Programming an HTTP Proxy Server

Introduction

In this programming assignment you will write an HTTP proxy-server that implements a limited subset of the entire HTTP specification.

The proxy server gets an HTTP request from the client, and performs some predefined checks on it. If the request is found legal, it forwards the request to the appropriate web server, and sends the response back to the client. Otherwise, it sends a response to the client without sending anything to the server. Only IPv4 connections should be supported.

Background

What is a proxy server?

A proxy server is a server that acts as an intermediary for requests from clients seeking resources from other servers. A client connects to the proxy server, requesting some service, such as a file, connection, web page, or other resource available from a different server. The proxy server evaluates the request according to its filtering rules. For example, it may filter traffic by several headers in the packet. If the request is validated by the filter, the proxy provides the resource by connecting to the relevant server and requesting the service on behalf of the client.

A proxy server may optionally alter the client's request or the server's response, and sometimes it may serve the request without contacting the specified server. In this case, it 'caches' responses from the remote server, and returns subsequent requests for the same content directly. However, in this exercise, the proxy does not cache responses (it only evaluates the request according to its filtering rules).

Program Description and What You Need to Do:

You will write two source files, proxyServer.c and threadpool.c (the executable file should be called proxyServer). The server should handle the connections with the clients. As we saw in class, when using TCP, a server creates a socket for each client it talks to. In other words, there is always one socket where the server listens to connections and for each client connection request, the server opens another socket. In order to enable multithreaded program, the server should create threads that handle the connections with the clients. Since, the server should maintain a limited number of threads, it construct a thread pool. In other words, the server create the pool of threads in advanced and each time it needs a thread to handle a client connection, it take one from the pool or enqueue the request if there is no available thread in the pool.

Command line usage: proxyServer <port> <pool-size> <max-number-of-request> <filter>

- Port is the port number your proxy server will listen on.
- pool-size is the number of threads in the pool

- number-of-request is the maximum number of requests your server will handle before it terminates (we count also unsuccessful requests). This parameter implies that our proxy server does not run forever.
- Filter is an absolute path to the filter file. This file contains Host names and sub-networks separated by new line that the proxy will filter. The sub-networks are in the format: n1.n2.n3.n4/x such that each number n1,n2,n3,n4 is between 0 to 255 and x is between 1 and 32. For a rule n1.n2.n3.n4/x, all IP addresses whose first (most significant) x bits match n1.n2.n3.n4 should be denied. We assume that host names do not begin with numbers. You may see an example of a filter in file filterExample.

In your program you need to:

Generally, once your server is up, it's ready to get requests. The server should parse the request, get the host name, get the IP from the host name, check the host name and the IP against the filter file, and if everything is ok, connect the origin server and send it the request. Then, read the response from the server and sends it back to the client.

In details:

- Read http-request from socket
- Check input: the request first line should contain method, path and protocol, and there should be a Host header. Here, you only have to check that these tokens exist, that the protocol is one of the http versions, and that Host header exists. Other checks on the method and the path will be checked later. In case the request is wrong, send 400 "Bad Request" respond, as in file 400.pdf
- You should support only the GET method, if you get another method, return error message 501 "Not Supported", as in file 501.pdf
- If you can't get the IP of the server (from the request's Host header), send 404 "Not Found" response, as in file 404.pdf
- If the requested URL direct to an IP address or host that appears in the filter-list, return error message 403 "Forbidden" as in file 403.pdf
- Otherwise, the http request is legal. Connect to the server (you have information about the server in the field "Host:" of the request), send the request to the server, get its response, and send it back to the client.

Few comments:

1. Our proxy server closes connection after sending the response, add header "Connection: close".
2. Don't use files to send error responses, this is very inefficient.
3. Each user should be handled in new thread.
4. When you fill your sockaddr_in struct, you can use htonl(INADDR_ANY) when assigning sin_addr.s_addr, meaning that the proxy server listen to requests in any of its addresses.

The threadpool

The pool is implemented by a queue. When the server gets a connection (getting back from accept()), it should put the connection in the queue. When there will be available thread (can be immediately), it will handle this connection (read request and write response).

You should implement the functions in threadpool.h.

The server should first init the thread pool by calling the function create_threadpool(int). This function gets the size of the pool.

create_threadpool should:

1. Check the legacy of the parameter.
2. Create threadpool structure and initialize it:
    a. num_thread = given parameter
    b. qsize=0
    c. threads = pointer to <num_thread> threads
    d. qhead = qtail = NULL
    e. Init lock and condition variables.
    f. shutdown = dont_accept = 0
    g. Create the threads with *do_work* as execution function and the *pool* as an argument.

do_work should run in an endless loop and:

1. If destruction process has begun, exit thread
2. If the queue is empty, wait (no job to make)
3. Check again destruction flag.
4. Take the first element from the queue (*work_t)
5. If the queue becomes empty and destruction process wait to begin, signal destruction process.
6. Call the thread routine.

dispatch gets the pool, pointer to the thread execution routine and argument to thread execution routine. dispatch should:

1. Create work_t structure and init it with the routine and argument.
2. If destroy function has begun, don't accept new item to the queue
3. Add item to the queue

destroy_threadpool

1. Set don't_accept flag to 1
2. Wait for queue to become empty
3. Set shutdown flag to 1
4. Signal threads that wait on empty queue, so they can wake up, see shutdown flag and exit.
5. Join all threads
6. Free whatever you have to free.

Program flow:

1. Server creates pool of threads, threads wait for jobs.
2. Server accept a new connection from a client (aka a new socket fd)
3. Server dispatch a job - call dispatch with the main negotiation function and fd as a parameter. dispatch will add work_t item to the queue.
4. When there will be an available thread, it will takes a job from the queue and run the negotiation function.

Error handling:

1. In any case of wrong command usage, print

"Usage: proxyServer <port> <pool-size> <max-number-of-request> <filter>\n"

2. In any case of a failure before connection with client is set, use perror("error: <sys_call>\n") and exit the program.

3. In any case of a failure after the connection with client is set, and in case the error is due to some server-side error (like failure in malloc), send a 500 "Internal Server Error", as in file 500.pdf.

Don't forget to enter new line after each error message (in the first two cases).

Additional Details:

Useful function:

1. Strchr
2. Strstr
3. strtok
4. strcat
5. strftime
6. gmtime
7. scandir
8. opendir
9. readdir
10. stat
11. S_ISDIR
12. S_ISREG

Assumptions:

You should relate only to the method GET (always in the first line), and the header Host (the headers of HTTP request are separated from the possible-message by empty line; the Host doesn't have to be the first header!). You can ignore anything else.

Compile the proxy server:

Remember that you have to compile with the –lpthread flag.

What to submit:

You should submit a tar file called ex2_b.tar with proxyServer.c, threadpool.c and README. Find README instructions in the course web-site. **DON'T SUBMIT threadpool.h**

tar –cvf ex2_b.tar proxyServer.c threadpool.c README

Test your code:

You can use a browser; you should configure your browser to use a proxy. The proxy's name is the computer name, and the port is the port of your program. If you run the browser from the same machine as the server, the computer-name is localhost.