CS 4480: Computer Networks

Programming Assignment 1
HTTP Web Proxy Server
Three parts: PA1–A, PA1–B, and PA1–Final
See Canvas for due dates

The goal of this programming assignment is to develop a simple caching and filtering HTTP web proxy.[1]

1 Background: 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. Upon receiving a client's request, the proxy opens a connection to the server and passes on the 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 as both an HTTP client (to the remote server) and an HTTP server (to the initial client).

Why use a proxy? There are a few reasons:

- **Performance.** By saving copies of the objects[2] that it fetches, a proxy can reduce the need to create connections to remote servers. This can reduce the overall delay involved in retrieving an object, particularly if a server is remote or under heavy load.
- **Content filtering and transformation.** While in the simplest case the proxy merely fetches an object without inspecting it, there is nothing that says that a proxy is limited to blindly fetching and serving objects. 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.** Web servers generally log all incoming requests. 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 URL. If a client does not want to have this personally identifiable information recorded, routing HTTP requests through a

[1] Credit: This programming assignment was derived from similar assignments at Stanford (Nick McKeown) and Princeton (Jennifer Rexford - COS-461) as described in Kurose and Ross.

[2] An *object* is an entity that is accessible at a URL: e.g., an HTML document, an image or video, a JavaScript file, a data file, or something else.

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 many 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.

2 Assignment Details

Your task is to implement a HTTP/1.0 proxy with basic object-caching and domain-blocking features. The proxy should be capable of serving *multiple concurrent requests*. Your proxy only needs to support the HTTP GET method.

It is **strongly recommended** that you approach the assignment as a multi-step process. First develop a proxy that is capable of receiving a request from a client, passing that through to the origin (real) server, and then passing the server's response to the client. Second, extend that basic capability so that your proxy is capable of serving multiple clients concurrently. Finally, enhance the multi-client proxy by adding the caching and blocking features.

2.1 PA1–A: Basic Proxy

Your first task is to build a web proxy capable of accepting HTTP requests, forwarding requests to remote (origin) servers, and returning response data to a client. You will only be responsible for implementing the GET method. *All other request methods received by the proxy should elicit a "501 Not Implemented" error. (See RFC 1945 Section 9.5 — Server Error 5xx.)*

You should not assume that the origin server will be running on a particular IP address, or use a particular TCP port number, or that clients will be coming from a predetermined IP. In other words, your proxy should correctly handle port numbers being specified in the URL.

Listening. When your proxy starts, the first thing that it will need to do is establish a socket that it can use to listen for incoming connections. Your proxy should listen on a *port specified on the command line* (see instructions in Section 3) 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, you should ensure that the proxy receives a request that contains a valid request line:

```
<METHOD> <URL> <HTTP VERSION>
```

All other headers just need to be properly formatted:

```
<HEADER NAME>: <HEADER VALUE>
```

In this assignment, client requests to the proxy must be in their absolute URI form (see RFC 1945, Section 5.1.2)—as a browser will send if properly configured to explicitly use a proxy.[3]

An invalid request from the client should be answered with an appropriate error code, e.g., "400 Bad Request" for malformed requests or "501 Not Implemented" for valid HTTP methods other than GET. Similarly, if headers are not properly formatted for parsing, your client should also generate a 400 response.

[3]This is in contrast to a transparent on-path proxy that some ISPs deploy, unbeknownst to their users.

Getting data from the remote server. Once the proxy has parsed the URL, it can make a connection to the origin server *(using the appropriate remote port, or the default of 80 if none is specified)* and send the HTTP request for the appropriate object. The proxy should always send the request in the relative URL + Host header format regardless of how the request was received from the client.

For example, accept from client:

```
GET http://www.google.com/ HTTP/1.0
```

Send to the origin server:

```
GET / HTTP/1.0
Host: www.google.com
Connection: close
(Additional client-specified headers, if any.)
```

Your proxy should *always* send an HTTP/1.0 "Connection: close" header to the server, so that it will close the connection after its response is fully transmitted, as opposed to keeping open a persistent connection. So while your proxy should pass the client headers it receives on to the server, it should make sure you replace any "Connection" header received from the client with one specifying "close" as shown.

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.

Testing your proxy. Assuming your proxy listens on port number `<port>` and is bound to the `localhost` interface, then as a basic test, try requesting an HTML 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.flux.utah.edu/cs4480/simple.html HTTP/1.0
```

Be sure to enter a blank line after the GET line.[4] If your proxy is working correctly, the headers and body of an "200 OK" HTTP response—containing a real, simple HTML document—should be displayed in your terminal window.

A good sanity check of proxy behavior would be to compare the HTTP response (headers and body) obtained via your proxy with the response you get from a direct `telnet` connection to the origin server. For example:

```
# telnet www.flux.utah.edu 80
Trying 155.98.63.201...
Connected to web.flux.utah.edu.
Escape character is '^]'.
GET /cs4480/simple.html HTTP/1.0
Host: www.flux.utah.edu
Connection: close
```

[4]If you've forgotten why this is important, review Kurose and Ross, Section 2.2.3.

Again, be sure to enter a blank line after the "Connection" line. See Section 5 for more advice about testing your proxy.

2.2 PA1–B: Multi-client Proxy

Once you have the basic proxy working, enhance it to handle multiple, concurrent client requests. (A proxy that can handle only a single request at a time is quite limited!) There are multiple possible approaches to implementing this feature. For example, you might use Python's `multiprocessing` package, or you might use Python's `threading` module.

Listening. Like the basic proxy, when the multi-client proxy starts, the first thing it will need to do is establish a socket that it uses to listen for incoming connections, on a port specified on the command line (Section 3). Unlike the basic proxy, as each new client establishes a connection, the multi-client proxy should start a new process or thread to handle the incoming request. Individual client requests should otherwise be handled as described in Section 2.1: the proxy reads data from the client, checks the validity of the HTTP request, and so on.

You can assume a reasonable limit on the number of processes/threads that your proxy will need to create at any one time, e.g., 100.

Testing your multi-client proxy. You should test the multi-client functionality of your proxy by requesting an object using `telnet` concurrently from two different shells. If you request a very short/simple object, it might be difficult to show concurrency as the object downloads very quickly. A simple way to work around this is to download a large object so that you can verify that it is concurrently being served to both clients. For example, you could create a large file and serve it up with your own web server, using something like Python's `http.server` module.

Alternatively, you can test correct multi-client functionality by using `telnet` from two different shells (as above), but leaving the first shell in the "telnet connected" state: i.e., not actually issuing the GET request, and then issuing a GET request from the second shell.

Of course, all of the HTTP proxying features of the basic web proxy (Section 2.1) should continue to work in the multi-client proxy.

2.3 PA1–Final: Caching and Filtering Proxy

Finally, complete your HTTP proxy by implementing some simple caching and domain-blocking features.

Caching. When a proxy is deployed for performance reasons, it typically caches web objects each time one of the clients makes a request for the first time. Basic HTTP/1.0 caching functionality works as follows. When the proxy receives a request, it checks if the requested object is cached, that is, stored locally at the proxy. If the object is stored locally, it returns the object from the cache, without contacting the origin server. If the object is not cached, the proxy retrieves the object from the origin server, returns it to the client and caches a copy for future requests.

In practice, objects on the web change over time. A proxy must therefore check that any cached object is *still up to date* before using that object in a response to a client. To do this, your proxy must implement the following steps. When your proxy's cache is enabled and a client requests object *Obj*:

1. The proxy checks if *Obj* is in the proxy's cache.
   (a) If not, the proxy requests *Obj* from the origin server using a GET request, as described earlier in this assignment.
   (b) If so, the proxy verifies that is cached copy of *Obj* is up to date by issuing a "conditional GET" to the origin server. The proxy receives the response from the origin server.
      i. If the server's response indicates that *Obj* has not been modified since it was cached by the proxy, then the proxy already has an up-to-date copy of *Obj*.
      ii. Otherwise, the server's response will contain an updated version of *Obj*.
2. If necessary, the proxy updates its cache with the up-to-date version of *Obj* and the time at which *Obj* was last modified.
3. The proxy responds to the client with the up-to-date version of *Obj*.

Conditional GET requests are described in RFC 1945, Sections 8.1 and 10.9, and in Kurose and Ross, Section 2.2.5.

Only objects returned in successful ("200 OK") responses from the origin server should be cached. In particular, your proxy should *not* try to remember the results of requests that return errors (e.g., "404 Not Found" responses).

As described later in this assignment, your proxy's cache can be *enabled* or *disabled*. When it is enabled, it follows the steps outlined above. When it is disabled, your proxy does not consult or update its cache; it simply relays HTTP messages between clients and origin servers as described in Section 2.1. The initial state of the cache (when you start your proxy process) is *disabled*, and the cache is empty. Do not persist the contents of the cache across separate runs of your proxy.

You might notice that the algorithm above does not pay attention to the "Expires" and "Pragma: no-cache" headers defined in RFC 1945. It also does not pay attention to the "Cache-control" header defined in RFC 2068 (for HTTP/1.1). This is intentional. Do not implement support for these headers in your proxy; just stick to the simple algorithm described above.

Note that real caching proxies employ sophisticated strategies to limit the total size of the cache. You do not need implement any such functionality for this programming assignment.

Domain blocking. Some web proxies are deployed in order to limit clients' access to selected sites or content. Your proxy must implement a simple "blocklist" that will prevent its clients from obtaining objects from certain websites.

When your proxy's blocklist is enabled and your proxy receives a request that refers to a blocked site, your proxy must *not* relay the request to the (blocked) origin server. In addition, your proxy should not consult or update its cache. Instead, your proxy should simply respond to the client with a "403 Forbidden" message.

The blocklist is simply a list of strings to check against the *host* portion—and *only* the host portion—of the URI in the client's request. If any string in the blocklist is a substring of the host portion of the URI, the client's request must be blocked. For example, suppose that "google" is a string in your proxy's blocklist, the blocklist is enabled, and your proxy receives following request:

```
GET http://www.google.com/ HTTP/1.0
```

Your proxy must respond to the client with an HTTP message indicating that the request was forbidden.

A detailed below, your proxy's blocklist can be *enabled* or *disabled*. When it is enabled, it performs the domain-blocking checks described above. When it is disabled, your proxy does not filter clients' requests against the blocklist. The initial state of the blocklist (when you start your proxy process) is *disabled*, and the blocklist is empty. Do not persist the contents of the blocklist across separate runs of your proxy.

Cache and blocklist control. To allow some control over your proxy's caching and domain-blocking features, your proxy must implement the interface described below. By sending certain GET requests to your proxy while it is running, a client can dynamically enable, disable, and configure these features of your proxy.

The control interface is based on GET requests for specific of URLs. In these requests, the host and port portions of the URL are ignored; only the "absolute path" portion of the URL is important.

When your proxy receives a request containing one of the special absolute paths described below, it does not consult is cache or blocklist, nor does it forward the request to an origin server. Instead, it performs an operation on its cache or blocklist:

`/proxy/cache/enable`

Enable the proxy's cache; if it is already enabled, do nothing. This request does not affect the contents of the cache. Future requests will consult the cache.

`/proxy/cache/disable`

Disable the proxy's cache; if it is already disabled, do nothing. This request does not affect the contents of the cache. Future requests will not consult the cache.

`/proxy/cache/flush`

Flush (empty) the proxy's cache. This request does not affect the enabled/disabled state of the cache.

`/proxy/blocklist/enable`

Enable the proxy's blocklist; if it is already enabled, do nothing. This request does not affect the contents of the blocklist. Future requests will consult the blocklist.

`/proxy/blocklist/disable`

Disable the proxy's blocklist; if it is already disabled, do nothing. This request does not affect the contents of the blocklist. Future requests will not consult the blocklist.

`/proxy/blocklist/add/`*`<string>`*

Add the specified string to the proxy's blocklist; if it is already in the blocklist, do nothing.

`/proxy/blocklist/remove/`*`<string>`*

Remove the specified string from the proxy's blocklist; if it is not already in the blocklist, do nothing.

`/proxy/blocklist/flush`

Flush (empty) the proxy's blocklist. This request does not affect the enabled/disabled state of the blocklist.

For any of these requests, your proxy must respond to the client with an "200 OK" response.

Testing your caching and filtering proxy. To thoroughly test the caching features of your proxy, you will want to observe the requests that it sends to origin servers, as well as the responses it receives from those origin servers, when your proxy processes its clients' requests in various situations. You can observe these messages in various ways. A straightforward way would be to have your proxy write those messages to a log file.[5] A more sophisticated approach would be to capture the packets with Wireshark. A third way would be to set up your own web server for testing (perhaps using Python's `http.server` module) and have it log the messages it receives from and sends to your proxy.

Testing the domain-blocking features is more straightforward. You can use tools like `telnet` (Section 2.1), `nc`, and `curl` (Section 5) to send requests to your proxy. Use these tools to issue requests that add and remove strings from your proxy's blocklist. Then use those tools to issue requests that should be affected by the blocklist.

Of course, all the features of the basic web proxy (Section 2.1) and the multi-client web proxy (Section 2.2) should continue to work in your final, completed proxy.

3 Implementation Details

Python. The current version of the textbook covers socket programming in Python and **you are required to use Python to complete this assignment.** For the basic proxy functionality, you are not allowed to use any libraries other than the standard socket libraries.

To help you get started, we will provide you with a very basic Python program skeleton for your proxy. You are not required to use it if you do not want to. The skeleton will be made available via Canvas.

You must implement your proxy in a single source file named `HTTPproxy.py`. This requirement makes it easy for you to submit your solution via Gradescope, and for Gradescope to test your submission.

Command-line interface. Your proxy must accept and process the following command-line arguments. (The skeleton we give to you handles basic command-line parsing.)

`-a` *`interface`*

Listen for incoming client connections on the specified network interface. This option can appear on the command line at most once. If `-a` is not specified on the command line, the proxy should listen on the `localhost` interface.

`-p` *`portnum`*

[5] If you do this, be sure to turn the logging off in the proxy you submit for grading!

Listen for incoming client connections on the specified port number. This option can appear on the command line at most once. If `-p` is not specified on the command line, the proxy should listen on port 2100.

You may implement other command-line arguments for your own purposes, e.g., for debugging. If you do this, those command-line arguments must be optional. When we grade your proxy, we will not run your proxy with any of your "custom" command-line arguments.

Implementing the cache. Because cached objects can be quite large, your proxy should store cached objects *on disk*, not in memory. In other words, when your proxy needs to save a retrieved object, it should write it to a disk file; when there is a cache "hit," your proxy should read the previously saved object from the file. For this, you need to implement some internal data structure in the proxy to keep track of which objects are cached and where they are on the disk. This bookkeeping data structure can be maintained in main memory; there is no need to make it persist across proxy shutdowns.

The Gradescope environment. Your proxy will be tested and graded via Gradescope. **Your proxy must run in the Gradescope environment that we have set up for this assignment.** You can develop your program(s) on any OS platform or machine, but it is your responsibility to ensure that it runs in Gradescope. You will not get any credit if the TA is unable to run your program(s).

The Gradescope environment is a container with Ubuntu 18.04 (Linux) and Python 3.7.5 installed. When your proxy is run within Gradescope:

- It will be invoked like this: `python HTTPproxy.py`
- The current working directory (i.e., the "`.`" directory) of the proxy process will be writable. The proxy can write cache files there.

To help you make sure your proxy works within Gradescope, the TAs will configure Gradescope to automatically test your submitted code.

4 Grading and Evaluation

To encourage you to start early and systematically work on the assignment, there will be three submissions as outlined below. The primary purpose of dividing the work like this is to help you to systematically work through the assignment. At the discretion of the instructor, the early parts of the assignment may or may not be substantively graded. In other words, you should not expect feedback on a particular part before the next part will be due.

4.1 PA1–A: Basic Proxy

For the first part of the assignment, the focus will be on basic (single client) proxy functionality. Specifically, we will evaluate your code by performing tests similar to those described in Section 2.1.

What to hand in. Submit your completed sub-assignment via Gradescope by the due date. Your submission should consist of a single Python source file named `HTTPproxy.py`, as described in Section 3. If you want to share information about your submission with the TAs, put that information in comments at the top of your Python source file.

4.2 PA1–B: Multi-client Proxy

For this part of the assignment, you are to develop the multi-client proxy as described in Section 2.2. At this point, the emphasis will be on the *multi-client* aspect of the proxy. We will evaluate your code by performing tests similar to those described in Section 2.2. (Of course, the basic proxy functionality should continue to work, too!)

What to hand in. Submit your completed sub-assignment via Gradescope by the due date. Your submission should consist of a single Python source file named `HTTPproxy.py`, as described in Section 3. If you want to share information about your submission with the TAs, put that information in comments at the top of your Python source file.

4.3 PA1–Final: Complete Assignment

For your final submission, you should implement the remaining required functionality in your proxy. Your final submission will be tested more thoroughly, using tests like those described in Sections 2.1, 2.2, and 2.3. The TAs will also inspect your code for general quality and robustness, including inline documentation and appropriate exception handling.

What to hand in. Submit your completed sub-assignment via Gradescope by the due date. Your submission should consist of a single Python source file named `HTTPproxy.py`, as described in Section 3. If you want to share information about your submission with the TAs, put that information in comments at the top of your Python source file.

4.4 Grading

Criterion	**Points**
PA1–A sub-assignment	10
PA1–B sub-assignment	10
PA1–Final	
Program functionality	70
Inline documentation	5
Exception handling in code	5
Total	**100**

5 Additional and Important Notes

Individual work! As stated in the course syllabus, every programming assignment in this course must be done individually by a student. No teaming or pairing is allowed. **Note that your code will be checked for similarity against other submissions.**

Gradescope. As described in Section 3, your proxy will be tested and graded via Gradescope. Your proxy must run in the Gradescope environment that we have set up for this assignment. You can develop your program(s) on any OS platform or machine, but it is your responsibility to ensure that it runs in Gradescope. You will not get any credit if the TA is unable to run your program(s).

CADE Lab. If you use the CADE Lab machines to work on your proxy, use TCP port numbers 2100 through 2120. These ports are only accessible from inside the CADE Lab network.

HTTP/1.0. To keep as assignment simple, your proxy should support version 1.0 of the HTTP protocol, as defined in RFC 1945. Do not support later versions of the protocol.

An implication of using HTTP/1.0 is that you can infer that you have received all the content from the server when the server closes the connection.

As specified in the assignment, the request forwarded to the origin server should contain a "Connection: close" header line to ensure that it closes the connection. (This might imply replacing a "Connection: keep-alive" header if the client had that in its request.)

Using command-line tools. There are several command-line Linux programs that are useful for testing your proxy. One of these is `telnet`, which allows you to connect to your running proxy via TCP and send data (ideally, well-formed HTTP/1.0 requests) to your proxy interactively, using your keyboard. Earlier sections of this assignment describe some simple tests that you can carry out via `telnet`. A downside to `telnet` is that it is interactive, meaning that you need to type out the requests.

Netcat (`nc`) is a program that can connect to a host/port and send data from a file. For example, to connect to the web server running on host `www.flux.utah.edu` and send it the data contained in file `request.txt`, you could use the following command:

```
# nc -c www.flux.utah.edu 80 < request.txt
```

(The `-c` option tell `nc` to send CRLF line endings, as required by HTTP.) The response from the server appears in the terminal window. Using `nc`, you can easily create a collection of test inputs in files and then send them to your proxy on demand.

`curl` is another program that you might find handy for testing. It is a multi-purpose tool for transferring data to and from servers, but what is important to us is that it can function as an HTTP/1.0 client. For example, the following command uses HTTP/1.0 to fetch the object at the URL shown:

```
# curl --http1.0 http://www.flux.utah.edu/cs4480/simple.html
```

The `--http1.0` command-line option is important: it tells `curl` to use HTTP/1.0 rather than some other version of HTTP. The following option tells `curl` to connect to an explicit inline proxy using the HTTP/1.0 protocol:

```
--proxy1.0 <proxyhost[:port]>
```

For more details about using `curl`, consult its online documentation.

Using a real web browser. A significant simplification that comes from limiting the proxy functionality to HTTP 1.0 is that your proxy does not need to support persistent connections. Recall that the default behavior for HTTP 1.0 is non-persistent connections. Note, however, that most browsers, including the latest version of Firefox, automatically add the optional "Connection: keep-alive" header line. Also, with current versions of Firefox, it is no longer possible to disable this behavior.

You do not need to test your proxy in conjunction with a real web browser. Instead, you can use the various command-line tools mentioned earlier to exercise your proxy.

However, if you want to test with a real web browser, you will probably need to download an older version of that browser. This is a potential topic for discussion via Canvas. *IMPORTANT:* Older versions of browsers have security vulnerabilities. I therefore strongly suggest that if you get an older version of a browser to use for this programming assignment, you do not use it for general web browsing, but only for the assignment.