HTTP Proxy Server

Syed Zaeem Shakir

Z5643027

Summary

This report describes a complete HTTP proxy server built in Python that acts as a middleman between web browsers (clients) and websites (servers). The proxy server handles all the communication between the web browser and websites. When you want to visit a website, instead of connecting directly to that website, the client is connected to our proxy first. The proxy then fetches the website for it and sends it back

Programming Language and Code Organisation

The HTTP Proxy Server was implemented using, selected for its ease of use, extensive standard library, and built-in support for networking and multithreading.

The entire implementation resides in a single source file, “Proxy.py”

This file contains all classes and functions necessary to run the proxy server, including HTTP request/response parsing, cache management, multithreaded connection handling, and HTTPS tunneling.

Python’s standard libraries were used:

socket – for networking and connection handling.

threading – to handle multiple clients concurrently.

collections.OrderedDict – to implement the LRU caching mechanism.

datetime, urllib.parse, sys, re – for operations like logging, URL parsing, and error handling.

A log file named **proxy.log** is created automatically to record all HTTP transactions in a Common Log Format. All interactions, cache status, and response sizes are tracked and written both to the terminal and the log file.

Key Features:

1. **Smart Caching System (LRU):** The proxy saves the websites visited recently and stores copies of them. When asked for the same website again, the proxy can instantly provide the saved copy instead of getting it again from the server. This makes websites load much faster and reduces internet traffic. the code uses an ordered dictionary and removes not used site from the storage to prevent it from getting full.

2. **HTTPS Support Through Tunneling**: for CONNECT command, the proxy doesn't read or modify the encrypted data; it just passes it back and forth safely to the origin server thus tunneling the data.
3. **Persistent Connections with Clients**: The proxy server supports "persistent connections," which means once client connects to the proxy, it can keep that connection open and send multiple requests through the same connection.
4. **Multiple Users at Once**: The proxy can handle many users simultaneously. When multiple people try to use the proxy at the same time, it creates separate "threads" for each communication.
5. **Detailed Record Keeping**: The proxy keeps a detailed log of every request it handles, including who made the request, what they asked for, whether it was found in the cache, and how long it took.

How Different Types of Requests Are Handled:

- **Website Requests (GET, HEAD, POST)**: the proxy first checks if it has a recent copy saved. If yes, it sends you the saved copy immediately (very fast). If no, it gets the content from the origin server, saves a copy for future use, and sends it to you.
- **Requests (CONNECT):** The proxy creates a tunnel directly between your Client and the origin server. The data passes through the proxy without any formatting.

The proxy successfully handles all the complex technical requirements of modern web browsing including managing different types of HTTP requests, maintaining secure connections, efficiently storing and retrieving cached content, handling multiple users simultaneously, and keeping detailed logs of all activities.

How And What Each Class do:

1. **HTTPMessage Class**:
    - **parse_request(data):** Parses an HTTP request message from the provided data.
    - **parse_response(data):** Parses an HTTP response message from the provided data.
2. **HTTPCache Class:**
    - **normalize_url(url):** Normalizes the URL for use as a cache key.
    - **get(url):** Retrieves a cached response for the given URL.
    - **put(url, response_data):** Caches the response data for the given URL, evicting the least recently used items if necessary.
3. **HTTPProxy Class:**

- **start():** Starts the proxy server, listens for client connections, and handles them in separate threads.
- **handle_client(client_socket, client_address)**: Handles a client connection, receiving and processing requests.

- **receive_http_message(sock):** Receives a complete HTTP message from the socket.
- **process_request(client_socket, client_address, request, request_data):** Processes an HTTP request, dispatching to the appropriate handler based on the request method.
- **handle_connect(client_socket, client_address, request, timestamp):** Handles the CONNECT method for HTTPS tunneling.
- **tunnel_data(client_socket, server_socket):** Performs bidirectional data tunneling for HTTPS connections.
- **handle_http_request(client_socket, client_address, request, request_data, timestamp):** Handles GET, HEAD, and POST requests, forwarding them to the origin server and caching the responses.
- **transform_request(request, path**): Transforms the client's HTTP request for forwarding to the origin server.
- **transform_response(response, original_request):** Transforms the origin server's HTTP response for forwarding to the client.
- **send_error_response(client_socket, status_code, reason_phrase, error_message**): Sends an error response to the client.
- **is_self_loop(host, port):** Checks if the request is pointing to the proxy itself.
- **log_request(client_address, cache_status, timestamp, request_line, status_code, body_length):** Logs the request in the Common Log Format.

Request Processing Logic

Based on the HTTP method, the proxy routes requests through different processing paths:

For CONNECT Requests:

- Validates the target (must be port 443)
- Establishes connection to target server
- Sends "200 Connection Established" response
- Initiates bidirectional data tunneling

For GET/HEAD/POST Requests:

- Parses the absolute-form URL to extract host, port, and path
- Checks cache for GET requests (cache hit/miss determination)
- Transforms request from absolute-form to origin-form
- Forwards request to origin server
- Processes and caches responses (if cacheable)

Response Processing and Caching

- **Response Parsing**: Extracts status code, headers, and body
- **Cache Evaluation**: Determines if response should be cached (GET, status 200, size limits)

- **LRU Management**: Evicts least recently used items when cache is full
- **Response Transformation**: Modifies headers for client compatibility

Data Structure Used

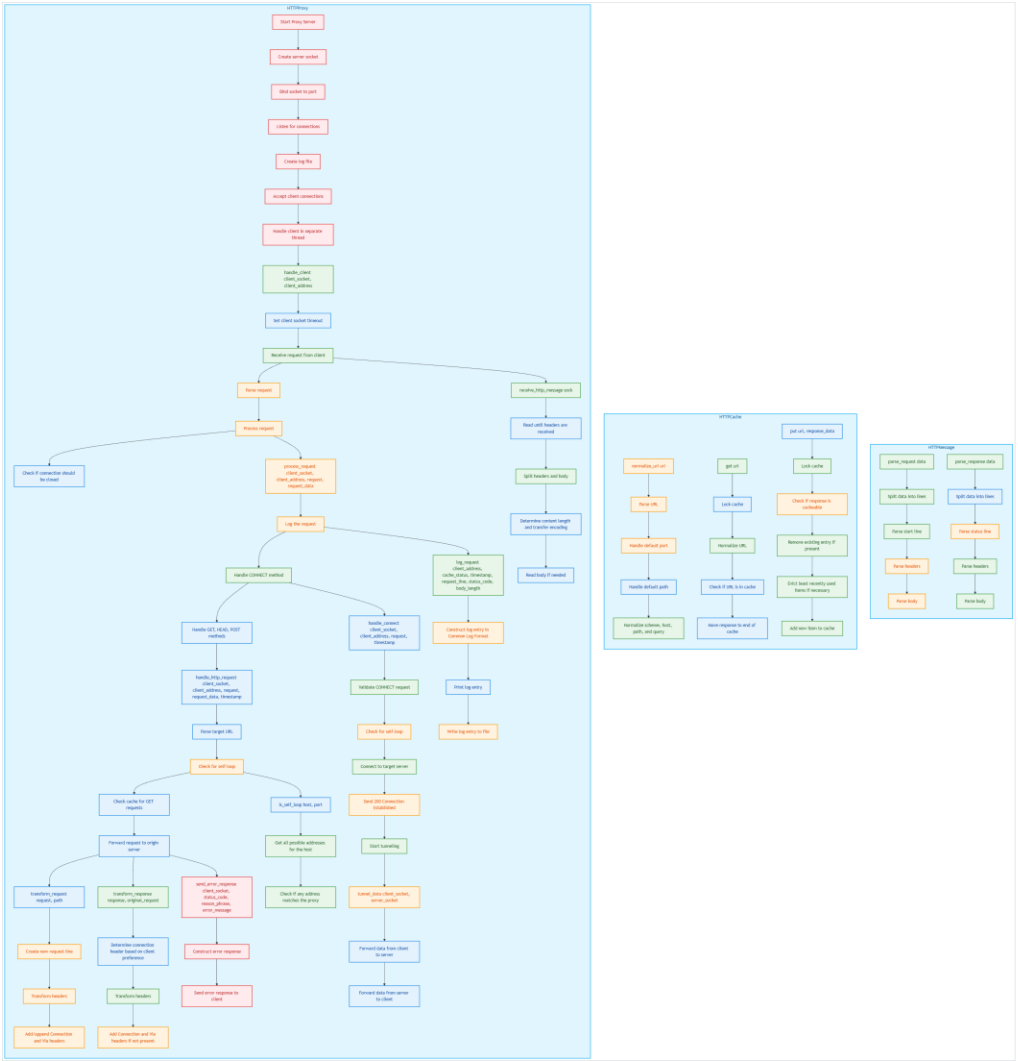
1-. OrderedDict: storing caches and LRU implementation

2- Dict and list: storing header info

3-Threading.Lock

4- Socket: to make a connection between proxy,client,origin

FLOW CHART



Limitations

- **Cache is time-unaware** and does not support expiry headers; it only uses size-based eviction (LRU).
- **Broken or invalid requests** may not be handled gracefully and can cause errors.
- **No persistent connections** with the origin server; each request opens a new connection.
- **CONNECT method works only for port 443**; HTTPS on other ports is blocked.
- **Chunked responses** are not decoded properly.
- **Log file may grow indefinitely**, as there is no size limit or rotation mechanism.

Acknowledgement

For flowchart and debugging

https://codetoflow.com/

for understanding and implementing

https://www.youtube.com/watch?v=Ve94yNM0OCU

https://www.youtube.com/watch?v=N-WW_v0C0Ac&t=87s

https://www.youtube.com/watch?v=-Zea7GB2OwA

https://www.youtube.com/watch?v=tb8gHvYlCFs