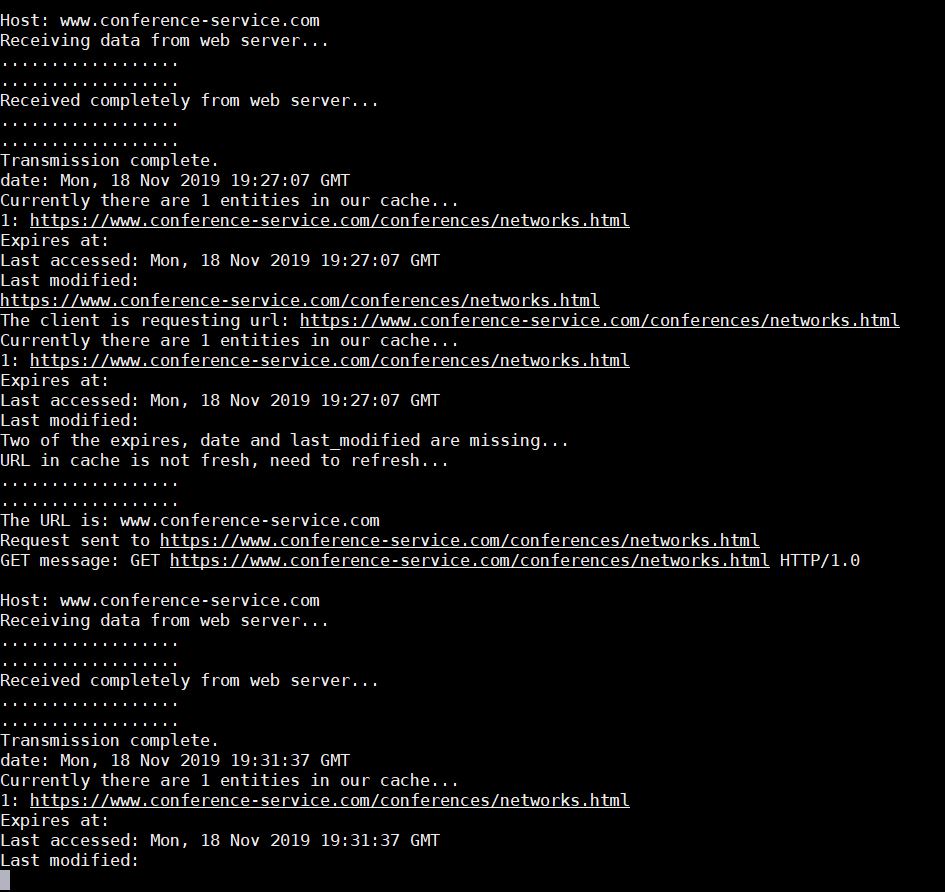




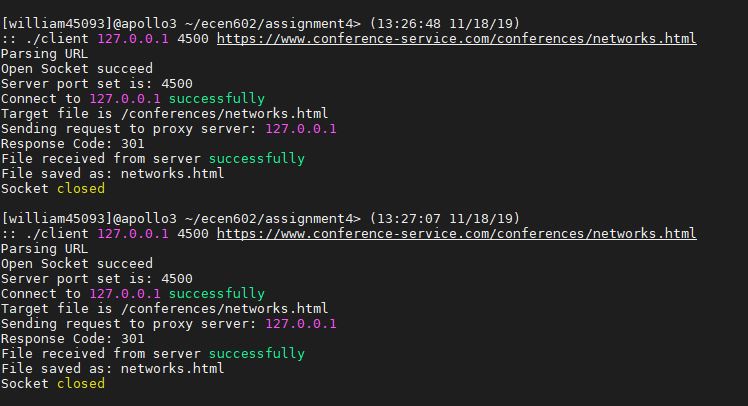


1. A stale Expires header in the cache is accessed, the cache entry is replaced with a fresh copy, and the fresh data is delivered to the requester

Server

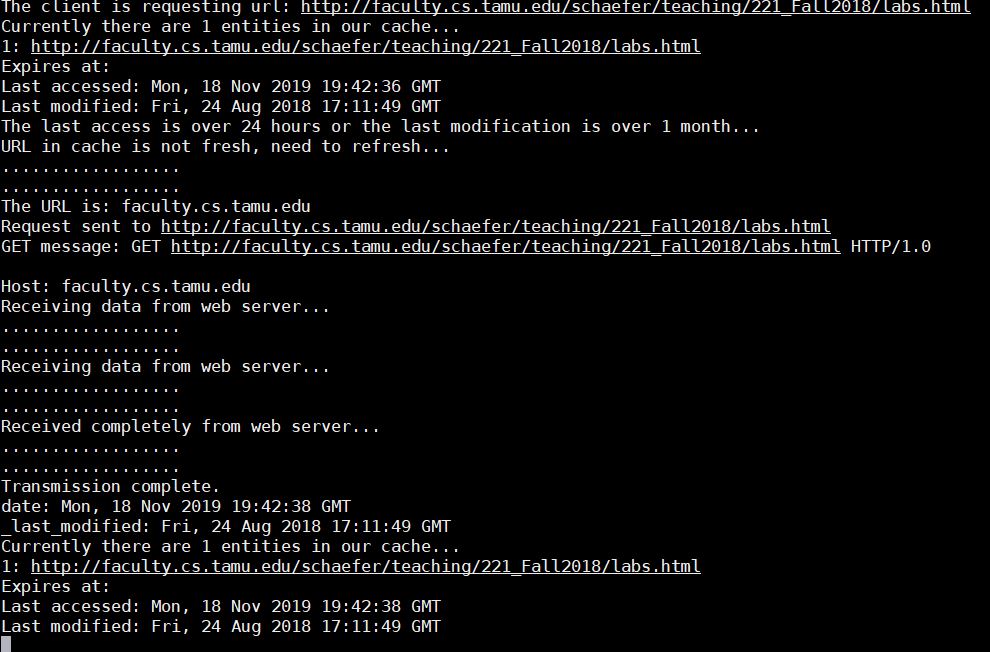


Client

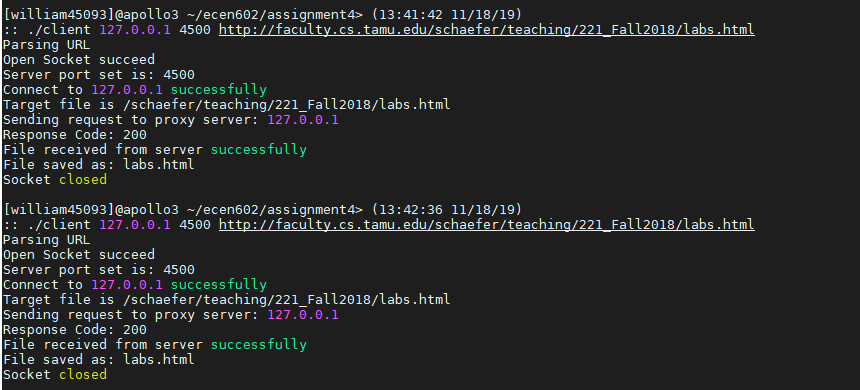


1. A stale entry in the cache without an Expires header is determined based on the last Web server access time and last modification time, the stale cache entry is replaced with fresh data, and the fresh data is delivered to the requester

Server

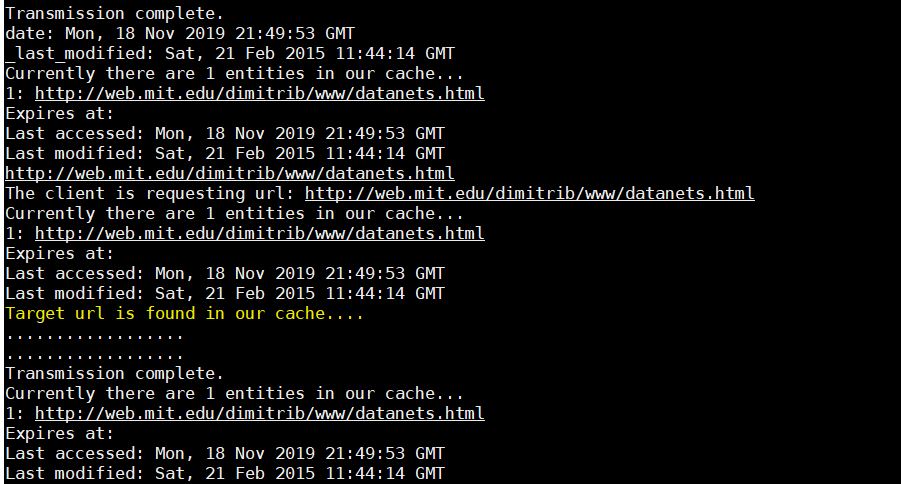


Client

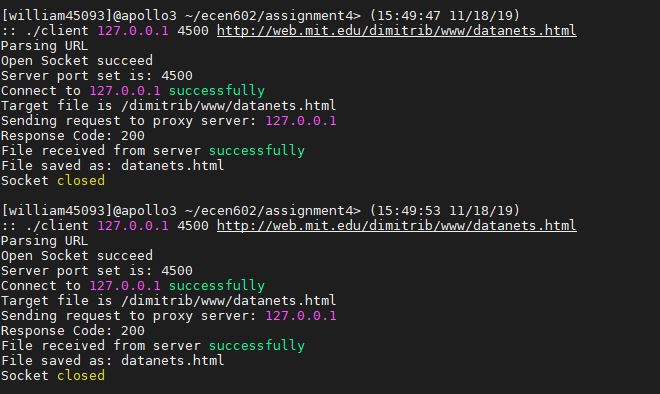


1. A cache entry without an Expires header that has been previously accessed from the Web server in the last 24 hours and was last modified more than one month ago is returned to the requester

Server

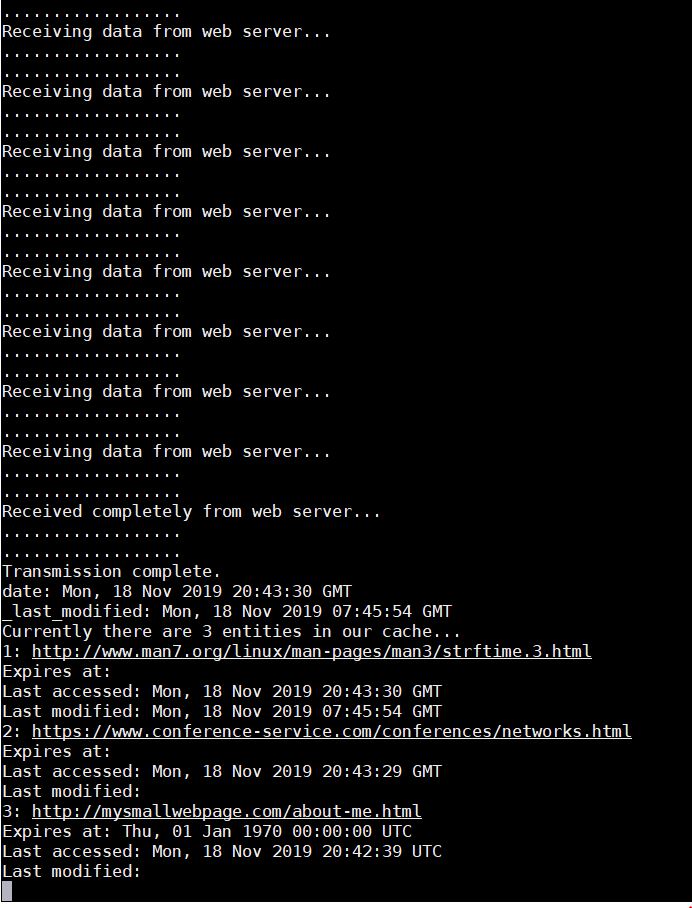


Client

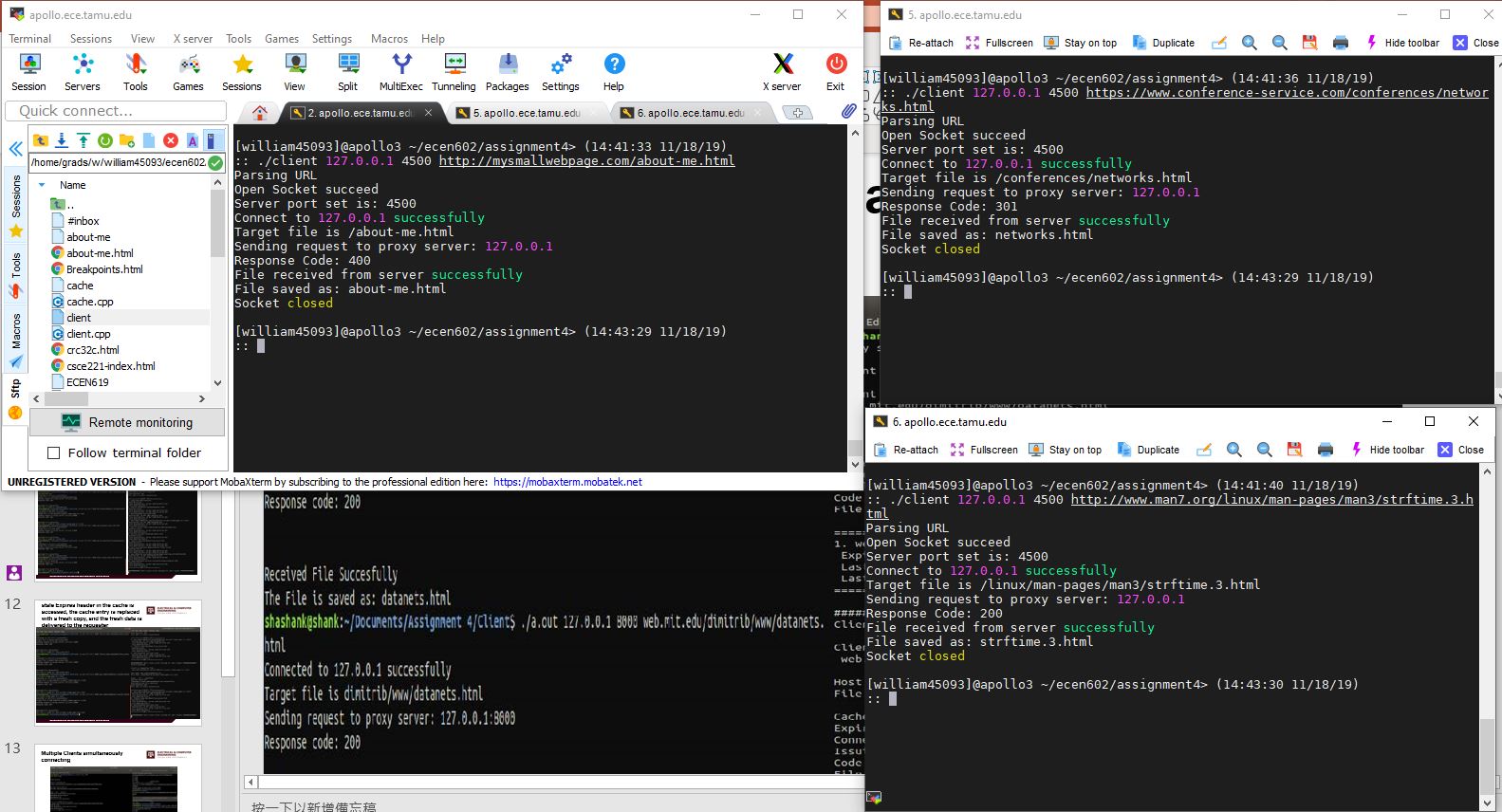


1. three clients can simultaneously access the proxy server and get the correct data

Server



Client



1. **Code**

server.cpp

#include <stdlib.h>

#include <iostream>

#include <vector>

#include <string>

#include <unordered\_map>

#include <unistd.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <sys/types.h>

#include <arpa/inet.h>

#include <sys/time.h>

#include <netdb.h>

#include <stdlib.h>

#include <errno.h>

#include <string.h>

#include <sys/wait.h>

#include <signal.h>

#include <sys/stat.h>

#include <time.h>

using namespace std;

#define MAX\_DATA\_SIZE 1024

#define MAX\_BUFFER\_SIZE 102400

// DESIGNED A DATA STRUCTURE CALLED NODE TO SAVE EACH ENTITY IN LRU CACHE.

struct Node{

string key;

string data;

string expireat;

string date;

string last\_modified;

Node\* next;

Node(): key(""), data(""), expireat(""), date(""), last\_modified(""), next(NULL) {}

Node(string key, string data, string expireat, string date, string last\_modified){

this->key = key;

this->data = data;

this->next = NULL;

this->expireat = expireat;

this->date = date;

this->last\_modified = last\_modified;

}

};

// LRU CACHE IS USED TO CACHE RECENT USED K ENTITIES. WHEN A REQUEST ARRIVES, WE FIRSTLY CHECK IF IT IS CONTAINED IN THE CACHE. IF NOT WE SEND REQUEST TO THE WEB SERVER

class LRUcache{

public:

unordered\_map<string, Node\*> map;

int size;

int capacity;

Node\* header;

Node\* tail;

LRUcache(int capacity) {

this->header = new Node();

this->tail = header;

this->size = 0;

this->capacity = capacity;

map.clear();

}

// MOVE THE NODE TO THE END OF END OF THE QUEUE TO MAINTAIN LRU CACHE

void movetoTail(Node\* prev){

if (prev->next == tail)

return;

Node\* temp = prev->next;

prev->next = temp->next;

map[temp->next->key] = prev;

map[temp->key] = tail;

tail->next = temp;

tail = tail->next;

}

Node\* get(string key) {

if (map.find(key) == map.end())

return NULL;

movetoTail(map[key]);

return map[key]->next;

}

void push(string key, string data, string \_expireat, string \_date, string \_last\_modified) {

if (map.find(key) != map.end()){

map[key]->next->data = data;

map[key]->next->expireat = \_expireat;

map[key]->next->date = \_date;

map[key]->next->last\_modified = \_last\_modified;

movetoTail(map[key]);

}

else{

Node \*temp = new Node(key, data, \_expireat, \_date, \_last\_modified);

map[key] = tail;

tail->next = temp;

tail = tail->next;

size += 1;

if (size > capacity){

Node\* temp2 = header->next;

map.erase(temp2->key);

header->next = temp2->next;

if (header->next != NULL)

map[temp2->next->key] = header;

size -= 1;

}

}

}

void print\_status(){

cout << "Currently there are " << map.size() << " entities in our cache..." << endl;

int cnt = 0;

for (auto &x: map){

cout << ++cnt << ": " << x.first << endl;

cout << "Expires at: " << map[x.first]->next->expireat << endl;

cout << "Last accessed: " << map[x.first]->next->date << endl;

cout << "Last modified: " << map[x.first]->next->last\_modified << endl;

}

}

};

int findsubstr(string str, string a){

for (int i = 7; i < str.length() - a.length(); i++){

if (str.substr(i, a.length()) == a)

return i;

}

return -1;

};

void print\_status(std::string s){

std::cout << s << "..." << std::endl;

std::cout << ".................." << std::endl;

std::cout << ".................." << std::endl;

}

double TimeDiffToNow (string t) {

int t\_length = t.length();

double difference;

char\* time\_in = new char[t\_length + 1];

time\_t curr\_time;

time\_t tm\_in;

struct tm\* now;

struct tm tm;

// current time

curr\_time = time(NULL);

now = gmtime(&curr\_time);

curr\_time = mktime(now);

// input time

strcpy(time\_in, t.c\_str());

strptime(time\_in, "%a, %d %b %Y %H:%M:%S %Z", &tm);

tm\_in = mktime(&tm);

difference = difftime(curr\_time, tm\_in);

char\* cur;

char\* in;

delete[] time\_in;

return difference;

}

void parsing\_URL(char\* URL, char\* host\_name, char\* path\_name) {

char\* temp\_URL = (char\*) malloc(MAX\_DATA\_SIZE \* sizeof(char));

char\* temp\_host;

char\* temp\_path;

char\* temp\_file;

char\* temp;

int length\_host;

int length\_path;

int length\_file;

// PARSING THE REQUESTED URL

memset(temp\_URL, 0, MAX\_DATA\_SIZE \* sizeof(char));

memcpy(temp\_URL, URL, strlen(URL));

if (strstr(temp\_URL, "https://") != NULL) {

temp\_host = temp\_URL + 8 \* sizeof(char); // point to the next char after "https:\\"

}

else if (strstr(temp\_URL, "http://") != NULL) {

temp\_host = temp\_URL + 7 \* sizeof(char); // point to the next char after "http:\\"

}

else {

temp\_host = temp\_URL; // point to the head of URL if no "http:\\" included

}

// find the second "/" and str before it would be path name

temp\_path = strtok(temp\_host, "/");

temp\_path = strtok(NULL, "/") - 1 \* sizeof(char);

memcpy(temp\_URL, URL, strlen(URL));

length\_host = strlen(temp\_host) - strlen(temp\_path);

length\_path = strlen(temp\_path) + 1;

memcpy(host\_name, temp\_host, length\_host);

memcpy(path\_name, temp\_path, length\_path);

temp = strtok(temp\_path, "/");

// find the file name

while (temp != NULL) {

temp\_file = temp;

temp = strtok(NULL, "/");

}

free(temp\_URL);

}

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

if (argc != 3){

errno = EPERM;

perror("Illegal Input! Please only input your ip address and port number. ");

exit(EXIT\_FAILURE);

}

string ID\_addr = argv[1];

string Port = argv[2];

int web\_socket;

int server\_socket;

int client\_socket;

struct addrinfo currinfo, \*serverinfo, \*p;

int current;

int yes = 1;

struct sockaddr\_storage client\_addr;

fd\_set master\_set;

fd\_set curr\_set;

socklen\_t addrlen;

int fdmax;

char buffer[MAX\_DATA\_SIZE];

char to\_get\_buffer[MAX\_BUFFER\_SIZE];

char to\_receive\_buffer[MAX\_BUFFER\_SIZE + 1];

char ipv4[30];

LRUcache mycache(10);

char path\_name[50];

char timestamp[30];

FD\_ZERO(&master\_set);

FD\_ZERO(&curr\_set);

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

currinfo.ai\_family = AF\_INET;

currinfo.ai\_socktype = SOCK\_STREAM;

currinfo.ai\_flags = AI\_PASSIVE;

if (getaddrinfo(NULL, argv[2], &currinfo, &serverinfo) != 0) {

perror("Fail to get address info");

exit(EXIT\_FAILURE);

}

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

if ((server\_socket = socket(p->ai\_family, p->ai\_socktype,p->ai\_protocol)) < 0){

perror("server: socket");

continue;

}

if (setsockopt(server\_socket, SOL\_SOCKET, SO\_REUSEADDR, &yes, sizeof(int)) < 0){

perror("Fail to create socket");

exit(EXIT\_FAILURE);

}

if (::bind(server\_socket, p->ai\_addr, p->ai\_addrlen) < 0) {

close(server\_socket);

perror("Fail to bind");

continue;

}

break;

}

if (!p){

perror("Fail to bind");

exit(EXIT\_FAILURE);

}

print\_status("Socket has been created");

print\_status("Socket has been binded");

if (listen(server\_socket, 10) < 0) {

perror("Fail to listen");

exit(EXIT\_FAILURE);

}

FD\_SET(server\_socket, &master\_set);

fdmax = server\_socket;

socklen\_t addr\_length;

while (true){

curr\_set = master\_set;

if (select(fdmax + 1, &curr\_set, NULL, NULL, NULL) < 0){

perror("Fail to select");

exit(EXIT\_FAILURE);

}

for (int i = 0; i <= fdmax; i++){

if (FD\_ISSET(i, &curr\_set)){

if (i == server\_socket){

addr\_length = sizeof(client\_addr);

client\_socket = accept(server\_socket, (struct sockaddr \*)&client\_addr, &addr\_length);

if (client\_socket < 0)

perror("Fail to accept");

else {

FD\_SET(client\_socket, &master\_set);

if (client\_socket > fdmax)

fdmax = client\_socket;

}

}

else{

size\_t received\_size = 0 ;

memset(buffer,0, MAX\_DATA\_SIZE);

received\_size = recv(i, buffer, MAX\_DATA\_SIZE, 0);

for (auto &x: buffer)

cout << x;

cout << endl;

if (received\_size <= 0) {

if (received\_size == 0) {

print\_status("No more data is received");

} else {

perror("Fail to received");

}

close(i);

FD\_CLR(i, &master\_set);

break;

}

cout << "The client is requesting url: " << buffer << endl;

string url\_in\_str(buffer);

mycache.print\_status();

Node\* res = mycache.get(url\_in\_str);

bool url\_expired = false;

// IF THE URL IS IN THE CACHE, WE NEED TO CHECK IF IT EXPIRED

if (res != NULL) {

if (res->expireat == "") {

if (((res->date) == "") || ((res->last\_modified) == "")) {

cout << "Two of the expires, date and last\_modified are missing..." <<endl;

print\_status("URL in cache is not fresh, need to refresh");

url\_expired = true;

}

else if ((TimeDiffToNow(res->date) > 86400.00) && (TimeDiffToNow(res->last\_modified) > 2592000.00)) {

cout << "The last access is over 24 hours or the last modification is over 1 month..." <<endl;

print\_status("URL in cache is not fresh, need to refresh");

url\_expired = true;

}

}

else if (TimeDiffToNow(res->expireat) > 0.00) {

cout << "URL in cache is expired..." <<endl;

print\_status("URL in cache is not fresh, need to refresh");

url\_expired = true;

}

}

if ((res == NULL) || (url\_expired == true)){

if (res == NULL) {

print\_status("Currently the target url is not cached");

}

memset(ipv4, 0, sizeof(ipv4));

memset(path\_name, 0, sizeof(path\_name));

parsing\_URL(buffer, ipv4, path\_name);

cout<<"The URL is: " << ipv4 << endl;

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

currinfo.ai\_family = AF\_INET;

currinfo.ai\_socktype = SOCK\_STREAM;

if (getaddrinfo(ipv4, "http", &currinfo, &serverinfo) != 0) {

perror("Fail to get address info");

exit(EXIT\_FAILURE);

}

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

if ((web\_socket = socket(p->ai\_family, p->ai\_socktype,p->ai\_protocol)) < 0){

perror("server: socket");

continue;

}

if (connect(web\_socket, p->ai\_addr, p->ai\_addrlen) < 0) {

close(web\_socket);

perror("Fail to conenct");

continue;

}

break;

}

freeaddrinfo(serverinfo);

memset(to\_get\_buffer, 0, MAX\_BUFFER\_SIZE);

strcpy(to\_get\_buffer, "GET ");

if (url\_in\_str.substr(0, 4) != "http")

strcat(to\_get\_buffer, "http://");

strcat(to\_get\_buffer, buffer);

strcat(to\_get\_buffer, " HTTP/1.0\r\n\r\n");

strcat(to\_get\_buffer, "Host: ");

strcat(to\_get\_buffer, ipv4);

received\_size = send(web\_socket, to\_get\_buffer, sizeof(to\_get\_buffer), 0);

cout << "Request sent to " << buffer << endl;

cout << "GET message: " << to\_get\_buffer << endl;

memset(to\_receive\_buffer, 0, MAX\_BUFFER\_SIZE);

received\_size = 0;

bool received = true;

char \*read\_ptr;

read\_ptr = to\_receive\_buffer;

size\_t sent\_size;

while (received){

received\_size = recv(web\_socket, read\_ptr, MAX\_DATA\_SIZE\*sizeof(char), 0);

if (strstr(read\_ptr, "404") != NULL){

print\_status("404 Not Found");

sent\_size = send(web\_socket, "404 Not Found", 10, 0);

close(web\_socket);

received = false;

break;

}

if (received\_size <= 0){

print\_status("Received completely from web server");

close(web\_socket);

received = false;

break;

}

print\_status("Receiving data from web server");

read\_ptr += received\_size;

}

sent\_size = send(i, to\_receive\_buffer, sizeof(to\_receive\_buffer), 0);

if (sent\_size <= 0){

perror("Fail to send");

exit(EXIT\_FAILURE);

}

cout << "Transmission complete." << endl;

// PARSING FOR EXPIRES, DATE, LAST\_MODIFIED

char\* ptr;

string \_expireat, \_date, \_last\_modified;

if ((ptr = strstr(to\_receive\_buffer, "expires:")) != NULL) {

memset(timestamp, 0, sizeof(timestamp));

memcpy(timestamp, ptr + 9, sizeof(timestamp));

\_expireat = timestamp;

cout << "expires: " << \_expireat << endl;

}

else if ((ptr = strstr(to\_receive\_buffer, "Expires:")) != NULL) {

memset(timestamp, 0, sizeof(timestamp));

memcpy(timestamp, ptr + 9, sizeof(timestamp));

\_expireat = timestamp;

cout << "expires: " << \_expireat << endl;

}

else {

\_expireat = "";

}

if ((ptr = strstr(to\_receive\_buffer, "date:")) != NULL) {

memset(timestamp, 0, sizeof(timestamp));

memcpy(timestamp, ptr + 6, sizeof(timestamp));

\_date = timestamp;

cout << "date: " << \_date << endl;

}

else if ((ptr = strstr(to\_receive\_buffer, "Date:")) != NULL) {

memset(timestamp, 0, sizeof(timestamp));

memcpy(timestamp, ptr + 6, sizeof(timestamp));

\_date = timestamp;

cout << "date: " << \_date << endl;

}

else {

\_date = "";

}

if ((ptr = strstr(to\_receive\_buffer, "Last-Modified:")) != NULL) {

memset(timestamp, 0, sizeof(timestamp));

memcpy(timestamp, ptr + 15, sizeof(timestamp));

\_last\_modified = timestamp;

cout << "\_last\_modified: " << \_last\_modified << endl;

}

else {

\_last\_modified = "";

}

// IF EXPIRES AND LAST-MODIFIED ARE MISSING, NOT CACHE THE URL (WE ARE UNSURE IF WE NEED TO CACHE IT IN THIS CASE)

// if ((\_expireat == "") &&(\_last\_modified =="")) {

// print\_status("Expires and Last-Modified are missing, the URL will not be cached");

// close(i);

// FD\_CLR(i, &master\_set);

// break;

// }

mycache.push(url\_in\_str, to\_receive\_buffer, \_expireat, \_date, \_last\_modified);

mycache.print\_status();

memset(buffer, 0, MAX\_DATA\_SIZE);

memset(to\_get\_buffer, 0, MAX\_BUFFER\_SIZE);

close(i);

FD\_CLR(i, &master\_set);

break;

}

else{

print\_status("Target url is found in our cache.");

strcpy(to\_receive\_buffer, res->data.c\_str());

int sent = send(i, to\_receive\_buffer, sizeof(to\_receive\_buffer), 0);

if (sent <= 0){

perror("Fail to send");

exit(EXIT\_FAILURE);

}

cout << "Transmission complete." << endl;

mycache.print\_status();

memset(buffer, 0, MAX\_DATA\_SIZE);

memset(to\_get\_buffer, 0, MAX\_BUFFER\_SIZE);

close(i);

FD\_CLR(i, &master\_set);

break;

}

}

}

}

}

return 0;

}

client.cpp

#include <stdio.h>

#include <string.h>

#include <unistd.h>

#include <sys/errno.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <arpa/inet.h>

#include <stdlib.h>

#include <iostream>

#define MAX\_BUFFER\_SIZE 1024 \* 1024

using namespace std;

void parsing\_URL(char\* URL, char\* host\_name, char\* path\_name, char\* file\_name) {

char\* temp\_URL = (char\*) malloc(150 \* sizeof(char));

char\* temp\_host;

char\* temp\_path;

char\* temp\_file;

char\* temp;

int length\_host;

int length\_path;

int length\_file;

// Parsing the requested URL

memset(temp\_URL, 0, 150 \* sizeof(char));

memcpy(temp\_URL, URL, strlen(URL));

if (strstr(temp\_URL, "https://") != NULL) {

temp\_host = temp\_URL + 8 \* sizeof(char); // point to the next char after "https:\\"

}

else if (strstr(temp\_URL, "http://") != NULL) {

temp\_host = temp\_URL + 7 \* sizeof(char); // point to the next char after "http:\\"

}

else {

temp\_host = temp\_URL; // point to the head of URL if no "http:\\" included

}

// find the second "/" and str before it would be path name

temp\_path = strtok(temp\_host, "/");

temp\_path = strtok(NULL, "/") - 1 \* sizeof(char);

memcpy(temp\_URL, URL, strlen(URL));

length\_host = strlen(temp\_host) - strlen(temp\_path);

length\_path = strlen(temp\_path) + 1;

memcpy(host\_name, temp\_host, length\_host);

memcpy(path\_name, temp\_path, length\_path);

temp = strtok(temp\_path, "/");

// find the file name

while (temp != NULL) {

temp\_file = temp;

temp = strtok(NULL, "/");

}

temp\_file = temp\_file;

memcpy(temp\_URL, URL, strlen(URL));

length\_file = strlen(temp\_file);

memcpy(file\_name, temp\_file, length\_file);

free(temp\_URL);

}

int main(int argc, char \*\*argv){

if (argc != 4) {

errno = EPERM;

printf("INPUT\_ERROR: ERRNO: \t%s\n", strerror(errno));

return -1;

}

char\* server\_name = argv[1];

char\* \_server\_port = argv[2];

char\* \_URL\_name = argv[3];

int server\_port = -1;

int c;

int socketfd = -1;

char host\_name[50];

char path\_name[100];

char file\_name[50];

char GET\_msg[150];

struct sockaddr\_in server\_addr;

char buffer[MAX\_BUFFER\_SIZE + 1];

int length\_recv = 0;

char response\_code[3];

char\* file\_head;

int file\_length;

char\* head\_response;

// Parsing the requested URL

printf("Parsing URL\n");

parsing\_URL(\_URL\_name, host\_name, path\_name, file\_name);

if (file\_name[strlen(file\_name) - 1] == '/') {

file\_name[strlen(file\_name) - 1] = '\0';

}

if ((socketfd = socket(AF\_INET, SOCK\_STREAM, 0)) < 0) {

printf("SOCKET\_OPEN: ERRNO: \t%s\n", strerror(errno));

return -1;

}

printf("Open Socket succeed\n");

// Input the server port to be connected

server\_port = atoi(\_server\_port);

printf("Server port set is: %d\n", server\_port);

// Setup the server address

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

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

server\_addr.sin\_port = htons(server\_port);

if (inet\_pton(AF\_INET, server\_name, &server\_addr.sin\_addr) < 0) {

printf("ADDR\_TRANS: ERRNO: \t%s\n", strerror(errno));

close(socketfd);

return -1;

}

// Connect the socket to the server

if ((connect(socketfd, (struct sockaddr \*)&server\_addr, sizeof(server\_addr))) < 0) {

printf("CONNECT: RRNO: \t%s\n", strerror(errno));

close(socketfd);

return -1;

}

printf("Connect to %s successfully\n", server\_name);

// Create GET message

sprintf(GET\_msg, "GET %s HTTP/1.0\r\nHOST: %s\r\n\r\n", path\_name, host\_name);

// Send GET request to proxy server

if ((write(socketfd, \_URL\_name, strlen(\_URL\_name))) < 0) {

printf("Send GET message: ERRNO: \t%s\n", strerror(errno));

close(socketfd);

return -1;

}

//printf("\*\*\*\*\*\*GET message was sent\*\*\*\*\*\*\n%s", GET\_msg);

printf("Target file is %s\n", path\_name);

printf("Sending request to proxy server: %s\n", server\_name);

// Received requested file from socket

memset(buffer, 0, (MAX\_BUFFER\_SIZE \* sizeof(char) + 1));

// Received file and save in the buffer

length\_recv = recv(socketfd, buffer, MAX\_BUFFER\_SIZE \* sizeof(char), 0);

if (length\_recv < 0) {

printf("Recv: ERRNO: \t%s\n", strerror(errno));

close(socketfd);

return -1;

}

// If no data received, close socket and return

else if (length\_recv == 0) {

printf("No data received from server\n");

close(socketfd);

return 0;

}

// Show response code

if ((head\_response = strstr(buffer, "HTTP/1.0")) != NULL){

memcpy(response\_code, head\_response + 9, 3 \* sizeof(char));

}

else if ((head\_response = strstr(buffer, "HTTP/1.1")) != NULL){

memcpy(response\_code, head\_response + 9, 3 \* sizeof(char));

}

printf("Response Code: %s\n", response\_code);

printf("File received from server successfully\n");

// Create file

FILE\* fd = fopen(file\_name, "w");

file\_head = buffer;

file\_length = strlen(buffer);

// Write the buffer to the file

fwrite(file\_head, sizeof(char), file\_length, fd);

printf("File saved as: %s\n", file\_name);

fclose(fd);

close(socketfd);

printf("Socket closed\n");

return 0;

}