Lavigne Tom

CSU33032 Project 1
A Web Proxy Server

1. Introduction to ProxyServer Functionality:

This proxy server allows clients to access web content through it, with the ability to cache web pages and block certain sites. It provides a basic command-line interface for server management. Here is the detail of the protocol :

Server Initialization: The ProxyServer class initializes a server socket on a specified port to listen for incoming client connections.

```
/**
 * Initializes the server socket.
 */
1 usage
public void init() {
    try {
        socketServer = new ServerSocket(port);
        System.out.println("Waiting for clients on port " + socketServer.getLocalPort() + "..");
        running = true;
    } catch (Exception e) {
        System.out.println("Error starting the server: " + e.getMessage());
    }
}
```

Main Method: The main method creates an instance of ProxyServer, initializes it, and starts accepting connections.

```
public static void main(String[] args) {
    int port = 5555; // Default port for HTTP
    ProxyServer proxyServer = new ProxyServer(port);
    proxyServer.init();
    proxyServer.acceptConnections();
}
```

Run Method: The run method, implemented as part of the Runnable interface, provides a command-line interface for the user to interact with the proxy server. The user can input commands to block new sites, view blocked sites, view cached sites, or close the server.

```
@Override
public void run() {
    Scanner scanner = new Scanner(System.in);
    String command;
    while (running) {
        System.out.println("Enter a site to block, or type \"blocked\" to see blocked sites, \"cached\" to see cached sites, or \"close\" to close the server.");
        command = scanner.nextLine();
        if (command.equalsIgnoreCase( anotherString: "blocked")) {
            System.out.println("\nCurrently Blocked Sites");
            for (String key : blockedSites.keySet()) {
                System.out.println(key);
            }
            System.out.println();
        } else if (command.equalsIgnoreCase( anotherString: "cached")) {
            System.out.println("\nCurrently Cached Sites");
            for (String key : cache.keySet()) {
                System.out.println(key);
            }
            System.out.println();
        } else if (command.equals("close")) {
            running = false;
            closeServer();
        } else {
            blockedSites.put(command, command);
            System.out.println("\n" + command + " blocked successfully \n");
        }
    }
    scanner.close();
}
```

Cache and Blocking Setup: It maintains two HashMaps: cache for storing cached websites and blockedSites for storing blocked URLs. These data structures are initialized and loaded from files (cache.txt and block.txt) if they exist, otherwise new files are created.

```
public ProxyServer(int port) {
    this.port = port;
    cache = new HashMap<>();
    blockedSites = new HashMap<>();
    threadList = new ArrayList<>();
    new Thread( task: this).start();

    try {
        // Load cached sites from file if they exist
        File cachedSites = new File( pathname: "cache.txt");
        if (!cachedSites.exists()) {
            System.out.println("No cached sites found - creating new file");
            cachedSites.createNewFile();
        } else {
            FileInputStream fileInputStream = new FileInputStream(cachedSites);
            ObjectInputStream objectInputStream = new ObjectInputStream(fileInputStream);
            cache = (HashMap<String, File>) objectInputStream.readObject();
            fileInputStream.close();
            objectInputStream.close();
        }

        // Load blocked sites from file if they exist
        File blockedSitesTxtFile = new File( pathname: "block.txt");
        if (!blockedSitesTxtFile.exists()) {
            System.out.println("No blocked sites found - creating new file");
            blockedSitesTxtFile.createNewFile();
        } else {
            FileInputStream fileInputStream = new FileInputStream(blockedSitesTxtFile);
            ObjectInputStream objectInputStream = new ObjectInputStream(fileInputStream);
            blockedSites = (HashMap<String, String>) objectInputStream.readObject();
            fileInputStream.close();
            objectInputStream.close();
        }
    } catch (IOException e) {
        System.out.println("Error loading previously cached sites file");
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        System.out.println("Class not found loading in previously cached sites file");
        e.printStackTrace();
    }

}
```

Accept Connections: The acceptConnections method continuously listens for incoming client connections. When a new client connection is accepted, it spawns a new ClientHandler thread to handle the client's request.

```
/**
 * Accepts incoming client connections and starts a new thread to handle each connection.
 */
1 usage
public void acceptConnections() {
    while (running) {
        try {
            Socket clientSocket = socketServer.accept();
            System.out.println("New connection: " + clientSocket);
            ClientHandler clientHandler = new ClientHandler(clientSocket);
            Thread thread = new Thread(clientHandler);
            threadList.add(thread);
            thread.start();
        } catch (Exception e) {
            System.out.println("Error with the new connection: " + e.getMessage());
        }
    }
}
```

Close Server: The closeServer method is called when the user decides to close the server. It saves the cache and blocked sites to files, waits for all client handling threads to finish, and then closes the server socket.

```
private void closeServer() {
    System.out.println("\nClosing Server..");
    running = false;
    try {
        // Save cache to file
        FileOutputStream fileOutputStream = new FileOutputStream( name: "cache.txt");
        ObjectOutputStream objectOutputStream = new ObjectOutputStream(fileOutputStream);
        objectOutputStream.writeObject(cache);
        objectOutputStream.close();
        fileOutputStream.close();
        System.out.println("Cache saved");

        // Save blocked sites to file
        FileOutputStream fileOutputStream2 = new FileOutputStream( name: "block.txt");
        ObjectOutputStream objectOutputStream2 = new ObjectOutputStream(fileOutputStream2);
        objectOutputStream2.writeObject(blockedSites);
        objectOutputStream2.close();
        fileOutputStream2.close();
        System.out.println("Blocked Sites saved");

        // Wait for all client threads to close
        try {
            for (Thread thread : threadList) {
                if (thread.isAlive()) {
                    System.out.print("Waiting on " + thread.getId() + " to close..");
                    thread.join();
                    System.out.println(" closed");
                }
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
            }
        } catch (IOException e) {
            System.out.println("Error saving cache/blocked sites");
            e.printStackTrace();
        }
        try {
            // Close server socket
            System.out.println("Terminating Connection");
            socketServer.close();
        } catch (Exception e) {
            System.out.println("Exception closing proxy's server socket");
            e.printStackTrace();
        }

    }
```

Utility Methods: Utility methods like getCachedPage, addCachedPage, and isBlocked are provided to manipulate the cache and check if a site is blocked.

```
/**
 * Retrieves a cached web page for a given URL.
 * @param url The URL of the web page.
 * @return The cached file if it exists, null otherwise.
 */
2 usages
public static File getCachedPage(String url) {
    return cache.get(url);
}

/**
 * Adds a web page to the cache.
 * @param urlString The URL of the web page.
 * @param fileToCache The file to be cached.
 */
2 usages
public static void addCachedPage(String urlString, File fileToCache) { cache.put(urlString, fileToCache); }

/**
 * Checks if a URL is blocked.
 * @param url The URL to be checked.
 * @return True if the URL is blocked, false otherwise.
 */
2 usages
public static boolean isBlocked(String url) { return blockedSites.get(url) != null; }
```

2. Explanation of the ClientHandler Class:

The ClientHandler class encapsulates the logic for handling client requests within the ProxyServer. It implements the Runnable interface, enabling concurrent handling of multiple client connections. Upon receiving a client request, the ClientHandler parses the request, determines its type (HTTP or HTTPS), and processes it accordingly. It interacts with the client's input and output streams to relay data and responses. Additionally, the ClientHandler manages error handling, caching, and HTTPS tunneling, making it a crucial component for mediating client-server communication. Here is the detail of the protocol :

ClientHandler Initialization: The ClientHandler class is responsible for managing client requests. Upon receiving a client socket, it initializes input and output streams to communicate with the client.

```
public ClientHandler(Socket clientSocket){
    this.clientSocket = clientSocket;
    try{
        this.clientSocket.setSoTimeout(20000);
        proxyClientReader = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
        proxyClientWriter = new BufferedWriter(new OutputStreamWriter(clientSocket.getOutputStream()));
    }
    catch (IOException e) {
        e.printStackTrace();
    }
}
```

Handling Client Requests: The run method of ClientHandler processes incoming client requests. It reads the request header to determine the type of request and the requested URL. If the URL is not blocked, it calls the appropriate method (HTTPS, HTTP GET,HTTP POST), otherwise it treats the blocked site. If the HTTP GET request is found in the cache it call the method which will send this file directly.

```
@Override
public void run() {

    // Get Request from the client
    String requestString;
    try{
        requestString = proxyClientReader.readLine();
    } catch (IOException e) {
        e.printStackTrace();
        System.out.println("Error reading request from the client");
        return;
    }
    if(requestString!=null) {
        // Parsing out URL

        // Get request type + URL
        String request = requestString.substring(0, requestString.indexOf(' '));

        String urlString = requestString.substring( beginIndex: requestString.indexOf(' ') + 1);
        urlString = urlString.substring(0, urlString.indexOf(' '));

        // Prepend http:// if necessary to form a correct URL
        if (urlString.startsWith("/")) {
            urlString =urlString.substring( beginIndex: 1);
        }
        if (!urlString.startsWith("http")) {
            String temp = "http://";
            urlString = temp + urlString;
        }

        // Check if the site is blocked
```

```
        if (ProxyServer.isBlocked(urlString)) {
            System.out.println("Blocked site requested: " + urlString);
            blockedSiteRequested();
            return;
        }

        // Check the request type
        if (request.equals("CONNECT")) {
            System.out.println("HTTPS Request for: " + urlString + "\n");
            handleHTTPS(urlString);
        } else if (request.equals("POST")) {
            System.out.println("HTTP POST for: " + urlString + "\n");
            // Handling POST request
            handlePOSTRequest(urlString, requestString);
        } else {
            // Check if there's a cached copy
            File file;
            if ((file = ProxyServer.getCachedPage(urlString)) != null) {
                System.out.println("Cached Copy found for: " + urlString + "\n");
                sendCached(file);
            } else {
                System.out.println("HTTP GET for: " + urlString + "\n");
                sendNonCached(urlString);
            }
        }
    }else{
        //System.out.println("null request");
    }
}
```

Cached Page Delivery: If the requested page is cached, the sendCached method retrieves the cached page from disk and sends it to the client. If the page is an image, it reads and transmits the image data. For text-based files, it reads and transmits the file line by line.

```
private void sendCached(File cachedFile){
    // Sending a File containing cached web page
    try{
        String extension = cachedFile.getName().substring(cachedFile.getName().lastIndexOf( ch: '.'));
        String resp;
        // Identifying images, write data to the client using buffered image.
        if((extension.contains(".png")) || extension.contains(".jpg") ||
                extension.contains(".jpeg") || extension.contains(".gif")){
            // Reading in the image from storage
            BufferedImage image = ImageIO.read(cachedFile);

            if(image == null ){
                System.out.println("Image " + cachedFile.getName() + " was null");
                resp = "HTTP/1.0 404 NOT FOUND \n" +
                        "Proxy-agent: ProxyServer/1.0\n" +
                        "\r\n";
                proxyClientWriter.write(resp);
                proxyClientWriter.flush();
            } else {
                resp = "HTTP/1.0 200 OK\n" +
                        "Proxy-agent: ProxyServer/1.0\n" +
                        "\r\n";
                proxyClientWriter.write(resp);
                proxyClientWriter.flush();
                ImageIO.write(image, extension.substring( beginIndex: 1), clientSocket.getOutputStream());
            }
        }
```

```
        // Standard text-based file requested
        else {
            BufferedReader cachedFileBufferedReader = new BufferedReader(new InputStreamReader(new FileInputStream(cachedFile)));

            resp = "HTTP/1.0 200 OK\n" +
                    "Proxy-agent: ProxyServer/1.0\n" +
                    "\r\n";
            proxyClientWriter.write(resp);
            proxyClientWriter.flush();

            String line;
            while((line = cachedFileBufferedReader.readLine()) != null){
                proxyClientWriter.write(line);
            }
            proxyClientWriter.flush();

            // Closing resources
            if(cachedFileBufferedReader != null){
                cachedFileBufferedReader.close();
            }
        }
        // Closing down resources
        if(proxyClientWriter != null){
            proxyClientWriter.close();
        }

    } catch (IOException e) {
        System.out.println("Error Sending Cached file to client");
        e.printStackTrace();
    }
}
```

Non-Cached Page Handling: If the requested page is not cached, the sendNonCached method retrieves the page from the remote server. Whether the page is an image or a text-based file, it downloads and caches the page (in “cached” directory after threat the filename) before sending it to the client.

```
private void sendNonCached(String urlString) {
    try {
        // Separation of the file name & extension
        int fileExtensionIndex = urlString.lastIndexOf( str: ".");
        String extension;
        extension = urlString.substring(fileExtensionIndex, urlString.length());
        String fileName = urlString.substring(0, fileExtensionIndex);
        fileName = fileName.substring( beginIndex: 7);
        //treatment of both to remove special character
        fileName = fileName.replace( target: "/", replacement: "__");
        fileName = fileName.replace( oldChar: '.', newChar: '_');
        extension = extension.replace( oldChar: '?', newChar: '_');
        fileName = fileName.replace( oldChar: '?', newChar: '_');
        urlString=urlString.substring( beginIndex: 7);
        urlString="https://"+urlString;
        if (extension.contains("/")) {
            extension = extension.replace( target: "/", replacement: "__");
            extension = extension.replace( oldChar: '.', newChar: '_');
            extension += ".html";
        }

        fileName = fileName + extension;
        boolean caching = true;
        File fileToCache = null;
        BufferedWriter fileToCacheBufferedWriter = null;

        //Create the File in the "cached" directory to put the data
        try {
            fileToCache = new File( pathname: "cached/" + fileName);
```

```
        File parentDir = fileToCache.getParentFile();
        if (parentDir != null && !parentDir.exists()) {
            parentDir.mkdirs();
        }
        if (!fileToCache.exists()) {
            new File( pathname: "cached/").mkdirs();
            fileToCache.createNewFile();
        }
        fileToCacheBufferedWriter = new BufferedWriter(new FileWriter(fileToCache));
    } catch (IOException e) {
        System.out.println("Couldn't cache: " + fileName);
        caching = false;
        e.printStackTrace();
    } catch (NullPointerException e) {
        System.out.println("NPE opening file");
    }

    URL remoteURL = new URL(urlString);
    // If the document is an image, using of beffered image to write in the file
    if ((extension.contains(".png")) || extension.contains(".jpg") ||
            extension.contains(".jpeg") || extension.contains(".gif") || extension.contains(".ico")) {
        BufferedImage image = ImageIO.read(remoteURL);
        if (image != null) {
            assert fileToCache != null;
            ImageIO.write(image, extension.substring( beginIndex: 1), fileToCache);

            String line = "HTTP/1.0 200 OK\n" +
                    "Proxy-agent: ProxyServer/1.0\n" +
                    "\r\n";
```

```
            proxyClientWriter.write(line);
            proxyClientWriter.flush();
            ImageIO.write(image, extension.substring( beginIndex: 1), clientSocket.getOutputStream());
        } else {
            System.out.println("Sending 404 to client as image wasn't received from server"
                    + fileName);
            String error = "HTTP/1.0 404 NOT FOUND\n" +
                    "Proxy-agent: ProxyServer/1.0\n" +
                    "\r\n";
            proxyClientWriter.write(error);
            proxyClientWriter.flush();
            return;
        }
    } else {
        // If the document is a text, retrieve data from the server and write it line by line
        HttpURLConnection proxyServerConnection = (HttpURLConnection) remoteURL.openConnection();
        proxyServerConnection.setRequestProperty("Content-Type",
                "application/x-www-form-urlencoded");
        proxyServerConnection.setRequestProperty("Content-Language", "en-US");
        proxyServerConnection.setUseCaches(false);
        proxyServerConnection.setDoOutput(true);
        BufferedReader proxyToServerBufferedReader = new BufferedReader(new InputStreamReader(proxyServerConnection.getInpu

        String line = "HTTP/1.0 200 OK\n" +
                "Proxy-agent: ProxyServer/1.0\n" +
                "\r\n";
        proxyClientWriter.write(line);

        while ((line = proxyToServerBufferedReader.readLine()) != null) {
```

```
            proxyClientWriter.write(line);
            if (caching) {
                if (fileToCacheBufferedWriter != null) {
                    fileToCacheBufferedWriter.write(line);
                }
            }
        }

        proxyClientWriter.flush();
        if (proxyToServerBufferedReader != null) {
            proxyToServerBufferedReader.close();
        }
    }
    //add file to hashmap
    if (caching) {
        if (fileToCacheBufferedWriter != null) {
            fileToCacheBufferedWriter.flush();
        }
        urlString=urlString.substring( beginIndex: 8);
        urlString="http://"+urlString;
        ProxyServer.addCachedPage(urlString, fileToCache);
    }

    if (fileToCacheBufferedWriter != null) {
        fileToCacheBufferedWriter.close();
    }

    if (proxyClientWriter != null) {
        proxyClientWriter.close();
    }
```

HTTPS Connection Handling: The handleHTTPS method manages HTTPS requests by establishing a connection with the remote server, relaying data between client and server, and handling timeouts. It manages the sending of server data to the client and creates a thread to do the same in the other direction.

```
private void handleHTTPS(String urlString){
    // Extracting the URL and port of the remote
    String url = urlString.substring( beginIndex: 7);
    String[] pieces = url.split( regex: ":");
    url = pieces[0];
    int port  = Integer.parseInt(pieces[1]);

    try{
        // discard the rest of the initial data on the stream
        for(int i=0;i<5;i++){
            proxyClientReader.readLine();
        }

        // Get IP with URL thanks to DNS
        InetAddress address = InetAddress.getByName(url);

        // Opening a socket to the remote server
        Socket proxyToServerSocket = new Socket(address, port);
        proxyToServerSocket.setSoTimeout(50000);

        // Confirm connexion to the client
        String line = "HTTP/1.0 200 Connection established\r\n" +
                "Proxy-Agent: ProxyServer/1.0\r\n" +
                "\r\n";
        proxyClientWriter.write(line);
        proxyClientWriter.flush();
        // Both client and remote will start sending data to proxy at this point
        // Proxy needs to asynchronously read data from each party and send it to the other party

        // Creating a Buffer between proxy and remote
        BufferedWriter proxyToServerBufferedWriter = new BufferedWriter(new OutputStreamWriter(proxyToServerSocket.getOutput
        BufferedReader proxyToServerBufferedReader = new BufferedReader(new InputStreamReader(proxyToServerSocket.getInputSt

        // Creating a new thread to listen to the client and transmit to the server
        ClientServerHTTPS clientToServerHttps =
                new ClientServerHTTPS(clientSocket.getInputStream(), proxyToServerSocket.getOutputStream());
        HTTPSClientServer = new Thread(clientToServerHttps);
        HTTPSClientServer.start();


        // Listening to the remote server and relaying to the client
        try {
            byte[] buffer = new byte[4096];
            int r;
            do {
                r = proxyToServerSocket.getInputStream().read(buffer);
                if (r > 0) {
                    clientSocket.getOutputStream().write(buffer,  off: 0, r);
                    if (proxyToServerSocket.getInputStream().available() < 1) {
                        clientSocket.getOutputStream().flush();
                    }
                }
            } while (r >= 0);
        }
        catch (SocketTimeoutException e) {
        }
        catch (IOException e) {
            e.printStackTrace();
        }
        // Closing down resources
        if(proxyToServerSocket != null){
            proxyToServerSocket.close();
        }
        if(proxyToServerBufferedReader != null){
            proxyToServerBufferedReader.close();
        }
        if(proxyToServerBufferedWriter != null){
            proxyToServerBufferedWriter.close();
        }
        if(proxyClientWriter != null){
            proxyClientWriter.close();
        }
    } catch (SocketTimeoutException e) {
        String line = "HTTP/1.0 504 Timeout Occurred after 10s\n" +
                "User-Agent: ProxyServer/1.0\n" +
                "\r\n";
        try{
            proxyClientWriter.write(line);
            proxyClientWriter.flush();
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }
    catch (Exception e){
```

Thread Management: For HTTPS requests, a separate thread (ClientServerHTTPS) is created to transmit data between the client and server. Thread resources are properly closed after data transmission.

```
class ClientServerHTTPS implements Runnable{
    3 usages
    InputStream proxyClientInput;
    3 usages
    OutputStream proxyServerOutput;
    1 usage
    public ClientServerHTTPS(InputStream proxyClientInput, OutputStream proxyServerOutput) {
        this.proxyClientInput = proxyClientInput;
        this.proxyServerOutput = proxyServerOutput;
    }
    @Override
    public void run(){
        try {
            // Reading byte by byte from the client and sending directly to the server
            byte[] buffer = new byte[4096];
            int r;
            do {
                r = proxyClientInput.read(buffer);
                if (r > 0) {
                    proxyServerOutput.write(buffer, off: 0, r);
                    if (proxyClientInput.available() < 1) {
                        proxyServerOutput.flush();
                    }
                }
            } while (r >= 0);
        }
        catch (SocketTimeoutException ste) {...}
        catch (IOException e) {...}
    }
}
```

Blocked Site Response: If a requested site is blocked, a 403 Forbidden response is sent to the client to indicate access restriction.

```
private void blockedSiteRequested(){
    try {
        BufferedWriter bufferedWriter = new BufferedWriter(new OutputStreamWriter(clientSocket.getOutputStream()));
        String line = "HTTP/1.0 403 Access Forbidden \n" +
                "User-Agent: ProxyServer/1.0\n" +
                "\r\n";
        bufferedWriter.write(line);
        bufferedWriter.flush();
    } catch (IOException e) {
        System.out.println("Error requesting a blocked site");
        e.printStackTrace();
    }
}
```

** I also created two functions to handle HTTP POST, as I needed them to run my Firefox browser on my proxy, but it wasn't in the instructions, so I didn't expand on it in the report.

3. Overview of the Caching Mechanism and its Benefits:

The ProxyServer includes a caching mechanism to store frequently accessed resources locally. This mechanism improves performance by reducing latency and bandwidth usage. Cached resources are served directly to clients without fetching them from the web server again, resulting in faster response times and decreased network congestion. Additionally, caching enhances scalability by offloading server load and improving overall system efficiency.

Here you can see my data collected for the www.wikipedia.org site before and after caching, as well as on a standard browser (Chrome). We can see that even though caching takes time, we manage to improve our performance once the file is cached.

Nom	État	Type	Initiateur	Taille	Durée
www.wikipedia.org	200	document	Autre	19.5 kB	54 ms

```
Waiting for clients on port 5555..
Enter a site to block, or type "blocked" to see blocked sites, "cached" to see cached sites, or "close" to close the server.
New connection: Socket[addr=/127.0.0.1,port=17790,localport=5555]
HTTP GET for: http://www.wikipedia.org/

Time taken without cache: 1443 ms
Data saved :0.618128Mbits
Bandwith :0.042836313236313234Mbits/s
New connection: Socket[addr=/127.0.0.1,port=17793,localport=5555]
Cached Copy found for: http://www.wikipedia.org/

Time taken from cache: 30 ms
Data saved :0.618128Mbits
Bandwith :20.604266666666668Mbits/s
```

4. Recommendations for Further Improvements or Enhancements:

This is a very simple implementation of a proxy server, there's so much we can do to improve it. We could also consider a cache for HTTPS for example.

Or by implementing expiration policies and cache validation strategies, we could ensure the freshness and validity of cached resources, thereby improving the overall user experience.

Additionally, optimizing the caching mechanism to meet high network requirements in terms of bandwidth will enable it to handle increased traffic more efficiently.

Lastly, the incorporation of advanced security features such as content filtering, malware detection, and intrusion prevention will bolster the system's defenses against threats and vulnerabilities, safeguarding both the network and the users' data.

Incorporating these recommendations will not only address the current limitations of the caching system but also position it for future scalability and resilience in the face of evolving technological landscapes and security challenges.