M1522.000800 System Programming, Fall 2017
Proxy Lab: HTTP and Proxy Servers
Assigned: Mon, Nov. 27, Due: Thurs, Dec. 7, 13:59

Introduction

An HTTP server (or web server) is a program that receives HTTP requests from a web browser, parses this information, then sends back the according HTTP response. This could be a simple HTML page, an image or something more complex. When we type a web address in the address bar of our browser we are sending an HTTP GET request to an HTTP server asking for the page with the given address. Examples of common HTTP servers in use today are Apache or Nginx.

Proxies are used for many purposes in the real world. Sometimes, proxies are used in firewalls, here the proxy is the only way for a web browser inside the firewall to contact a web server outside the firewall. A proxy may make modifications to external pages, for instance, by formatting them so they are viewable on a mobile phone. Proxies are also used as *anonymizers*. By stripping a request of all identifying information, a proxy can make the browser anonymous to the end server. Proxies can even be used to cache web objects, here they store a copy of a requested file when it is first requested, in future requests it can server the cached web object instead of going to the end server.

In this lab, you will be implementing two servers: a simple HTTP server and a simple proxy server.

- In the first part of the lab, you will write a simple sequential HTTP server that serves static content, e.g. HTML pages and images. This server waits repeatedly for HTTP GET requests, parses the requests, and, if the requested web object exists, sends it back to the client (browser) as part of an HTTP response. This part will give you some experience with network programming and the HTTP protocol.
- In the second part of the lab, you will implement a simple proxy server that will accept HTTP requests from the client (browser), forward these to the HTTP server, and then return the response of the HTTP server back to the client. Here the proxy is acting as a middleman between the two entities, anonymizing the client from the HTTP server.

Hand Out Instructions

You can clone the `proxylab` repository from GIT (http://git.csap.snu.ac.kr/sysprog17/proxylab.git). The two files you will be modifying are http.c (the HTTP server) and proxy.c (the Proxy Server). When you have completed the lab, you will commit all files containing your solution as well as a hard-copy of the report.

The `proxylab` directory contains the following code:

- `http.c`: This is the skeleton code of the HTTP server you will be modifying and handing in.
- `proxy.c`: This is the skeleton code of the proxy server you will be modifying and handing in.
- `csapp.c`: This is the file of the same name as the one described in the textbook. It contains error handling wrappers and helper functions such as the RIO (Robust I/O) package (textbook 10.5), `open_clientfd` and `open_listenfd` (textbook 11.4.8).
- `csapp.h`: This file contains a few manifest constants, type definitions, and prototypes for the functions in `csapp.c`.
- `Makefile`: Combines and links `proxy.c` and `csapp.c` into the executable `proxy`. Combines and links `http.c` and `csapp.c` into the executable `http`.
- `./pages`: This is a directory of containing webpages and images you can use to test your solution.

The `http.c` and `proxy.c` contain comments to guide you in your solution as well as some variable declarations that have been commented out. You can use these variables in your solution, you should not need any more to implement the requested functionality, but feel free to add any additional variables you feel are necessary. Your `http.c` and `proxy.c` files may call any function in the `csapp.c` file. However, since only the `http.c` and `proxy.c` files will be evaluated please **do NOT** modify the `csapp.c` file. If you want different versions of the functions found in `csapp.c`, write a new function in the `proxy.c` or `http.c` file.

Part I: Implementing an HTTP Server

In this part you will implement a sequential HTTP server. Your HTTP server should open a socket and listen for a connection request from a web browser. You can start your HTTP server by running:

```
devel@gentoo $ ./http 1234
```

You may use any port number p, where $1024 \leq p \leq 65536$, and where p is not currently being used by any other system or user services. See `/etc/services` for a list of the port numbers reserved by other services on your system. You can send an HTTP request to your HTTP server by typing the following address into the browser address bar:

```
http://127.0.0.1:1234
```

Instead of 127.0.0.1 you may type `localhost`, this is a loopback address pointing to your own computer.

Before implementing the full HTTP server it is useful to change some settings in your browser, for simplicity please just use Firefox for testing your solutions. Modern browsers often do automatic error correction on mis-typed address, or addresses that don't respond. This can be inconvenient when building your own web server. To disable this autocorrection type `about:config` into the address bar search for: `browser.fixup.alternate.enabled` and `network.captive-portal-service.enabled` and turn off both of these flags. This will stop the browser from autocorrecting your requests.

Your first task is to accept requests from the browser. Write a loop in the `main` function to accept connections and pass this to the `doit` function to handle them. Once this has been implemented, when sending a request to 127.0.0.1:1234 you should see the HTTP request being printed (probably multiple times) in your terminal. As your webserver doesn't respond to requests yet, the browser keeps sending requests as it believes there is a connection error. The next step is to parse the request sent by the browser. HTTP requests arrive in the form:

```
method URI version
Host: <host>:<port>
```

You are implementing the GET method, here you must find the file specified by the URI and send it back to the browser. You are also expected to handle 3 types of error:

```
501 - if the HTTP method is not GET
404 - if the requested page does not exist
403 - if the user requests a directory rather than a file
```

Firstly, you should implement these errors by inspecting the requested file on the local computer then passing the correct arguments to the `clienterror` function. Additionally a `parse_uri` function is provided to turn the requested URI to a path on your local machine e.g. ./pages/index.html. You can use this to find the requested file. Please do not modify the `clienterror` or `parse_uri` functions.

Once you can handle the aforementioned errors, if the file exists you should send the a requested file back to the web browser. This should be done by implementing the `serve_static` function.

Firstly you should build the response header, then send the file data (content). The format of the response should be:

```
version status-code status-message
Server: <serverName>
Content-length: <contentLength>
Content-type: <contentType>

<content>
```

Note that the two new lines between the `<contentType>` and `<content>` is represented as the string "\r\n\r\n". The format in which the HTTP protocol should be implemented is specified in a Request for Comments (RFC) document, you can check: https://tools.ietf.org/html/rfc2616 for a specification of the HTTP 1.1 protocol.

You can use the `get_filetype` function to determine the type of the requested file, currently only html files are supported, please extend this function to support images: jpgs, pngs, and gifs. Once the response has been sent the client connection should be closed. You should now see the webpage or image displayed in your browser.

A directory called pages has been included in the handout, this contains a number of pages with which you can test your web browser. However please feel free to test with your own html and image files. The defined of behaviour some of these files is:

```
http://127.0.0.1:1234/pages/index.html -> send index.html to the browser
http://127.0.0.1:1234/pages/image.jpg -> send image.jpg to the browser
http://127.0.0.1:1234/pages/abcd.html -> 404 error
http://127.0.0.1:1234/pages/ -> 403 error
```

You can test your webserver with some of the files in the pages folder along with some of your own files. You have now built an HTTP server!

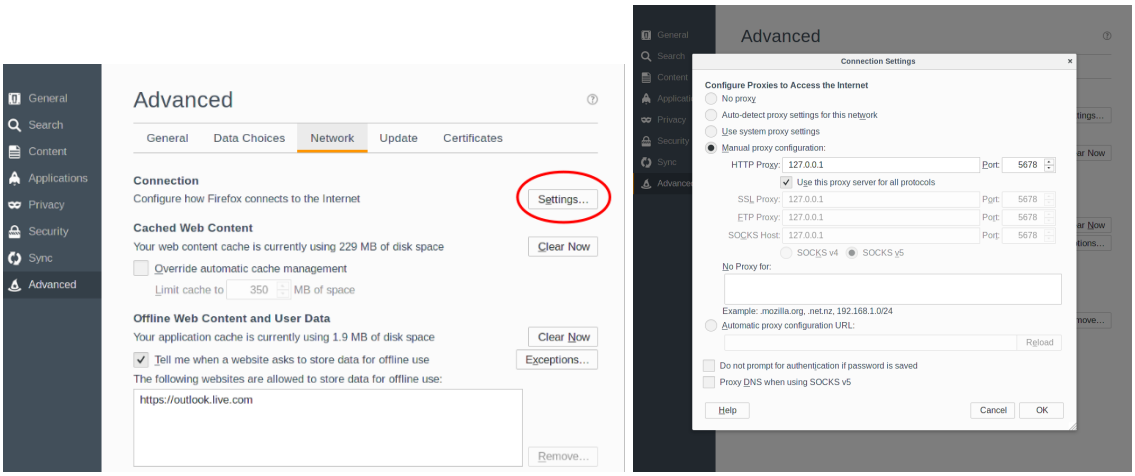
Part II: Implementing a Proxy Server

Now that you have built a web server that you can communicate with using your browser, you will implement a proxy server to communicate between the browser and this server. Since your proxy is a middleman between the browser and the HTTP server, it will act as a server to the browser and a client to the HTTP server. Thus you will get experience with both client and server programming.

You can run the proxy file as shown below, you will have to choose a different port to your HTTP server as you will be running both simultaneously.

```
devel@gentoo $ ./proxy 5678
```

In order to get the browser to connect to your proxy server rather than other web servers directly you will have to change some of your browser settings. In Firefox, go to `Preferences -> Advanced -> Network` and click on the settings button next to connection. Change your settings as follows:



Set the HTTP proxy as 127.0.0.1 (your local machine) and set the port to the port number that you will run your proxy on. Click the 'use this proxy server for all protocols' checkbox. Finally, remove '127.0.0.1' and 'localhost' from the 'No proxy for:' box. Now, Firefox will send all HTTP requests to your proxy server instead of connecting directly to the outside world, to test this you can try to visit any web page with Firefox, you should no longer be able to connect to them.

You should now implement your proxy server to accept connections from your browser, forward these to your HTTP server, wait for a response, and then send this response back to the browser. You should start both your HTTP and your proxy server on different ports. You should then make a request to the HTTP server in your browser as in section 1, your proxy will intercept this request. Accepting connections will work in the same way as with the HTTP server.

You should send a request to your HTTP server as before (http://127.0.0.1:1234/pages/index.html). The browser will send an HTTP request in the same format as before. Your first job is to parse this to determine the end destination of the request, you should then open a connection to that destination and then forward the request to there. For our purposes, forwarding just the first line of the request to the end server should be enough.

After forwarding the request from the client to the requested host, we should read the HTTP response sent back by the server. This is the response sent back from our HTTP server. We should first read the response headers, we know these end in an empty line to terminate them. We should parse the Content-length value from this response header and use it to read the correct amount of data from the response body. We should then send both the response header and the content back to the client (browser).

After implementing the proxy you should be able to use your HTTP server in the same way as in section 1, i.e. by making requests with your browser configured to point at the proxy. This has the effect of anonymizing the client from the end server, i.e. the HTTP server is not aware of your browser. You are currently running these locally on your machine with different ports, but it should be possible to run your HTTP server on one machine, your proxy on another and your browser on yet another, and this would still work.

Evaluation

- (30 Points) HTTP server. Your HTTP server should correctly accept connections from the browser, if the path is valid it should serve static html or image content to the browser. Based on the request it should also do some basic error handling (404, 403, 501 errors).
- (40 Points) Proxy server. Your proxy server should correctly accept connections from the browser and forward this to the correct server. It should then read the response from the end server and send this back to the browser.
- (10 points) Code style. You can get up to ten points for well written and commented code. Your code should begin with a short block describing how your proxy works. Also, each function should have a comment block describing what that functions does. Make sure you free any memory you allocate dynamically.
- (20 points) Report. In your report you should describe your implementation. Also, you should add a section regarding difficulties and suggestions for this lab, similar to the previous lab (shell lab).

Hand-in Instructions

To submit your solution simply commit, and push the changes to your remote repository.

```
devel@gentoo  /proxylab $ git add <list of modified files>
devel@gentoo  /proxylab $ git commit -m <commit message>
devel@gentoo  /proxylab $ git push origin master
```

We wont consider any commit which timestamp is greater than the labs deadline. If for some reason you cannot push to the git server and there are 5 minutes before the deadline then, and only then, create a tar file of your submission and send it via email to the TAs.

Hints

- You can use the RIO (Robust I/O) package (textbook 11.4) performing I/O on sockets.
- You will have to perform a lot of string manipulation in this lab, the `sprintf` and `sscanf` functions will be very helpful to you. When using `sscanf` you can use the string `"%[^:]"` to read in a string up until the `:` character.
- You can use the `stat` system call to find out information about a file based on it's file path.