



# Proxy Server

Submitted by: Tamer Kobba

Student ID(s): 202104873

Instructor's Name: Ayman Tajeddine

December 9, 2024

## Contents

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Introduction</b>                                  | <b>2</b>  |
| <b>2</b> | <b>Requirements</b>                                  | <b>2</b>  |
| <b>3</b> | <b>Implementation Details</b>                        | <b>3</b>  |
| 3.1      | A. Basic Proxy Functionality . . . . .               | 3         |
| 3.2      | B. Socket Programming . . . . .                      | 3         |
| 3.3      | C. Request Parsing and Header Manipulation . . . . . | 3         |
| 3.4      | D. Threading . . . . .                               | 4         |
| 3.5      | E. Logging . . . . .                                 | 4         |
| 3.6      | F. Content Caching and Invalidation . . . . .        | 4         |
| 3.7      | G. Domain-Based Filtering . . . . .                  | 5         |
| <b>4</b> | <b>Full Code Listing</b>                             | <b>5</b>  |
| <b>5</b> | <b>Demonstration of Functionality</b>                | <b>12</b> |
| <b>6</b> | <b>Conclusion</b>                                    | <b>17</b> |

# 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

```
1 if method.upper() == "CONNECT":
2     # HTTPS tunnel
3     self.handle_https(webserver, port, connection, buff_size,
4         requested_file)
5 else:
6     # HTTP request
7     self.handle_http(webserver, port, connection, method, url, headers,
8         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

```
1 listener = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
2 listener.bind(('', port))
3 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

```

1 method, url, headers = self.parse_http_request(connection, buff_size)
2 headers['host'] = host_str
3 if 'proxy-connection' in headers:
4     del headers['proxy-connection']
5 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

```

1 client_thread = threading.Thread(target=self.handle_client_request,
2     args=(client_conn, client_addr, buffer_size))
3 client_thread.daemon = True
4 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

```

1 self.log_message("Received connection from " + client_addr[0] + ":" +
2     str(client_addr[1]))
3 self.log_message("HTTP request detected: " + method.upper() + " " + url
4     )

```

### 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

```
1 if os.path.exists(cache_path) and os.path.exists(meta_path):
2     if self.is_cache_fresh(meta_path):
3         # Serve from cache
4     else:
5         # 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

```
1 if not self.is_allowed_website(webserver):
2     self.send_error_response(connection, 403, "Forbidden")
3     connection.close()
4     return
5
6 if self.is_blocked_website(webserver):
7     self.send_error_response(connection, 403, "Forbidden")
8     connection.close()
9     return
```

## 4 Full Code Listing

Below is the complete updated code:

```
1 import sys
2 import time
3 import datetime
4 import socket
5 import threading
6 import os
7 import select
8
9 class ProxyServer:
10
11     DEFAULT_TTL = 60 # time-to-live in seconds if no max-age is found
12
13     def __init__(self, blocked_sites=None, allowed_sites=None):
14         self.blocked_sites = blocked_sites if blocked_sites else []
15         self.allowed_sites = allowed_sites if allowed_sites else []
16         self.log_file_path = "log/log.txt"
17         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()).
30             strftime('%Y-%m-%d %H:%M:%S') + "]"
31
32     def start(self, max_connections=5, buffer_size=4096, listen_port
33         =8080):
34         self.log_message("\n\nStarting the Proxy Server\n")
35         try:
36             self.listen_for_clients(max_connections, buffer_size,
37                                     listen_port)
38         except KeyboardInterrupt:
39             print(self.current_timestamp(), "Server interrupted by user
40                 .")
41             self.log_message("Server interrupted by user.")
42             time.sleep(0.5)
43         finally:
44             print(self.current_timestamp(), "Shutting down the server
45                 ...")
46             self.log_message("Shutting down the server.")
47             self.print_log_file()
48             sys.exit()
49
50     def print_log_file(self):
51         if os.path.exists(self.log_file_path):
52             print("\n--- Full Log File Contents ---")
53             with open(self.log_file_path, "r", encoding="utf-8") as f:
54                 for line in f:
55                     print(line.strip())
56             print("--- End of Log File ---\n")
57
58     def listen_for_clients(self, max_conn, buffer_size, port):
59         try:
60             listener = socket.socket(socket.AF_INET, socket.SOCK_STREAM
61                                     )
62             listener.bind(('', port))
63             listener.listen(max_conn)
64             self.log_message(f"Proxy is now listening on port {port}")
65             self.log_message("Initialized socket and listening on port
66                             " + str(port))
67         except Exception as err:
68             self.log_message("Error: Unable to start listening - " +
69                             str(err))
70             sys.exit(1)
71
72     while True:
73         try:
74             client_conn, client_addr = listener.accept()
75             self.log_message("Received connection from " +

```

```

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

```



```

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 "").
184     replace(b"https://", b "").replace(b"/", b "_").replace(b".",
185     b "_")
186     return webserver, port, requested_file
187
188 def is_blocked_website(self, webserver):
189     try:
190         clean_ws = webserver.replace(b"http://", b "").replace(b"
191         https://", b "")
192         domain_parts = clean_ws.split(b".")
193         if len(domain_parts) > 1:
194             domain = domain_parts[-2].decode('utf-8', errors='
195             ignore')
196         else:
197             domain = domain_parts[0].decode('utf-8', errors='ignore
198             ')
199         if domain in self.blocked_sites:
200             return True
201     except:
202         pass
203     return False
204
205 def is_allowed_website(self, webserver):
206     if len(self.allowed_sites) > 0:
207         try:
208             clean_ws = webserver.replace(b"http://", b "").replace(b
209             "https://", b "")
210             domain_parts = clean_ws.split(b".")
211             if len(domain_parts) > 1:
212                 domain = domain_parts[-2].decode('utf-8', errors='
213                 ignore')
214             else:
215                 domain = domain_parts[0].decode('utf-8', errors='
216                 ignore')
217             return domain in self.allowed_sites
218         except:
219             return False
220     return True
221
222 def handle_http(self, webserver, port, conn, method, url, headers,
223 client_addr, buffer_size, requested_file):
224     cache_path = os.path.join("cache", requested_file.decode('utf-8

```

```

216         ', errors='ignore'))
217     meta_path = cache_path + ".meta"
218
219     if os.path.exists(cache_path) and os.path.exists(meta_path):
220         self.log_message("Cache file found for " + requested_file.
221             decode('utf-8', errors='ignore'))
222         if self.is_cache_fresh(meta_path):
223             self.log_message("Cache hit for " + requested_file.
224                 decode('utf-8', errors='ignore'))
225             with open(cache_path, "rb") as cached_file:
226                 cached_data = cached_file.read()
227                 conn.sendall(cached_data)
228                 conn.close()
229                 return
230         else:
231             self.log_message("Cache stale for " + requested_file.
232                 decode('utf-8', errors='ignore'))
233             os.remove(cache_path)
234             os.remove(meta_path)
235
236     try:
237         remote_socket = socket.socket(socket.AF_INET, socket.
238             SOCK_STREAM)
239         remote_socket.connect((webserver, port))
240
241         req_line = f"{method} {url} HTTP/1.1\r\n"
242         forward_headers = ""
243         for k, v in headers.items():
244             forward_headers += f"{k}: {v}\r\n"
245         forward_headers += "\r\n"
246
247         remote_socket.sendall(req_line.encode('utf-8') +
248             forward_headers.encode('utf-8'))
249
250         remote_socket.settimeout(5)
251         response_chunks = []
252         while True:
253             try:
254                 data = remote_socket.recv(buffer_size)
255                 if not data:
256                     break
257                 conn.sendall(data)
258                 response_chunks.append(data)
259             except socket.timeout:
260                 break
261
262         response_data = b''.join(response_chunks)
263         self.cache_response(cache_path, meta_path, response_data)
264
265         remote_socket.close()
266         conn.close()
267         self.log_message("Completed request for client " +
268             client_addr[0])
269     except Exception as err:
270         self.log_message("Error forwarding HTTP request: " + str(
271             err))
272         self.send_error_response(conn, 502, "Bad Gateway")
273         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',
275             errors='ignore').lower()
276         if "cache-control:" in header_block:
277             for line in header_block.split("\r\n"):
278                 if "cache-control:" in line and "max-age=" in line:
279                     parts = line.split("max-age=", 1)
280                     if len(parts) > 1:
281                         val = parts[1].split(',', 1)[0].strip()
282                         if val.isdigit():
283                             expiration_time = time.time() + int(val)
284                             break
285
286     with open(meta_path, "w", encoding="utf-8") as meta_file:
287         meta_file.write(str(expiration_time))
288
289 def is_cache_fresh(self, meta_path):
290     with open(meta_path, "r", encoding="utf-8") as meta_file:
291         expiration_str = meta_file.read().strip()
292
293     try:
294         expiration_time = float(expiration_str)
295         return time.time() < expiration_time
296     except ValueError:
297         return False
298
299 def handle_https(self, webserver, port, client_conn, buffer_size,
300     requested_file):
301     try:
302         remote_socket = socket.socket(socket.AF_INET, socket.
303             SOCK_STREAM)
304         remote_socket.connect((webserver, port))
305         reply = "HTTP/1.0 200 Connection established\r\nProxy-agent
306             : Proxy\r\n\r\n"
307         client_conn.sendall(reply.encode("utf-8"))
308
309         client_conn.setblocking(False)
310         remote_socket.setblocking(False)
311
312         self.log_message("HTTPS tunnel established with " +
313             webserver.decode('utf-8', errors='ignore'))
314
315         while True:
316             read_sockets, _, error_sockets = select.select([
317                 client_conn, remote_socket], [], [client_conn,
318                 remote_socket], 5)
319             if error_sockets:
320                 break
321
322             if not read_sockets:

```

```

316         pass
317
318     if client_conn in read_sockets:
319         try:
320             data_from_client = client_conn.recv(buffer_size
321             )
322             if data_from_client:
323                 remote_socket.sendall(data_from_client)
324             else:
325                 break
326         except:
327             pass
328
329     if remote_socket in read_sockets:
330         try:
331             data_from_server = remote_socket.recv(
332                 buffer_size)
333             if data_from_server:
334                 client_conn.sendall(data_from_server)
335             else:
336                 break
337         except:
338             pass
339
340     remote_socket.close()
341     client_conn.close()
342 except Exception as err:
343     self.log_message("Error in HTTPS tunneling: " + str(err))
344     self.send_error_response(client_conn, 502, "Bad Gateway")
345     client_conn.close()
346
347 def send_error_response(self, conn, code, message):
348     response = f"HTTP/1.1 {code} {message}\r\nServer: Proxy\r\
349     nContent-Length: 0\r\nConnection: close\r\n\r\n"
350     try:
351         conn.sendall(response.encode('utf-8'))
352     except:
353         pass
354
355 if __name__ == "__main__":
356     blocked_sites = ['facebook']
357     allowed_sites = ['google', 'example']
358     proxy = ProxyServer(blocked_sites=blocked_sites, allowed_sites=
359         allowed_sites)
360     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.

```
class ProxyServer: 1 usage
    def handle_https(self, webserver, port, client_conn, buffer_size, requested_file): 1 usage
        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): 7 usages
        response = f"HTTP/1.1 {code} {message}\r\nServer: Proxy\r\nContent-Length: 0\r\nConnection:
        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()
```

proxy\_server x

```
C:\Users\Tamer\PycharmProjects\proxy\.venv\Scripts\python.exe C:\Users\Tamer\PycharmProjects\proxy\pro
[2024-12-09 05:15:12]

Starting the Proxy Server

[2024-12-09 05:15:12] Proxy is now listening on port 8080
[2024-12-09 05:15:12] Initialized socket and listening on port 8080
```

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

```

C:\Users\Tamer>curl -v --proxy localhost:8888 http://example.com
* Host localhost:8888 was resolved.
* IP6s: :1
* IPv4: 127.0.0.1
* Trying [::1]:8888...
* Trying 127.0.0.1:8888...
* Connected to localhost (127.0.0.1) port 8888
> GET http://example.com/ HTTP/1.1
> Host: example.com
> User-Agent: curl/8.9.1
> Accept: */*
> Proxy-Connection: Keep-Alive
>
* Request completely sent off
< HTTP/1.1 200 OK
< Date: Mon, 09 Dec 2024 03:15:33 GMT
< Cache-Control: max-age=604800
< Content-Type: text/html; charset=utf-8
< Etag: "3147526947+gzip+ident"
< Last-Modified: Thu, 17 Oct 2019 07:18:26 GMT
< Server: ECAcc (dcd/7D55)
< Vary: Accept-Encoding
< X-Cache: HIT
< Content-Length: 1256
< Age: 0
< Expires: Mon, 16 Dec 2024 03:15:35 GMT
< Connection: close
<
<!doctype html>
<html>
<head>
  <title>Example Domain</title>

  <meta charset="utf-8" />
  <meta http-equiv="Content-type" content="text/html; charset=utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <style type="text/css">
    body {
      background-color: #f0f0f2;
      margin: 0;
      padding: 0;
      font-family: -apple-system, system-ui, BlinkMacSystemFont, "Segoe UI", "Open Sans", "Helvetica Neue", Helvetica, Arial, sans-serif;
    }
    div {
      width: 600px;
      margin: 5em auto;
      padding: 2em;
    }
  </style>
</head>
<body>
<div>
  <h1>Example Domain</h1>
  <p>This domain is for use in illustrative examples in documents. You may use this
  domain in literature without prior coordination or asking for permission.</p>
  <p><a href="https://www.iana.org/domains/example">More information...</a></p>
</div>
</body>
</html>
* shutting down connection #0

```

```

<title>Example Domain</title>

<meta charset="utf-8" />
<meta http-equiv="Content-type" content="text/html; charset=utf-8" />
<meta name="viewport" content="width=device-width, initial-scale=1" />
<style type="text/css">
  body {
    background-color: #f0f0f2;
    margin: 0;
    padding: 0;
    font-family: -apple-system, system-ui, BlinkMacSystemFont, "Segoe UI", "Open Sans", "Helvetica Neue", Helvetica, Arial, sans-serif;
  }
  div {
    width: 600px;
    margin: 5em auto;
    padding: 2em;
    background-color: #fdfdff;
    border-radius: 0.5em;
    box-shadow: 2px 3px 7px rgba(0,0,0,0.02);
  }
  a:link, a:visited {
    color: #38488f;
    text-decoration: none;
  }
  @media (max-width: 700px) {
    div {
      margin: 0 auto;
      width: auto;
    }
  }
</style>
</head>
<body>
<div>
  <h1>Example Domain</h1>
  <p>This domain is for use in illustrative examples in documents. You may use this
  domain in literature without prior coordination or asking for permission.</p>
  <p><a href="https://www.iana.org/domains/example">More information...</a></p>
</div>
</body>
</html>

```

```

class ProxyServer:
    def __init__(self, webserver, port, client_conn, buffer_size, response):
        self.webserver = webserver
        self.port = port
        self.client_conn = client_conn
        self.buffer_size = buffer_size
        self.response = response

    def handle_https(self, self, webserver, port, client_conn, buffer_size, response):
        try:
            if remote_socket in read_sockets:
                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}"
            try:
                conn.sendall(response.encode('utf-8'))
            except:
                pass

```

```

[2024-12-09 05:43:59] Starting the Proxy Server
[2024-12-09 05:43:59] Proxy is now listening on port 8888
[2024-12-09 05:44:39] Received connection from 127.0.0.1:58929
[2024-12-09 05:44:39] HTTP request detected: GET http://example.com/
[2024-12-09 05:44:39] Handling HTTP request...
[2024-12-09 05:44:40] Completed request for client 127.0.0.1

```

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

```

C:\Users\Tamer>curl -v --proxy localhost:8080 https://www.google.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
* CONNECT tunnel: HTTP/1.1 negotiated
* Allocate connect buffer
* Establish HTTP proxy tunnel to www.google.com:443
* CONNECT www.google.com:443 HTTP/1.1
> Host: www.google.com:443
> User-Agent: curl/8.9.1
> Proxy-Connection: Keep-Alive
* HTTP/1.0 200 Connection established
* Proxy-agent: Proxy
*
* CONNECT phase completed
* CONNECT tunnel established, response 200
* schannel: disabled automatic use of client certificate
* ALPN: curl offers http/1.1
* ALPN: server accepted http/1.1
> GET / HTTP/1.1
> Host: www.google.com
> User-Agent: curl/8.9.1
> Accept: */*
*
* Request completely sent off
* schannel: remote party requests renegotiation
* schannel: renegotiating SSL/TLS connection
* schannel: SSL/TLS connection renegotiated
* schannel: failed to decrypt data, need more data
* HTTP/1.1 200 OK
* Date: Mon, 09 Dec 2024 03:16:52 GMT
* Expires: -1
* Cache-Control: private, max-age=0
* Content-Type: text/html; charset=ISO-8859-1
* Content-Security-Policy-Report-Only: object-src 'none';base-uri 'self';script-src 'nonce-y08Vh0mI1Nv170KChKaw' 'strict-dynamic' 'report-sample' 'unsafe-eval' 'unsafe-inline' https: http:report-uri https://csp.withgoogle.com/csp/gms/other-hp
* Accept-CH: Sec-CH-Prefers-Color-Scheme
* PSP: CH="This is not a PSP policy! See g.co/p3phelp for more info."
* Server: gse
* X-XSS-Protection: 0
* X-Frame-Options: SAMEORIGIN
* Set-Cookie: SEC_AZFC=V08933J0veT0G6d0v0BteGbuZBSLg242a1pVnm0owP4vtfFMe2A; expires=Sat, 07-Jun-2026 03:16:52 GMT; path=/; domain=.google.com; Secure; HttpOnly; SameSite=Lax
* Set-Cookie: NID=519=3gkHQDFfzad9L_2d3j5S4McPqN64F4dEX5U4Nhy9Hhc9wJndq22zjCEgDl4GUBHt0kUMTzVLwiy1YPLP3z0pD1GHTK0278A9X0MED778y9hmqcC-HdjsaNBMOG91M3F0s1617Nob17mWwV6-BDvz2_v8FDXUgkth4WzRL39Q6GZArfy3YwQ

```

(a) An HTTPS request using the CONNECT method for [www.google.com](https://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.

```

<doctype html>
<html>
<head>
<title>Example Domain</title>

<meta charset="utf-8" />
<meta http-equiv="Content-type" content="text/html; charset=utf-8" />
<meta name="viewport" content="width=device-width, initial-scale=1" />
<style type="text/css">
body {
background-color: #f9f9f2;
margin: 0;
padding: 0;
font-family: -apple-system, system-ui, BlinkMacSystemFont, "Segoe UI", "Open Sans", "Helvetica Neue", Helvetica, Arial, sans-serif;
}
div {
width: 600px;
margin: 5em auto;
padding: 2em;
background-color: #fdfdff;
border-radius: 0.5em;
box-shadow: 2px 3px 7px 2px rgba(0,0,0,0.02);
}
a:link, a:visited {
color: #28488f;
text-decoration: none;
}
@media (max-width: 700px) {
div {
margin: 0 auto;
width: auto;
}
}
</style>
</head>
<body>
<div>
<h1>Example Domain</h1>
<p>This domain is for use in illustrative examples in documents. You may use this domain in literature without prior coordination or asking for permission.</p>
<p><a href="https://www.iana.org/domains/example">More information...</a></p>
</div>
</body>
</html>
shutting down connection #0

```

```

class ProxyServer:
    def __init__(self, webserver, port, client_conn, buffer_size, request):
        self.webserver = webserver
        self.port = port
        self.client_conn = client_conn
        self.buffer_size = buffer_size
        self.request = request

    def handle_https(self):
        try:
            pass
        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)
            except:
                break
        else:
            pass

        remote_socket.close()
        client_conn.close()

    except Exception as e:
        self.log_message("Error in HTTPS tunneling: " + str(e))
        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}"
        conn.sendall(response.encode("utf-8"))

```

```

Run proxy_server
[2024-12-09 05:43:59] Initialized socket and listening on port 8080
[2024-12-09 05:44:39] Received connection from 127.0.0.1:58920
[2024-12-09 05:44:39] HTTP request detected: GET http://example.com/
[2024-12-09 05:44:39] Handling HTTP request...
[2024-12-09 05:44:40] Completed request for client 127.0.0.1
[2024-12-09 05:44:57] Received connection from 127.0.0.1:58937
[2024-12-09 05:44:57] HTTP request detected: GET http://example.com/
[2024-12-09 05:44:57] Handling HTTP request...
[2024-12-09 05:44:57] Cache file found for example.com
[2024-12-09 05:44:57] Cache hit for example.com

```

(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.

```

[2024-12-09 05:15:12] Proxy is now listening on port 8080
[2024-12-09 05:15:12] Initialized socket and listening on port 8080
[2024-12-09 05:15:34] Received connection from 127.0.0.1:52320
[2024-12-09 05:15:34] HTTP request detected: GET http://example.com/
[2024-12-09 05:15:34] Handling HTTP request...
[2024-12-09 05:15:35] Completed request for client 127.0.0.1
[2024-12-09 05:16:51] Received connection from 127.0.0.1:52360
[2024-12-09 05:16:51] HTTPS request detected (CONNECT)
[2024-12-09 05:16:51] Handling HTTPS request...
[2024-12-09 05:16:51] HTTPS tunnel established with www.google.com

```

(a) An HTTPS request through the proxy to [www.google.com](https://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](https://github.com/Tamerkobba/Proxy_server)