CSE351 Computer Networks

Fall, 2025, Programming Assignment #2
Due: October 12, 2025 (Sun), 23:59 KST

Instructor: Taesik Gong (taesik.gong@unist.ac.kr), TA: Seyeong Park (seyeongpark@unist.ac.kr)

Submission Instructions. You shall submit this assignment as a single ZIP file named PA2_StudentNumber_ Name.zip (e.g., PA2_20251234_HongGildong.zip) on the Blackboard. The ZIP must include proxy.c, a Makefile (running $ make must build the executables proxy), and report.pdf. You may include additional .c/.h files if needed, but they must be built by the Makefile and briefly explained in the report. No skeleton files are provided—you must implement everything from scratch.

Collaboration Policy. You are welcome to discuss the homework with your classmates to understand the concepts, and you may also use code snippets from the provided guide links. However, if your report or code is found to be unreasonably identical to someone else's work, it will be considered plagiarism and will result in no points for all involved.

Project Overview. 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.

What to Do. Your job is to build a basic web proxy capable of (i) accepting HTTP requests from clients, (ii) sending requests to / Receiving responses from remote servers, and (iii) returning data to a client. Additionally, your proxy should (i) serve multiple clients, (ii) work in real web browsers, and (iii) Support HTTP caching functionality.

[Description 1] 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)[1]. 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[2].
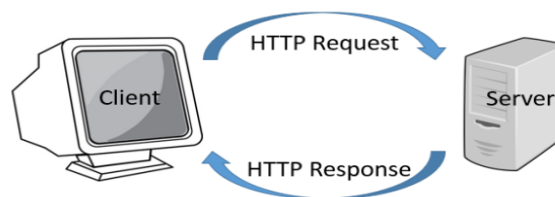

Figure 1. The Hypertext Transfer Protocol

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.

---

[1] Source: https://github.com/VanHakobyan/HTTP-Protocol-Manipulation
[2] 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 /uai/cse351.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*[3] (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.[4]

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.[5]

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)

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

---

[3] Uniform Resource Identifier. E.g., URL is a subset of URI. (RFC3986)
[4] For more information, refer to 5.1-5.2, 8.1-8.3, 10, D.1 of RFC 1945.
[5] For more information, refer to 6.1-6.2, 9.1-9.5, 10 of RFC 1945.

2. Issue a request by typing "GET / HTTP/1.0" and "Host: www.google.com". (Figure 3)

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):

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.

[Description 2] 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[6]). 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 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.

---

[6] Source of figure 5 and 6: previous ee323 precept slides

[Problem] Build a basic web proxy

Your proxy is capable of:

(a) Accepting HTTP requests from clients,
(b) Sending requests to / Receiving responses from remote servers, and
(c) Returning data to a client.

Additionally, your proxy should:

(a) Serve multiple clients,
(b) Work in real web browsers, and
(c) Support HTTP caching functionality.

You should complete this project in **C**. It should be compiled and run without errors inside the container built from the provided Dockerfile, 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**.

You should return formatted error message (400 Bad Request) in the following cases:

- When a request from a client does NOT have a "host" header field. This is because some web servers require the "host" http header[7], so it is better to add this header whenever making a request.

- When HTTP methods other than "GET" are used while requesting data. This does not mean that HTTP version 1.0 cannot use other methods. We simplified the assignment for your convenience.

- When different HTTP versions (other than v1.0) are used while requesting data.

- When invalid "host" header fields are passed (hint: use gethostbyname() to check if the host name is invalid).

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.

---

[7] Required in HTTP 1.1, but some 1.0 servers may complain if the request is missing the header.

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:

./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).

Figure 7. Run your proxy with command

format `./proxy <port>`

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. 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.

(easier instruction with visual aids is in the "FirefoxSettingGuide.pptx"!)

1. Select "Setting" from the top-right menu.

2. Navigate to the "Network Settings" section in the "General" tab.

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 Cache Functionality

Your proxy should support caching. When a client sends a request to the proxy, the proxy forwards it to the original server and receives the HTTP response. The proxy should then check the response headers, such as "Expires", "Pragma", or "Cache-Control", to decide whether to cache the content. Although Cache-Control was formally introduced in HTTP/1.1, it is also widely used in HTTP/1.0 responses, so your proxy should handle it.

If the response is cached, subsequent requests for the same resource should be served from the local cache (in memory or on disk) instead of requesting it from the original server. Through this task, you will have a better understanding of the mechanism of HTTP caching and how web performance can be improved by avoiding unnecessary network requests.

The HTTP header might look like the examples below (see Figure 9, 10). Actually, Cache-Control has many options, but in this assignment we only consider a subset of caching directives: max-age, private, and public.

- If a response has *Cache-Control: max-age=<seconds>*, your proxy should store the response and reuse it for the given number of seconds. During this period, any request for the same resource should be served directly from the cache. After <max-age> seconds, the cached response is considered expired. When the next request comes from the client, the proxy should forward the request to the original server again and refresh the cache.
- *Cache-Control: private* indicates the response is intended for a single user and should be stored only in a browser's private cache, not in a shared proxy.
- *Cache-Control: public* Indicates the response may be cached by any cache, including proxy servers.

In this assignment, your proxy should only store responses when they are marked as **public (or when max-age exists without private).**
- You can implement the cache using either memory or files; both approaches are allowed. (if your code uses *fork* for multi-client supporting, using files for the cache might be easier, or you would need to use shared memory.)

Figure 9. Cache-Control of www.google.com

Figure 10. Cache-Control of www.example.com

[Appendix] Grading criteria

As a default, your assignment should create a binary named proxy. The first argument should be the port your proxy will listen on. For example, ./proxy 3128 should listen on port 3128. Any status message or diagnostic output should be off by default. (It should not print anything to the standard output.) You should complete the assignment in C. Assuming these, your script will be evaluated via the following criteria:

1. <u>(60%) Proxy Tester</u>

   We will check if your proxy works correctly with a small number of web pages with this testing program. We will provide a <u>binary executable</u> for you so that you can test your proxy and check your points.

   - Before testing your code with the provided binary executable, make sure that you already compiled your proxy.c and have your executable file (proxy).

   - The provided tester file is a **.zip file** that contains a program and associated files. You don't need to know about the associated files, but these should be present in the same directory as the program when you test your code with the program.

   - (1) After downloading the script, send the zip file to the testing container.

   - 2) Unzip the provided proxy_tester.zip folder

      - $ unzip proxy_tester.zip

      - 'proxy_tester' is the tester file you'll be using.

   - (3) Give executable permission to the file:

      - $ chmod +x ~/proxy_tester/proxy_tester

   - (4) Run the executable with the following format:

      - $ ./proxy_tester/proxy_tester ./proxy <port>

2. <u>(10%) Firefox Test</u>

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

3. <u>(20%) Cache Test</u>

   We will check the HTTP caching functionality. For a server that allows caching, if you request the same URL multiple times and the response comes back without sending a request to the external server, you will earn points.

4. <u>(10%) Multiple Client Support</u>

   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' 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_GildongHong_project2.zip)

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

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 field is different, should I return an error message? For example:

"GET http://www.google.co.kr:4567/ HTTP/1.0

  Host: www.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 the formatted error message (including CRLF) "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?

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.

Q. When testing my program with the provided proxy_tester, it usually gives a full score but sometimes it fails. For example, below is the message I get:

Testing: http://info.in2p3.fr/

Proxy: date: Thu, 11 Sep 2025 06:23:39 GMT

Direct: date: Thu, 11 Sep 2025 06:23:40 GMT

http://info.in2p3.fr/: [FAILED] (0/5 points)

Is it an implementation problem which I have to fix?

A. The Expires header is a response header, so it would be the one the remote server is returning. The millisecond difference would be due to the time that proxy spends for validating the client message and sending it to the remote server, while direct connection doesn't need this. Also, when grading, we will test your code up to 10 times when we observe such cases and give you the highest score. Including the symptom in your report.pdf would be fine.

[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. (Settings -> Cookies and Site Data ->Clear Data -> Clear)

Q. Will the cache functionality be tested on the Firefox browser?

A. No.

[General]

Q. Should the connection between the client and server be closed after the transaction is completed?

A. Yes.

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"?

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.