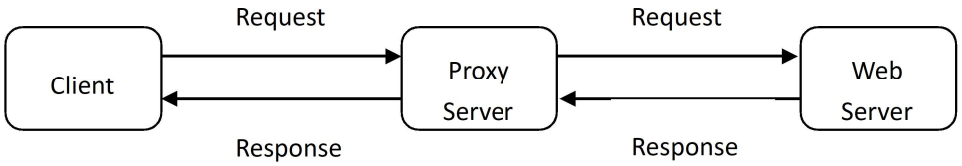
Lab 4: HTTP Web Proxy Server

In this lab, you will learn how web proxy servers work and one of their basic functionalities – caching.

Your task is to develop a small web proxy server which is able to cache web pages. It is a very simple proxy server which only understands simple GET-requests, but is able to handle all kinds of objects - not just HTML pages, but also images.

Generally, when the client makes a request, the request is sent to the web server. The web server then processes the request and sends back a response message to the requesting client. In order to improve the performance we create a proxy server between the client and the web server. Now, both the request message sent by the client and the response message delivered by the web server pass through the proxy server. In other words, the client requests the objects via the proxy server. The proxy server will forward the client's request to the web server. The web server will then generate a response message and deliver it to the proxy server, which in turn sends it to the client.



Setup

You will be provided a zip file with these instructions, starter code for the proxy, a Flask test application, and a test suite. Unzip this file to a workspace on your machine. The setup instructions will assume that you have done this and have navigated to the resulting directory on your command line.

The zip file includes:

- programming_hw_proxyserver.pdf – These instructions
- proxy.py – Starter code for your proxy server
- app.py – A Flask application to test with. This will take the place of the "Web Server" in the above diagram.
- test_proxy.py – A Python test suite to run the autograder tests. The tests will take the place of the "Client" in the above diagram.

To get your environment set up, you will want to make sure you have Python 3 installed. Please take a moment to verify this by running the command:

```
python --version
```

If this outputs a version of Python >= Python 3.9, you're all set. If it outputs a version of Python older than 3.9, you will want to upgrade Python to at least 3.9. If it outputs a version of Python 2, then you may have to use

```
python3 --version
```

If you do not have Python 3 installed on your system, you will need to visit the Python website and follow the installation instructions for your system.

Whichever is the correct Python command for your system, you will need to use it for the following step to create a virtual environment.

Setting Up a Python Virtual Environment

In the project directory, you will want to create a virtual environment. To do this, run the following command (or an alternate Python command as discussed above).

```
python -m venv venv
```

Then activate the virtual environment on Windows by running

```
venv\Scripts\activate
```

or on Mac/Linux by running

```
source ./venv/bin/activate
```

After this, your command line prompt should change to show the activated virtual environment. While in the virtual environment, all of the remaining steps can be followed by using the python command (and not python3 or any alternate alias you may have been using before).

Installing Dependencies

The tests require the Flask module and the requests module to be installed. Install them into the virtual environment with these commands:

```
python -m pip install requests
python -m pip install Flask
```

Part One: Basic HTTP Proxy

In this section, you will implement a web proxy server that is capable of receiving an HTTP GET request from a client and forwarding it to the destination server. It should then receive the HTTP response from the server and forward it back to the client.

The starter code in proxy.py includes boilerplate and some structure to get you somewhat close to accomplishing this, but it is incomplete. Fill in the blanks between the comments *Fill in start* and *Fill in end*. Study the flow of the `proxyServer(port)` function to understand what the proxy is trying to do. Some functions like `parse_http_headers(sockf)` are provided for you in their entirety to give you an idea of how to parse HTTP messages.

You are encouraged to use the HTTP specification found here as a reference.

When you are finished with this section, you should be able to pass the basic200 and basic404 tests. You can run them with the following commands:

```
python -m unittest test_proxy.TestProxy.testBasic200
python -m unittest test_proxy.TestProxy.testBasic404
```

Each passed test is worth 1 point.

Part Two: HTTP POST Requests

In this section, you will extend your proxy to be able to handle HTTP POST requests that include data in the body. The starter code for `forward_request` likely only has enough for you to handle a basic GET request with no body. You are free to change the function, the argument list, and anything else in the code to implement HTTP POST proxying. The only real requirement is that you must keep a function with the signature `proxyServer(port)`

When you are finished with this section, you should be able to pass the testPosts test.

```
python -m unittest test_proxy.TestProxy.testPosts
```

This test is worth 1 point.

Part Three: Caching

A typical proxy server will cache the web pages each time the client makes a particular request for the first time. The basic functionality of caching works as follows. When the proxy

gets a request, it checks if the requested object is cached, and if yes, it returns the object from the cache, without contacting the server. If the object is not cached, the proxy retrieves the object from the server, returns it to the client and caches a copy for future requests.

In this section, you will implement the simple caching functionality described above. Your implementation will need to be able to write responses to the disk (i.e., the cache) and fetch them from disk when you get a cache hit. Only GET requests can be cached; POST requests or any other requests that may have side effects like PUTs or DELETEs cannot be cached.

In practice, the proxy server must verify that the cached responses are still valid and that they are the correct responses to the client's requests. You can read more about caching and how it is handled in HTTP in RFC2068. You do NOT need to implement any replacement or validation policies. It will be sufficient to cache a page forever.

The starter code defines a `cacheDir` location, which is a subdirectory named `cache` in the same directory as proxy.py. You may define the directory structure of your cache how you like, but it should all go in this `cache` directory. There is also starter code that will open a cache file for you, but you will need to decide where the cache file should live and how to write to it.

When you are finished with this section, you should be able to pass the testCacheGets and testCachePosts tests:

```
python -m unittest test_proxy.TestProxy.testCacheGets
python -m unittest test_proxy.TestProxy.testCachePosts
```

Each passed test is worth 1 point.

Testing

To run the entire test suite, you can run this command:

```
python -m unittest test_proxy
```

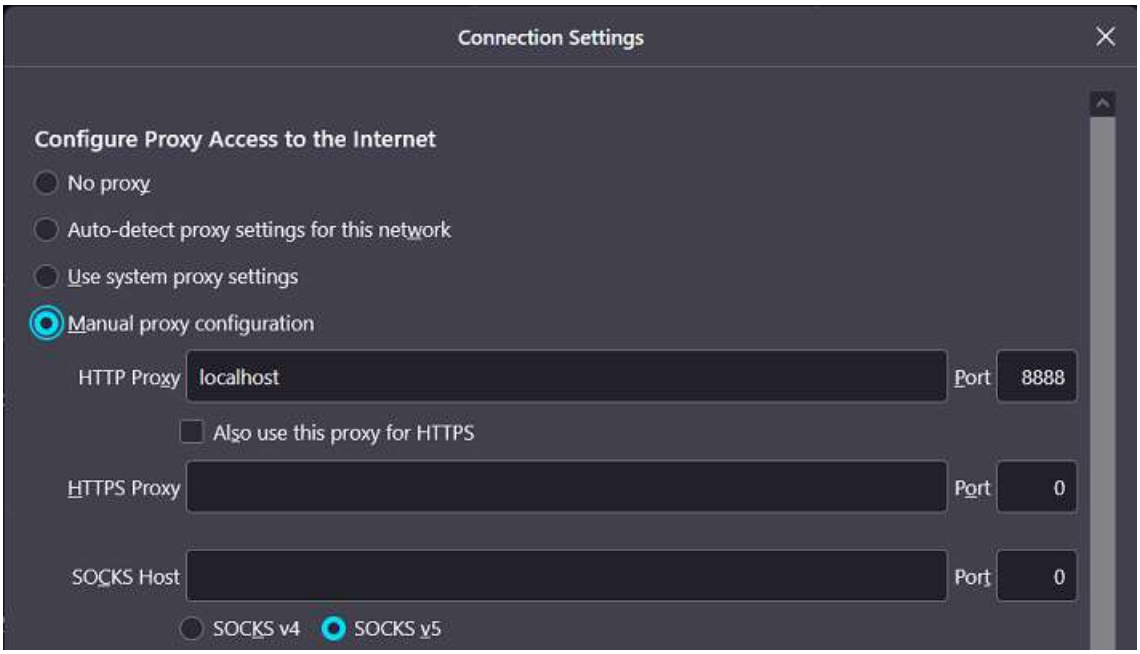
The test suite will run an instance of your proxy server on port 8888 in one subprocess and an instance of the Flask test application on port 5000 in another subprocess. It uses Requests, a Python module for making HTTP requests, to send requests to the Flask server through the proxy.

It is also a good idea to test the proxy outside of the test suite and see if it works on remote sites. You can run the proxy with:

```
python proxy.py
```

This will start the proxy server on your machine on port 8888. You can then configure your browser to use the proxy. The instructions for this will differ per browser.

On Firefox, you can go to Settings → General → Network Settings. You'll end up on a screen like this one:



Select "Manual proxy configuration" and set localhost:8888 as the HTTP Proxy.

Chrome uses your system's proxy settings. In Chrome, you can go to Settings → Advanced → System → "Open your computer's proxy settings"

Once you have your browser configured to use the proxy, you can continue with using the browser as normal, except now all HTTP requests will first go to your proxy server. In order for this to be interesting, you should test against sites that allow the use of HTTP. Most websites today will automatically redirect you to the HTTPS version of their site. Here are a few sites that we've tested with that will not redirect:

- http://gnu.org
- http://nginx.org
- http://go.com

You will likely need to manually type out "http://" in the address bar, otherwise your browser will automatically request for HTTPS.

What To Hand In

You should submit a file named proxy.py that contains your code. Optionally you may also submit a PDF report explaining your code, or you may document your code with comments in proxy.py. You should submit no other files.

Tips

1. Remember that Wireshark is a good tool for inspecting the traffic originating from and arriving at your machine. If you're troubleshooting the Python test suite, you can start a capture on the loopback interface. If you're troubleshooting against a remote server, you can capture on your normal, primary network interface.
2. If your proxy passes the Python test cases, you can be pretty confident it will work on the autograder. However, it is still a good idea to test with your browser, which will let you verify that your proxy forwards non-text objects properly and behaves well in other edge cases.