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 (that is, between the browser and the web 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 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. (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.)

- **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 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.

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.

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 1945Links to an external site. Section 9.5Links to an external site. — Server Error 5xx.)*

You should not assume that the origin server will be running on a particular IP address (your proxy should handle both DNS and IPv4 URLs. Hint: Python sockets perform DNS resolution on their own), or use a particular TCP port number (your proxy should handle clients connecting to a web server through a port other than 80, for example http://localhost:88/Links to an external site.), or that clients will be coming from a predetermined IP (your proxy should be able to handle any incoming IPv4 client). You may assume this assignment will be working only with IPv4 addresses and hosts (we will discuss IPv4 and IPv6 later in the course).

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 the "Implementation Details" section below) 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 1945Links to an external site., Section 5.1.2Links to an external site.)—as a browser will send if properly configured to explicitly use a proxy. (This is in contrast to a transparent on-path proxy that some ISPs deploy, unbeknownst to their users.)

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.

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 `2100` and is bound to the `localhost` interface, then as a basic test, try requesting an HTML page. There are different tools you can use to test this, but one way to send the request and print the response is by writing a tiny Python client. For example, save the following as simple-test.py:

```
from socket import *
```

```
sock = socket(AF_INET, SOCK_STREAM)
sock.connect(('localhost', 2100))
sock.sendall(b'GET http://www.flux.utah.edu/cs4480/simple.html HTTP/1.0\r\n\r\n')

print(sock.makefile('rb').read())
```

Then, you can run this test client while your proxy is running:

```
# python3.10 simple-test.py
b'HTTP/1.1 200 OK\r\nDate: Tue, 02 Jan 2024 20:05:47 GMT\r\nServer: Apache/2.2.24
(FreeBSD) PHP/5.3.10 with Suhosin-Patch mod_ssl/2.2.24 OpenSSL/1.0.1h
DAV/2\r\nLast-Modified: Sat, 22 Jan 2022 23:19:27 GMT\r\nETag:
"206accd-c2-5d633f776fdc0"\r\nAccept-Ranges: bytes\r\nContent-Length: 194\r\nConnection:
close\r\nContent-Type: text/html\r\n\r\n<!DOCTYPE html>\n<html lang="en">\n<head>\n  <meta
charset="utf-8">\n  <title>Simple Page</title>\n</head>\n\n<body>\n<h1>Simple
Page</h1>\n\n<p>\n  Hello!  This is simple HTML page.\n</p>\n</body>\n\n</html>\n'
```

Be sure to enter a blank line after the GET line (if you've forgotten why this is important, review Kurose and Ross, Section 2.2.3; this is why the test client sends \r\n\r\n). 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 or by sending a request directly to the origin server using another simple Python script. For example:

```
from socket import *

sock = socket(AF_INET, SOCK_STREAM)
sock.connect(('www.flux.utah.edu', 80))
sock.sendall(b'GET http://www.flux.utah.edu/cs4480/simple.html HTTP/1.0\r\n\r\n')

print(sock.makefile('rb').read())
```

```
# python3.10 simple-test-skip-proxy.py
b'HTTP/1.1 200 OK\r\nDate: Tue, 02 Jan 2024 20:05:47 GMT\r\nServer: Apache/2.2.24
(FreeBSD) PHP/5.3.10 with Suhosin-Patch mod_ssl/2.2.24 OpenSSL/1.0.1h
DAV/2\r\nLast-Modified: Sat, 22 Jan 2022 23:19:27 GMT\r\nETag:
"206accd-c2-5d633f776fdc0"\r\nAccept-Ranges: bytes\r\nContent-Length: 194\r\nConnection:
close\r\nContent-Type: text/html\r\n\r\n<!DOCTYPE html>\n<html lang="en">\n<head>\n  <meta
charset="utf-8">\n  <title>Simple Page</title>\n</head>\n\n<body>\n<h1>Simple
Page</h1>\n\n<p>\n  Hello!  This is simple HTML page.\n</p>\n</body>\n\n</html>\n'
```

See the "Additional and Important Notes" section for more advice about testing your proxy. For example, using a true HTTP client to test your proxy is very helpful, since you can quickly try out different URLs and requests with more complex and realistic headers.

curl --http1.0 http://www.flux.utah.edu/cs4480/simple.html --proxy localhost:2100

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!) You should use Python's `threading` module to do this.

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 (see the "Implementation Details" section). Unlike the basic proxy, as each new client establishes a connection, the multi-client proxy should start a new thread to handle the incoming request. Individual client requests should otherwise be handled as described in part PA1-A: 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 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 objects concurrently with two different clients (e.g. two separate Python processes or concurrent instances of `curl`, etc.). 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.

One good strategy to test multi-client concurrency is to connect one client without sending the whole HTTP request (e.g. sending the request without the last \r\n). This keep the thread handling that client blocked at the server awaiting more data from the client. If, while this first request is stalled, you can make a second request from another client without it getting "stuck" behind the first client's request, then your solution is likely working.

Of course, all of the HTTP proxying features of the basic web proxy (see part PA1-A) should continue to work in the multi-client proxy.

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.
    - If not, the proxy requests *Obj* from the origin server using a GET request, as described earlier in this assignment.
    - 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.
    - 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*.
    - 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 1945Links to an external site., Sections 8.1Links to an external site. and 10.9Links to an external site., 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 part PA1-A. 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 1945Links to an external site.. It also does not pay attention to the “Cache-control” header defined in RFC 2068Links to an external site. (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. The optional port in the URI is considered part of the host part of the URI, so it should be possible to block specific ports or specific domain/port combinations using the blocklist.

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 its 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 (if you do this, be sure to turn the logging off in the proxy you submit for grading). 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 a simple Python client like the ones shown in PA1-A or `curl` (see below) 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 (part PA1-A) and the multi-client web proxy (part PA1-B) should continue to work in your final, completed proxy.

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.**

You are not allowed to use any libraries other than the standard socket libraries.

To help you get started, we have provided you with a very basic Python program skeleton for your proxy

You are not required to use it if you do not want to.

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`

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.

Important: immediately after you create your proxy's listening socket add the following code (where `skt` is the name of the socket here):
`skt.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)`
Without this code the autograder may cause some tests to fail spuriously.

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.

For this assignment it is okay to *keep cached objects in memory in your proxy*. In practice, a real proxy would need to deal with the fact that objects could be large (e.g. by caching objects in files), and it would need to deal with eviction (forgetting about old objects in order to make space for newer ones). The cache should not be persistent. Each time your proxy restarts it should forget everything that was cached previously and revert to its default state of not caching objects.