Requirement 1:

The handling of HTTP and HTTPS requests by my program is encompassed by the request function. Whenever a new connection is made, the request function is called.

It begins by displaying the new request on the management console. It then initialises the destination socket, which will be bound to an IP and socket later once we've determined how to handle the request.

```
req_data = client_socket.recv(BUFFER_SIZE)
print("Received new request:", req_data)
dest_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

We dissect the request data line-by line to first find the type of request, and set the parameters for the destination socket appropriately.

```
lines = req_data.split(b'\r\n')
line1 = lines[0].split(b' ')
req_type = line1[0]
print("Request type:", req_type)

if req_type == b'CONNECT':
   dest_url = None
   dest = line1[1].decode('utf-8')
   dest_address, dest_port = dest.split(":")

elif req_type == b'GET':
   dest_url = line1[1]
   for line in lines:
       if b'Host' in line:
           dest_address = line.split(b' ')[1].decode('utf-8')
   dest_port = '80'
```

We then bind the destination socket by the parameters we determined, and handle the incoming request appropriately; either confirming the connection if it is a CONNECT request, forwarding the data if it is not.

```
dest_socket.connect((dest_address, int(dest_port)))

if req_type == b'CONNECT':
   client_socket.sendall(b'HTTP/1.1 200 Connection Established\r\n\r\n')
else:
   dest_socket.sendall(req_data)
```

Finally, with the sockets we need initialised, we handle the connection until it terminates by setting two threads to work; one to handle client to server, and the other to handle server to client communication.

```
client_to_server_thread = threading.Thread(target=client_to_server,
args=[client_socket, dest_socket])
```

```
server_to_client_thread = threading.Thread(target=server_to_client,
args=[dest_socket, client_socket, dest_url])
client_to_server_thread.start()
server_to_client_thread.start()
```

Requirement 2:

When the program starts, it sets a thread running a simple function to manage the blocking of urls.

```
block_url_thread = threading.Thread(target=block_url)
block_url_thread.start()
```

The function itself simply reads input from the user, and adds the input to its list of blocked urls if it is not in the list, and removes it if it is.

```
def block_url():
   while True:
       url = input("")
       if url in blocked:
           print("UNBLOCKING:", url)
           blocked.remove(url)
       else:
           print("BLOCKING:", url)
           blocked.append(url)
```

Finally, in the request function explained in part 1, before running the threads for server/client and client/server communication, we check if the connecting url is a substring of a blocked url or if a blocked url is a substring of the url. If it is, we simply don't initialise those threads, and so communication is stopped.

```
skip = False
for url in blocked:
   if (url in dest_address) or (dest_address in url):
       skip = True
       break

if not skip:
   dest_socket.connect((dest_address, int(dest_port)))

   if req_type == b'CONNECT':
       client_socket.sendall(b'HTTP/1.1 200 Connection Establ...
```

Requirement 3:

Caching is handled within the server_to_client function. The url parameter is used to distinguish between HTTP and HTTPS requests;

as the url cannot be extracted from a HTTPS request, it is passed as None for HTTPS requests, and it is set for HTTP requests.
If we are handling a HTTPS request, we simply forward the data as normal. However if it is a HTTP request, we first check if we have cached data for this url before. If we have, we forward the cached data, rather than the data on the receiving socket. If we haven't cached data from this url, then we record it in a dict where the url indexes to the data for use later.

```
while True:
    loop_start = time.time()
    if not (url is None):
        if not (url in cached):
            data = dest_socket.recv(BUFFER_SIZE)
            if not data:
                break
            cached.update({url : data})
        else:
            data = cached[url]
            print("Cached data used,", len(cached[url]), "bytes saved.")

    else:
        data = dest_socket.recv(BUFFER_SIZE)

    client_socket.sendall(data)
    print("Time taken:", (time.time() - loop_start))
```

Requirement 4:

The implementation of a threaded server is quite simple; the request function, discussed above, is simply called in a thread each time a new connection is made.

```
def start_proxy_server():
    srv_soc = socket.socket(family=socket.AF_INET, type=socket.SOCK_STREAM)
    srv_soc.bind((PROXY_IP, CLIENT_PORT))
    srv_soc.listen(5)

    block_url_thread = threading.Thread(target=block_url)
    block_url_thread.start()

    while True:
        client_socket, client_address = srv_soc.accept()

        client_thread = threading.Thread(target=request,
                                         args=(client_socket,))
        client_thread.start()
```

Raw code below:

```
import socket
import threading
import time

PROXY_IP    = '127.0.0.1'
CLIENT_PORT = 4000
BUFFER_SIZE = 4096
blocked = []
cached = {}


# function assumes the user's intent
# by the state of the blocklist; if
# an entered url is not already
# blocked, it is added to the list.
# if it is already on the list, it
# is removed. Runs in its own thread
def block_url():
   while True:
       url = input("")
       if url in blocked:
           print("UNBLOCKING:", url)
           blocked.remove(url)
       else:
           print("BLOCKING:", url)
           blocked.append(url)

#initialised with a socket for receiving
# client data and a socket for
# forwarding to the destination.
# multiple instances run in threads
def client_to_server(client_socket, dest_socket):
   try:
       while True:
           data = client_socket.recv(BUFFER_SIZE)
           if not data:
               break
           dest_socket.sendall(data)
   except Exception:
       return

#initialised with a socket for receiving
# server data and a socket for
# forwarding to the client. url
# parameter is used to manage the
# caching of HTTP requests. multiple
# instances run in threads
def server_to_client(dest_socket, client_socket, url):
   try:
       while True:
           loop_start = time.time()
```

```
            if not (url is None):
                if not (url in cached):
                    data = dest_socket.recv(BUFFER_SIZE)
                    if not data:
                        break
                    cached.update({url : data})
                else:
                    data = cached[url]
                    print("Cached data used,", len(cached[url]), "bytes
saved.")

            else:
                data = dest_socket.recv(BUFFER_SIZE)

            client_socket.sendall(data)
            print("Time taken:", (time.time() - loop_start))

    except Exception:
        return

#called by the start function whenever a new
# connection is made
def request(client_socket):
    #display request data on the management console
    req_data = client_socket.recv(BUFFER_SIZE)
    print("Received new request:", req_data)
    dest_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    #split the request data by line and get the request type from the first
line
    lines = req_data.split(b'\r\n')
    line1 = lines[0].split(b' ')
    req_type = line1[0]
    print("Request type:", req_type)

    #handle CONNECT requests
    if req_type == b'CONNECT':
        dest_url = None
        dest = line1[1].decode('utf-8')
        dest_address, dest_port = dest.split(":")

    #handle GET requests
    elif req_type == b'GET':
        dest_url = line1[1]
        for line in lines:
            if b'Host' in line:
                dest_address = line.split(b' ')[1].decode('utf-8')
        dest_port = '80'

    #check if the url has been blocked, in which case we simply don't respond
    skip = False
    for url in blocked:
        if (url in dest_address) or (dest_address in url):
```

```
            skip = True
            break

    if not skip:
        #initialse destination socket with appropriate data
        dest_socket.connect((dest_address, int(dest_port)))

        #confirm CONNECT requests, respond appropriately to others
        if req_type == b'CONNECT':
            client_socket.sendall(b'HTTP/1.1 200 Connection
Established\r\n\r\n')
        else:
            dest_socket.sendall(req_data)

        #set server to client and client to server communication running in
threads, using the sockets we'd initialised
        # for this particular connection
        client_to_server_thread = threading.Thread(target=client_to_server,
args=[client_socket, dest_socket])
        server_to_client_thread = threading.Thread(target=server_to_client,
args=[dest_socket, client_socket, dest_url])
        client_to_server_thread.start()
        server_to_client_thread.start()

        #wait for threads to finish, and close the relevant sockets once they
have done
        client_to_server_thread.join()
        server_to_client_thread.join()
        dest_socket.close()
        client_socket.close()

#this function can be thought of as the
# defacto "main".
def start_proxy_server():
    #initialise a socket for listening on the specified proxy IP and port
    srv_soc = socket.socket(family=socket.AF_INET, type=socket.SOCK_STREAM)
    srv_soc.bind((PROXY_IP, CLIENT_PORT))
    srv_soc.listen(5)

    #set the block url function running in its own thread
    block_url_thread = threading.Thread(target=block_url)
    block_url_thread.start()

    while True:
        #when a connection is detected on the server socket, initialise a
socket to respond to it
        client_socket, client_address = srv_soc.accept()

        #handle the connection in a thread
        client_thread = threading.Thread(target=request,
args=(client_socket,))
        client_thread.start()
```

```
if __name__ == "__main__":
    start_proxy_server()
```