Trinity College, Dublin
CSU34031: Advanced Telecommunications

Web Proxy Server with Caching

Claire Cassidy

12/03/2021

Contents

1 Problem Statement

A Web Proxy Server is a server that acts as a middleman between a client and a real third-party server. The web proxy intercepts all outgoing HTTP(S) and WebSocket requests from the client, and fulfils those requests on the client's behalf. This can either be achieved by contacting the third-party server directly to obtain the required resource(s), or by delivering an in-date cached copy of the resource that has been obtained by the server on a recent request for the same resource. In doing so, the proxy server can reduce response time for the client and bandwidth on the local network by negating the need to establish a connection with the third-party server and transfer the requested resources from a potentially distant source. This can result in increased performance of the network overall. Furthermore, the proxy can also be configured to dynamically block resources, including whole websites or specific resources on those websites, making the proxy a more general tool for a network administrator to manage network access.

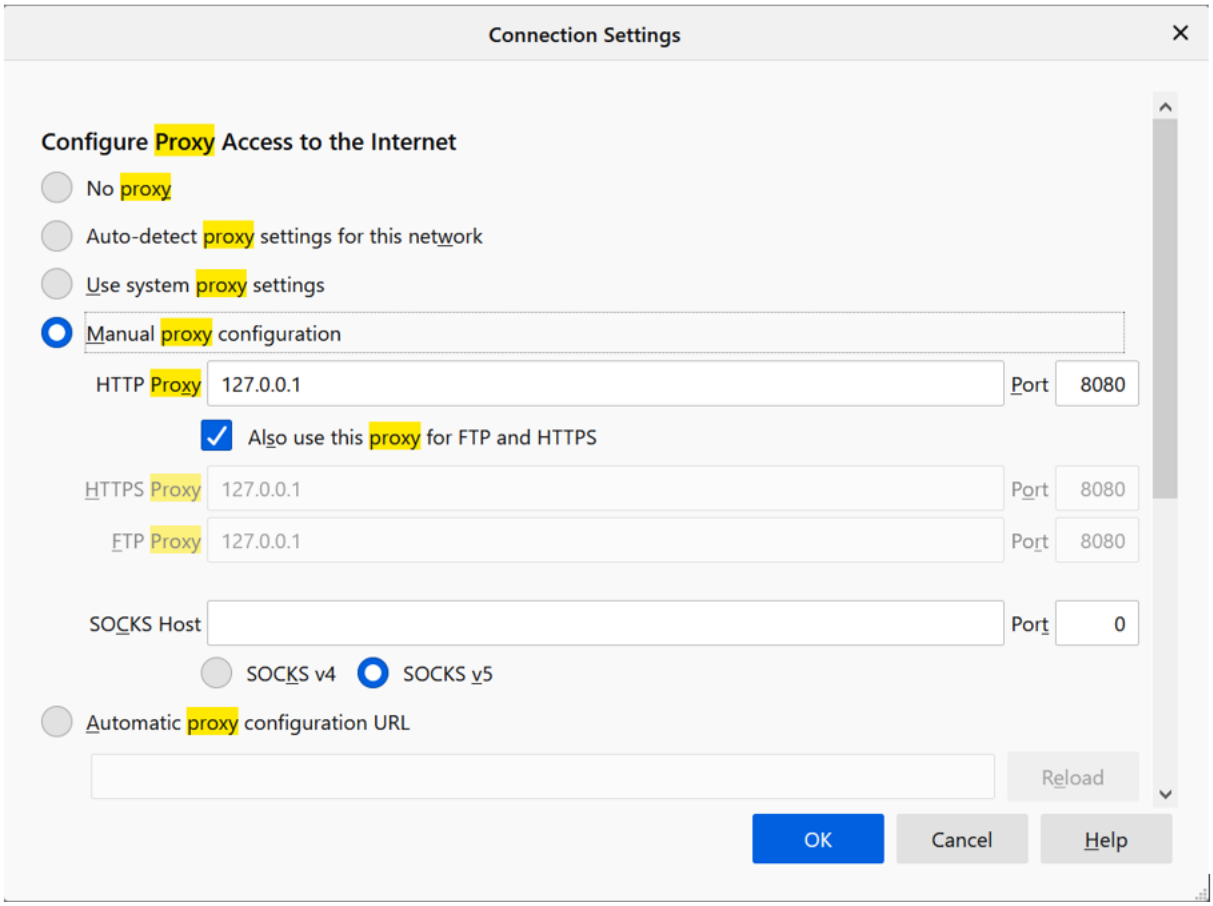
2 Program Setup

Once downloaded, the program can be run by compiling the source files and running the `Launcher` class.

It is recommended to run this program by configuring it as an **IntelliJ** project and interacting with the program via the built-in terminal, since some terminals such as the generic Windows `cmd` terminal don't support multi-coloured printing. Powershell appears to show some colours but doesn't disambiguate between normal console logs and stack traces and may be more confusing to read.

```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
Server established on port 8080
ProxyServer Running
Enter a command to configure the Proxy Server:
    BLOCK [url]          - block the specified URL
    UNBLOCK [url]        - remove URL from blocklist
    HELP                 - print this message again
    QUIT                 - save blocked/cache config and safely exit program
    BLOCKLIST            - print list of blocked sites
true
Establishing ConnectionHandlerThread...
```

By default, the program is configured to use `localhost:8080`. The program was tested in Firefox 86.0.1, due to the ease of configuring it to use a proxy at the desired port. This can be done by going to Options → type "proxy" in the options searchbar → Settings... → Manual Proxy Configuration and entering the following:



Once this is done, all requests will automatically be routed through the program.

3 Program Design

I will now discuss the precise specifications given in the assignment and how my implementation tackles them at a high level. Subsequent sections will discuss each of the classes and their code in detail.

1. *Respond to HTTP and HTTPs requests and display each request on a management console. It should forward the request to the web server and relay the response in the browser.*

   The project contains a `MangementConsole` class, which runs on a separate thread and exists solely to handle configuration input from the user. It polls `stdin` for input, parses it in relation to a set of recognised commands, such as **BLOCK**; **UNBLOCK**; **QUIT**; etc., and performs the appropriate action to satisfy the user's request. To disambiguate between messages logged by the worker threads and communication between the end user and the management console, messages originating from the management console are printed in a separate colour.

   As the entry and exit point for the program, the Management Console also has jurisdiction over creating and maintaining the persistent data that the proxy server needs to operate; namely the Cache directory and the file containing the blocked sites. These aspects will be discussed in further detail shortly.

   The Management Console is booted up before the proxy server itself is started, so that the files are ensured to be created and appropriate data is loaded before operation begins. Once this is complete, the Management console creates a `ProxyServer` thread, which can now begin servicing requests. The `ManagementConsole` remains listening until it receives the **QUIT** command, at which point it will signal the blockedSites instance object to be written to the `blockedsites.txt` file. **For this reason, it is recommended that the program is only terminated after the QUIT command has been entered, to ensure persistent features work as intended.**

   The `ProxyServer` thread actually doesn't do much of the hard work; its job is just to accept incoming requests for resources from the browser and spawn a new `ConnectionHandlerThread` to service each request, passing it the socket on which the connection was made. Communication between the proxy and the client then takes place on a per-request basis, with each request being serviced in its entirety by its own `ConnectionHandlerThread`. HTTP and HTTPS requests alike are passed to the same kind of worker thread, although the method for dealing with HTTPS requests is slightly different, as will be discussed later. The other function of the `ProxyServer` is to respond to shutdown requests from the `ManagementConsole`, which are received on receipt of a **QUIT** command from the user.

   The `ConnectionHandlerThread` checks the method type of the request and prints out whether it's a HTTP or HTTPS request to the console, as well as the requested resource URL and the socket information for the connection to the client. The `ConnectionHandlerThread` then decides how to service the request based on whether

the resource falls under a blocked resource and whether or not it is cached already. The details of this will be discussed later. Given that the resource is not blocked, the thread either delivers the cached version of the resource or performs a HTTP(S) request to the endpoint, and forwards the reply back to the client. The result of this can be seen in the user's web browser as required.

2. *Handle Websocket connections.*

   Due to time restrictions and lack of library support, I was unable to complete this portion of the problem statement. However, a form of full-duplex communication similar to that adopted by WebSockets can be seen in the way the program handles HTTPs requests (discussed in a later section), which may form part of the solution. What would remain would be to expand the `ConnectionHandlerThread` class to parse requests using the **ws://** and **wss://** headers.

3. *Dynamically block selected URLs via the management console.*

   As described above, the Management Console is launched at start-up and listens for **BLOCK/UNBLOCK** requests from the user at the console. This can be used to configure blocked sites on the fly, and block and unblock a resource in the same session without needing to reload the program. Each `ConnectionHandlerThread` consults a dynamically-updated `ArrayList` containing the URLs of blocked resources before serving a request. The changes to the blocklist are persistent and written to/loaded from the `blocked.txt` file in the `res` folder to preserve the list between sessions.

4. *Efficiently cache HTTP requests locally and thus save bandwidth. You must gather timing and bandwidth data to prove the efficiency of your proxy.*

   The `res` folder contains a subfolder called `cache` which contains each of the cached HTTP resources the proxy has access to. When a URL is fetched from a remote endpoint, this will mean the file wasn't found in the cache. The fetched HTTP resource will then automatically be written to the cache as it is being sent back to the client, using an encoding scheme that preserves the original URL of the resource. When a new request comes in, the URL will be converted into this encoded form, checked against the filenames in the cache folder, and be retrieved and sent back to the user if it exists.

   The cache files also take into account the expiry dates given in the HTTP response headers supplied by the endpoint, where they exist. Cache files are written with the expiry date/time on the first like, followed by the actual resource on the second line. Thus, once we find a match, we can also check the expiry date against the current time to ensure the user is getting a fresh copy of the resource they requested.

   Timing and bandwidth data will be provided in the section *Measuring Cache Performance*.

4 Implementation Details:

Having given an overview of my implementation in response to the problem statement, I elect in this section to go into further details regarding my implantation on a class-by-class basis.

4.1 Launcher.java

This class contains the main method for the program and simply creates the `ManagementConsole` thread and leaves it to do the rest of the work

```
import java.io.IOException;


public class Launcher {
    public static void main(String[] args) throws IOException {

        Thread mgmtConsole = new Thread(new ManagementConsole());
        mgmtConsole.start();

    }
}
```

4.2 ManagementConsole.java

The management console keeps member variables for the Proxy, so that it may send it the shut down signal, and the blocked sites list, so that worker threads may consult a reference of blocked sites. The worker threads obtain this resource by calling the following method:

```
    public static ArrayList<String> getBlockedSites() {
        return blockedSitesInstance;
    }
```

On creation, the Management Console must ensure the `res` folder and its contents are all created. This will only be necessary on the first running of the program. The blocked sites list must also be loaded into memory. The following method, called in the class constructor, achieves this:

```
    private void createAndLoadResources() throws IOException {

        // create blocked.txt if doesn't exist
        File blockedSitesFile = new File("res\\blocked.txt");
        if (!blockedSitesFile.getParentFile().exists()) {
            if (blockedSitesFile.getParentFile().mkdirs()) {
                printMgmtStyle("Successfully created directory \"res\".");// create 'res' dir if
            } else {
                printMgmtStyle("Error creating directory \"res\".");
                throw new IOException();
            }
```

```
        }

        // create res/cache if doesn't exist
        File cacheDir = new File("res\\cache");
        if (!cacheDir.exists()) {
            if (cacheDir.mkdir()) {
                printMgmtStyle("Successfully created directory \"res\\cache\".");
            } else {
                printMgmtStyle("Error creating directory \"res\".");
                throw new IOException();
            }
        }

        blockedSitesFile.getParentFile().mkdir();

        // load the blocked sites into memory
        try {
            if (!blockedSitesFile.exists()) {
                printMgmtStyle(this.toString() + "Couldn't find blocked sites file - Creating ..
                blockedSitesFile.createNewFile();
                printMgmtStyle("Created \"res/blocked.txt\"");
            }


            Scanner sc = new Scanner(blockedSitesFile);     //file to be scanned
            blockedSitesInstance = new ArrayList<>();

            String site;
            while (sc.hasNextLine()) {
                site = sc.nextLine();
                blockedSitesInstance.add(site); // update dynamic blocked sites
            }

        } catch (FileNotFoundException e) {
            printMgmtStyle(this.toString()+"Failed to open blocked sites file");

            e.printStackTrace();
        }

    }
```

Once the resources have been created and loaded, the management console creates a `ProxyServer` instance, which is responsible for servicing incoming connections `localhost:8080`.

The management console then enters the `run` method, where it enters an endless loop of reading commands from the user on `std.in`.

```
    while (this.isRunning()) {
```

```
            // parse next user command
            String nextInput = in.nextLine();
            try {

                String[] nextInputTokens = nextInput.split(" ");

                if (nextInputTokens[0].toLowerCase().equals("block")) {

                    ...

                } else {
                    printMgmtStyle("Couldn't recognised command - please retry or enter HELP to s

                }
            } catch (Exception e) {
                printMgmtStyle("Couldn't recognised command - please retry or enter HELP to see i

                //e.printStackTrace();
            }
        }

        // reached after QUIT entered
        in.close();
    }
```

`this.isRunning()` resolves to false once the user enters the **QUIT** command, which prompts the console to send the kill signal to the Proxy Server and shut itself down. Before shutting down, the dynamic instance of the blocked sites list is written back to the file.

```
    // On shutdown, called to write dynamic blocked files arraylist to
    // blocked.txt for persistent blocking
    private void saveBlockedSitesConfig() {
        try {
            FileWriter fw = new FileWriter("res\\blocked.txt");
            for(String str:blockedSitesInstance) {
                fw.write(str + System.lineSeparator());
            }
            fw.close();
        } catch (IOException e) {
            printMgmtStyle("Error writing to blocked.txt");
            e.printStackTrace();
        }
    }
```

4.3 ProxyServer.java

The `ProxyServer` class is responsible for spawning `ConnectionHandlerThread`s which perform the functions of the proxy server. It achieves this by entering an infinite loop of receiv-

ing connections on its `ServerSocket` and then spawning a new `ConnectionHandlerThread`, giving it the socket instance on which to communicate.

```
@Override
public void run() {
    System.out.println("ProxyServer Running");

    try {
        // while the server socket should still be accepting incoming connections
        while (this.isRunning() && serverSocket.isBound() && !serverSocket.isClosed()) {

            // wait for connection and when received, assign to incoming
            Socket incoming = serverSocket.accept();

            // create a new thread and give it the socket to begin communications
            ConnectionHandlerThread newThread = new ConnectionHandlerThread(incoming);
            newThread.start();
        } // loop and listen for new connections
    } catch (IOException e) {
        System.out.println("Error creating server socket @PORT:" + port);
        e.printStackTrace();
    }


}
```

4.4 ConnectionHandlerThread.java

This class is where the majority of the work of the proxy server is done.

Once the thread enters the run method, it begins by extracting the input and output streams on which it will exchange data with the client. The HTTP(S) request in its entirety is read in as a `String` from the input stream:

```
// read in the request from the client and store in 'request'
BufferedReader in = new BufferedReader(new InputStreamReader(inputStream));
String request = "";
String inputLine;
while ((inputLine = in.readLine()) != null && !inputLine.equals("")) {
    request += inputLine;
}
```

Before contacting the endpoint, the thread first checks if 1) the resource is blocked, 2) the resource is cached. If either of these checks are true, the endpoint is not contacted by the thread.

To check the resource is not on the blocklist, the thread consults the blocked sites `ArrayList`. It must not be from a website containing a URL substring on the blocklist - the blocklist is intended to hold website names, such as **www.youtube.com**. In this way,

all resources from YouTube, such as embedded videos on unrelated sites, get caught in the blocking process. We then set a boolean to tell the control logic what path to take based on the URL's blocked status.

```
            // check resource not on blocklist
            ArrayList<String> blockedSites = ManagementConsole.getBlockedSites();
            boolean endpointIsBlocked = false;

            if (blockedSites != null) {
                for (String site : blockedSites) {
                    // capture "www.youtube.com/..." as well as "www.youtube.com"
                    if (endpointUrl.toLowerCase().contains(site.toLowerCase())) {
                        endpointIsBlocked = true;
                        break;
                    }
                }
            }
```

If the endpoint is blocked, we send back a 403 response and notify the user in the management console. In FireFox, the 403 response usually results in a message saying the proxy isn't accepting connections, but the response receipt can be verified by using the network tab in the inspector.

```
    ...
    } else {    // resource was on blocklist
    ManagementConsole.printMgmtStyle("Access to blocked site \"" + endpointUrl + "\" denied.\n"
    + "\tIf this is unexpected, you may have blocked a parent resource -\n" +
            "\tEnter \"BLOCKLIST\" to view your blocked sites. \n" +
            "\tEnter \"UNBLOCK [site]\" to access.");

    sendForbidden();    // respond to browser with 403
}

...

// triggered as a response to a request for a blocked resource
private void sendForbidden() throws IOException {
    System.out.println("Sending forbidden...");
    String fullResponse = "HTTP/1.1 " + FORBIDDEN_CODE + " " + FORBIDDEN // [HTTP_VERSION] [RESP(
            + CRLF // HEADER
            + CRLF // tells client were done with header
            + CRLF + CRLF;
    outputStream.write(fullResponse.getBytes());
    outputStream.flush();
}
```

If the resource was not blocked, we proceed to check if we have a cached copy. As discussed before, the URLs are cached with the intent to preserve their original name. However, since URLs contain the '/' character, we cannot cache them as-is since they are mistaken for

filepaths. Thus, I encode them by 1) replace all _ with ,,, 2) replace all / with _. The choice of ,,, was made with the intention of choosing a short and uncommon pattern that won't be present in the vast majority of URLs. Of course, this is not a perfect strategy, since it is still *legal* for URLs to contain this subsequence. The consequence of a URL containing the subsequence is that it simply won't be retrieved from the cache correctly because the URL will be malformed on decoding. However, given the infrequency of this pattern I considered this was a reasonable trade-off. I considered other methods for encoding such as Base64 encoding but they almost always produced filenames that were too long to be saved if the URL was of a modest length.

```
@Override
public String toString() {
    return "ConnectionHandlerThread:" + id;
}

// should be a url like www.example.com/files/resource
public static String getCacheFilenameFromUrl(String url) {
    //  www.w3.org/Icons/w3c_logo.svg
    //      convert _ to ,,,
    //  www.w3.org/Icons/w3c,,,logo.svg
    //      convert / to _
    //  www.w3.org_Icons_w3c,,,logo.svg

    // trim if hasn't been trimmed already; removes 'http(s)://' and trailing '/'
    String filename = trimUrl(url);
    filename = filename.replaceAll("_", ",,,");
    filename = filename.replaceAll("/", "_");

    return filename;
}
```

This method is used both when the request URL is begin checked for its existence in the cache, and for when saving contents to the cache.

If there is a hit on the filename in the cache, a final check needs to be performed - is the cached resource in date? Cache files are written with the expiry date/time on the first like, followed by the actual resource on the second line. Thus, once we find a match, the expiry date is checked against the current time to ensure the user is getting a fresh copy of the resource they requested.

```
ArrayList<File> cacheFiles = getCacheFiles();

    for (File curFile : cacheFiles) {   // compare against filenames in the cache

        String filename = curFile.getName();

        if (filename.equals(filenameForUrl)) {  // if it matches, it may not be in date
            // if it hasn't expired, don't send request to server and just send cached file
```

```
            if (isInDate(curFile)) {
                sendCached = true;
                cachedFile = curFile;
                break;
            }
        }
    }

    if (sendCached) {   // if cached and in date, send the cached version
        sendCachedFile(cachedFile);
    } else {    // otherwise need to contact endpoint
        ...
    }
```

The `isInDate` method takes a File, reads its first line to get the expiry date, and uses a timezone-agnostic Date format, `ZonedDateTime`, to compare the current time to the expiry time on the cached file.

The `sendCachedFile` method simply reads the second line from the passed file and sends this in a **200 OK** response to the client.

Finally, if we have exhausted all other options, the thread will contact the endpoint on the client's behalf. Firstly, the thread needs to determine if the request uses the HTTP or HTTPS protocol. A HTTP request will simply have a header with a **GET** method, whereas a HTTPS request will have a header with a **CONNECT** method. The two protocols have different implementations, so they call different methods:

```
if (sendCached) {   // if cached and in date, send the cached version
    sendCachedFile(cachedFile);
} else {    // otherwise need to contact endpoint

    if (method.toLowerCase().equals("connect")) {   // https connect request
        handleHttpsRequest(endpointUrl);
    } else if (method.toLowerCase().equals("get")) { // http get request
        handleHttpRequest(endpointUrl);
    } else {    // only service GET and CONNECT
        sendNotImplementedResponse();
    }
}
```

We will discuss `handleHttpRequest` first. Originally, I built the proxy to read the output streams using String buffers, which I only realised at a later point would cause an issue when reading image files, since they contain byte sequences that cannot be saved as an ASCII characters and thus will become corrupted. Thus, I added a separate case for transmission of images from the proxy to the client, that makes use of Java's built-in `ImageIO` class, which can connect to an input stream and transfer data as needed. The determination of when to use this case is made by examining the URL extension. The image data is stored in a `BufferedImage` object whose bytes are then written over the output stream to the client. In the future, the code would be better refactored to exclusively use byte arrays since they don't impose a type on the data being transmitted.

```
if (isImageType(urlExtension)) {    // handle image stream separately
        try {
            URL url = new URL(urlString);

            HttpURLConnection connection = (HttpURLConnection) url.openConnection();
            // fetch the resource and store it in 'imgResource'
            BufferedImage imgResource = ImageIO.read(connection.getInputStream());
            if (imgResource != null) {                          // found image at given url
                String responseHeader = "HTTP/1.1 " + OK_CODE + " " + OK
                        + CRLF + CRLF;
                outputStream.write(responseHeader.getBytes());
                ImageIO.write(imgResource, urlExtension, outputStream);
            } else {
                sendNotFound(); // couldn't fetch the resource, so send user a 404
            }

        } catch (MalformedURLException e) {

            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    } else {    // send normal text type
        ...
    }
```

When submitting plaintext, the method is as simple as copying the data into a string object, and passing that along as the body of a **200 OK** response to the client. In the process of extracting the response headers to print them, the method also notes the expiry date for caching purposes.

```
    } else {    // send normal text type
        try {
    URL url = new URL(urlString);
    HttpURLConnection connection = (HttpURLConnection) url.openConnection();    // connect to enc
    connection.setRequestMethod("GET");         // indicate method type in header
    connection.setConnectTimeout(5000);         // cancel long connection
    connection.setReadTimeout(5000);            // cancel long-awaited reply

    // read response from requested endpoint, store in 'content'
    BufferedReader in = new BufferedReader(
            new InputStreamReader(connection.getInputStream()));
    String inputLine;
    StringBuffer content = new StringBuffer();
    while ((inputLine = in.readLine()) != null) {
        content.append(inputLine);
    }

    // get response code
```

```
int responseCode = connection.getResponseCode();
String responseMsg = connection.getResponseMessage();

...
// printing...
...

String fullResponse = "";

// given good response code, contents is what we want to send back
if (responseCode >= 200 && responseCode < 300) { // alles gut


    fullResponse = "HTTP/1.1 " + responseCode + " " + responseMsg // [HTTP_VERSION] [RESPONSE
            + CRLF
            + "Content-Length: " + content.toString().getBytes().length + CRLF // HEADER
            + CRLF // tells client were done with header
            + content.toString() // response body
            + CRLF + CRLF;
    outputStream.write(fullResponse.getBytes());
    outputStream.flush();
    // cache the response

    String justTheUrl = trimUrl(urlString);
    String filenameFromUrl = getCacheFilenameFromUrl(justTheUrl); // name cache file accordin

    if (expiryDate != null) {   // don't bother caching if we can't give an expiry date - wil
        try {

            File cacheFile = new File("res\\cache\\" + filenameFromUrl);
            ManagementConsole.printMgmtStyle("Writing to " + cacheFile.getCanonicalPath());
            if (cacheFile.createNewFile()) {                        // creates the file if it d
                System.out.println(this.toString() + " Creating cache file for \"" + filename
            } else {
                System.out.println(this.toString() + " Overwriting cache file for \"" + justT
            }
            // write expiry date and response to file

            FileWriter fw = new FileWriter(cacheFile, false);
            fw.write(expiryDate + System.lineSeparator());
            fw.write(content.toString() + System.lineSeparator());
            fw.flush();
            fw.close();
        } catch (IOException e) {
            System.out.println(this.toString() + " Error writing to cachefile for resource \"
            e.printStackTrace();
        }
    }
```

```
}
```

Finally, I will discuss the `handleHttpsRequest` method.

A HTTPs connection is required to be secure, and the two communicating parties alone contain the key to decrypt the communication from the opposite party. Thus, a HTTP proxy necessarily cannot read data from the client to endpoint (and vice versa), nor can it act on the client's behalf to contact the server, since HTTPS is designed to block middlemen intercepting the exchange of messages. Therefore, the proxy must perform what is known as *HTTP tunneling*, which essentially stitches the input and output streams of both the client and endpoint together so that they can communicate. Instead of intercepting and relaying the messages individually, the proxy simply supports the conduit through which communication takes place. From Wikipedia:

The most common form of HTTP tunneling is the standardized HTTP CONNECT method. In this mechanism, the client asks an HTTP proxy server to forward the TCP connection to the desired destination. The server then proceeds to make the connection on behalf of the client. Once the connection has been established by the server, the proxy server continues to proxy the TCP stream to and from the client. Only the initial connection request is HTTP - after that, the server simply proxies the established TCP connection. This mechanism is how a client behind an HTTP proxy can access websites using SSL or TLS (i.e. HTTPS).

In order to achieve this 'tunnelling', the thread creates two `HttpsConnectorThread`s, whose job it is to connect the input and output streams of a socket between the client and endpoint in a single direction (hence the need for two to facilitate bi-directional transfer). Since the two 'tunnels' in either direction are independent, the communication can be considered full-duplex. It is for this reason I suggested a similar implementation WebSocket connections.

The method first takes note of the URL and the target port for the request (a specific feature of HTTPS requests). It then creates a new socket to establish communication to the endpoint. However, instead of using its own input/output streams, the method passes a reference to the client's input and output streams. This means the client and endpoint will be in direct contact.

It is the proxy thread's responsibility to close the Socket connection to the endpoint, and so it waits until communication has ceased in either direction before doing so.

```
// extract endpoint target port from url:
    String[] urlComponents = urlString.split(":");
    String urlItself = urlComponents[0];
    int endpointPort = Integer.parseInt(urlComponents[1]);

    try {
        sendConnectionEst();    // respond to connect request

        // set up socket connection to endpoint
```

```
        Socket endpointSocket = new Socket(urlItself, endpointPort);
        endpointSocket.setSoTimeout(3000);

        InputStream endpointSocketIn = endpointSocket.getInputStream();
        OutputStream endpointSocketOut = endpointSocket.getOutputStream();

        // link from client to endpoint
        // Needs endpointSocket's inputStream, incoming's outputStream
        Thread clientToEndpointThread = new HttpsConnectorThread(endpointSocketIn, outputStream)
        clientToEndpointThread.start();

        // link from endpoint to client
        // Needs incoming's input stream, endpointSocket's outputStream
        Thread endpointToClientThread = new HttpsConnectorThread(inputStream, endpointSocketOut)
        endpointToClientThread.start();

        // wait for them to finish before continuing
        clientToEndpointThread.join();
        endpointToClientThread.join();

        // close the endpoint socket when communication has ceased
        endpointSocket.close();

    } catch ...
```

4.5 HttpsConnectorThread

As discussed above, the function of the `HttpsConnectorThread` is the facilitate uni-directional data transfer between a client and endpoint over a TCP connection.

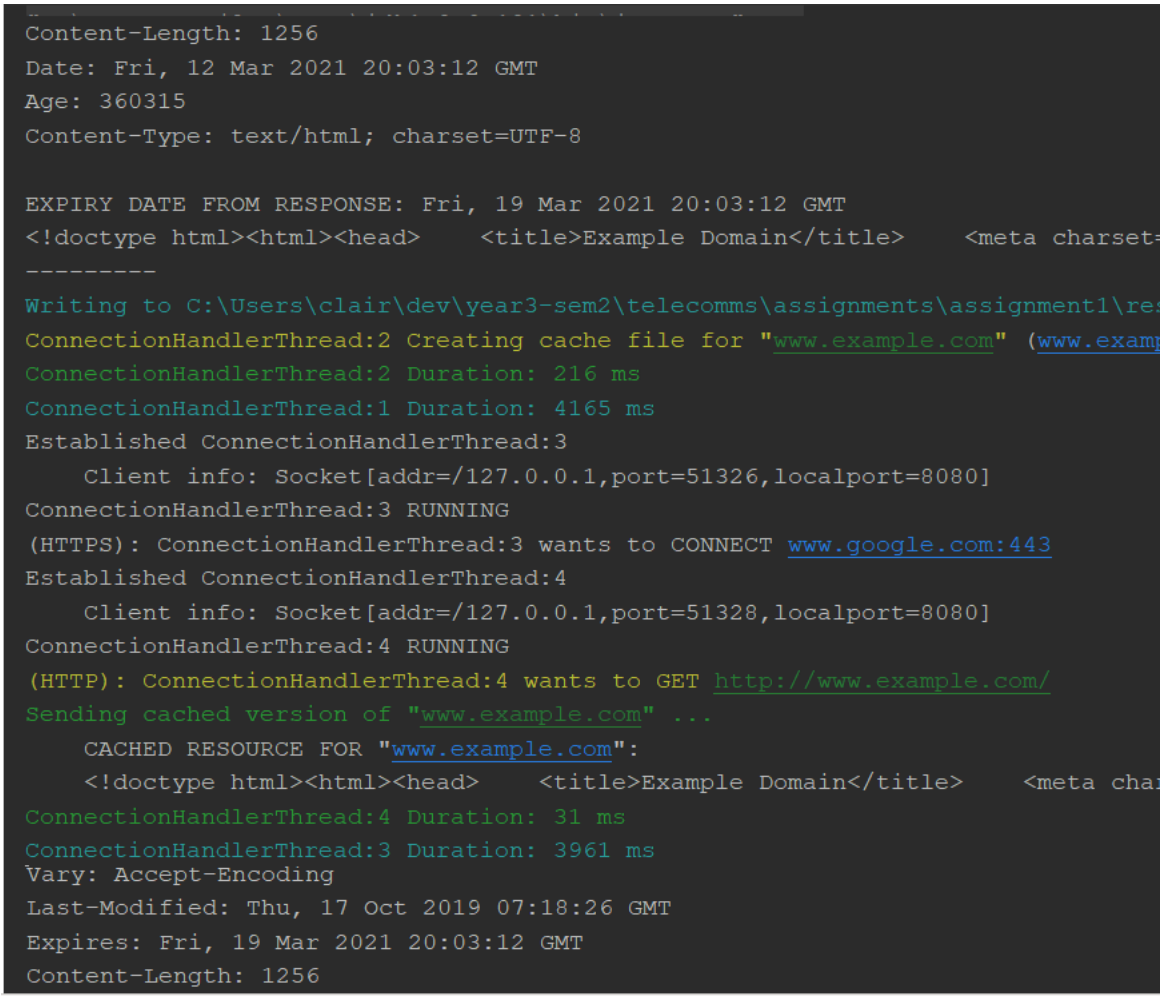
The functionality of the thread is comparatively simple, in that it simply forwards a buffer of bytes from one side to another, while there are still bytes left to be read.

```
try {
    byte[] responseBuffer = new byte[8192]; // buffer received data
    // will be > 0 if last read operation got some bytes
    int amountBytesToSend;

    // keep checking we have bytes
    while ((amountBytesToSend = in.read(responseBuffer, 0 , responseBuffer.length)) != -1) {
        out.write(responseBuffer, 0, amountBytesToSend);          // send them
        out.flush();
    }
} catch ...
```

5 Measuring Cache Performance:

By using a fresh cache and attempting to access a HTTP page, **www.example.com**, then immediately sending another request for the same resource, I received the following timing information by measuring method execution time directly (see highlighted output):

```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
Server established on port 8080
ProxyServer Running
Enter a command to configure the Proxy Server:
    BLOCK [url]          - block the specified URL
    UNBLOCK [url]        - remove URL from blocklist
    HELP                 - print this message again
    QUIT                 - save blocked/cache config and safely exit program
    BLOCKLIST            - print list of blocked sites
Established ConnectionHandlerThread:1
    Client info: Socket[addr=/127.0.0.1,port=51320,localport=8080]
ConnectionHandlerThread:1 RUNNING
(HTTPS): ConnectionHandlerThread:1 wants to CONNECT www.google.com:443
Established ConnectionHandlerThread:2
    Client info: Socket[addr=/127.0.0.1,port=51322,localport=8080]
ConnectionHandlerThread:2 RUNNING
(HTTP): ConnectionHandlerThread:2 wants to GET http://www.example.com/
---------
Repsonse from http://www.example.com/:

200 OK
X-Cache: HIT
Server: ECS (bsa/EB18)
Etag: "3147526947+ident"
Cache-Control: max-age=604800
Vary: Accept-Encoding
Last-Modified: Thu, 17 Oct 2019 07:18:26 GMT
Expires: Fri, 19 Mar 2021 20:03:12 GMT
Content-Length: 1256
```



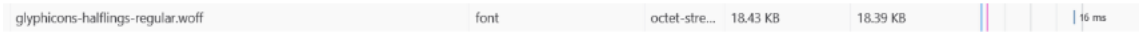
In the above, **duration** is the time in milliseconds from the time of request parsing to the time of thread exit, after the response has been sent. As the log output shows, the first attempt at fetching the resource required 216 ms to complete to fetch from the source and deliver to the client. Once the resource had been cached, it took only 31 ms.

I tried this again with another (mostly) HTTP site, apache.org. To get the base webpage without caching took 330 ms, whereas with caching it took only 43 ms. (Images show request when uncached vs. requesting again once cached).

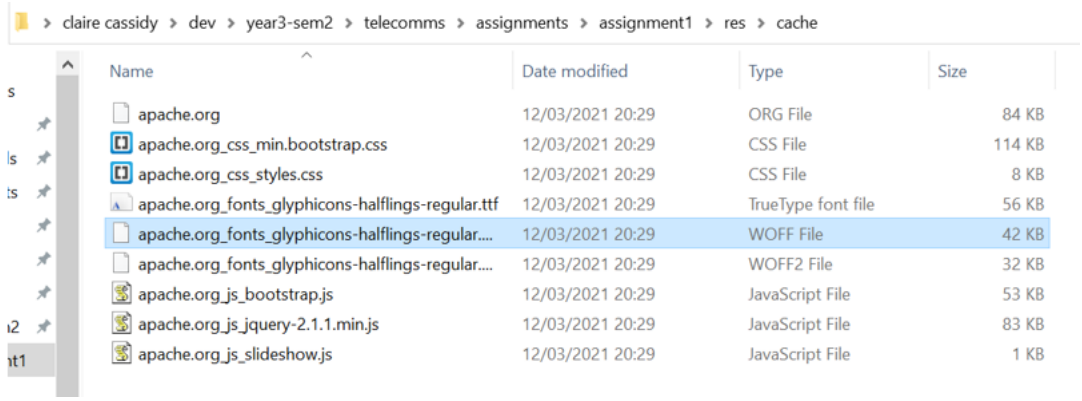
```
ConnectionHandlerThread:3 RUNNING
(HTTP): ConnectionHandlerThread:3 wants to GET http://apache.org/
---------
Repsonse from http://apache.org/:

200 OK
Keep-Alive: timeout=5, max=2000
Server: Apache
Connection: Keep-Alive
Last-Modified: Fri, 12 Mar 2021 20:10:31 GMT
Date: Fri, 12 Mar 2021 20:12:21 GMT
Accept-Ranges: bytes
Cache-Control: max-age=3600
ETag: "15466-5bd5c7df614c8"
Vary: Accept-Encoding
Expires: Fri, 12 Mar 2021 21:12:21 GMT
Content-Length: 87142
Content-Type: text/html

EXPIRY DATE FROM RESPONSE: Fri, 12 Mar 2021 21:12:21 GMT
<!DOCTYPE html><html lang="en"><head>  <meta charset="utf-8">  <meta http-equiv="X-UA-Compatible
---------
Writing to C:\Users\clair\dev\year3-sem2\telecomms\assignments\assignment1\res\cache\apache.org
ConnectionHandlerThread:3 Creating cache file for "apache.org" (apache.org) ...
ConnectionHandlerThread:3 Duration: 330 ms
```

```
(HTTP): ConnectionHandlerThread:67 wants to GET http://apache.org/
Sending cached version of "apache.org" ...
    CACHED RESOURCE FOR "apache.org":
    <!DOCTYPE html><html lang="en"><head>  <meta charset="utf-8">  <me
ConnectionHandlerThread:67 Duration: 43 ms
```

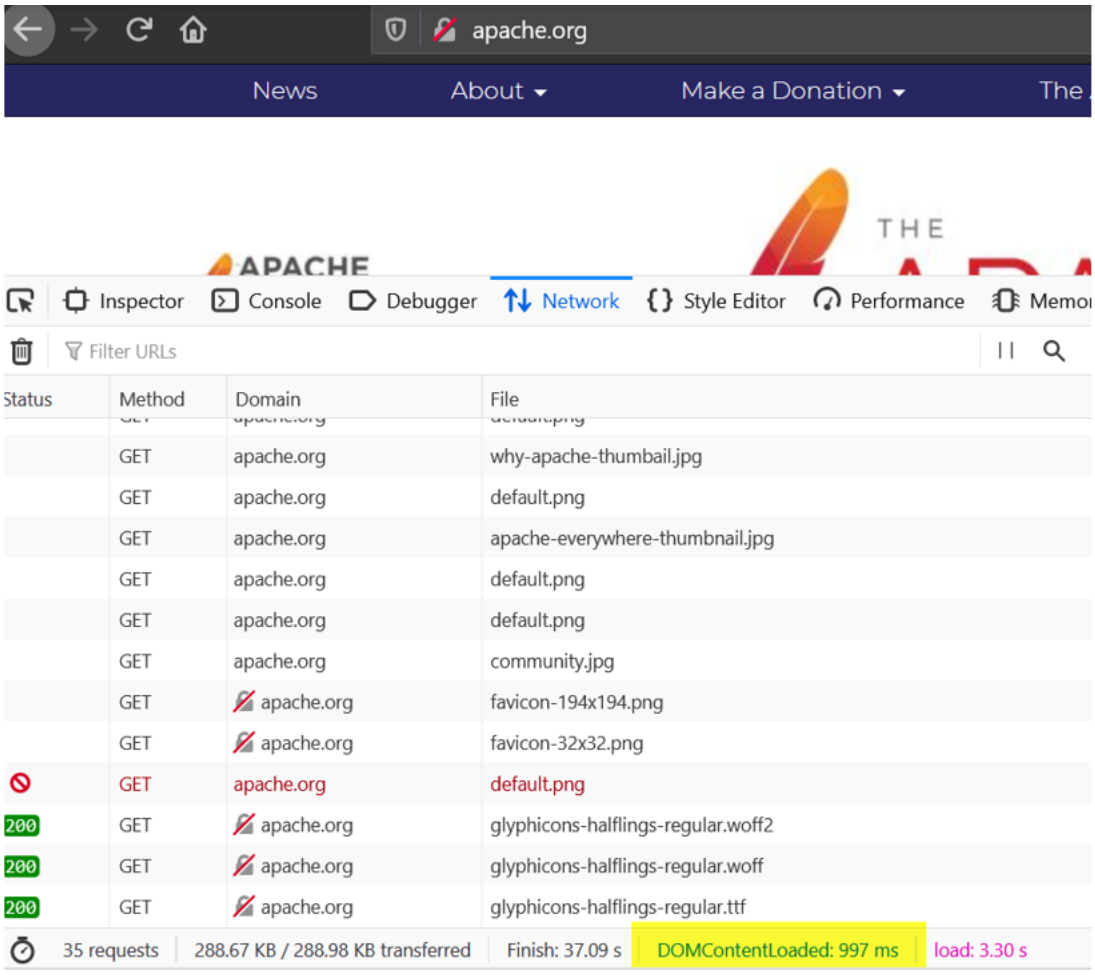
This method of timing cannot take into account the time to transmit to the client, however. When taking timing information from the browser itself (ensuring caching in the browser is turned off), caching also proved to achieve significant speedups. For example, see the before and after for the following files that are cached when visiting apache.org:
Before caching (16ms):

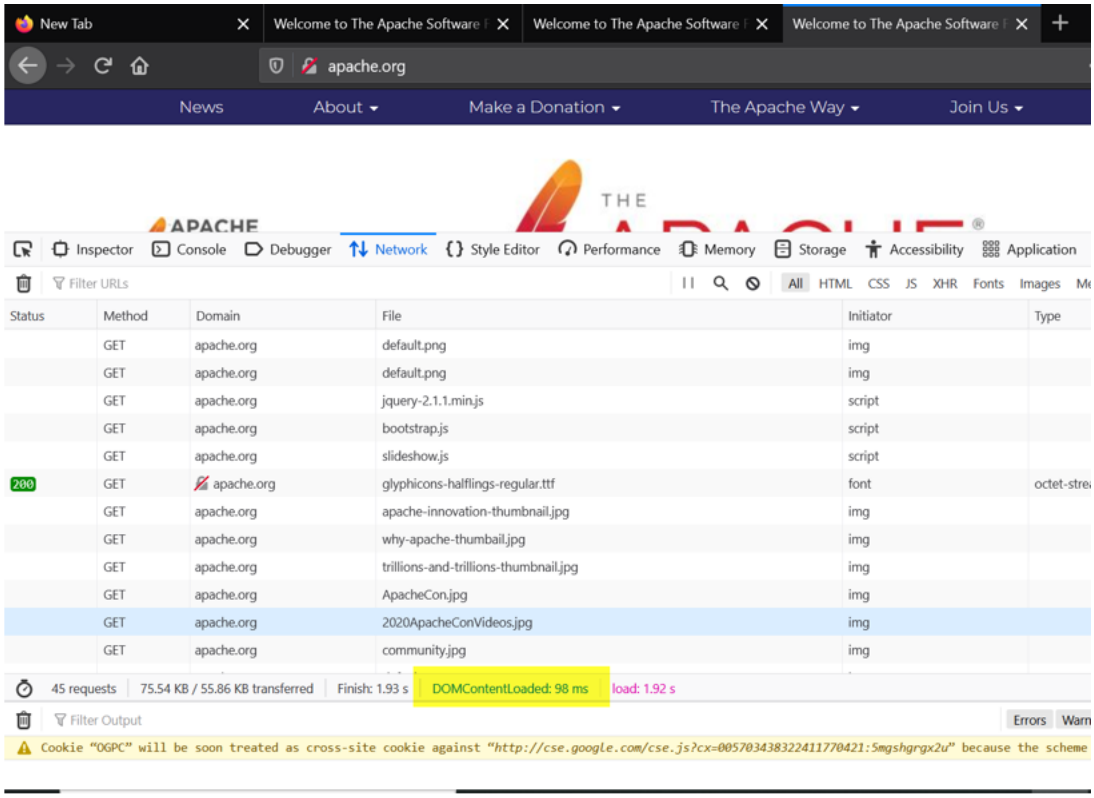


After caching (8ms):



Overall, there is a significant speedup in loading the DOM after caching:





Note: by default, the timing code has been commented out in the source files to prevent cluttering the output. You may uncomment them to verify timing information.

6 Full Code Listing

6.1 Launcher.java

```
import java.io.IOException;


public class Launcher {
    public static void main(String[] args) throws IOException {

        Thread mgmtConsole = new Thread(new ManagementConsole());
        mgmtConsole.start();

    }
}
```

6.2 ManagementConsole.java

```
    import java.io.*;
import java.util.ArrayList;
import java.util.Scanner;

public class ManagementConsole implements Runnable {

    private volatile boolean stopped = false;
    private static ArrayList<String> blockedSitesInstance = null;   // dynamic copy of blocked.tx
    private ProxyServer proxy;          // maintain reference so can stop proxy server from consc

    // for easily distinguishing management console msgs
    public static final String ANSI_CYAN = "\u001B[36m";
    public static final String ANSI_RESET = "\u001B[0m";
    private static final String INSTRUCTIONS = "Enter a command to configure the Proxy Server:\n"
            "\tBLOCK [url] \t\t- block the specified URL\n"+
            "\tUNBLOCK [url] \t\t- remove URL from blocklist\n"+
            "\tHELP \t\t\t\t- print this message again\n"+
            "\tQUIT \t\t\t\t- save blocked/cache config and safely exit program\n"+
            "\tBLOCKLIST \t\t\t- print list of blocked sites";



    public ManagementConsole() throws IOException {

        // 1. create necessary resources if they don't exist
        // 2. load blocked sites so worker threads can intercept requests
        // 3. create proxy server
        // 4. repeatedly listen for user input at stdin

        createAndLoadResources();

        // create the proxy server
        proxy = new ProxyServer(8080);

        Thread proxyThread = new Thread(proxy);
        proxyThread.start();
    }

    public static ArrayList<String> getBlockedSites() {
        return blockedSitesInstance;
    }

    // may be called from Launcher to stop the mgmt console
    public synchronized void stop() {

        printMgmtStyle("Shutting down application ...");
```

```
    this.stopped = true;
}

private synchronized boolean isRunning() {
    return this.stopped == false;
}

// disambiguate between thread print statements and communication between console and end use
public static void printMgmtStyle(String toPrint) { // threads can also ask the console to pr
    System.out.println(ANSI_CYAN + toPrint + ANSI_RESET);
}

private void createAndLoadResources() throws IOException {

    // create blocked.txt if doesn't exist
    File blockedSitesFile = new File("res\\blocked.txt");
    if (!blockedSitesFile.getParentFile().exists()) {
        if (blockedSitesFile.getParentFile().mkdirs()) {
            printMgmtStyle("Successfully created directory \"res\".");// create 'res' dir if
        } else {
            printMgmtStyle("Error creating directory \"res\".");
            throw new IOException();
        }
    }

    // create res/cache if doesn't exist
    File cacheDir = new File("res\\cache");
    if (!cacheDir.exists()) {
        if (cacheDir.mkdir()) {
            printMgmtStyle("Successfully created directory \"res\\cache\".");
        } else {
            printMgmtStyle("Error creating directory \"res\".");
            throw new IOException();
        }
    }

    blockedSitesFile.getParentFile().mkdir();

    // load the blocked sites into memory
    try {
        if (!blockedSitesFile.exists()) {
            printMgmtStyle(this.toString() + "Couldn't find blocked sites file - Creating ..
            blockedSitesFile.createNewFile();
            printMgmtStyle("Created \"res/blocked.txt\"");
        }


        Scanner sc = new Scanner(blockedSitesFile);     //file to be scanned
```

```
            blockedSitesInstance = new ArrayList<>();

            String site;
            while (sc.hasNextLine()) {
                site = sc.nextLine();
                blockedSitesInstance.add(site); // update dynamic blocked sites
            }

        } catch (FileNotFoundException e) {
            printMgmtStyle(this.toString()+"Failed to open blocked sites file");

            e.printStackTrace();
        }

    }

    @Override
    public void run() {

        Scanner in = new Scanner(System.in);

        // instructions for user
        printMgmtStyle(INSTRUCTIONS);

        while (this.isRunning()) {

            // parse next user command
            String nextInput = in.nextLine();
            try {

                String[] nextInputTokens = nextInput.split(" ");

                if (nextInputTokens[0].toLowerCase().equals("block")) {

                    printMgmtStyle("You want to block " + nextInputTokens[1]);
                    blockedSitesInstance.add(nextInputTokens[1]);
                    printMgmtStyle(nextInputTokens[1] + " successfully blocked.");

                } else if (nextInputTokens[0].toLowerCase().equals("help")) {

                    printMgmtStyle("\n" + INSTRUCTIONS + "\n");

                } else if (nextInputTokens[0].toLowerCase().equals("unblock")) {

                    int index = blockedSitesInstance.indexOf(nextInputTokens[1]);
                    if (index >= 0) {
                        blockedSitesInstance.remove(index);
                        printMgmtStyle("Successfully removed "+nextInputTokens[1]+" from blocklis
```

```
                } else {    // was not blocked
                    printMgmtStyle("\""+nextInputTokens[1] + "\" not found on blocklist.");
                }

            } else if (nextInputTokens[0].toLowerCase().equals("blocklist")) {

                printMgmtStyle("Contents of blocklist.txt: ");
                for (String site:blockedSitesInstance) {
                    printMgmtStyle(site);
                }

            } else if (nextInputTokens[0].toLowerCase().equals("quit")) {
                printMgmtStyle("Quitting ... ");

                printMgmtStyle("\t Writing blocked sites ...");
                saveBlockedSitesConfig();

                // stop the proxy and the management console
                proxy.stop();
                this.stop();

            } else {
                printMgmtStyle("Couldn't recognised command - please retry or enter HELP to s

            }
        } catch (Exception e) {
            printMgmtStyle("Couldn't recognised command - please retry or enter HELP to see

            //e.printStackTrace();
        }
    }

    // reached after QUIT entered
    in.close();
}

// On shutdown, called to write dynamic blocked files arraylist to blocked.txt for persistent
private void saveBlockedSitesConfig() {
    try {
        FileWriter fw = new FileWriter("res\\blocked.txt");
        for(String str:blockedSitesInstance) {
            fw.write(str + System.lineSeparator());
        }
        fw.close();
    } catch (IOException e) {
        printMgmtStyle("Error writing to blocked.txt");
        e.printStackTrace();
    }
```

```
    }

    @Override
    public String toString() {
        return "\tMGMT-CONSOLE: ";
    }
}
```

6.3 ProxyServer.java

```
    import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;

public class ProxyServer implements Runnable {

    private volatile boolean stopped = false;
    private int port;
    private ServerSocket serverSocket;

    // creates a new ServerSocket given a port number that listens for connections
    public ProxyServer(int port) throws IOException {

        this.port = port;

        serverSocket = new ServerSocket(port);
        System.out.println("Server established on port " + port);
    }

    // may be called from ManagementConsole to stop the server; prevents response to new connecti
    //      and thus the spawning of new ConnectionHandlerThreads
    public synchronized void stop() {
        ManagementConsole.printMgmtStyle("ProxyServer stopping ...");
        this.stopped = true;
    }


    // checked after each connection to see if it should still be running
    private synchronized boolean isRunning() {
        return this.stopped == false;
    }

    @Override
    public void run() {
        System.out.println("ProxyServer Running");

        try {
```

```
            // while the server socket should still be accepting incoming connections
            while (this.isRunning() && serverSocket.isBound() && !serverSocket.isClosed()) {

                // wait for connection and when received, assign to incoming
                Socket incoming = serverSocket.accept();

                // create a new thread and give it the socket to begin communications
                ConnectionHandlerThread newThread = new ConnectionHandlerThread(incoming);
                newThread.start();
            } // loop and listen for new connections
        } catch (IOException e) {
            System.out.println("Error creating server socket @PORT:" + port);
            e.printStackTrace();
        }


    }
}
```

6.4 ConnectionHandlerThread

```
    import javax.imageio.ImageIO;
import java.awt.image.BufferedImage;
import java.io.*;
import java.net.*;
import java.time.ZonedDateTime;
import java.time.format.DateTimeFormatter;
import java.util.*;

public class ConnectionHandlerThread extends Thread {

    private Socket incoming;
    private static int global_id;   // helps identify the threads; incr'd every time thread is cr
    private int id;                 // local id
    private InputStream inputStream;
    private OutputStream outputStream;

    // HTTP Status codes
    final String CRLF = "\r\n";
    final String FORBIDDEN = "FORBIDDEN";
    final String FORBIDDEN_CODE = "403";
    final String NOT_IMPLEMENTED = "NOT IMPLEMENTED";
    final String NOT_IMPLEMENTED_CODE = "501";
    final String OK = "OK";
    final String OK_CODE = "200";
    final String CONNECTION_ESTABLISHED = "CONNECTION ESTABLISHED";
    final String NOT_FOUND = "NOT FOUND";
```

```
    final String NOT_FOUND_CODE = "404";

    // for configuring how to send http resource
    final String JPG = "jpg";
    final String JPEG = "jpeg";
    final String GIF = "gif";
    final String PNG = "png";


    public ConnectionHandlerThread(Socket incoming) throws IOException {

        this.incoming = incoming;       // record the client socket
        incoming.setSoTimeout(3000);    // set timeout window

        // assign id to thread
        setId();

    }

    // prevent race condition among threads
    private synchronized void setId() {
        global_id++;
        id = global_id;
        System.out.println("Established ConnectionHandlerThread:" + id);
        System.out.println("\tClient info: " + incoming.toString());
    }

    @Override
    public void run() {
        System.out.println(this.toString() + " RUNNING");

        try {
            inputStream = incoming.getInputStream();
            outputStream = incoming.getOutputStream();

            // read in the request from the client and store in 'request'
            BufferedReader in = new BufferedReader(new InputStreamReader(inputStream));
            String request = "";
            String inputLine;
            while ((inputLine = in.readLine()) != null && !inputLine.equals("")) {
                request += inputLine;
            }

//            long startTime = System.nanoTime();

            // get the type of the http(s) request and the resource url
            String method = getMethod(request);
            String endpointUrl = getURL(request);
```

```
if (method.toLowerCase().equals("get")) {
    System.out.print("(HTTP): ");
} else if (method.toLowerCase().equals("connect")) {
    System.out.print("(HTTPS): ");
}
System.out.println(this.toString() + " wants to " + method + " " + endpointUrl);

// check blocklist -> check cache -> fetch from source

// check resource not on blocklist
ArrayList<String> blockedSites = ManagementConsole.getBlockedSites();
boolean endpointIsBlocked = false;

if (blockedSites != null) {
    for (String site : blockedSites) {
        // capture "www.youtube.com/..." as well as "www.youtube.com"
        if (endpointUrl.toLowerCase().contains(site.toLowerCase())) {
            endpointIsBlocked = true;
            break;
        }
    }
}


if (!endpointIsBlocked) {   // next check in cache for resource before contacting enc

    // compare the encoded version of the desired resource url against the filenames
    String justTheUrl = trimUrl(endpointUrl);
    String filenameForUrl = getCacheFilenameFromUrl(justTheUrl);    // get the encode

    boolean sendCached = false;
    File cachedFile = null;

    ArrayList<File> cacheFiles = getCacheFiles();

    for (File curFile : cacheFiles) {   // compare against filenames in the cache

        String filename = curFile.getName();

        if (filename.equals(filenameForUrl)) {  // if it matches, it may not be in da
            // if it hasn't expired, don't send request to server and just send cache
            if (isInDate(curFile)) {
                sendCached = true;
                cachedFile = curFile;
                break;
            }
        }
```

```
                }

                if (sendCached) {   // if cached and in date, send the cached version
                    sendCachedFile(cachedFile);
                } else {    // otherwise need to contact endpoint

                    if (method.toLowerCase().equals("connect")) {   // https connect request
                        handleHttpsRequest(endpointUrl);
                    } else if (method.toLowerCase().equals("get")) { // http get request
                        handleHttpRequest(endpointUrl);
                    } else {    // only service GET and CONNECT
                        sendNotImplementedResponse();
                    }
                }

            } else {    // resource was on blocklist
                ManagementConsole.printMgmtStyle("Access to blocked site \"" + endpointUrl + "\"
                        "\tIf this is unexpected, you may have blocked a parent resource -\n" +
                        "\tEnter \"BLOCKLIST\" to view your blocked sites. \n" +
                        "\tEnter \"UNBLOCK [site]\" to access.");

                sendForbidden();    // respond to browser with 403
            }

//            long endTime = System.nanoTime();
//            long duration = (endTime - startTime)/1000000;  //divide by 1000000 to get millisec
//            ManagementConsole.printMgmtStyle(this.toString()+" Duration: "+duration+" ms");

            // close streams and sockets
            inputStream.close();
            outputStream.close();
            incoming.close();
        } catch (SocketTimeoutException e) {    // ignore; usually as result of trying to contact
        } catch (IOException e) {
            System.out.println("IOException in ConnectionHandlerThread " + id);
            e.printStackTrace();
        }


    }

    // Takes a raw http request and attempts to extract the method
    private String getMethod(String request) {
        String method = null;
        try {

            String[] requestLines = request.split("\r\n");  // split into lines
```

```
        String methodLine = requestLines[0];    // get first line
        String[] methodLineTokens = methodLine.split(" ");  // get the individual tokens
        method = methodLineTokens[0];

    } catch (Exception e) {
        System.out.println(this.toString() + " Error parsing request method type - invalid fo
    } finally {
        return method;
    }
}

// Takes a raw http request and attempts to extract the endpoint URL
private String getURL(String request) {
    String endpoint = null;
    try {

        String[] requestLines = request.split("\r\n");  // split into lines

        String methodLine = requestLines[0];    // get first line
        String[] methodLineTokens = methodLine.split(" ");  // get the individual tokens
        endpoint = methodLineTokens[1];

    } catch (Exception e) {
        System.out.println(this.toString() + " Error parsing endpoint URL - invalid format?")
    } finally {
        return endpoint;
    }
}

private void handleHttpsRequest(String urlString) {

    /*
    The most common form of HTTP tunneling is the standardized HTTP CONNECT method.[1][2]
    In this mechanism, the client asks an HTTP proxy server to forward the TCP connection
    to the desired destination. The server then proceeds to make the connection on behalf
    of the client. Once the connection has been established by the server, the proxy server
    continues to proxy the TCP stream to and from the client. Only the initial connection
    request is HTTP - after that, the server simply proxies the established TCP connection.
    This mechanism is how a client behind an HTTP proxy can access websites using SSL or
    TLS (i.e. HTTPS).
    */

    // extract endpoint target port from url:
    String[] urlComponents = urlString.split(":");
    String urlItself = urlComponents[0];
    int endpointPort = Integer.parseInt(urlComponents[1]);

try {
```

```
            sendConnectionEst();    // respond to connect request

            // set up socket connection to endpoint
            Socket endpointSocket = new Socket(urlItself, endpointPort);
            endpointSocket.setSoTimeout(3000);

            InputStream endpointSocketIn = endpointSocket.getInputStream();
            OutputStream endpointSocketOut = endpointSocket.getOutputStream();

            // link from client to endpoint
            // Needs endpointSocket's inputStream, incoming's outputStream
            Thread clientToEndpointThread = new HttpsConnectorThread(endpointSocketIn, outputStream);
            clientToEndpointThread.start();

            // link from endpoint to client
            // Needs incoming's input stream, endpointSocket's outputStream
            Thread endpointToClientThread = new HttpsConnectorThread(inputStream, endpointSocketOut);
            endpointToClientThread.start();

            // wait for them to finish before continuing
            clientToEndpointThread.join();
            endpointToClientThread.join();

            // close the endpoint socket when communication has ceased
            endpointSocket.close();

        } catch (UnknownHostException e) {
            System.out.println(this.toString() + " Unable to connect to HTTPS host "+urlItself);
            e.printStackTrace();
        } catch (IOException e) {
            System.out.println(this.toString() + " IOException when connecting to HTTPS "+urlItself);
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    private void handleHttpRequest(String urlString) {   // simple http get (non-cached)

        String expiryDate = null;   // for caching purposes
        String urlExtension = getResourceExtension(urlString);  // decide what kind of connection
        if (isImageType(urlExtension)) {    // handle image stream separately
            try {
                URL url = new URL(urlString);

                HttpURLConnection connection = (HttpURLConnection) url.openConnection();
                // fetch the resource and store it in 'imgResource'
                BufferedImage imgResource = ImageIO.read(connection.getInputStream());
```

```
            if (imgResource != null) {                          // found image at given url
                String responseHeader = "HTTP/1.1 " + OK_CODE + " " + OK
                        + CRLF + CRLF;
                outputStream.write(responseHeader.getBytes());
                ImageIO.write(imgResource, urlExtension, outputStream);
            } else {
                sendNotFound(); // couldn't fetch the resource, so send user a 404
            }

        } catch (MalformedURLException e) {

            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    } else {    // send normal text type
        try {
            URL url = new URL(urlString);
            HttpURLConnection connection = (HttpURLConnection) url.openConnection();    // co
            connection.setRequestMethod("GET");         // indicate method type in header
            connection.setConnectTimeout(5000);         // cancel long connection
            connection.setReadTimeout(5000);            // cancel long-awaited reply

            // read response from requested endpoint, store in 'content'
            BufferedReader in = new BufferedReader(
                    new InputStreamReader(connection.getInputStream()));
            String inputLine;
            StringBuffer content = new StringBuffer();
            while ((inputLine = in.readLine()) != null) {
                content.append(inputLine);
            }


            // check everything was okay ->
            System.out.println("---------\nRepsonse from " + urlString + ":\n");

            // get response code
            int responseCode = connection.getResponseCode();
            String responseMsg = connection.getResponseMessage();

            StringBuilder builder = new StringBuilder();
            builder.append(responseCode)
                    .append(" ")
                    .append(responseMsg)
                    .append("\n");

            // get headers
            Map<String, List<String>> map = connection.getHeaderFields();
```

```
            for (Map.Entry<String, List<String>> entry : map.entrySet()) {
                if (entry.getKey() == null)
                    continue;
                builder.append(entry.getKey())
                        .append(": ");

                if (entry.getKey().toLowerCase().equals("expires")) {
                    expiryDate = entry.getValue().get(0);
                }

                List<String> headerValues = entry.getValue();
                Iterator<String> it = headerValues.iterator();
                if (it.hasNext()) {
                    builder.append(it.next());

                    while (it.hasNext()) {
                        builder.append(", ")
                                .append(it.next());
                    }
                }

                builder.append("\n");
            }

            System.out.println(builder.toString());
            System.out.println("EXPIRY DATE FROM RESPONSE: " + expiryDate);
            System.out.println(content + "\n---------");

            String fullResponse = "";

        // given good response code, contents is what we want to send back
        if (responseCode >= 200 && responseCode < 300) { // alles gut


            fullResponse = "HTTP/1.1 " + responseCode + " " + responseMsg // [HTTP_VERSION] [RESP
                    + CRLF
                    + "Content-Length: " + content.toString().getBytes().length + CRLF // HEADER
                    + CRLF // tells client were done with header
                    + content.toString() // response body
                    + CRLF + CRLF;
            outputStream.write(fullResponse.getBytes());
            outputStream.flush();
            // cache the response

            String justTheUrl = trimUrl(urlString);
            String filenameFromUrl = getCacheFilenameFromUrl(justTheUrl); // name cache file acco

            if (expiryDate != null) {   // don't bother caching if we can't give an expiry date -
```

```
try {

    File cacheFile = new File("res\\cache\\" + filenameFromUrl);
    ManagementConsole.printMgmtStyle("Writing to "
    + cacheFile.getCanonicalPath());
    if (cacheFile.createNewFile()) {                        // creates the file if it doesn't alrea
        System.out.println(this.toString() + " Creating cache file for \"" + filenameFromUrl
        + "\" ("+justTheUrl+") ... ");
    } else {
        System.out.println(this.toString() + " Overwriting cache file for \"" + justTheUrl
        + "\" ("+justTheUrl+") ... ");
    }
    // write expiry date and response to file

    FileWriter fw = new FileWriter(cacheFile, false);
    fw.write(expiryDate + System.lineSeparator());
    fw.write(content.toString() + System.lineSeparator());
    fw.flush();
    fw.close();
} catch (IOException e) {
    System.out.println(this.toString() + " Error writing to
    cachefile for resource \"" + filenameFromUrl +
    "\" ("+justTheUrl+")");
    e.printStackTrace();
}
            }


        }

            } catch (ProtocolException e) {
                System.out.println(this.toString() + " experienced ProtocolException -
                check headers");
                e.printStackTrace();
            } catch (MalformedURLException e) {
                System.out.println(this.toString() + " received malformed url: " + urlString);
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    // sends 200 CONNECTION ESTABLISHED
    private void sendConnectionEst() throws IOException {
        String response = "HTTP/1.1 " + OK_CODE + " " + CONNECTION_ESTABLISHED
                + CRLF // HEADER
                + CRLF // tells client were done with header
```

```
            + CRLF + CRLF;
    outputStream.write(response.getBytes());
    outputStream.flush();
}

// sent when a request is a received for an image that can't be sourced, etc.
private void sendNotFound() throws IOException {
    System.out.println("Sending not found ...");
    String fullResponse = "HTTP/1.1 " + NOT_FOUND_CODE + " " + NOT_FOUND // [HTTP_VERSION] [
            + CRLF // HEADER
            + CRLF // tells client were done with header
            + CRLF + CRLF;
    outputStream.write(fullResponse.getBytes());
    outputStream.flush();
}

// check resource type before choosing send method
private boolean isImageType(String urlExtension) {
    String local = urlExtension.toLowerCase();
    return local.equals(JPG) ||
            local.equals(JPEG) ||
            local.equals(PNG) ||
            local.equals(GIF);
}

// triggered as a response to a request for a blocked resource
private void sendForbidden() throws IOException {
    System.out.println("Sending forbidden...");
    String fullResponse = "HTTP/1.1 " + FORBIDDEN_CODE + " " + FORBIDDEN // [HTTP_VERSION] [
            + CRLF // HEADER
            + CRLF // tells client were done with header
            + CRLF + CRLF;
    outputStream.write(fullResponse.getBytes());
    outputStream.flush();
}   // will say "proxy is refusing connections" -> go to inspector and network and see that


// triggered as a response to a non-GET or non-CONNECT method type in request
private void sendNotImplementedResponse() throws IOException {
    System.out.println("Sending not implemented...");
    String fullResponse = "HTTP/1.1 " + NOT_IMPLEMENTED_CODE + " " + NOT_IMPLEMENTED// [HTTP_
            + CRLF // HEADER
            + CRLF // tells client were done with header
            + CRLF + CRLF;
    outputStream.write(fullResponse.getBytes());
    outputStream.flush();
}
```

```
// for comparison against the requested resource url
private ArrayList<File> getCacheFiles() {

    ArrayList<File> files = new ArrayList<>();

    try {

        File cacheDir = new File("res\\cache");     // file representing the directory
        File[] cacheFiles = cacheDir.listFiles();

        for (File cacheFile : cacheFiles) {     // copy to arraylist
            if (cacheFile.isFile()) {           // ignore subdirectories
                files.add(cacheFile);
            }
        }

    } catch (Exception e) {
        System.out.println(this.toString() + " Experienced error
        retrieving cache files - check path");
        e.printStackTrace();
    }

    return files;
}

private boolean isInDate(File cachedFile) {
    // file cached with format
    // 1    EXPIRY_DATE
    // 2    HTML etc.   (one line)

    try (Scanner sc = new Scanner(cachedFile)) {    // ensure resource is closed
        String expiryDate;
        if (sc.hasNextLine()) { // just need the first line
            expiryDate = sc.nextLine();

            ZonedDateTime expiryZdt = ZonedDateTime.parse(expiryDate,
            DateTimeFormatter.RFC_1123_DATE_TIME);
            ZonedDateTime nowZdt = ZonedDateTime.now();

            // diff == 1 if expiryZdt in future, diff = -1 if its expired, diff == 0 if its t
            long diff = expiryZdt.compareTo(nowZdt);

            if (diff < 1) { // i.e. expired or expires now; not suitable for sending
                return false;
            } else return true;
        }
    } catch (FileNotFoundException e) {
        System.out.println(this.toString() + " Unexpected error
```

```
            encountered opening Cached File \""
                    + cachedFile.getName() + "\".");
            e.printStackTrace();
        } catch (Exception e) { // catch date parse errors
            System.out.println(this.toString() + " Invalid date format in file \""
            + cachedFile.getName() + "\"");
            e.printStackTrace();
        }

        return false;   // if we can't parse an expiry date from the cache, re-fetch the resource
    }

    private void sendCachedFile(File cachedFile) {

        // file cached with format
        // 1    EXPIRY_DATE
        // 2    HTML etc.   (one line)

        ManagementConsole.printMgmtStyle("Sending cached version of
        \"" + cachedFile.getName() + "\" ...");
        String resource;

        try (Scanner sc = new Scanner(cachedFile)) {    // ensure resource is closed
            if (sc.hasNextLine()) sc.nextLine();        // skip expiry date line
            if (sc.hasNextLine()) {
                resource = sc.nextLine();
                System.out.println("\tCACHED RESOURCE FOR \"" + cachedFile.getName() + "\": \n\t"

                String fullResponse = "HTTP/1.1 " + OK_CODE + " " + OK // [HTTP_VERSION] [RESPONS
                        + CRLF
                        + "Content-Length: " + resource.getBytes().length + CRLF // HEADER
                        + CRLF // tells client were done with header
                        + resource // response body
                        + CRLF + CRLF;

                outputStream.write(fullResponse.getBytes());
                outputStream.flush();
            }


        } catch (FileNotFoundException e) {
            System.out.println(this.toString() + " Unexpected error
            encountered opening Cached File \"" + cachedFile.getName() + "\".");
            e.printStackTrace();
        } catch (IOException e) {
            System.out.println(this.toString() + " Error writing cached resource to client.");
            e.printStackTrace();
        }
```

```
}

// trims unecessary stuff from request url so that it can be effectively matched to cache fil
private static String trimUrl(String rawUrl) {
    String justTheUrl = rawUrl;

    if (rawUrl.startsWith("https")) {
        justTheUrl = rawUrl.substring(8);
    } else if (rawUrl.startsWith("http")) {
        justTheUrl = rawUrl.substring(7);
    }

    if (justTheUrl.endsWith("/")) {
        justTheUrl = justTheUrl.substring(0, justTheUrl.length() - 1); // trim trailing '/'
    }

    return justTheUrl;
}

// need to disambiguate between image-type resources and text resources and handle streams ap
private String getResourceExtension(String url) {

    int extStartIndex = url.lastIndexOf('.');
    if (extStartIndex > -1 && extStartIndex != url.length()-1) {
        String extension = url.substring(extStartIndex+1);
        if (extension.contains(":")) {  // port
            extension = extension.substring(0, extension.indexOf(':'));
        }
        return extension;
    } else {
        return null;
    }
}

@Override
public String toString() {
    return "ConnectionHandlerThread:" + id;
}

public static String getCacheFilenameFromUrl(String url) { // should be a url like www.exampl
    //  www.w3.org/Icons/w3c_logo.svg
    //      convert _ to ,,,
    //  www.w3.org/Icons/w3c,,,logo.svg
    //      convert / to _
    //  www.w3.org_Icons_w3c,,,logo.svg

    String filename = trimUrl(url); // trim if hasn't been trimmed already; removes 'http(s)
```

```
        filename = filename.replaceAll("_", ",,,");
        filename = filename.replaceAll("/", "_");

        return filename;
    }

    // reverses the encoding process for cache filenames and extracts true url
    public static String getUrlFromCacheFilename(String filename) {
        String url = filename;
        url = url.replaceAll("_", "/");
        url = url.replaceAll(",,,", "_");

        return url;
    }

}
```

6.5 HttpsConnectorThread.java

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.SocketTimeoutException;

/* services a one-way HTTPS connection (tunnel) between the client and the desired endpoint;
*   a second HttpsConnectorThread services data flow in the opoosite direction */
public class HttpsConnectorThread extends Thread {

    InputStream in;
    OutputStream out;

    // helps identify the threads; incremented every time a thread is created in the session
    private static int global_id;
    private int id;                 // local id

    public HttpsConnectorThread(InputStream in, OutputStream out) {
        this.in = in;
        this.out = out;

        // assign id to thread
        setId();
    }

    // prevent race condition
    private synchronized void setId() {
```

```
        global_id++;
        id = global_id;
//        System.out.println("Established HttpsConnectorThread:" + id);
    }

    @Override
    public void run() {

        try {
            byte[] responseBuffer = new byte[8192]; // buffer received data
            // will be > 0 if last read operation got some bytes
            int amountBytesToSend;

            // keep checking we have bytes
            while ((amountBytesToSend = in.read(responseBuffer, 0 ,
            responseBuffer.length)) != -1) {
                out.write(responseBuffer, 0, amountBytesToSend);          // send them
                out.flush();
            }
        } catch (IOException e) { // don't print these as they tend to flood console
            // System.out.println(this.toString()+" experienced IOException");
            //e.printStackTrace();
        }
    }

    @Override
    public String toString() {
        return "HttpsConnectorThread:"+id;
    }
}
```