HTTP Proxy Server

Mahan Madani

Abstract

For this project, I used python to implement a proxy server designed to handle HTTP requests and responses between clients and servers. The proxy server can be accessed locally via browser, using localhost and the port number 8888. All requests sent using this address and port will go through the proxy server. Additionally, I implemented a caching system to improve performance by reducing redundant data transfer.

What Is a Proxy Server?

A proxy server acts as an intermediary between client devices (such as web browsers) and servers, facilitating communication by forwarding requests and responses. It also masks the client's IP address enhancing its privacy and security.

Proxy servers can be used to bypass network restrictions, access blocked content, and optimize internet traffic by caching frequently accessed resources. They are also used for monitoring internet usage, making them essential tools for both personal privacy and network management.

The proxy server implemented for this project mainly focuses on the HTTP protocol. Only basic python libraries were used (such as sockets) and due to this, more advanced protocols (such as HTTPS) are not fully supported.

Setting Up the Proxy Server

The proxy server has 4 main goals:

1. Receive an HTTP request from the client
2. Modify the request and send it to the specified web server
3. Transfer the server’s response back to the client
4. Cache frequently accessed content

Due to the nature of the proxy server, each connection is closed after one request-response cycle. Meaning if a client has multiple requests, it must establish a separate connection for each request. Note that the server utilizes multithreading to be able to offer its services to multiple clients simultaneously.

All of the code for the proxy server can be found in the ProxyServer.py file. The server first creates a Server Socket, to which clients can connect to. A new thread is created after a client establishes a connection with the server. The client's request is handled within that thread, and after a response is sent to the client, the connection is closed and the client gets disconnected.

Modifying The Original Request

Once a client has successfully connected to the proxy server, it can send one HTTP request to the proxy server. The HTTP request my contain either the GET or POST method. The client should also send the URL of the requested web page, which contains both the web address of the server and the path to the specific file or web page it has requested.

For example, if the request URL is:

`localhost:8888/www.geeksforgeeks.org/courses/coding-for-everyone`

Then the web address of the server would be: `www.geeksforgeeks.org`

And the path to the webpage would be: `/courses/coding-for-everyone`

This information is present in the request line of the HTTP request sent to the proxy server and can be easily extracted. The web address is the address of the host the original request is meant to be sent to, and the file path is the resource it requires from the host.

The proxy server must modify the request to include these details. Additionally, the request is sent to the web address via a separate socket, using the web address of the target host and port 80 (all HTTP messages must use this port number).

The modified request's header will contain the following lines:

```
`HTTP method   /file_path   HTTP_version`

`Host: web_address`
```

Going with the previous example, these lines would be:

```
`GET  /courses/coding-for-everyone  HTTP/1.1`
`Host: www.geeksforgeeks.org`
```

Note that the `Connection` header is also set to `close` to ensure each connection is closed after a full message transfer between the web server and the client.

Logging and Error Handling

The proxy server I implemented utilizes extensive logging and error handling. Every significant action is logged (such as receiving a request, receiving a response, caching a response, etc.), and the details of each request is also displayed. There is also the option to print each individual request and response if necessary.

Additionally, I handled every exception I ran into and took appropriate measures for each one. The proxy server also sends a custom `Not Found` response in the event of not receiving a response from the web server.

Web Caching

As a bonus objective, I implemented a simple caching system for the proxy server. Each time a client requests a resource, a copy of the server's response is stored as a file in a `cache` folder. The file's name is derived from the requested URL, so each resource is uniquely identifiable.

If a client requests a web page, the proxy server first checks the `cache` directory to see if the client's desired file has already been cached. If a cached version exists, the proxy server reads the file (which contains both a response and the requested file) and directly sends it to the client instead of receiving another copy from the original server.

Web caching allows the proxy server to handle clients much faster and prevents unnecessary traffic on the server side. A proper proxy server would also update these cached websites; however, my implementation simply caches each resource once and reuses the cached version indefinitely.

Example Usage of the Proxy Server

While the proxy server can technically access both HTTP and HTTPS websites, it works most correctly with HTTP websites. As an example, I accessed this website (Figure 1) in my browser, from the proxy server.

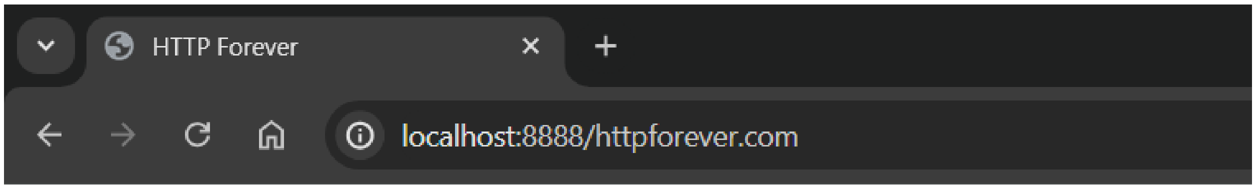
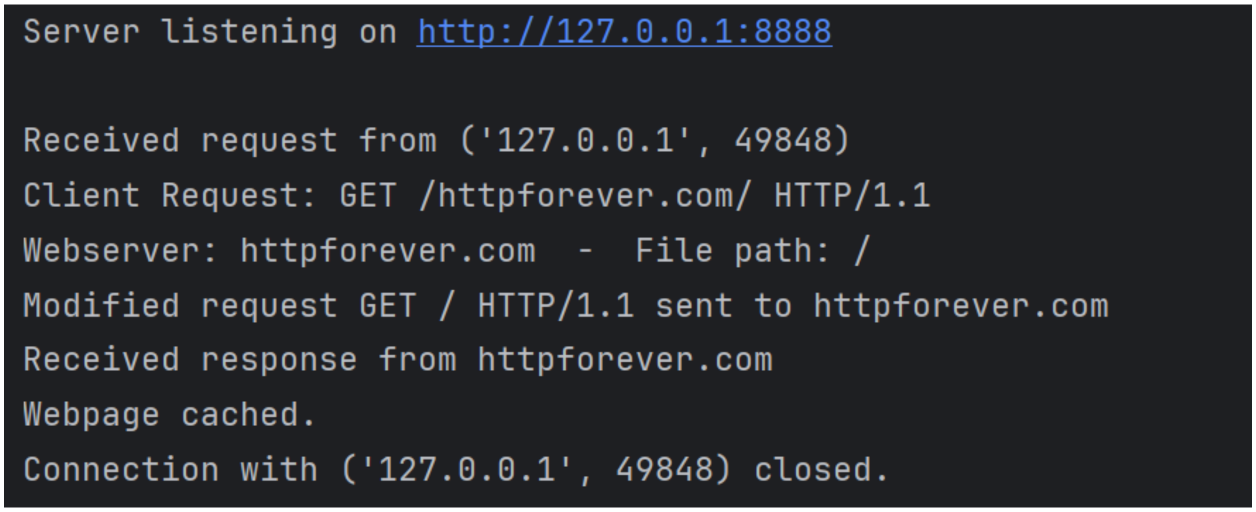


Figure 1

The proxy server's log for accessing the main page can be viewed in Figure 2. A GET request is received from the client. A modified version of the request is sent to the server and a response is received, which is then transferred to the client.

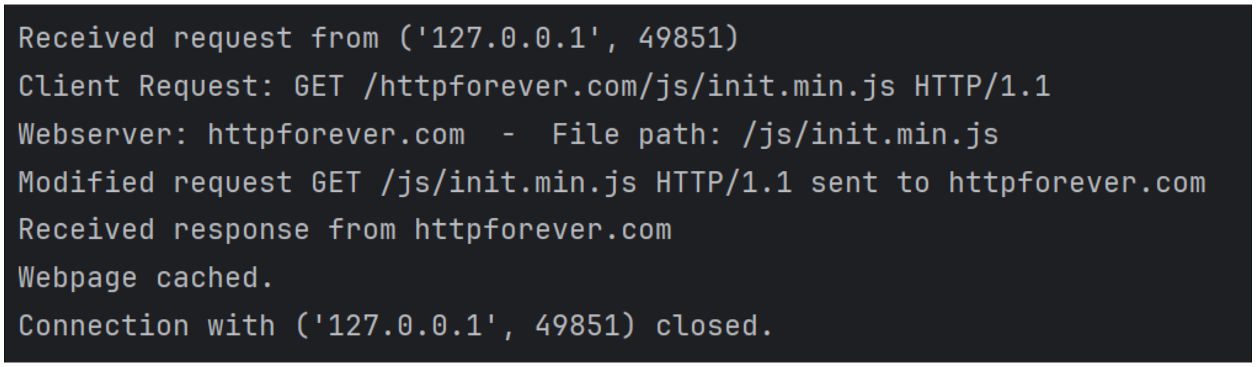


```
Server listening on http://127.0.0.1:8888

Received request from ('127.0.0.1', 49848)
Client Request: GET /httpforever.com/ HTTP/1.1
Webserver: httpforever.com  -  File path: /
Modified request GET / HTTP/1.1 sent to httpforever.com
Received response from httpforever.com
Webpage cached.
Connection with ('127.0.0.1', 49848) closed.
```

Figure 2

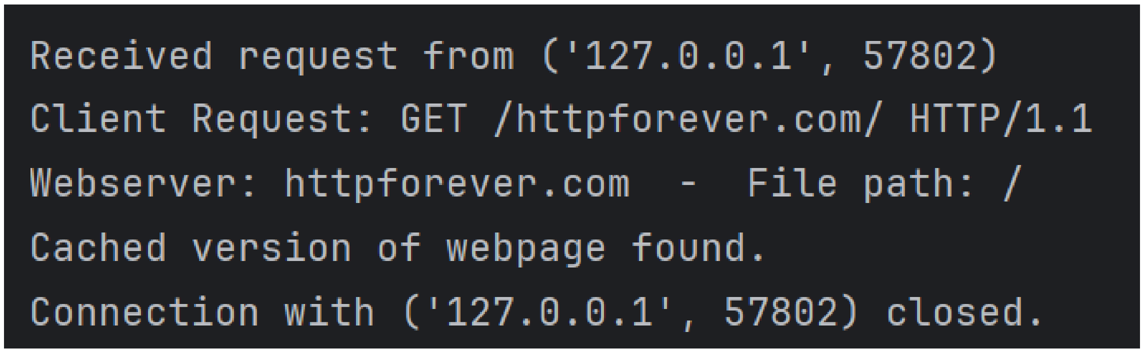
Several other requests are sent afterward, each requesting a different resource from the website. One example is shown in Figure 3, where a JavaScript file is requested by the client. Note that each resource is cached separately, so each request-response cycle also includes a caching step.



```
Received request from ('127.0.0.1', 49851)
Client Request: GET /httpforever.com/js/init.min.js HTTP/1.1
Webserver: httpforever.com  -  File path: /js/init.min.js
Modified request GET /js/init.min.js HTTP/1.1 sent to httpforever.com
Received response from httpforever.com
Webpage cached.
Connection with ('127.0.0.1', 49851) closed.
```

Figure 3

Requesting the same webpage again results in the cached version of the resources being sent, as shown in Figure 4.



```
Received request from ('127.0.0.1', 57802)
Client Request: GET /httpforever.com/ HTTP/1.1
Webserver: httpforever.com  -  File path: /
Cached version of webpage found.
Connection with ('127.0.0.1', 57802) closed.
```

Figure 4

Conclusion

This project demonstrates the usage of proxy servers and the challenges I faced during the implementation of one. Only utilizing basic python libraries was the biggest challenge for me, as there are several more powerful tools for creating an HTTP web server that massively simplify this process. Ultimately, creating a more advanced proxy server that is able to handle various protocols requires more effort, in addition to the usage of advanced libraries.