Proxy Server

Submitted by: Tamer Kobba
Student ID(s): 202104873
Instructor's Name: Ayman Tajeddine

December 9, 2024

Contents

1 Introduction

This report presents an Python-based proxy server application. The proxy supports both HTTP and HTTPS (via the CONNECT method) requests, implements content caching with cache invalidation, provides comprehensive logging, uses multithreading to handle concurrent connections, and includes domain-based filtering using allowlists and blocklists.

2 Requirements

The requirements for the proxy server remain consistent with the original goals:

- **A. Basic Proxy Functionality:**
  - Accept and parse requests from clients, forward these requests to the target server, and relay the responses back to the clients.
  - Support both HTTP and, as a bonus, HTTPS by establishing a tunnel (CONNECT method) without inspecting encrypted data.
- **B. Socket Programming:**
  - Utilize sockets to handle connections between clients and target servers.
  - Listen for incoming client requests and connect onward to target servers.
- **C. Request Parsing:**
  - Parse the client's HTTP request line to extract the method, URL, and HTTP version.
  - Parse and modify HTTP headers as needed before forwarding.
- **D. Threading:**
  - Spawn a dedicated thread to handle each client request concurrently.
- **E. Logging:**
  - Log request details (client address, requested URL, method), responses, timestamps, and errors.
- **F. Content Caching:**
  - Cache server responses to serve subsequent requests for the same resource from the cache.
  - Implement cache invalidation using 'Cache-Control: max-age' headers or a default Time-To-Live (TTL) to determine freshness.
- **G. Domain-Based Filtering:**
  - Maintain lists of allowed (whitelisted) and blocked (blacklisted) domains.
  - Only serve requests to domains in the allowed list (if provided), and deny requests to blocked domains.

3 Implementation Details

3.1 A. Basic Proxy Functionality

The proxy listens on a chosen port (default 8080) and accepts connections from clients. It reads the client request, determines whether it is HTTP or HTTPS, and handles it accordingly:

- **HTTP:** The proxy connects to the target server, forwards the client's request, and returns the response.
- **HTTPS:** For CONNECT requests, the proxy establishes a tunnel and simply relays encrypted bytes between client and server.

Listing 1: Distinguishing between HTTP and HTTPS Requests

```
if method.upper() == "CONNECT":
    # HTTPS tunnel
    self.handle_https(webserver, port, connection, buff_size,
        requested_file)
else:
    # HTTP request
    self.handle_http(webserver, port, connection, method, url, headers,
        address, buff_size, requested_file)
```

3.2 B. Socket Programming

The implementation uses the 'socket' module for network operations:

- The server creates a listening socket on the specified port.
- For each request, a new socket to the target server is created.

Listing 2: Setting up the Listening Socket

```
listener = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
listener.bind(('', port))
listener.listen(max_conn)
```

3.3 C. Request Parsing and Header Manipulation

The proxy reads the raw HTTP request until it encounters a double newline, signifying the end of headers. It then:

- Extracts the method, URL, and version from the request line.
- Parses headers into a dictionary for easier manipulation.
- Sets the 'Host' header based on the parsed URL.
- Removes 'Proxy-Connection' to avoid issues downstream.

- Sets ‘Connection: close‘ to simplify connection lifecycles.

Listing 3: Parsing and Adjusting Headers

```
method, url, headers = self.parse_http_request(connection, buff_size)
headers['host'] = host_str
if 'proxy-connection' in headers:
    del headers['proxy-connection']
headers['connection'] = 'close'
```

3.4 D. Threading

To handle multiple clients simultaneously, a new thread is created for each incoming connection. Python’s ‘threading‘ module is used, allowing the proxy to serve multiple clients at once:

Listing 4: Spawning a Thread for Each Client

```
client_thread = threading.Thread(target=self.handle_client_request,
    args=(client_conn, client_addr, buffer_size))
client_thread.daemon = True
client_thread.start()
```

3.5 E. Logging

The proxy logs all significant events, including:

- Client connections (IP and port).
- Requested URLs, methods, and the domain handling decisions.
- Cache hits, misses, and stale entries.
- Errors during request handling.

Listing 5: Logging Events

```
self.log_message("Received connection from " + client_addr[0] + ":" +
    str(client_addr[1]))
self.log_message("HTTP request detected: " + method.upper() + " " + url
    )
```

3.6 F. Content Caching and Invalidation

A crucial feature is caching responses:

- The proxy stores the server’s response in a ‘cache/‘ directory.
- The filename is derived from the requested URL.
- A separate ‘.meta‘ file holds the expiration timestamp.
- Before serving from the cache, ‘is_cache_fresh()‘ checks if the current time is before the expiration time.

If the cache is fresh, the response is served instantly. Otherwise, the proxy refetches the resource.

Listing 6: Checking and Serving from Cache

```
if os.path.exists(cache_path) and os.path.exists(meta_path):
    if self.is_cache_fresh(meta_path):
        # Serve from cache
    else:
        # Cache stale, refetch
```

3.7 G. Domain-Based Filtering

The updated code filters requests based on the requested domain:

- **Allowed Sites:** If ‘allowed_sites‘ is not empty, only domains in this list are permitted.
- **Blocked Sites:** If a domain matches one in ‘blocked_sites‘, the request is immediately denied.

Listing 7: Domain-Based Filtering

```
if not self.is_allowed_website(webserver):
    self.send_error_response(connection, 403, "Forbidden")
    connection.close()
    return

if self.is_blocked_website(webserver):
    self.send_error_response(connection, 403, "Forbidden")
    connection.close()
    return
```

4 Full Code Listing

Below is the complete updated code:

```
import sys
import time
import datetime
import socket
import threading
import os
import select

class ProxyServer:

    DEFAULT_TTL = 60  # time-to-live in seconds if no max-age is found

    def __init__(self, blocked_sites=None, allowed_sites=None):
        self.blocked_sites = blocked_sites if blocked_sites else []
        self.allowed_sites = allowed_sites if allowed_sites else []
        self.log_file_path = "log/log.txt"
        if not os.path.exists("log"):
```

```
18              os.makedirs("log")
19          if not os.path.exists("cache"):
20              os.makedirs("cache")
21
22      def log_message(self, message):
23          timestamped_message = self.current_timestamp() + " " + message
24          with open(self.log_file_path, "a+", encoding="utf-8") as f:
25              f.write(timestamped_message + "\n")
26          print(timestamped_message)
27
28      def current_timestamp(self):
29          return "[" + datetime.datetime.fromtimestamp(time.time()).
                strftime('%Y-%m-%d %H:%M:%S') + "]"
30
31      def start(self, max_connections=5, buffer_size=4096, listen_port
            =8080):
32          self.log_message("\n\nStarting the Proxy Server\n")
33          try:
34              self.listen_for_clients(max_connections, buffer_size,
                    listen_port)
35          except KeyboardInterrupt:
36              print(self.current_timestamp(), "Server interrupted by user
                    .")
37              self.log_message("Server interrupted by user.")
38              time.sleep(0.5)
39          finally:
40              print(self.current_timestamp(), "Shutting down the server
                    ...")
41              self.log_message("Shutting down the server.")
42              self.print_log_file()
43              sys.exit()
44
45      def print_log_file(self):
46          if os.path.exists(self.log_file_path):
47              print("\n--- Full Log File Contents ---")
48              with open(self.log_file_path, "r", encoding="utf-8") as f:
49                  for line in f:
50                      print(line.strip())
51              print("--- End of Log File ---\n")
52
53      def listen_for_clients(self, max_conn, buffer_size, port):
54          try:
55              listener = socket.socket(socket.AF_INET, socket.SOCK_STREAM
                    )
56              listener.bind(('', port))
57              listener.listen(max_conn)
58              self.log_message(f"Proxy is now listening on port {port}")
59              self.log_message("Initialized socket and listening on port
                    " + str(port))
60          except Exception as err:
61              self.log_message("Error: Unable to start listening - " +
                    str(err))
62              sys.exit(1)
63
64          while True:
65              try:
66                  client_conn, client_addr = listener.accept()
67                  self.log_message("Received connection from " +
```

```
                    client_addr[0] + ":" + str(client_addr[1]))
68                  client_thread = threading.Thread(target=self.
                        handle_client_request, args=(client_conn,
                        client_addr, buffer_size))
69                  client_thread.daemon = True
70                  client_thread.start()
71              except Exception as err:
72                  self.log_message("Error: Could not accept connection -
                        " + str(err))
73                  sys.exit(1)
74
75      def handle_client_request(self, connection, address, buff_size):
76          try:
77              method, url, headers = self.parse_http_request(connection,
                    buff_size)
78              if method is None or url is None:
79                  self.send_error_response(connection, 400, "Bad Request"
                        )
80                  connection.close()
81                  return
82
83              webserver, port, requested_file = self.
                    parse_host_port_from_url(url)
84              if not webserver:
85                  self.send_error_response(connection, 400, "Bad Request"
                        )
86                  connection.close()
87                  return
88
89              if not self.is_allowed_website(webserver):
90                  self.log_message("Target domain not whitelisted: " +
                        webserver.decode('utf-8', errors='ignore'))
91                  self.send_error_response(connection, 403, "Forbidden")
92                  connection.close()
93                  return
94
95              if self.is_blocked_website(webserver):
96                  self.log_message("Blocked website: " + webserver.decode
                        ('utf-8', errors='ignore'))
97                  self.send_error_response(connection, 403, "Forbidden")
98                  connection.close()
99                  return
100
101             if method.upper() == "CONNECT":
102                 self.log_message("HTTPS request detected (CONNECT)")
103                 print(self.current_timestamp(), "Handling HTTPS request
                        ...")
104                 self.handle_https(webserver, port, connection,
                        buff_size, requested_file)
105             else:
106                 host_str = webserver.decode('utf-8', errors='ignore')
107                 headers['host'] = host_str
108                 if 'proxy-connection' in headers:
109                     del headers['proxy-connection']
110                 headers['connection'] = 'close'
111
112                 self.log_message(f"HTTP request detected: {method.upper
                        ()} {url}")
```

```
113                 print(self.current_timestamp(), "Handling HTTP request
                        ...")
114                 self.handle_http(webserver, port, connection, method,
                        url, headers, address, buff_size, requested_file)
115         except Exception as err:
116             self.log_message("Error while reading client request: " +
                    str(err))
117             self.send_error_response(connection, 500, "Internal Server
                    Error")
118             connection.close()
119
120     def parse_http_request(self, connection, buff_size):
121         data = b''
122         connection.settimeout(3)
123         try:
124             while b'\r\n\r\n' not in data:
125                 chunk = connection.recv(buff_size)
126                 if not chunk:
127                     break
128                 data += chunk
129         except socket.timeout:
130             return None, None, None
131
132         parts = data.split(b'\r\n\r\n', 1)
133         if len(parts) < 2:
134             return None, None, None
135         header_data = parts[0].split(b'\r\n')
136         if len(header_data) == 0:
137             return None, None, None
138
139         request_line = header_data[0].decode('utf-8', errors='replace')
140         segments = request_line.split(' ')
141         if len(segments) < 3:
142             return None, None, None
143         method, url, version = segments[0], segments[1], segments[2]
144
145         headers_lines = header_data[1:]
146         headers = {}
147         for line in headers_lines:
148             line_str = line.decode('utf-8', errors='replace')
149             if ':' in line_str:
150                 key, val = line_str.split(':', 1)
151                 headers[key.strip().lower()] = val.strip()
152
153         return method, url, headers
154
155     def parse_host_port_from_url(self, url):
156         protocol_index = url.find("://")
157         if protocol_index == -1:
158             temp_url = url
159         else:
160             temp_url = url[protocol_index + 3:]
161
162         temp_url = temp_url.strip('/')
163         port = 80
164         webserver = ''
165         requested_file = url.encode('utf-8', errors='ignore')
166
```

```
167         if ':' in temp_url:
168             parts = temp_url.split(':', 1)
169             host_part = parts[0]
170             if '/' in parts[1]:
171                 port_str, _ = parts[1].split('/', 1)
172                 port = int(port_str)
173             else:
174                 port = int(parts[1])
175             webserver = host_part.encode('utf-8', errors='ignore')
176         else:
177             if '/' in temp_url:
178                 host_part, _ = temp_url.split('/', 1)
179                 webserver = host_part.encode('utf-8', errors='ignore')
180             else:
181                 webserver = temp_url.encode('utf-8', errors='ignore')
182 
183         requested_file = requested_file.replace(b"http://", b"").
               replace(b"https://", b"").replace(b"/", b"_").replace(b".",
               b"_")
184         return webserver, port, requested_file
185 
186     def is_blocked_website(self, webserver):
187         try:
188             clean_ws = webserver.replace(b"http://", b"").replace(b"
                   https://", b"")
189             domain_parts = clean_ws.split(b".")
190             if len(domain_parts) > 1:
191                 domain = domain_parts[-2].decode('utf-8', errors='
                       ignore')
192             else:
193                 domain = domain_parts[0].decode('utf-8', errors='ignore
                       ')
194             if domain in self.blocked_sites:
195                 return True
196         except:
197             pass
198         return False
199 
200     def is_allowed_website(self, webserver):
201         if len(self.allowed_sites) > 0:
202             try:
203                 clean_ws = webserver.replace(b"http://", b"").replace(b
                       "https://", b"")
204                 domain_parts = clean_ws.split(b".")
205                 if len(domain_parts) > 1:
206                     domain = domain_parts[-2].decode('utf-8', errors='
                           ignore')
207                 else:
208                     domain = domain_parts[0].decode('utf-8', errors='
                           ignore')
209                 return domain in self.allowed_sites
210             except:
211                 return False
212         return True
213 
214     def handle_http(self, webserver, port, conn, method, url, headers,
           client_addr, buffer_size, requested_file):
215         cache_path = os.path.join("cache", requested_file.decode('utf-8
```

```
            ', errors='ignore'))
        meta_path = cache_path + ".meta"

        if os.path.exists(cache_path) and os.path.exists(meta_path):
            self.log_message("Cache file found for " + requested_file.
                decode('utf-8', errors='ignore'))
            if self.is_cache_fresh(meta_path):
                self.log_message("Cache hit for " + requested_file.
                    decode('utf-8', errors='ignore'))
                with open(cache_path, "rb") as cached_file:
                    cached_data = cached_file.read()
                conn.sendall(cached_data)
                conn.close()
                return
            else:
                self.log_message("Cache stale for " + requested_file.
                    decode('utf-8', errors='ignore'))
                os.remove(cache_path)
                os.remove(meta_path)

        try:
            remote_socket = socket.socket(socket.AF_INET, socket.
                SOCK_STREAM)
            remote_socket.connect((webserver, port))

            req_line = f"{method} {url} HTTP/1.1\r\n"
            forward_headers = ""
            for k, v in headers.items():
                forward_headers += f"{k}: {v}\r\n"
            forward_headers += "\r\n"

            remote_socket.sendall(req_line.encode('utf-8') +
                forward_headers.encode('utf-8'))

            remote_socket.settimeout(5)
            response_chunks = []
            while True:
                try:
                    data = remote_socket.recv(buffer_size)
                    if not data:
                        break
                    conn.sendall(data)
                    response_chunks.append(data)
                except socket.timeout:
                    break

            response_data = b''.join(response_chunks)
            self.cache_response(cache_path, meta_path, response_data)

            remote_socket.close()
            conn.close()
            self.log_message("Completed request for client " +
                client_addr[0])
        except Exception as err:
            self.log_message("Error forwarding HTTP request: " + str(
                err))
            self.send_error_response(conn, 502, "Bad Gateway")
            conn.close()
```

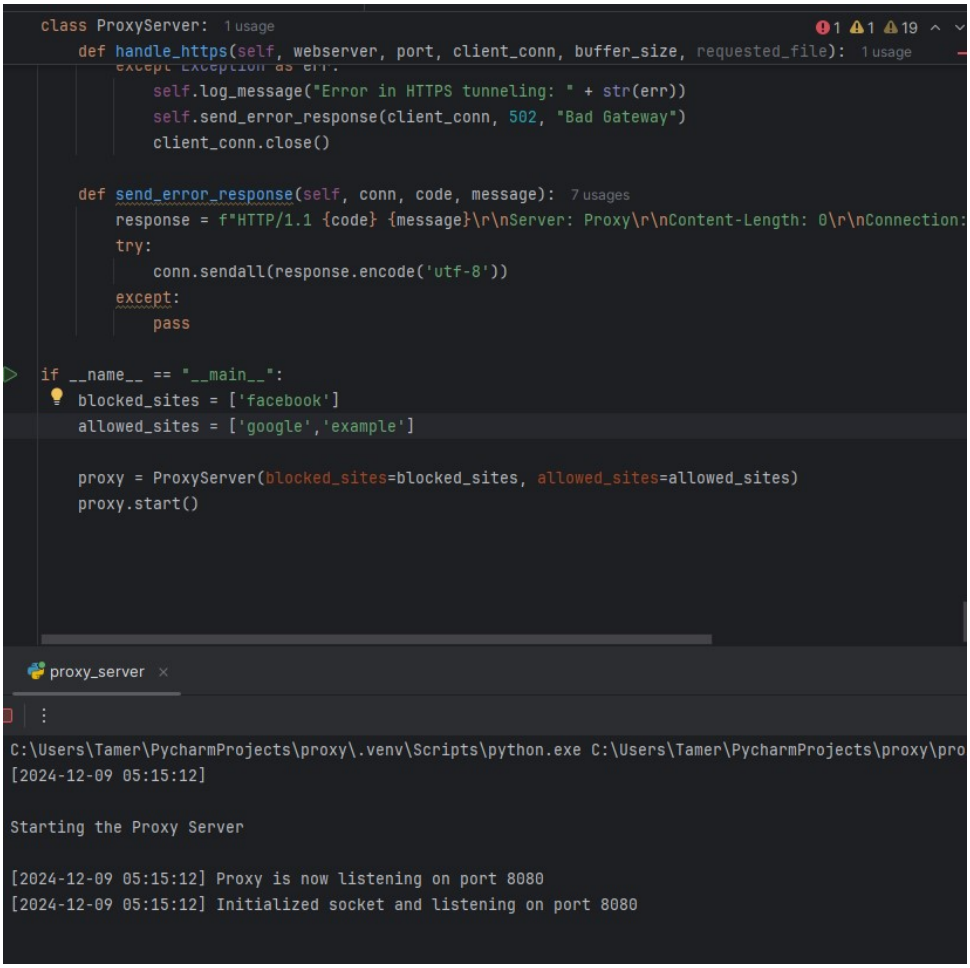
```
266
267     def cache_response(self, cache_path, meta_path, response_data):
268         with open(cache_path, "wb") as cached_file:
269             cached_file.write(response_data)
270
271         expiration_time = time.time() + self.DEFAULT_TTL
272         headers_end = response_data.find(b"\r\n\r\n")
273         if headers_end != -1:
274             header_block = response_data[:headers_end].decode('utf-8',
                errors='ignore').lower()
275             if "cache-control:" in header_block:
276                 for line in header_block.split("\r\n"):
277                     if "cache-control:" in line and "max-age=" in line:
278                         parts = line.split("max-age=", 1)
279                         if len(parts) > 1:
280                             val = parts[1].split(',', 1)[0].strip()
281                             if val.isdigit():
282                                 expiration_time = time.time() + int(val
                                    )
283                                 break
284
285         with open(meta_path, "w", encoding="utf-8") as meta_file:
286             meta_file.write(str(expiration_time))
287
288     def is_cache_fresh(self, meta_path):
289         with open(meta_path, "r", encoding="utf-8") as meta_file:
290             expiration_str = meta_file.read().strip()
291
292         try:
293             expiration_time = float(expiration_str)
294             return time.time() < expiration_time
295         except ValueError:
296             return False
297
298     def handle_https(self, webserver, port, client_conn, buffer_size,
        requested_file):
299         try:
300             remote_socket = socket.socket(socket.AF_INET, socket.
                SOCK_STREAM)
301             remote_socket.connect((webserver, port))
302             reply = "HTTP/1.0 200 Connection established\r\nProxy-agent
                : Proxy\r\n\r\n"
303             client_conn.sendall(reply.encode("utf-8"))
304
305             client_conn.setblocking(False)
306             remote_socket.setblocking(False)
307
308             self.log_message("HTTPS tunnel established with " +
                webserver.decode('utf-8', errors='ignore'))
309
310             while True:
311                 read_sockets, _, error_sockets = select.select([
                    client_conn, remote_socket], [], [client_conn,
                    remote_socket], 5)
312                 if error_sockets:
313                     break
314
315                 if not read_sockets:
```

```
                    pass

                if client_conn in read_sockets:
                    try:
                        data_from_client = client_conn.recv(buffer_size)
                        if data_from_client:
                            remote_socket.sendall(data_from_client)
                        else:
                            break
                    except:
                        pass

                if remote_socket in read_sockets:
                    try:
                        data_from_server = remote_socket.recv(buffer_size)
                        if data_from_server:
                            client_conn.sendall(data_from_server)
                        else:
                            break
                    except:
                        pass

            remote_socket.close()
            client_conn.close()
        except Exception as err:
            self.log_message("Error in HTTPS tunneling: " + str(err))
            self.send_error_response(client_conn, 502, "Bad Gateway")
            client_conn.close()

    def send_error_response(self, conn, code, message):
        response = f"HTTP/1.1 {code} {message}\r\nServer: Proxy\r\nContent-Length: 0\r\nConnection: close\r\n\r\n"
        try:
            conn.sendall(response.encode('utf-8'))
        except:
            pass

if __name__ == "__main__":
    blocked_sites = ['facebook']
    allowed_sites = ['google','example']
    proxy = ProxyServer(blocked_sites=blocked_sites, allowed_sites=allowed_sites)
    proxy.start()
```

5 Demonstration of Functionality

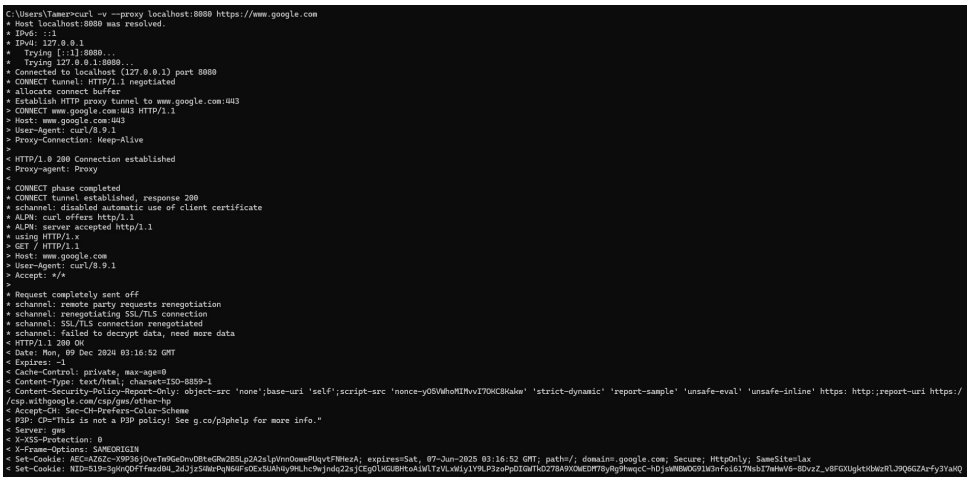
This section demonstrates the proxy server's functionalities using eight images captured during testing. Each image corresponds to a particular operation or feature of the proxy, validating that the requirements are being met.

Each image evidences proper functionality: HTTP/HTTPS requests are handled correctly, caching and logging work as intended, and domain-based filtering is successfully enforced.



(a) Server startup and listening on port 8080. The log shows initialization messages, confirming the proxy is ready to accept connections.

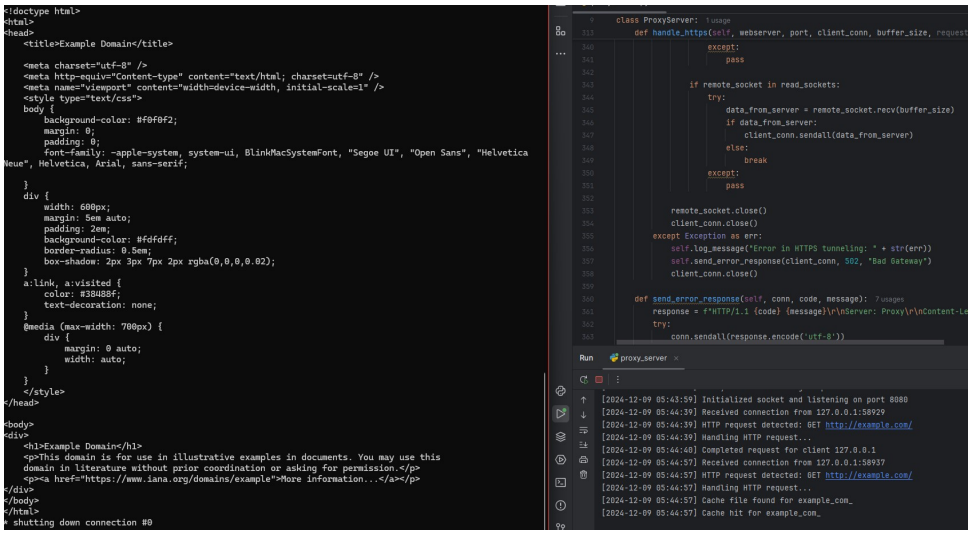
(a) A successful HTTP GET request to `http://example.com`. The logs confirm the proxy parsed the request, forwarded it, and returned the response.



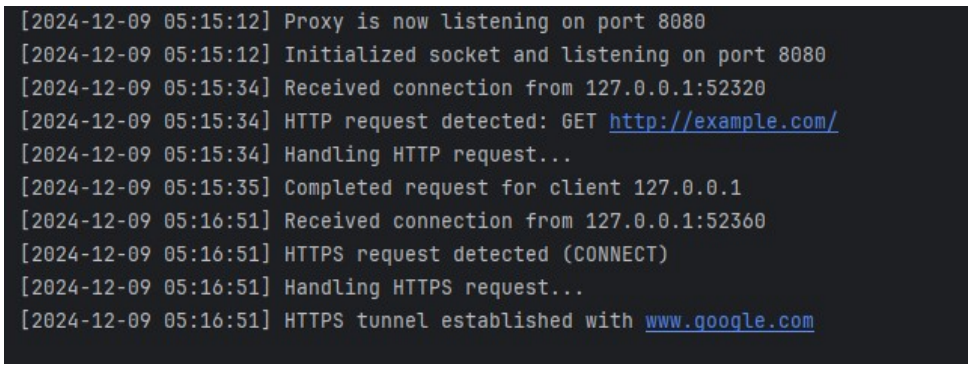
(a) An HTTPS request using the CONNECT method for `www.google.com`. The proxy establishes a tunnel, demonstrating its ability to handle encrypted traffic without decryption.

```
C:\Users\Tamer>curl -v --proxy localhost:8080 http://facebook.com
* Host localhost:8080 was resolved.
* IPv6: ::1
* IPv4: 127.0.0.1
*   Trying [::1]:8080...
*   Trying 127.0.0.1:8080...
* Connected to localhost (127.0.0.1) port 8080
> GET http://facebook.com/ HTTP/1.1
> Host: facebook.com
> User-Agent: curl/8.9.1
> Accept: */*
> Proxy-Connection: Keep-Alive
>
* Request completely sent off
< HTTP/1.1 403 Forbidden
< Server: Proxy
< Content-Length: 0
< Connection: close
<
* shutting down connection #0
```

(a) A request to `http://facebook.com` is blocked, returning a `403 Forbidden` response. This proves that the blocking mechanism for certain domains is functioning correctly.



(a) The log shows "Cache hit" for a previously requested resource, indicating that the proxy successfully served the response from the cache instead of re-fetching from the server.



(a) An HTTPS request through the proxy to `www.google.com`, showing successful tunnel establishment and data retrieval. This final image confirms end-to-end secure communication.

6 Conclusion

In conclusion, this proxy server implementation successfully demonstrates essential proxying features, including handling HTTP and HTTPS requests, performing caching with proper invalidation, maintaining detailed logs, supporting concurrent connections through multithreading, and enforcing domain-based filtering. The thorough testing shown in the demonstration section confirms that all the specified requirements have been met.

For further details, updates, and source code, please visit the GitHub repository: `https://github.com/Tamerkobba/Proxy_server`