

HOMEWORK 3

Welcome to your last homework, The following are the following are the **rules** of submitting homework. Read them carefully before doing your first homework:

You have to submit the code/report in pdf, compiled from markdown. Any report written in word document or so will be discarded and score 0.

You have to submit your homework via Email.

Your Email subject shall be **Course Z6110X0002 Linux Programming+Homework #+{Your Name}+{Your ID}**.

The deadline of your homework will be **June 17th, 11:59PM**.

HTTP Proxy Problem

Acknowledge: This is originally a Princeton network programming homework. I modified here for your challenge.

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 a good understanding 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 this 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 a 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.
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 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.
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 Linux prompt, type:

```
telnet www.baidu.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 218.64.56.32...
Connected to www.ncu.edu.cn.
Escape character is '^['.
```

type the following:

```
GET http://www.ncu.edu.cn/ HTTP/1.0
```

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



```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<META content="南昌大学是一所“理工医渗透、产学研结合”的综合性大学，是江西省人民政府和教育部共建的国家“211工程”重点建设大学，是国家“中西部高校提升
<META name=keywords content="南昌大学">
...

```

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 our university homepage: 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 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 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 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.

Assignment Details

The Basics

Your first task is to build a basic web proxy capable of accepting HTTP requests, forwarding requests to remote (origin) servers, and returning response data to a client. The proxy does NOT need to handle concurrent requests, i.e. no need for threaded, forked, or event-based non-blocking operation. Rather, the proxy should handle requests sequentially. 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 (using g++), producing a binary called **proxy** that takes as its first argument a port to listen from. 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.

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

And a Host header, if the specified resource is a PATH: Host: All other headers just need to be properly formatted:

:

An invalid request from the client should be answered with an appropriate error code, i.e. "Bad Request" (400) or "Not Implemented" (501) for valid HTTP methods other than GET. See the note on network programming for more guidelines on how to handle real world clients and semi-valid requests.

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. You will need to parse the URL (absolute or relative) specified in the request line. Note that since a relative URL request, i.e. (/) does not include a hostname it must include the Host header in addition to the standard request line. Otherwise the proxy will not know where to retrieve the original resource. If the hostname, indicated in either the absolute URL or in the Host header, does not have a port specified, 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.ncu.edu.cn/ HTTP/1.0
```

or

```
GET / HTTP/1.0
Host: www.ncu.edu.cn
```

Send to remote server:

```
GET / HTTP/1.0
Host: www.ncu.edu.cn
(Additional client specified headers, if any...)
```

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. Once the transaction is complete, the proxy should close the connection to the client. Note: the proxy should terminate the connection to the remote server once the response has been fully received. For HTTP 1.0, the remote server will terminate the connection once the transaction is complete.

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.qq.com/ HTTP/1.0
```

If your proxy is working correctly, the headers and HTML of the Google homepage should be displayed on your terminal screen. Notice here that we request the absolute URL (`http://www.qq.com/`) instead of just the relative URL (`/`). Again, your proxy should support both of these formats from

the client, and only send the relative URL format along with a Host header. 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.

Debugging hints

Here are some debugging tips. If you are still having trouble, ask.

- There are defined buffer size and queue length constants in the scaffolding code. Use them. If they are not defined in a particular file, you don't need them. If you are not using one of them, either you have hard-coded a value, which is bad style, or you are very likely doing something wrong.
 - There are multiple ways to read and write from stdin/stdout in C, Python, and Go. Any method is acceptable as long as it does not read an unbounded amount into memory at once and does not modify the message.
 - If you are using buffered I/O to write to stdout, make sure to call flush or the end of a long message may not write.
 - Remember to close the socket at the end of the client program.
 - When testing, make sure you are using 127.0.0.1 as the server IP argument to the client and the same server port for both client and server programs.
 - Last hint, use your lab 6 code, wisely.
-