Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

CSU34031 Advanced Telecommunications
Assignment #1: Web Proxy Server

Jack Gilbride, 17340868

February 25, 2020

Contents

1 Introduction

The task for this assignment was to design and implement our own Web Proxy Server. It defined five requirements for the server. The server should be able to:

1. Respond to HTTP & HTTPS requests, displaying each request on a management console. It should forward the request to the Web server and relay the response to the browser.
2. Handle Websocket connections.
3. Dynamically block selected URLs via a management console.
4. Efficiently cache requests locally and thus save bandwidth.
5. Handle multiple requests simultaneously by implementing a threaded server.

In this report I will document my approach towards this assignment, starting with the theory of the topic, then describing my own implementation in Java. I will finish with a summary. This will be followed by an appendix of the code deliverables produced for the assignment.

2 Theory of Topic

In this section I will describe the concepts that were used to realise a solution to my final web proxy server. In particular I will discuss the concept of a web proxy server itself, as well as HTTP and HTTPS.

2.1 Web Proxy Server

A web proxy is a local server, which fetches items from the web on behalf of a web client (e.g. a web browser) instead of the client fetching them directly. Proxy servers have two main purposes: to improve performance and to filter requests. Proxy servers improve performance by saving results of all requests for a certain amount of time. If the client is looking for a page from the server that the web proxy is already storing, the server will send that page to the client rather than requesting it from the server, provided the page has not changed since the proxy last saved it. This means that the proxy does not have to wait for the server to send the entire page, and also means that traffic is reduced on an institution's access link to the internet. This process is known as web caching. Proxy servers may filter requests by, for example, blocking a particular URL. An institution may use this to prevent its members from accessing a specific set of websites.

2.2 HTTP

Hyper Text Transfer Protocol, or HTTP, is an application layer protocol in the internet stack, which sits on top of Transmission Control Protocol (TCP). Typically, the client initiates a TCP connection (creates a socket) to the server on a particular port number. The server accepts the TCP connection. HTTP messages are exchanged between the client and the server. The TCP connection is then closed. HTTP is "stateless", i.e. the server maintains no information about past client request (because TCP already does it). There are two types of HTTP messages; requests and responses. Both types are in ASCII, human readable format. A HTTP request can have one of multiple method types. For example a GET request requests data from a specified resource. HTTP response messages are sent in response to requests, containing metadata along with (sometimes) a body of data. Part of a HTTP response message is a status code which indicates whether the request was successful, and if not, why not.

2.3 HTTPS

Hyper Text Transfer Protocol Secure (HTTPS) is an extension of HTTP. It is used for secure communication over a computer network, and is widely used on the Internet. In HTTPS, the communication protocol is encrypted using Transport Layer Security (TLS) or its predecessor Secure Sockets Layer (SSL). The contents of a HTTPS message are in the same format as that of a HTTP message, but it adds extra security benefit. While in HTTP data is sent in clear text, in HTTPS it is encrypted. This means that if someone is eavesdropping on your internet traffic, they cannot understand its contents. There is an added benefit other than encryption; when you connect to a HTTPS-secured server, your web browser checks the website's security certificate and verifies it was issued by a legitimate certificate authority. This helps you ensure that the site you're connected to are who they say they are.

3 Implementation

Building on the topics mentioned above, I will now discuss my implementation of the web proxy server in Java. The program works with the concepts of caching, blocking, HTTP and HTTPS. In particular I will talk about the overall architecture, responding to HTTP requests, responding to HTTPS requests, handling Websocket connections, dynamically blocking URLs and and finally, caching.

3.1 Overall Architecture

The program is made up of two main classes. The first class is *ProxyMultiThread*, which is broken up into static functionality and non static functionality. There are three static data structures to keep track of cached sites, blocked sites and connection threads. There are also static methods to access these from outside the class. The mainline creates an instance of *ProxyMultiThread* and starts it. Non static functionality of *ProxyMultiThread* is broken up into two separate parts. One part listens for connections to a specified port number from a browser (port 9999 in this case) and creates an instance of *ConnectionThread* to handle it. The other part is the management console, which runs on a separate thread. This watches System.in and allows the user to dynamically block URLs, view the blocked list and view the cached list. The one thread runs the connection functionality in *listen()* while the other runs the management console in *run()* to make the threading as simple as possible.
The other class of the program is *ConnectionThread*. As the name suggests, the class is a thread. As mentioned above, a new instance of this thread is created every time a connection is initiated by the browser. Each *ConnectionThread* is responsible for a particular connection between the client and the server. It is responsible for refusing access to blocked sites, returning cached HTTP pages, caching non-cached fetched HTTP pages, and setting up HTTPS connections between the client and server. In the HTTPS functionality, messages from the server to client are sent in *ConnectionThread*, while a separate Thread class called *ClientToServerHttpsTransmitter* is instantiated to also allow the client to send messages to the client in parallel.

3.2 Responding to HTTP

The code in Listing 1 sums up the functionality of how HTTP requests are handled within the *ConnectionThread*. Once the URL has been parsed out from the GET request, a HTTP Connection is made to the server at this URL. Some metadata is set for the connection, and a reader is setup from the connection. The HTTP content is read from the server, and a 200 "OK" message is sent back to signify that the transfer was successful. The HTTP content is then written to the client (the browser) to interpret and display to the user.

```
URL remoteURL = new URL(requestUrl);
HttpURLConnection serverConnection = (HttpURLConnection)
    remoteURL.openConnection();
serverConnection.setRequestProperty("Content-Type",
    "application/x-www-form-urlencoded");
serverConnection.setRequestProperty("Content-Language", "en-US");
serverConnection.setUseCaches(false);
serverConnection.setDoOutput(true);
BufferedReader proxyToServerBR = new BufferedReader(
                new InputStreamReader(serverConnection.getInputStream()));
String line = getResponse(200, false);
clientWriter.write(line);
while ((line = proxyToServerBR.readLine()) != null) {
        clientWriter.write(line);
        if (caching) {
                cacheWriter.write(line);
        }
}
clientWriter.flush();
```

Listing 1: Handling HTTP requests.

The code in Listing 1 is in the case that the HTTP content is in text form. There is also code to handle when the content is an image, as seen in Listing 2. If the file to be fetched is an image, the program uses *ImageIO* to get the content from the server, and again writes it back to the client.

```
if (isImage(fileExtension)) {
        URL remoteURL = new URL(requestUrl);
```

```
        BufferedImage image = ImageIO.read(remoteURL);

        if (image != null) {
                ImageIO.write(image, fileExtension.substring(1), fileToCache);
                String line = getResponse(200, false);
                clientWriter.write(line);
                clientWriter.flush();
                ImageIO.write(image, fileExtension.substring(1),
                    browserSocket.getOutputStream());

        } else {
                String error = getResponse(404, false);
                clientWriter.write(error);
                clientWriter.flush();
                return;
        }
}
```

Listing 2: Handling HTTP requests when the content is an image.

3.3 Responding to HTTPS

HTTP content could be handled in the manner that it was in Listing 1 and Listing 2 as data was transmitted in clear text. As a result the GET request and corresponding response could all be parsed easily. As mentioned earlier, HTTPS connections are encrypted so the same functionality cannot be used. Only the original CONNECT request can be parsed to identify the address and port number of the server. Any data sent after that is encrypted. As a result the web proxy must set up a secure end-to-end connection between the client and server. In Listing 3 we begin to set up the connection between the client and server.
The socket to the client already existed as it was passed from *ProxyMultiThread*, and we have been reading from and writing to it so the objects to do that already exist in the program. We need to create a socket to the server and IO objects for that also. Once these are created, a separate thread is created to handle messages from the client to the server.

```
// Create a new connection to the server
InetAddress serverAddress = InetAddress.getByName(url);
Socket serverSocket = new Socket(serverAddress, serverPort);
serverSocket.setSoTimeout(5000);

String line = getResponse(200, true);
clientWriter.write(line);
clientWriter.flush();

// Create the IO tools to read from and write to the server
BufferedWriter serverWriter = new BufferedWriter(new
    OutputStreamWriter(serverSocket.getOutputStream()));
BufferedReader serverReader = new BufferedReader(new
    InputStreamReader(serverSocket.getInputStream()));

// Spin off a seperate thread to send from the client to the server. This
    thread
// will handle communication from the server to the client.
ClientToServerHttpsTransmitter clientToServerHttps = new
    ClientToServerHttpsTransmitter(
                browserSocket.getInputStream(),
                    serverSocket.getOutputStream());
clientToServerHttpsThread = new Thread(clientToServerHttps);
```

```
clientToServerHttpsThread.start();
```

Listing 3: Handling HTTPS connections. Set up the IO tools and a separate thread to run alongside the current one.

Once the relevant IO objects are instantiated, communication between the client and server is relatively simple. While there is data to forward, the program listens to the server and forwards it to the client. When there is no more data to send, it stops. This can be seen in Listing 4.

```
byte[] buffer = new byte[4096];
int read;
do {
        read = serverSocket.getInputStream().read(buffer);
        if (read > 0) {
                browserSocket.getOutputStream().write(buffer, 0, read);
                if (serverSocket.getInputStream().available() < 1) {
                        browserSocket.getOutputStream().flush();
                }
        }
} while (read >= 0);
```

Listing 4: Handling HTTPS messages from the server to the client.

Similarly, the separate thread *ClientToServerHttpsTransmitter* handles messages from the client to the server. This time it listens to the client and forwards information to the server while there is still information to forward. This is in a different thread to allow messages to go back and forward between the client and server until all relevant information has been exchanged. This can be seen in Listing 5.

```
byte[] buffer = new byte[4096];
int read;
do {
        read = clientStream.read(buffer);
        if (read > 0) {
                serverStream.write(buffer, 0, read);
                if (clientStream.available() < 1) {
                        serverStream.flush();
                }
        }
} while (read >= 0);
```

Listing 5: Handling HTTPS messages from the client to the server.

3.4 Handling Websocket Connections

Websockets are automatically created between the client and the server when HTTP/HTTPS connections are made. This is to allow persistent connections between the client and server so that we do not have to re-establish connection on every request and response that we send. These approaches have already been mentioned in the previous two sections.

3.5 Dynamically Blocking URLs

The functionality to dynamically block URLs is found within the management console of *ProxyMultiThread*. The *run()* method scans user input for a list of predefined keywords. The keywords allow the user to see cached sites, see blocked sites, close the proxy server and ask for help. If one of these keywords is not entered, the input is assumed to be a URL to block. The URL is added to the map of blocked sites. When the proxy starts this data is pulled from a local file and when the proxy is closed this data is written back to the file. This is to allow the list of blocked sites to be maintained beyond one execution time of the proxy server, which would be needed in the real world.

```
default:
        blockedMap.put(userInput.toLowerCase(), userInput.toLowerCase());
        System.out.println("\n" + userInput + "␣blocked␣successfully␣\n");
        break;
```

Listing 6: Dynamically blocking URLs through the management console. If a keyword such as CACHED or HELP is not entered into the management console, the program interprets the input as a URL and adds it to the blocked sites.

Sites can be blocked dynamically as seen in Listing 6. However we also need the functionality to check this *blockedMap* now that there are URLs in it. This is the job of the code in Listing 7. If the URL contains any keyword from the blocked map, it is considered a blocked URL. There is a benefit to the *contains()* method rather than looking for an exact match, e.g. if someone wanted to block the site for Facebook in the management console, they may enter any of; "*Facebook*", "*www.facebook.com*", "*http:www.facebook.com:443*", and so on. Any of these are contained within the URL of Facebook so Facebook would be blocked. This enables users to easily block sites without having to know the exact URL.

```
public static boolean isBlocked(String url) {
        for (String key : blockedMap.keySet()) {
                if (url.contains(key)) {
                        return true;
                }
        }
        return false;
}
```

Listing 7: Checking which sites have been blocked by the management console.

When a website is blocked, there is no point in the proxy fetching it from its respective server. None of the previously mentioned HTTP or HTTPS functionality occurs, and an error code is sent back to the browser to signify that access has been blocked by the proxy server. This can be seen in Listing 8.

```
private void blockedSiteRequested() {
        try {
                BufferedWriter bufferedWriter = new BufferedWriter(new
                    OutputStreamWriter(browserSocket.getOutputStream()));
                String line = getResponse(403, false);
                bufferedWriter.write(line);
                bufferedWriter.flush();
        } catch (IOException e) {
                System.out.println("Error␣writing␣to␣client␣when␣requested␣a␣
                    blocked␣site");
        }
}
```

Listing 8: Responding to requests to blocked websites by returning an error code and returning.

So to summarize the blocking process; during the lifecycle of the program, any URL or subset of a URL may be added to the blocked sites, which is updated immediately. When a site is requested by the client, it is checked against the map of blocked sites. If the site is blocked, it is not fetched and instead a 403 error code is sent to the client. If the site is not blocked, the proxy server sets up a HTTP or HTTPS connection as normal. The file containing the blocked sites is updated at the end of every execution of the proxy server, and this file will be pulled from at the start of the next execution.

3.6 Caching

Just like the blocked sites, the cached sites are stored in a local data structure and pulled to and pushed to a file on startup and closing respectively. The data structure is a hashmap, where the URL is the key and

the webpage is the data. This allows files to be looked up quickly by the proxy server. As only one cache structure exists, it is static and is accessed in the static functions seen in Listing 9.

```
public static File getCachedPage(String url) {
        return cachedMap.get(url);
}

public static void addCachedPage(String urlString, File fileToCache) {
        cachedMap.put(urlString, fileToCache);
}
```

Listing 9: Accessing the static cache in ProxyMultiThread. The methods are public so allow individual ConnectionThreads to access the cache.

The obvious time to check the cache is after we check if the page is blocked, but before we try to pull it from its server. If the page is found in the cache, we can use this file instead of querying the server. The code to do this is much like the code to pull from the server, but is much faster as we are pulling directly from the local machine. This can be seen in Listing 10. Similarly, in the case where the cached file is an image, it is pulled from the cache and sent to the client as before.

```
BufferedReader cachedFileBufferedReader = new BufferedReader(
                new InputStreamReader(new FileInputStream(cachedFile)));
String response = getResponse(200, false);
clientWriter.write(response);
clientWriter.flush();
String line;
while ((line = cachedFileBufferedReader.readLine()) != null) {
        clientWriter.write(line);
}
clientWriter.flush();
```

Listing 10: Responding to the client when a cached file has been found. Much like with a non-cached request, it is outputted to the client through a BufferedWriter. However, it is not pulled from the server and instead straight from the cache.

For the cache to work effectively, it must also cache pages that are not already cached when they are pulled from the server. This can already be seen in Listing 1, where a HTTP page is written to the *cacheWriter* when caching is enabled. Once the request has completed, the contents of the cache writer are placed into the cache using the method in Listing 9.
One thing to note is that the process of checking the cache and caching received pages within the proxy only occurs for HTTP requests, i.e. not for HTTPS requests. This is because HTTPS is built for secure end to end connections. This proxy server uses a simple file structure so it is a security risk to store HTTPS pages in simple clear text when they are built for secure connections. The browser will also notice that the page is coming from the proxy server rather than the original site, i.e. the page will not come from somewhere with a valid security certificate. This would weaken one of the major benefits of HTTPS. For this reason caching is only implemented for HTTP requests. In practise, the proxy within a web browser will normally cache HTTPS pages but that is not practical in this implementation.

4 Summary

This report has described my attempt at creating a web proxy server through Java. To summarize, the following functionality was implemented:

- *ProxyMultiThread*
  - Static functionality
    - Access the cache
    - Access the list of blocked sites

        * Keep track of currently running *ConnectionThread*s
    - Main Thread
        * Initialize static variables
        * Initialize management console
        * Listen for connection requests from client
        * Create new *ConnectionThread*s
    - Management Console Thread
        * Block URLs
        * View blocked URLs
        * View cached URLs
        * Close the proxy server
        * View list of commands
- *ConnectionThread*
    - Handle requests
        * Check protocol of request
        * Check if the URL is blocked
        * Refuse blocked requests
        * Check if the URL's page is cached
        * Call HTTP functionality
        * Call HTTPS functionality
    - HTTP
        * Send cached files to client
        * Fetch non-cached files from server
        * Send non-cached files to client
        * Cache non-cached files
    - HTTPS
        * Set up socket with server
        * Set up read/write from server to client
        * Create new client to server thread
            · Set up read/write from client to server

5 Reflection

Upon reflection, this assignment was a very valuable and worthwhile exercise. It helped to cement my knowledge in course topics such as HTTP and caching, as a thorough understanding of these topics was needed to implement them in software. It also helped me to expand my knowledge of telecommunications beyond the scope of the course; for example I gained a deeper understanding of HTTPS and insight into how web browsers interact with proxy servers. Simply seeing the breakdown of requests in my terminal as I interacted with the browser was an interesting exercise. I will definitely look into similar projects in my own time in the near future to expand my knowledge on high level telecommunications concepts such as clients, servers and web proxies.

6 Appendix

```
/*
# Web Proxy Server
The objective of the exercise is to implement a Web Proxy Server.
```

```
- A Web proxy is a local server, which fetches items from the Web on behalf
    of a Web client instead of the client fetching them directly.
- This allows for caching of pages and access control.
The program should be able to:
1. Respond to HTTP & HTTPS requests, and should display each request on a
    management console. It should forward the request to the Web server and
    relay the response to the browser.
2. Handle Websocket connections.
3. Dynamically block selected URLs via the management console.
4. Effectively cache requests locally and thus save bandwidth. You must
    gather timing and bandwidth data to prove the efficiency of your proxy.
5. Handle multiple requests simultaneously by implementing a threaded server.
The program can be written in a programming language of your choice. However,
    you must ensure that you do not overuse any API or Library functionality
    that implements the majority of the work for you.
*/

import java.io.*;
import java.net.*;
import java.util.*;

public class ProxyMultiThread implements Runnable {

        // Constants for user input
        private static final String BLOCKED = "BLOCKED";
        private static final String CACHED = "CACHED";
        private static final String CLOSE = "CLOSE";
        private static final String HELP = "HELP";

        // Static local variables. Data structures to keep track of cached
            sites, blocked sites and threads.
        private static HashMap<String, File> cachedMap;
        private static HashMap<String, String> blockedMap;
        private static ArrayList<Thread> threadList;

        /*
         * Static methods.
         */
        // Get the cached page from the cache hashmap
        public static File getCachedPage(String url) {
                return cachedMap.get(url);
        }

        // Add a page to the cache hashmap
        public static void addCachedPage(String urlString, File fileToCache) {
                cachedMap.put(urlString, fileToCache);
        }

        // Go through the blocked list to check whether the site is blocked
        public static boolean isBlocked(String url) {
                for (String key : blockedMap.keySet()) {
                        if (url.contains(key)) {
                                return true;
                        }
                }
```

```
        return false;
    }

    /*
     * main method. Create the proxy and start listening for a client
        connection.
     */
    public static void main(String[] args) {
        ProxyMultiThread proxy = new ProxyMultiThread();
        proxy.listen();
    }

    /*
     * Local variables. The port number for the browser to listen on, a
        server
     * socket to listen to this port, a boolean to declare whether the
        proxy is
     * running, and a seperate thread for the management console.
     */
    private int browserPort;
    private ServerSocket browserListener;
    private volatile boolean running = true;
    private Thread managementConsole;

    /*
     * Constructor for the Proxy.
     */
    public ProxyMultiThread() {
        // initialise the data structures and client port number
        cachedMap = new HashMap<>();
        blockedMap = new HashMap<>();
        threadList = new ArrayList<>();
        browserPort = 9999;
        // Spin off a seperate thread to handle the management console
        managementConsole = new Thread(this);
        managementConsole.start();
        // Initalize the maps of cached site and blocked sites
        initializeCachedSites();
        initializeBlockedSites();
        // Start listening to the browser
        try {
            browserListener = new ServerSocket(browserPort);
            System.out.println("Waiting for client on port " +
                browserListener.getLocalPort());
            running = true;
        } catch (SocketException e) {
            System.out.println("Socket Exception when connecting
                to client");
        } catch (SocketTimeoutException e) {
            System.out.println("Timeout occured while connecting
                to client");
        } catch (IOException e) {
            System.out.println("IO exception when connecting to
                client");
        }
```

```
        }

        /*
         * Non static methods.
         */

        // Spin off a new thread for every new connection that the browser
            requests
        public void listen() {
                while (running) {
                        try {
                                Socket socket = browserListener.accept();
                                Thread thread = new Thread(new
                                    ConnectionThread(socket));
                                threadList.add(thread);
                                thread.start();
                        } catch (SocketException e) {
                                System.out.println("Server closed");
                        } catch (IOException e) {
                                System.out.println("Error creating new Thread
                                    from ServerSocket.");
                        }
                }
        }

        // Initalize the data structure for the cached sites by reading from
            the cache
        // file. If a cache file does not exist, create one
        private void initializeCachedSites() {
                try {
                        File cachedSites = new File("cachedSites.txt");
                        if (!cachedSites.exists()) {
                                System.out.println("Creating new cache file");
                                cachedSites.createNewFile();
                        } else {
                                FileInputStream cachedFileStream = new
                                    FileInputStream(cachedSites);
                                ObjectInputStream cachedObjectStream = new
                                    ObjectInputStream(cachedFileStream);
                                cachedMap = (HashMap<String, File>)
                                    cachedObjectStream.readObject();
                                cachedFileStream.close();
                                cachedObjectStream.close();
                        }
                } catch (IOException e) {
                        System.out.println("Error loading previously cached
                            sites file");
                } catch (ClassNotFoundException e) {
                        System.out.println("Class not found loading in
                            preivously cached sites file");
                }
        }

        // Initalize the data structure for the blocked sites by reading from
            the
```

```
        // blocked file. If a blocked file does not exist, create one
        private void initializeBlockedSites() {
                try {
                        File blockedSitesTxtFile = new
                            File("blockedSites.txt");
                        if (!blockedSitesTxtFile.exists()) {
                                System.out.println("No blocked sites found - 
                                    creating new file");
                                blockedSitesTxtFile.createNewFile();
                        } else {
                                FileInputStream blockedFileStream = new
                                    FileInputStream(blockedSitesTxtFile);
                                ObjectInputStream blockedObjectStream = new
                                    ObjectInputStream(blockedFileStream);
                                blockedMap = (HashMap<String, String>)
                                    blockedObjectStream.readObject();
                                blockedFileStream.close();
                                blockedObjectStream.close();
                        }
                } catch (IOException e) {
                        System.out.println("Error loading previously cached 
                            sites file");
                } catch (ClassNotFoundException e) {
                        System.out.println("Class not found loading in 
                            preivously cached sites file");
                }
        }

        // Close the server. Write back to the cached and blocked files. Join
            the
        // threads.
        private void closeServer() {
                System.out.println("Closing server");
                running = false;
                try {
                        FileOutputStream cachedFileStream = new
                            FileOutputStream("cachedSites.txt");
                        ObjectOutputStream cachedObjectStream = new
                            ObjectOutputStream(cachedFileStream);

                        cachedObjectStream.writeObject(cachedMap);
                        cachedObjectStream.close();
                        cachedFileStream.close();
                        System.out.println("Cached sites written");

                        FileOutputStream blockedFileStream = new
                            FileOutputStream("blockedSites.txt");
                        ObjectOutputStream blockedObjectStream = new
                            ObjectOutputStream(blockedFileStream);
                        blockedObjectStream.writeObject(blockedMap);
                        blockedObjectStream.close();
                        blockedFileStream.close();
                        System.out.println("Blocked site list saved");
                        try {
                                for (Thread thread : threadList) {
```

```
                                        if (thread.isAlive()) {
                                                thread.join();
                                        }
                                }
                        } catch (InterruptedException e) {
                                System.out.println("Interrupted exception
                                    when closing server.");
                        }

                } catch (IOException e) {
                        System.out.println("Error saving cache/blocked
                            sites");
                }
                try {
                        System.out.println("Terminating connection");
                        browserListener.close();
                } catch (Exception e) {
                        System.out.println("Exception closing proxy's server
                            socket");
                        e.printStackTrace();
                }
        }

        // The functionality of the management console. Watch System.in and
            look out for
        // commands. If a defined command is not entered, assume that the
            input is a URL
        // to be blocked and block it.
        @Override
        public void run() {
                Scanner terminalScanner = new Scanner(System.in);
                String userInput;
                while (running) {
                        System.out.println("Please enter a command. Enter
                            HELP to see the list of commands.");
                        userInput = terminalScanner.nextLine().toUpperCase();

                        switch (userInput) {
                                case BLOCKED:
                                        System.out.println("\nCurrently
                                            Blocked Sites");
                                        for (String key :
                                            blockedMap.keySet()) {
                                                System.out.println(key);
                                        }
                                        System.out.println();
                                        break;
                                case CACHED:
                                        System.out.println("\nCurrently
                                            Cached Sites");
                                        for (String key : cachedMap.keySet())
                                            {
                                                System.out.println(key);
                                        }
                                        System.out.println();
```

```
                                        break;
                                case CLOSE:
                                        running = false;
                                        closeServer();
                                        break;
                                case HELP:
                                        System.out.println("Enter BLOCKED to
                                            view the list of blocked URLs.");
                                        System.out.println("Enter CACHED to
                                            view the list of caches
                                            webpages");
                                        System.out.println("Enter CLOSE to
                                            close the proxy server.");
                                        System.out.println("Enter HELP to see
                                            the list of possible commands");
                                        System.out.println("Otherwise, enter
                                            a URL to add it to the blocked
                                            list.");
                                        break;
                                default:
                                        blockedMap.put(userInput.toLowerCase(),
                                            userInput.toLowerCase());
                                        System.out.println("\n" + userInput +
                                            " blocked successfully \n");
                                        break;
                                }
                        }
                        terminalScanner.close();
                }
}
```

Listing 11: ProxyMultiThread.java, which has static and non static functionality. The static functionality includes keeping track of the cache and blocked sites. The non static functionality includes creating new threads for every connection request, and running the management console.

```
import java.awt.image.BufferedImage;
import javax.imageio.ImageIO;
import java.io.*;
import java.net.*;

public class ConnectionThread implements Runnable {

        /*
         * Variables for this thread. A socket to connect to the client
            (browser), a
         * reader and writer to the client, and a seperate thread to handle
            HTTPS
         * communication from the client to the server.
         */
        private Socket browserSocket;
        private BufferedReader clientReader;
        private BufferedWriter clientWriter;
        private Thread clientToServerHttpsThread;

        /*
```

```
        * Constants for this class. To identify connection types and file
            types.
        */
        private static final String CONNECT = "CONNECT";
        private static final String GET = "GET";
        private static final String JPG = ".jpg";
        private static final String JPEG = ".jpeg";
        private static final String PNG = ".png";
        private static final String GIF = ".gif";

        /*
        * Constructor for this thread. Initalize the local variables.
        */
        public ConnectionThread(Socket browserSocket) {
                this.browserSocket = browserSocket;
                try {
                        this.browserSocket.setSoTimeout(2000);
                        clientReader = new BufferedReader(new
                            InputStreamReader(browserSocket.getInputStream()));
                        clientWriter = new BufferedWriter(new
                            OutputStreamWriter(browserSocket.getOutputStream()));
                } catch (IOException e) {
                        System.out.println("Error initializing new thread.");
                }
        }

        /*
        * Take the request from the client. If the site is blocked, do not
            fulfil the
        * request. Identify whether it is a HTTP or HTTPS request. Handle by
            passing
        * off to the appropriate method.
        */
        @Override
        public void run() {
                String requestString, requestType, requestUrl;
                try {
                        requestString = clientReader.readLine();
                        String[] splitRequest = splitRequest(requestString);
                        requestType = splitRequest[0];
                        requestUrl = splitRequest[1];
                } catch (IOException e) {
                        System.out.println("Error reading request from
                            client.");
                        return;
                }

                if (ProxyMultiThread.isBlocked(requestUrl)) {
                        System.out.println("Blocked site " + requestUrl + "
                            requested.");
                        blockedSiteRequested();
                        return;
                }

                switch (requestType) {
```

```
                        case CONNECT:
                                System.out.println("HTTPS request for : " +
                                    requestUrl);
                                handleHTTPSRequest(requestUrl);
                                break;
                        default:
                                File file =
                                    ProxyMultiThread.getCachedPage(requestUrl);
                                if (file == null) {
                                        System.out.println("HTTP request for
                                            : " + requestUrl + ". No cached
                                            page found.");
                                        fulfilNonCachedRequest(requestUrl);
                                } else {
                                        System.out.println("HTTP request for
                                            : " + requestUrl + ". Cached page
                                            found.");
                                        fulfilCachedRequest(file);
                                }
                                break;
                }
        }

        // Parse a CONNECT or GET request and return an array containing the
            URL and
        // port number
        private String[] splitRequest(String requestString) {
                String requestType, requestUrl;
                int requestSeparatorIndex;
                requestSeparatorIndex = requestString.indexOf(' ');
                requestType = requestString.substring(0,
                    requestSeparatorIndex);
                requestUrl = requestString.substring(requestSeparatorIndex +
                    1);
                requestUrl = requestUrl.substring(0, requestUrl.indexOf(' '));
                if (!requestUrl.substring(0, 4).equals("http")) {
                        requestUrl = "http://" + requestUrl;
                }
                return new String[] { requestType, requestUrl };
        }

        // Fetch a page from the cache for the client
        private void fulfilCachedRequest(File cachedFile) {
                try {
                        String fileExtension =
                            cachedFile.getName().substring(cachedFile.getName().lastInc
                        if (isImage(fileExtension)) {
                                BufferedImage image =
                                    ImageIO.read(cachedFile);
                                if (image == null) {
                                        System.out.println("Image " +
                                            cachedFile.getName() + " was
                                            null");
                                        String response = getResponse(404,
                                            false);
```

```
                                clientWriter.write(response);
                                clientWriter.flush();
                        } else {
                                String response = getResponse(200,
                                    false);
                                clientWriter.write(response);
                                clientWriter.flush();
                                ImageIO.write(image,
                                    fileExtension.substring(1),
                                    browserSocket.getOutputStream());
                        }
                } else {
                        BufferedReader cachedFileBufferedReader = new
                            BufferedReader(
                                        new InputStreamReader(new
                                            FileInputStream(cachedFile)));
                        String response = getResponse(200, false);
                        clientWriter.write(response);
                        clientWriter.flush();
                        String line;
                        while ((line =
                            cachedFileBufferedReader.readLine()) !=
                            null) {
                                clientWriter.write(line);
                        }
                        clientWriter.flush();

                        if (cachedFileBufferedReader != null) {
                                cachedFileBufferedReader.close();
                        }
                }
                if (clientWriter != null) {
                        clientWriter.close();
                }
        } catch (IOException e) {
                System.out.println("Error sending cached file to
                    client");
        }
}

// Set up a server connection, fetch the appropriate content. Return
    the content
// to the client and also add it to the cache.
private void fulfilNonCachedRequest(String requestUrl) {
        try {
                int fileExtensionIndex = requestUrl.lastIndexOf(".");
                String fileExtension;
                fileExtension =
                    requestUrl.substring(fileExtensionIndex,
                    requestUrl.length());
                String fileName = requestUrl.substring(0,
                    fileExtensionIndex);
                fileName = fileName.substring(fileName.indexOf('.') +
                    1);
                fileName = fileName.replace("/", "__");
```

```
                fileName = fileName.replace('.', '_');

                if (fileExtension.contains("/")) {
                        fileExtension = fileExtension.replace("/",
                            "__");
                        fileExtension = fileExtension.replace('.',
                            '_');
                        fileExtension += ".html";
                }
                fileName = fileName + fileExtension;
                boolean caching = true;
                File fileToCache = null;
                BufferedWriter cacheWriter = null;

                try {
                        fileToCache = new File("cache/" + fileName);
                        if (!fileToCache.exists()) {
                                fileToCache.createNewFile();
                        }
                        cacheWriter = new BufferedWriter(new
                            FileWriter(fileToCache));
                } catch (IOException e) {
                        System.out.println("Error trying to cache " +
                            fileName);
                        caching = false;
                } catch (NullPointerException e) {
                        System.out.println("Null pointer opening file
                            " + fileName);
                }
                if (isImage(fileExtension)) {
                        URL remoteURL = new URL(requestUrl);
                        BufferedImage image = ImageIO.read(remoteURL);

                        if (image != null) {
                                ImageIO.write(image,
                                    fileExtension.substring(1),
                                    fileToCache);
                                String line = getResponse(200, false);
                                clientWriter.write(line);
                                clientWriter.flush();
                                ImageIO.write(image,
                                    fileExtension.substring(1),
                                    browserSocket.getOutputStream());

                        } else {
                                System.out.println("Sending 404 to
                                    client as image wasn't received
                                    from server" + fileName);
                                String error = getResponse(404,
                                    false);
                                clientWriter.write(error);
                                clientWriter.flush();
                                return;
                        }
                } else {
```

```
                                URL remoteURL = new URL(requestUrl);
                                HttpURLConnection serverConnection =
                                    (HttpURLConnection)
                                    remoteURL.openConnection();
                                serverConnection.setRequestProperty("Content-Type",
                                    "application/x-www-form-urlencoded");
                                serverConnection.setRequestProperty("Content-Language"
                                    "en-US");
                                serverConnection.setUseCaches(false);
                                serverConnection.setDoOutput(true);
                                BufferedReader proxyToServerBR = new
                                    BufferedReader(
                                                new
                                                    InputStreamReader(serverConnection
                                String line = getResponse(200, false);
                                clientWriter.write(line);
                                while ((line = proxyToServerBR.readLine()) !=
                                    null) {
                                        clientWriter.write(line);
                                        if (caching) {
                                                cacheWriter.write(line);
                                        }
                                }
                                clientWriter.flush();
                                if (proxyToServerBR != null) {
                                        proxyToServerBR.close();
                                }
                        }

                        if (caching) {
                                cacheWriter.flush();
                                ProxyMultiThread.addCachedPage(requestUrl,
                                    fileToCache);
                        }
                        if (cacheWriter != null) {
                                cacheWriter.close();
                        }
                        if (clientWriter != null) {
                                clientWriter.close();
                        }
                } catch (Exception e) {
                        System.out.println("Error sending non cached page to
                            client");
                }
        }

        // Handle a HTTPS CONNECT request
        private void handleHTTPSRequest(String requestUrl) {
                String url = requestUrl.substring(7);
                String pieces[] = url.split(":");
                url = pieces[0];
                int serverPort = Integer.valueOf(pieces[1]);

                try {
                        // Only the first line of the CONNECT request has
```

```
        been processed. Clear the
    // rest.
    for (int i = 0; i < 5; i++) {
            clientReader.readLine();
    }
    // Create a new connection to the server
    InetAddress serverAddress =
        InetAddress.getByName(url);
    Socket serverSocket = new Socket(serverAddress,
        serverPort);
    serverSocket.setSoTimeout(5000);

    String line = getResponse(200, true);
    clientWriter.write(line);
    clientWriter.flush();

    // Create the IO tools to read from and write to the
        server
    BufferedWriter serverWriter = new BufferedWriter(new
        OutputStreamWriter(serverSocket.getOutputStream()));
    BufferedReader serverReader = new BufferedReader(new
        InputStreamReader(serverSocket.getInputStream()));

    // Spin off a seperate thread to send from the client
        to the server. This thread
    // will handle communication from the server to the
        client.
    ClientToServerHttpsTransmitter clientToServerHttps =
        new ClientToServerHttpsTransmitter(
                    browserSocket.getInputStream(),
                        serverSocket.getOutputStream());
    clientToServerHttpsThread = new
        Thread(clientToServerHttps);
    clientToServerHttpsThread.start();

    // Handle communication from the server to the client
    try {
            byte[] buffer = new byte[4096];
            int read;
            do {
                    read =
                        serverSocket.getInputStream().read(buffer)
                    if (read > 0) {
                            browserSocket.getOutputStream().write
                                0, read);
                            if
                                (serverSocket.getInputStream().ava
                                < 1) {
                                    browserSocket.getOutputStream
                            }
                    }
            } while (read >= 0);
    } catch (SocketTimeoutException e) {
            System.out.println("Socket␣timeout␣during␣
                HTTPs␣connection");
```

```
                } catch (IOException e) {
                        System.out.println("Error handling HTTPs
                            connection");
                }

                // Close the resources
                closeResources(serverSocket, serverReader,
                    serverWriter, clientWriter);

            } catch (SocketTimeoutException e) {
                String line = getResponse(504, false);
                try {
                        clientWriter.write(line);
                        clientWriter.flush();
                } catch (IOException x) {
                }
            } catch (Exception e) {
                System.out.println("Error on HTTPS " + requestUrl);
            }
        }

        // Respond to the browser with a 403 Error Code
        private void blockedSiteRequested() {
            try {
                BufferedWriter bufferedWriter = new
                    BufferedWriter(new
                    OutputStreamWriter(browserSocket.getOutputStream()));
                String line = getResponse(403, false);
                bufferedWriter.write(line);
                bufferedWriter.flush();
            } catch (IOException e) {
                System.out.println("Error writing to client when
                    requested a blocked site");
            }
        }

        // Check whether the file extension is that of an image
        private boolean isImage(String fileExtension) {
            return fileExtension.contains(PNG) ||
                fileExtension.contains(JPG) ||
                fileExtension.contains(JPEG)
                        || fileExtension.contains(GIF);
        }

        // Return certain response strings based on the error code passed.
            There are two
        // cases for 200, 'OK' and 'Connection established', in which case
            check the
        // boolean.
        private String getResponse(int code, boolean connectionEstablished) {
            String response = "";
            switch (code) {
                case 200:
                    if (connectionEstablished) {
                        response = "HTTP/1.0 200 Connection
```

```
                                        established\r\nProxy-Agent:␣
                                        ProxyServer/1.0\r\n\r\n";
                                } else {
                                        response = "HTTP/1.0␣200␣
                                        OK\nProxy-agent:␣
                                        ProxyServer/1.0\n\r\n";
                                }
                                break;
                        case 403:
                                response = "HTTP/1.0␣403␣Access␣Forbidden␣
                                    \nUser-Agent:␣ProxyServer/1.0\n\r\n";
                                break;
                        case 404:
                                response = "HTTP/1.0␣404␣NOT␣FOUND␣
                                    \nProxy-agent:␣ProxyServer/1.0\n\r\n";
                                break;
                        case 504:
                                response = "HTTP/1.0␣504␣Timeout␣Occured␣
                                    after␣10s\nUser-Agent:␣
                                    ProxyServer/1.0\n\r\n";
                                break;
                        default:
                                break;
                }
                return response;
        }

        // Close the passed resources
        private void closeResources(Socket serverSocket, BufferedReader
            serverReader, BufferedWriter serverWriter,
                        BufferedWriter clientWriter) throws IOException {
                if (serverSocket != null) {
                        serverSocket.close();
                }
                if (serverReader != null) {
                        serverReader.close();
                }
                if (serverWriter != null) {
                        serverWriter.close();
                }
                if (clientWriter != null) {
                        clientWriter.close();
                }
        }

        // Seperate class to handle HTTPS transmission from the client to the
            server. In
        // practise it is spun off as a seperate thread to run alongside
            transmission
        // from the server to the client.
        class ClientToServerHttpsTransmitter implements Runnable {

                InputStream clientStream;
                OutputStream serverStream;
```

```
                public ClientToServerHttpsTransmitter(InputStream
                    proxyToClientIS, OutputStream proxyToServerOS) {
                        this.clientStream = proxyToClientIS;
                        this.serverStream = proxyToServerOS;
                }

                @Override
                public void run() {
                        try {
                                byte[] buffer = new byte[4096];
                                int read;
                                do {
                                        read = clientStream.read(buffer);
                                        if (read > 0) {
                                                serverStream.write(buffer, 0,
                                                    read);
                                                if (clientStream.available()
                                                    < 1) {
                                                        serverStream.flush();
                                                }
                                        }
                                } while (read >= 0);
                        } catch (SocketTimeoutException e) {
                        } catch (IOException e) {
                                System.out.println("Proxy to client HTTPS
                                    read timed out");
                        }
                }
        }
}
```

Listing 12: ConnectionThread.java, which handles individual HTTP and HTTP connections between the client and server.