Assignment 2: Building an HTTP Proxy

Overview

In this assignment, you will implement a simple web proxy that passes requests and data between a web client and a web server. This will give you a chance to get to know one of the most popular application protocols on the Internet- the Hypertext Transfer Protocol (HTTP)v. 1.0- and give you an introduction to the Berkeley sockets API. When you're done with the assignment, you should be able to configure your web browser to use your personal proxy server as a web proxy.

Introduction: The Hypertext Transfer Protocol

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 assignment we will be dealing only with version 1.0 of the HTTP protocol, defined in detail in RFC 1945. You should read through this RFC and refer back to it when deciding on the behavior of your proxy.

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.

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 an HTTP method (most often GET, but POST, PUT, and others are possible), 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 prompt, type:

telnet www.google.co.kr 80

This opens a TCP connection to the server at www.google.co.kr listening on port 80- the default HTTP port. You should see something like this:

Trying 216.58.197.99...
Connected to www.google.co.kr.
Escape character is '^]'.

type the following:
GET / HTTP/1.0

and hit enter twice. You should see something like the following:

HTTP/1.0 302 Found
Cache-Control: private
Content-Type: text/html; charset=UTF-8
Location: http://www.google.co.kr/?gfe_rd=cr&ei=vEr-VvG4McLC8geBq7zICg
Content-Length: 261
Date: Fri, 01 Apr 2016 10:17:32 GMT
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 google 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.

HTTP Proxies

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 an HTTP client (to the remote server) and an 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 other 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.

Links:

RFC 1945 The Hypertext Transfer Protocol, version 1.0
https://www.w3.org/Protocols/rfc1945/rfc1945

Assignment Details

The Basics

Your first task is to build a basic web proxy capable of accepting HTTP requests, making requests from remote servers, and returning data to a client.

This assignment can be completed in either C or C++. It should compile and run without errors from the EE323 Lab cluster, producing a binary called proxy that takes as its first argument a port to listen on. Don't use a hard-coded port number.

You shouldn't assume that your server will be running on a particular IP address, or that clients will be coming from a pre-determined IP.

The proxy should be able to receive requests from multiple clients. When multiple clients simultaneously try to send requests to the proxy, the proxy should take them one at a time (in any order). Note that you do not have to implement an event-driven or multi-threaded proxy. Serving one client at a time is enough, as you did in previous assignment.

Listening

When your proxy starts, the first thing that it will need to do is to 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.

Once a client has connected, the proxy should read data from the client and then check for a properly-formatted HTTP request. An invalid request from the client should be answered with an appropriate error code.

Parsing the URL

Once the proxy sees a valid HTTP request, it will need to parse the requested URL. The proxy needs at most three pieces of information: the requested host and port, and the requested path. See the URL (7) manual page for more info.

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 an HTTP request for the appropriate file. The proxy then sends the HTTP request that it received from the client to the remote server.

Returning Data to the Client

After the response from the remote server is received, the proxy should send the response message to the client via the appropriate socket. Once the transaction is complete, the proxy should close the connection.

Testing Your Proxy

Run your client with the following command:

./proxy <port>, where port is the port number that the proxy should listen on. 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.google.co.kr/ HTTP/1.0
Host: www.google.co.kr

If your proxy is working correctly, the headers and HTML of the Google homepage should be displayed on your terminal screen.

For a slightly more complex test, you can configure your web browser to use your proxy server as its web proxy. See the section below for details.

HTTP Host header

Some web servers require the "Host" HTTP header (required in HTTP 1.1 but some 1.0 servers may

complain if the request is missing the header), so make sure to add this header whenever you(/client) make a request. Your proxy should return an error message (400 Bad Request) when the request from a client does not have the "Host" header field.

Socket Programming

In order to build your proxy you will need to learn and become comfortable programming sockets. The Berkeley sockets library is the standard method of creating network systems on Unix. There are a number of functions that you will need to use for this assignment:

- Parsing addresses:
  - inet_addr
    - Convert a dotted quad IP address (such as 36.56.0.150) into a 32-bit address.
  - gethostbyname
    - Convert a hostname (such as argus.stanford.edu) into a 32-bit IP address.
  - getservbyname
    - Find the port number associated with a particular service, such as FTP.
- Setting up a connection:
  - socket
    - Get a descriptor to a socket of the given type
  - connect
    - Connect to a peer on a given socket
  - getsockname
    - Get the local address of a socket
- Creating a server socket:
  - bind
    - Assign an address to a socket
  - listen
    - Tell a socket to listen for incoming connections
  - accept

        - Accept an incoming connection
- Communicating over the connection:
    - read/write
        - Read and write data to a socket descriptor
    - htons, htonl / ntohs , ntohl
        - Convert between host and network byte orders (and vice versa) for 16 and 32-bit values

You can find the details of these functions in the Unix man pages and in Beej's guild to network programming. You may refer to both client and server codes for this assignment and the name and address conversion functions in the Beej's pdf.

Possible Extensions

Enable Page Compression

HTTP compression is a capability that can be built into web servers and web clients to make better use of available bandwidth, and provide greater transmission speeds between both.

HTTP data is compressed before it is sent from the server: compliant browsers will announce what methods are supported to the server before downloading the correct format: browsers that do not support compliant compression method will download uncompressed data. Compression as defined in RFC 2616 can be applied to the response data from the server (but not to the request data from the client).

You can implement some extensions for your proxy to reduce total bytes sent to the client: for example, page compression using gzip or deflate. When you implement this extension, you need to check how much total bytes sent to the client are reduced, and show the result as an output file. The output file include the total bytes sent to the client without compression, total bytes sent to the client with compression, and the reduction ratio for each response data.

~~Multiple Clients~~

~~Basically you can serve multiple clients by using fork(). Instead, you can implement extension by using the epoll framework.~~

Caching

Caching is one of the most common performance enhancements that web proxies implement. Caching takes advantage of the fact that most pages on the web don't change that often, and that any page that you visit once you (or someone else using the same proxy) are likely to visit again. A caching proxy server saves a copy of the files that it retrieves from remote servers. When another request comes in for the same resource, it returns the saved (or cached) copy instead of creating a new connection to a remote server. This saves a modest amount of time and CPU if the remote server is nearby and lightly trafficked, but can create more significant savings in the case of a more distant server or a remote server that is overloaded (it can also help reduce the load on heavily trafficked servers).

Caching introduces a few new complexities as well. First of all, a great deal of web content is dynamically generated, and as such shouldn't really be cached. Second, we need to decide how long to keep pages around in our cache. If the timeout is set too short, we negate most of the advantages of having a caching proxy. If the timeout is set too long, the client may end up looking at pages that are outdated or irrelevant.

There are a few steps to implementing caching behavior for your web proxy:

1. First, alter your proxy so that you can specify a timeout value (probably in seconds) on the command line.

2. Second, you'll need to alter how your proxy retrieves pages. It should now check to see if a page exists in the proxy before retrieving a page from a remote server. If there is a valid cached copy of the page, that should be presented to the client instead of creating a new server connection.

3. Finally, you will need to somehow implement cache expiration. The timing does not need to be exact (i.e. it's okay if a page is still in your cache after the timeout has expired, but it's not okay to serve a cached page after its timeout has expired), but you want to ensure that pages that are older than the user-set timeout are not served from the cache.

Althogh it is enough to support caching as suggested in the above, the HTTP caching semantics widely used in practice is slightly more complex. For the curious, please refer to the caching section in HTTP 1.1 specification (RFC 2068).

Readme

Use emacs to create a text file named readme that contains (not readme.txt, or README, or Readme, etc.)

Your name, student ID, and the assignment number.

A description of whatever help (if any) you received from others while doing the assignment, and the names of any individuals with whom you collaborated, as prescribed by the course Policy web page.

(Optionally) An indication of how much time you spent doing the assignment.

(Optionally) Your assessment of the assignment: Did it help you to learn? What did it help you to learn? Do you have any suggestions form improvement? Etc.

(Optionally) Any information that will help us to grade your work in the most favorable light. In particular, you should describe all known bugs.

Descriptions of your code should not be in the readme file. Instead they should be integrated into your code as comments. Your readme file should be a plain text file. Don't create your readme file using Microsoft Word, Hangul (HWP) or any other word processor. Your readme file should be written in English.

Grading

The deliverable is one tar file which contains "readme", "proxy.c" and "Makefile". 'make' should produce proxy from the source code.

Submission: 1 point

> Submission should include a correct set of files: proxy.c readme Makefile.
> Makefile should work without an error in lab machines

Basic functionality: 6 points

> Your proxy should work correctly with a small number of major web pages using the given script (Python).

Advanced functionality: 2 points

Your proxy should work correctly with a number of additional URLs and transactions that you will not know in advance. It will check the overall robustness of your proxy, and how your proxy handles certain corner cases.

Well written: 1 point

Good abstraction, error checking, readability and well commented code will get 1 point.

Extra credit:

Page compression, ~~Serve multiple clients (using epoll)~~, Caching (Details are in the precept material).