CPSC 5510 Computer Networks

Project 2: Group Project – Implementing a Simple HTTP Web Proxy

Note:
This is a group project. You cannot take it as an individual project. The max group size is 3.

I. Overview:

This project aims to implement a simple web proxy using HTTP 1.0. It only requires implementing GET method.

II. Background:

HTTP

The Hypertext Transfer Protocol or (HTTP) is the protocol used for communication on the web. That is, it is the protocol which defines how your web browser requests resources from a web server and how the server responds. For simplicity, in this project we will be dealing only with version 1.0 of the HTTP protocol, defined in detail in RFC 1945. You may refer back to the RFC while completing this assignment, but our instructions should be self-contained.

HTTP communications happen in the form of transactions. A transaction consists of a client sending a request to a server and then reading the response. Request and response messages share a common basic format:

- An initial line (a request or response line, as defined below)
- Zero or more header lines
- A blank line (CRLF)
- An optional message body.

The initial line and header lines are each followed by a "carriage-return line-feed (CRLF)" (\r\n) signifying the end-of-line.

For most common HTTP transactions, the protocol boils down to a relatively simple series of steps (important sections of RFC 1945 are in parenthesis):

1. A client creates a connection to the server.
2. The client issues a request by sending a line of text to the server. This **request line** consists of a HTTP *method* (most often GET, but POST, PUT, and others are possible in real world), a *request URI* (like a URL), and the protocol version that the client wants to use (HTTP/1.0). The message body of the initial request is typically empty. (5.1-5.2, 8.1-8.3, 10, D.1)
3. The server sends a response message, with its initial line consisting of a **status line**, indicating if the request was successful. The status line consists of the HTTP version (HTTP/1.0), a *response status code* (a numerical value that indicates whether or not the request was completed successfully), and a *reason phrase*, an English-language message providing description of the status code. Just as with the request message, there can be as many or as few header fields in the response as the server wants to return. Following the CRLF field separator, the message

body contains the data requested by the client in the event of a successful request. (6.1-6.2, 9.1-9.5, 10)

4. Once the server has returned the response to the client, it closes the connection. It's fairly easy to see this process in action without using a web browser. From a Unix/LINUX prompt, type:

   telnet www.yahoo.com 80

   This opens a TCP connection to the server at www.yahoo.com listening on port 80 – the default HTTP port. You should see something like this:

   Trying 209.131.36.158...
   Connected to www.yahoo.com (209.131.36.158).
   Escape character is '^]'.

   type the following:
   GET / HTTP/1.0
   Hostname:www.yahoo.com
   Connection:close
   and hit enter twice. You should see something like the following:

   HTTP/1.0 200 OK
   Date: Fri, 10 Nov 2006 20:31:19 GMT
   Connection: close
   Content-Type: text/html; charset=utf-8
   <html><head>
   <title>Yahoo!</title>
   (More HTML follows)

There may be some additional pieces of header information as well-setting cookies, instructions to the browser or proxy on caching behavior, etc. What you are seeing is exactly what your web browser sees when it goes to the Yahoo home page: the HTTP status line, the header fields, and finally the HTTP message body – consisting of the HTML that your browser interprets to create a web page. You may notice here that the server responds with HTTP 1.1 even though you requested 1.0. Some web servers refuse to serve HTTP 1.0 content.

HTTP Proxy

Ordinarily, HTTP is a client-server protocol. The client (usually your web browser) communicates directly with the server (the web server software). However, in some circumstances it may be useful to introduce an intermediate entity called a proxy. Conceptually, the proxy sits between the client and the server. In the simplest case, instead of sending requests directly to the server the client sends all its requests to the proxy. The proxy then opens a connection to the server, and passes on the client's request. The proxy receives the reply from the server, and then sends that reply back to the client. Notice that the proxy is essentially acting like both a HTTP client (to the remote server) and a HTTP server (to the initial client).

Why use a proxy? There are a few possible reasons:

• **Performance:** By saving a copy of the pages that it fetches, a proxy can reduce the need to create connections to remote servers. This can reduce the overall delay involved in retrieving a page, particularly if a server is remote or under heavy load.

• **Content Filtering and Transformation:** While in the simplest case the proxy merely fetches a resource without inspecting it, there is nothing that says that a proxy is limited to blindly fetching and serving files. The proxy can inspect the requested URL and selectively block access to certain domains, reformat web pages (for instances, by stripping out images to make a page easier to display on a handheld or other limited-resource client), or perform other transformations and filtering.

• **Privacy:** Normally, web servers log all incoming requests for resources. This information typically includes at least the IP address of the client, the browser or their client program that they are using (called the User-Agent), the date and time, and the requested file. If a client does not wish to have this personally identifiable information recorded, routing HTTP requests through a proxy is one solution. All requests coming from clients using the same proxy appear to come from the IP address and User-Agent of the proxy itself, rather than the individual clients. If a number of clients use the same proxy (say, an entire business or university), it becomes much harder to link a particular HTTP transaction to a single computer or individual.

III. Assignment Details

The Basics

Your task is to build a web proxy capable of accepting HTTP requests, forwarding requests to remote (origin) servers, and returning response data to a client. The proxy MUST handle **concurrent** requests by using multi-threading. You will only be responsible for implementing the GET method. All other request methods received by the proxy should elicit a "Not Implemented" (501) error (see RFC 1945 section 9.5 - Server Error).

This assignment can be completed in either C or C++. It should compile and run without errors or warnings from the cs2.seattleu.edu, producing a binary called proxy that takes as its first argument a port to listen from. Don't use a hard-coded port number. The proxy should be running on cs2 since cs1 is configured not to receive uninvited incoming TCP connections.

Listening

When your proxy starts, the first thing that it will need to do is establish a socket connection that it can use to listen for incoming connections. Your proxy should listen on the port specified from the command line and wait for incoming client connections. Each new client request is accepted, and a worker thread handles the request. To avoid overwhelming your server, you should not create more than a reasonable number of worker threads (for this experiment, 30 used to be a good number). You can use a thread pool and producer/consumer model used in undergraduate operating systems class.

Once a client has connected, the proxy should read data from the client and then check for a properly-formatted HTTP request. Specifically, you will parse the HTTP request to ensure that the proxy receives a request that contains a valid request line:

<METHOD> <URL> <HTTP VERSION>

All other headers just need to be properly formatted:
<HEADER NAME>: <HEADER VALUE>

In this assignment, client requests to the proxy must be in their absolute URI form (see RFC 1945, Section 5.1.2), e.g.,

GET http://www.seattleu.edu/index.html HTTP/1.0

Your browser will send absolute URI if properly configured to explicitly use a proxy (as opposed to a transparent on-path proxy). On the other form, your proxy should issue requests to the webserver properly specifying *relative* URLs, e.g.,

GET /index.html HTTP/1.0
Host: www.seattleu.edu

An invalid request from the client should be answered with an appropriate error code, i.e. 500 'Internal Error'. Similarly, if headers are not properly formatted for parsing, your proxy should also generate a type of 500 'Internal Error' message.

Parsing the URL
Once the proxy receives a valid HTTP request, it will need to parse the requested URL. The proxy needs at least three pieces of information: the requested host, port, and path. You will need to parse the absolute URL specified in the given request line. If the hostname indicated in the absolute URL does not have a port specified, you should use the default HTTP port 80.

Getting Data from the Remote Server
Once the proxy has parsed the URL, it can make a connection to the requested host (using the appropriate remote port, or the default of 80 if none is specified) and send the HTTP request for the appropriate resource. The proxy should always send the request in the relative URL + Host header format regardless of how the request was received from the client:

Accept from client:
GET http://www.cnn.com/ HTTP/1.0

Send to remote server:
GET / HTTP/1.0
Host: www.cnn.com
Connection: close
(Additional client specified headers, if any...)

Note that we always send HTTP/1.0 flags and a Connection: close header to the server, so that it will close the connection after its response is fully transmitted, as opposed to keeping open a persistent connection (as we learned in class). So while you should pass the client headers you receive on to the server, you should make sure you replace any Connection header received from the client with one specifying close, as shown. To add new headers or modify existing ones, do it when parsing the HTTP request.

Returning Data to the Client

After the response from the remote server is received, the proxy should send the response message as-is to the client via the appropriate socket. To be strict, the proxy would be required to ensure a Connection: close is present in the server's response to let the client decide if it should close its end of the connection after receiving the response. However, checking this is not required in this assignment for the following reasons. First, a well-behaving server would respond with a Connection: close anyway given that we ensure that we sent the server a close token. Second, we configure Firefox to always send a Connection: close by setting keepalive to false. Finally, we wanted to simplify the assignment so you wouldn't have to parse the server response.

The following summarizes how status replies should be sent from the proxy to the client:

1. For any error your proxy should return the status 500 'Internal Error'. This means for any request method other than GET, your proxy should return the status 500 'Internal Error' rather than 501 'Not Implemented'. Likewise, for any invalid, incorrectly formed headers or requests, your proxy should return the status 500 'Internal Error' rather than 400 'Bad Request' to the client. For any error that your proxy has in processing a request such as failed memory allocation or missing files, your proxy should also return the status 500 'Internal Error'. (This is what is done by default in this case.)
2. Your proxy should simply forward status replies from the remote server to the client. This means most 1xx, 2xx, 3xx, 4xx, and 5xx status replies should go directly from the remote server to the client through your proxy. Most often this should be the status 200 'OK'. However, it may also be the status 404 'Not Found' from the remote server. (While you are debugging, make sure you are getting valid 404 status replies from the remote server and not the result of poorly forwarded requests from your proxy.)

IV. Submission and Grading

This project has two checkpoint submissions.

IV-1 Phase #1

Given a client's HTTP request, your proxy server should establish a TCP connection to a web server, reconstruct the client's HTTP request (as described in Section III), forward it to the web server, receive the HTTP response from the web server, display the response on the screen, and exit.

In this phase, your proxy server does NOT have to:

(1) be a multi-threading proxy.
(2) parse the received base file and download the embedded objects.

Your proxy server should function like running "telnet www.cnn.com 80" when we do the following:

(1) Run your proxy server at cs2.seattleu.edu with the command:

./proxy <port#>

Where port# is the port number that the proxy should listen on.

(2) As a basic test of functionality, try requesting a page using telnet:

telnet localhost <port#>
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
GET http://www.cnn.com/ HTTP/1.0

Where port# is the port number specified in (1).

If your proxy is working correctly, the headers and HTML of the CNN homepage should be displayed on your terminal screen. Notice here that we request the absolute URL (http://www.cnn.com/) instead of just the relative URL (/) since your proxy needs to reconstruct the request based on the absolute URL and then send it to the server (as described in Section III). A good sanity check of proxy behavior would be to compare the HTTP response (headers and body) obtained via your proxy with the response from a direct telnet connection to the remote server.

Below are the grading criteria for this phase:

Label	Grading components	Comments
2-A	[1 pts] Complete file submission: all source files, Readme, & Makefile -- Readme file should list each team member's name and his/her contributions. -- Readme file should describe your code and the design decisions you made as well as strengths and weakness.	1. Missing Makefile & Readme: -0.5 pt 2. Makefile does not work: -0.5 pt
2-B	[3 pts]Basic Functionality: --- It displays the received HTTP response on the terminal screen given a user GET request.	As described in Phase #1
2-C	[1 pt] Your proxy should run silently: any status messages or diagnostic output should be off by default.	
2-D	Either the program cannot be compiled Or fatal segmentation faults	Zero point for this phase

The submission deadline: **11:59PM, Sunday, Oct 29, 2017**

In cs1.seattleu.edu, make your files into a tar package, named **project2.tar**, which includes Makefile, all source files, and a README file. E.g.,

tar -cvf project2.tar proxy.h proxy.cpp Makefile README

Run the command to submit:

$ /home/fac/zhuy/class/CPSC5510/submit p21 project2.tar

IV-2 Phase #2

In this phase, your "full-blown" proxy server should be a multi-threading server that handles multiple HTTP requests simultaneously (and thus synchronization & race condition). Your proxy should use PThread library in this assignment.

Your proxy should parse the received base file from the web server, download the embedded objects, and forward all to the client (web browser).

Not only should your proxy function correctly, but also it should have good performance (fast loading of web pages) and stay robust (i.e., do not crash upon client requests). **Regarding performance and robustness in the face of multiple clients, in-class demo is required. The date for in-class demo will be announced later.**

To test your proxy, you can repeat the process you did in Phase #1. But, you can try requesting a page using telnet concurrently from two different shells to test that it is a multi-client server.

For a slightly more complex test, you can configure Firefox to use your proxy server as its web proxy as follows (more recent versions of Firefox could differ slightly):

1. Go to the 'Edit' menu.
2. Select 'Preferences'. Select 'Advanced' and then select 'Network'.
3. Under 'Connection', select 'Settings...'.
4. Select 'Manual Proxy Configuration'. Enter the hostname and port where your proxy program is running.
5. Save your changes by selecting 'OK' in the connection tab and then select 'Close' in the preferences tab.

Because Firefox defaults to using HTTP/1.1 and your proxy speaks HTTP/1.0, there are a couple of minor changes that need to be made to Firefox's configuration. Fortunately, Firefox is smart enough to know when it is connecting through a proxy, and has a few special configuration keys that can be used to tweak the browser's behavior.

1. Type 'about:config' in the title bar.
2. In the search/filter bar, type 'network.http.proxy'

3. You should see three keys: network.http.proxy.keepalive, network.http.proxy.pipelining, and network.http.proxy.version.
4. Set keepalive to false. Set version to 1.0. Make sure that pipelining is set to false.

To configure **Internet Explorer**, you should also do the following to make Internet Explorer work in a HTTP 1.0 compatible mode with your proxy:

1. Under Internet Options, select the 'Advanced' tab.
2. Scroll down to HTTP 1.1 Settings. Uncheck 'Use HTTP 1.1 through proxy connections'

Note: Different versions of web browsers may differ in the proxy configuration process.

Below are the grading criteria for this phase:

Label	Grading components	Comments
2-A	[1 pts] Complete file submission: all source files, Readme, & Makefile -- Readme file should list each team member's name and contributions if it is a group project -- Readme file should describe your code and the design decisions you made as well as strengths and weakness.	1. Missing Makefile & Readme: -0.5 pt 2. Makefile does not work: -0.5 pt
2-B	[8 pts]Basic Functionality: -- Your proxy works with a few "light" web pages (probably with not many embedded objects). If it passes, your proxy gets 5 points -- Your proxy works with some "heavy" web pages (with many embedded objects). If it passes, your proxy gets 3 more points.	Excellent: 7-8 pts (the web pages are well displayed for both light and heavy ones) Good: 4-6 pts (The web pages are well displayed for light ones, but not so well for heavy ones.) Bad: 0-3 pts (The web pages are not loaded correctly)
2-C	[1 pt] Well-written code: for readability, error checking, resource management, and comments, etc.	
2-D	[1 pt] Your proxy should run silently: any status messages or diagnostic output should be off by default.	
2-E	[4 pts] --- Multi-client proxy: whether your proxy works with multiple concurrent clients.	Excellent: 4 pts (no unexpected abortion or crashes, the web pages are loaded fast) Good: 2-3 pts (The web pages are displayed,

	--- Performance: how fast is a web page loaded? How many embedded objects in a web page are downloaded? --- Robustness: How robust is your proxy in presence of multiple clients? In-class demo!	the server crashes or aborts occasionally) Bad: 0-1 pt (The web pages are not loaded correctly, the server crashes consistently)
2-F	Either the program cannot be compiled Or fatal segmentation faults	Zero points for this project!

The submission deadline: **11:59PM, Sunday, November 5, 2017**

In cs1.seattleu.edu, make your files into a tar package, named **project2.tar**, which includes Makefile, all source files, and a README file. E.g.,

tar -cvf project2.tar proxy.h proxy.cpp Makefile README

Run the command to submit:

$ /home/fac/zhuy/class/CPSC5510/submit p22 project2.tar

V. Resources

- RFC 1945: https://www.w3.org/Protocols/rfc1945/rfc1945
- HTTP Made Really Easy - A Practical Guide to Writing Clients and Servers: http://www.jmarshall.com/easy/http/
- POSIX Pthreads Programming: https://computing.llnl.gov/tutorials/pthreads/
- POSIX Pthreads Programming: http://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html