

## Building an HTTP Proxy

In this project, you will implement a simple web proxy that passes requests, responses, and data between a web client and a web server. The purpose of this project is to (i) help you get to know one of the most popular application protocols on the Internet - the Hypertext Transfer Protocol (HTTP) v1.0 and (ii) help you learn how a proxy works. By the end of this project, you will be able to configure your web browser to use your personal proxy server as a web proxy.

### Important Notice

The deadline for this project is **March 30th, Tuesday, 11:55 PM**. The submission link will be automatically closed after the deadline. Note that any kinds of late submissions are not allowed.

Submission Link: <https://bit.ly/ee323-proj2-2021-submit>

### Background: The Hypertext Transfer Protocol (HTTP)

HTTP is the protocol used for communication on the web. It is the protocol that defines how your web browser *requests* resources from a web server and how the server *responds* (Figure 1)<sup>1</sup>. For simplicity, in this assignment, we will be dealing only with version 1.0 of the HTTP protocol, defined in detail in [RFC 1945](#). When deciding on the behavior of your proxy, we recommend reading through this RFC<sup>2</sup>.



**Figure 1. The Hypertext Transfer Protocol**

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 as described below.

---

<sup>1</sup> Source: <https://github.com/VanHakobyan/HTTP-Protocol-Manipulation>

<sup>2</sup> Request For Comments: a document that describes the open standards, protocols, and technologies of the Internet TCP/IP.

Format	Request message	Response message
An initial line ( a request or response line)	GET /nmsl/ee323.txt HTTP/1.0	HTTP/1.0 200 OK
Zero or more header lines	Accept: text/* Accept-Language: en	Content-type: text/plain Content-length: 12
A blank line (CRLF)		
An optional message body		Hello World!

For most common HTTP transactions, the protocol boils down to a relatively simple series of steps:

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* (GET, POST, PUT, ...), a *request URI*<sup>3</sup> (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.<sup>4</sup>
3. The server sends a response message, with its initial line consisting of a **status line**, which indicates 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 the request was completed successfully), and a *reason phrase* (an English-language message providing a description of the status code). For example, a popular combination of a response status code and a reason phrase is "404: Not Found". Like 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 (a blank line), the message body contains the data requested by the client in the event of a successful request.<sup>5</sup>
4. Once the server has returned the response to the client, it closes the connection.

#### [Hands-on Experience]

It's fairly easy to see this process in action without using a web browser.

1. From a Unix prompt or a Linux terminal, type "telnet www.google.com 80". It opens a TCP connection to the server at www.google.com listening on port 80 - the default HTTP port. (Figure 2)

<sup>3</sup> Uniform Resource Identifier. E.g., URL is a subset of URI. ([RFC3986](#))

<sup>4</sup> For more information, refer to 5.1-5.2, 8.1-8.3, 10, D.1 of [RFC 1945](#).

<sup>5</sup> For more information, refer to 6.1-6.2, 9.1-9.5, 10 of [RFC 1945](#).

```
(base) yewon@chris:~$ telnet www.google.com 80
Trying 172.217.26.132...
Connected to www.google.com.
Escape character is '^]'.
|
```

Figure 2. Create a connection in a UNIX prompt / Linux Terminal

2. Issue a request by typing "GET / HTTP/1.0". (Figure 3)

```
(base) yewon@chris:~$ telnet www.google.com 80
Trying 172.217.26.132...
Connected to www.google.com.
Escape character is '^]'.
GET / HTTP/1.0|
```

Figure 3. Issue a request.

3. Hit the enter twice and see what is happening. At the beginning of the message, you should see something like the following (Figure 4):

```
(base) yewon@chris:~$ telnet www.google.com 80
Trying 172.217.25.4...
Connected to www.google.com.
Escape character is '^]'.
GET / HTTP/1.0

HTTP/1.0 200 OK
Date: Fri, 26 Feb 2021 05:44:08 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
P3P: CP="This is not a P3P policy! See g.co/p3phelp for more info."
Server: gws
X-XSS-Protection: 0
X-Frame-Options: SAMEORIGIN
Set-Cookie: 1P_JAR=2021-02-26-05; expires=Sun, 28-Mar-2021 05:44:08 GMT; path=/; domain=.google.com; Secure
Set-Cookie: NID=218=Zqr4zugbEPn8NN8hXIdgSo1GB3vhpw01bsNky_dfBLaDr0YDEqEpvdZ56NZ1ecrqmUDLX-v83BmIBFRg7YqhQPvwDqtMCocZFgJG
2WPn5aUI5rk0ZHoasCpBilmkf9UYuE8Vmjxp13qjXkyrISxTRL-EjqnhAeT4-J87FVbN9M; expires=Sat, 28-Aug-2021 05:44:08 GMT; path=/;
domain=.google.com; HttpOnly
Accept-Ranges: none
Vary: Accept-Encoding

<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="ko"><head><meta content="text/html; charset
```

Figure 4: Receive a response from the server. More HTML follows after the message.

What you are seeing is exactly what your web browser sees when it goes to the Google 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.

#### [HTTP Proxies]

Ordinarily, HTTP is a client-server protocol. The client (usually your web browser) communicates directly with the server (the web server software) (Figure 5). 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 (Figure 6<sup>6</sup>). In the simplest case, instead of sending requests directly to the server, the client sends all its requests to the proxy. The proxy

<sup>6</sup> Source of figure 5 and 6: previous ee323 precept slides

then opens a connection to the server and passes on the client's request. The proxy receives a reply from the server, and then sends that reply back to the client. That is, the proxy essentially acts like both an HTTP client (to the remote server) and an HTTP server (to the initial client).



**Figure 5. Communication between the client and the server (*without proxy*)**



**Figure 6. Communication *with proxy***

There are several advantages of using a proxy:

1. **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.
2. **Content Filtering and Transformation.** The proxy can inspect the requested URL and selectively block access to certain domains, reformat web pages (for instance, 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.
3. **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 (e.g., an entire business or university), it becomes much harder to link a particular HTTP transaction to a single computer or individual.

## What Should I Do?

Your job is to build a basic web proxy capable of:

1. Accepting HTTP requests from clients,
2. Sending requests to / Receiving responses from remote servers, and
3. Returning data to a client.

Additionally, your proxy should:

- A. Serve multiple clients,
- B. Work in real web browsers, and
- C. Support HTTP redirection functionality.

You should complete this project in **ANSI C**. It should be compiled and run without errors from the **Haedong lounge machine**, producing a **binary called proxy** that takes as its first argument a port to listen on. (Command line for example: `./proxy 5678`) You shouldn't assume that your server will be running on a particular IP address, or that clients will be coming from a predetermined IP.

Don't worry, we will guide you step by step.

### Step 0. Socket Programming

As you did in the first project, this project requires a socket program. The Berkeley sockets library is the standard method for creating network systems on Unix. We introduce you to a number of functions that you will need to use for this assignment below.

Category	Function Name	Description
Parsing addresses	inet_addr	Convert a dotted quad IP address (e.g., 36.56.0.150) into a 32-bit address.
	gethostbyname	Convert a hostname (e.g., 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.

Step 1-1. Starting Your Proxy: Listen for Incoming Connections

When your proxy starts, establish a socket connection 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**.

- Your proxy should return an error message (400 Bad Request) when the request from a client does not have the “Host” header field. This is because some web servers require the “Host” HTTP header<sup>7</sup>, so it is better to add this header whenever making a request.
- You should handle invalid HTTP requests by returning a 400 Bad Request error message. (e.g., methods other than “GET”, different HTTP versions, etc.)
- You should also handle invalid “Host” header fields (i. Invalid hostname, ii. Invalid IP address). In this case, return an HTTP error message (503 Service Unavailable).

Step 1-2. Parse 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: (i) the requested **host**, (ii) the requested **port**, and (iii) the requested **path**. Following functions would be helpful for this task.

strtok()	Breaks string into a series of tokens
strcmp() / strncmp()	Compares two strings
strlen()	Calculates the length of a string
strchr()	Searches for the first occurrence of a character in a string
strncpy() / strcpy() / memcpy()	Copies a string

Step 2. Get Data from the Remote Server

Once the proxy has parsed the URL, it can make a **connection** to the requested host. When making a connection, use the **appropriate remote port**, or the **default of 80** if none is specified. The proxy then sends the **HTTP request** that it received from the client to the remote server.

Step 3. Transfer Response of the Server 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**. **Close** the connection once the transaction is complete.

Testing If Your Proxy Works Properly

If you finish step 0 ~ 3, you have successfully built a proxy. You may want to test if your proxy is working properly. To do this, first, run your client with the following command:

---

<sup>7</sup> Required in HTTP 1.1, but some 1.0 servers may complain if the request is missing the header.

`./proxy <port>`, where `port` is the port number the proxy should listen on (Figure 7). Then, open the second terminal, and try requesting a page using **telnet** (Figure 8).

```
(base) yewon@s00ae:~$ ./proxy 5678
E.g., run proxy with port number 5678
```

Figure 7. Run your proxy with command format

`./proxy <port>`

[about:blank#blocked](#)

```
(base) yewon@s00ae:~$ telnet localhost 5678
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
GET http://www.google.com/ HTTP/1.0
Host: www.google.com
```

Run telnet with the same port number your proxy is listening on

Type these. Make sure that your proxy always need the "Host" header field.

Figure 8. Try requesting a page using telnet.

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

#### A. Serving Multiple Clients

Your proxy should be able to receive requests from multiple clients. When multiple clients simultaneously try to send requests to the proxy, do not block the incoming requests. Handle them simultaneously, as you did in project 1.

#### B. Working in Real Web Browsers

Here, you will configure a Firefox web browser to use a proxy. First, turn on your proxy with a port number on the Haedong Lounge machine or your local machine (Figure 7). Then set your web browser to use your proxy with the appropriate port number that your proxy is listening on. To configure the web browser, follow the instructions below. (Testing environment: version 86.0)

1. Select Options from the menu.
2. Click the "Settings" button in the "Network Settings" section.
3. Select 'Manual Proxy Configuration' from the options available. In the boxes, enter the host IP address and port where the proxy program is running.

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.

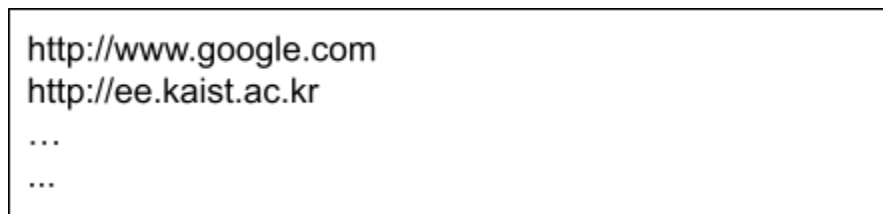
1. Type 'about:config' in the address bar.
2. Search for "network.http.proxy.version" and set it to 1.0.

If you write a **single-threaded** proxy server, you will probably see some problems when you use your proxy with a standard web browser. Because a web browser like Firefox issues multiple HTTP requests for each URL you request (for instance, to download images and other

embedded content), a single-threaded proxy will likely miss some requests, resulting in missing images or other minor errors. **That's OK.** You are more than welcome to use threading (or events) but that is not required in this assignment. As long as your proxy works correctly for a simple HTML document (e.g., <http://example.com/> ) and follows the RFC, you can still receive all the points for this assignment.

### C. Supporting HTTP Redirection Functionality

Your proxy should support redirection. Once the proxy gets blacklist pages as a standard input (for example, `./proxy 5678 < blacklist.txt`), it should block the request for those pages and redirect to the warning site (<http://warning.or.kr>). Thus, the proxy should not send requests to the pages on the blacklist file. Instead, it will send a request to the warning site, and return the content of the warning site to the client. Through this task, you will have a better understanding of the mechanism of Internet censorship. The blacklist text file would look like below (Figure 9). The presented websites are just examples, but the format of the file will follow it.



```
http://www.google.com
http://ee.kaist.ac.kr
...
...
```

**Figure 9. Example of blacklist.txt**

### What are the grading criteria?

This project is worth 4.5% of your total course grade. As a default, your assignment should create a binary named `proxy` that can be compiled and run on the EE323 lab cluster (Haedong Lounge server). The first argument should be the port your proxy will listen on. For example, `./proxy 3128` should listen on port 3128. Also, your proxy should **run silently** - any status message or diagnostic output should be off by default. You should complete the assignment in ANSI C. Assuming these, your script will be evaluated via the following criteria:

#### 1. (20%) Basic Functionality Test

We will first check if your proxy works correctly with a small number of major web pages with this [testing script](#). (A total of 4 test cases)

- After downloading the script, give executable permission to the script using `chmod`:  
`$ chmod +x proxy_tester.py`
- Run the script with the following format: `./proxy_tester.py ./proxy <port>`

#### 2. (10%) Firefox Test

Your proxy should work with Firefox. We will use <http://www.example.com/> for the testing.

#### 3. (20%) Blacklist Test (Hidden test cases)



We will check the HTTP redirection functionality. After inserting the blacklist text file as standard input to your proxy, you will earn a grade if your proxy properly blocks the request for the blacklist pages and redirects to a warning site.

4. (40%) Error Handling Test (Hidden test cases)

We will check a number of additional URLs and transactions that you will not know in advance.

5. (10%) Multiple Client Support (Hidden test cases)

We will check if your proxy server supports multiple clients.

### Useful Resources

- [Beej's Guide to Network Programming](#)
- [HTTP Make Really Easy - A Practical Guide to Writing Clients and Servers](#)

### How to submit?

Submit a zip file containing the following:

- All of the source code for your proxy
- A Makefile that builds your proxy ('\$ make all' should generate an executable file 'proxy'.)
- A report.pdf describing your code and the design decisions that you made.

Compress the above items into one zip file. Name it as

**{studentID}\_{name(in English)}\_project2.zip** (e.g., 20211234\_TaekyungLee\_project2.zip) and submit it to the link below.

Submission link: <https://bit.ly/ee323-proj2-2021-submit>

Please make sure that **we do not accept late submissions**. Start early, and submit early.

### Where can I ask questions?

We encourage students to actively participate in the course. Please share your experiences and questions in the course [Campuswire](#) group. However, if you think your question is not appropriate for sharing for some reason (such as it contains private information), send it to [ee323@nmsl.kaist.ac.kr](mailto:ee323@nmsl.kaist.ac.kr). However, TAs may share your question on Campuswire.

### Q&A

#### [Error Message Handling Issues]

Q. Is "HTTP/1.0" necessary? (If this message is omitted, should I return an error message?)

A. Yes.

Q. Regarding the method, the only functionality I should develop is “GET”? Nothing else?

A. Yes.

Q. If the host on the GET message and the host on the Host message is different, should I return an error message? For example:

“GET <http://google.co.kr:4567/> HTTP/1.0

Host: <http://example.com>”

A. In that case, you should return a 400 Bad Request error message.

Q. What would be the format for error messages? Is “HTTP/1.0 400 Bad Request” enough, or do we have to print anything else?

A. You just have to print “HTTP/1.0 400 Bad Request” for the empty host header field in this case.

Q. Sometimes some servers send a 400 Bad Request error. Should I send my own bad request error message or the server’s message? And if there is not enough memory to allocate, should I print out 500 Internal Server Error?

A. Your proxy should check the validity of incoming requests and send a 400 Bad Request response when the requests are invalid. It should not make a connection with the server when it receives an invalid request.

### [Firefox Test]

Q. I noticed I can receive a response from <http://example.com> even though I am not running the proxy but without changing the configuration of Firefox. Is it normal?

A. It is probably due to a cache problem. Try to remove the web cache from the Firefox setting and try again.

Q. Will the blacklist redirection functionality be tested on the Firefox browser?

A. No.

Q. “warning.or.kr” page doesn’t appear in the Firefox browser.

A. We do not test the blacklist functionality with Firefox. We will only test with telnet.

### [Redirection (Blacklist)]

Q. When I send a request like “GET / HTTP/1.0\r\nHost:warning.or.kr\r\n” to the warning.or.kr server, the server sometimes returns the right value and 200 response. However, sometimes, it responds with a 302 found message. How can I solve this problem?

A. 302 Found response is not a wrong response. You can see the same response with “telnet [www.warning.or.kr](http://www.warning.or.kr) 80”

### [General]

Q. Number of header lines: Can I assume there will not be more than a fixed header line (10 for example)?

A. In this project, only the Host header is necessary.

Q. What does “closing the connection” in lab slides #14 mean? (Step 3: Transfer Response to the Client) Does it mean the proxy server should be shut down, or just the connection between the client and server should be closed?

A. The latter one is right.

Q. Can I use server.c and client.c (that I implemented in project 1) as the base code for this project?

A. Yes.

Q. Does the test case always include “http://”? Or could there be “[www.google.com](http://www.google.com)”?

A. It always includes “http://”. Thus “<http://www.google.com>” is the right format.

Q. How to stop the proxy server?

A. We do not specify the conditions for stopping the proxy server. You may stop with ctrl+c.