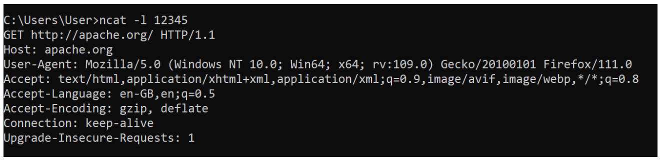
CS-421 Programming Assignment 1

Ege Çenberci, 21801618
Bilkent University
Spring 2023
CS-421
(Dated: March 31, 2023)

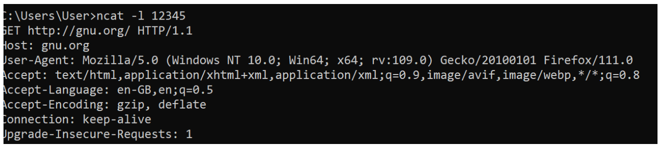
I. NETCAT RESULTS

The three HTTP website results using Netcat can be seen in Figs.(1,2,3).



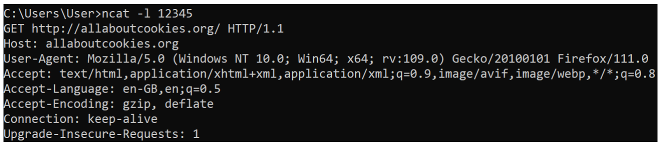
```
C:\Users\User>ncat -l 12345
GET http://apache.org/ HTTP/1.1
Host: apache.org
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:109.0) Gecko/20100101 Firefox/111.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
```

FIG. 1: Netcat result of apache.org



```
C:\Users\User>ncat -l 12345
GET http://gnu.org/ HTTP/1.1
Host: gnu.org
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:109.0) Gecko/20100101 Firefox/111.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
```

FIG. 2: Netcat result of gnu.org



```
C:\Users\User>ncat -l 12345
GET http://allaboutcookies.org/ HTTP/1.1
Host: allaboutcookies.org
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:109.0) Gecko/20100101 Firefox/111.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
```

FIG. 3: Netcat result of allaboutcookies.org

II. HTTP PROXY SERVER CODE STRUCTURE

```java
12 public class ProxyDownloader {
13     public static final int HTTP_PORT = 80;
14     public static final int CONNECT_PORT = 443;
15     public static void main(String[] args) throws IOException {
16         int port=0;
17         if (args.length < 1) {
18             System.err.println("Usage: java ProxyDownloader <port>");
19             System.exit(1);
20         }
21         else if(args.length > 1){
22             System.err.println("Too many arguments. Usage: java ProxyDownloader <port>");
23             System.exit(1);
24         }
25         else{
26             port = Integer.parseInt(args[0]);
27         }
28 
29         ServerSocket serverSocket = new ServerSocket(port);
```

Listing 1: Run Argument Handling and Initialization

First thing first, two port numbers are created as static final integers for easier use later, which can be seen in lines 13-14 of the Listing 1. On line 16, an integer is initialized to store the argument inputted by the user. Then, the conditions the program's arguments must meet are written through lines 17 to 27. If the argument given to the program is valid and the program didn't exit, a ServerSocket object is created to listen to the given port number in line 29.

```java
31         while (!serverSocket.isClosed()) {
32             Socket clientSocket = serverSocket.accept();
33 
34             InputStream clientIn = clientSocket.getInputStream();
35             OutputStream clientOut = clientSocket.getOutputStream();
36 
37             // Read request from client
38             byte[] buffer = new byte[4096]; // 4096 bytes is therequest length that can be read in a single time
39             int readBytesNum = clientIn.read(buffer);
40             String request = new String(buffer, 0, readBytesNum);
41 
42             // Extract host from the message
43             boolean downloadFile = false; // bool to get around file downloads in firefox's auto requests
44             String host;
45             String fullURL;
46             String fileName = "";
47             String requestType = request.substring(0, request.indexOf(" "));
48             String firstline = request.substring(0,request.indexOf("\n"));
```

Listing 2: Start of the While Loop

As seen from the beginning of Listing 2, line 31, the while loop the program runs on is created. In line 32, the Socket object *clientSocket* is initialized to listen to activity from the specified port. In lines 34-35, InputStream and OutputStream objects are initialized to listen to the client's requests and return responses to those requests. Through lines 36 to 40, the client request is read and stored in a String object and a byte array of size 4096. Through lines 42 to 48, some necessary declarations are made for further use, such as String object *host* and String object *fullURL*. Finally, in line 48, the first line of the HTTP message is extracted into the String object *firstLine*.

```
50          // Checks to avoid processing unneccessary automatic requests
51          boolean skipSteps = false;
52          boolean connectSkip = false;
53          if (firstline.contains("mozilla") | firstline.contains("firefox") | firstline.contains("r3.o.lencr.org") | firstline.contains("ocsp.digicert.com") | firstline.contains("ocsp.pki.goog") ) {
54              skipSteps = true;
55          }
56          if (!skipSteps) {
57              int firstSpace = request.indexOf(" ");
58              int secondSpace = request.indexOf(" ", firstSpace+1);
59              fullURL = request.substring(firstSpace, secondSpace);
60              int dslash = request.indexOf("//");
61              int slash = request.indexOf("/", dslash+2);
62              host = request.substring(dslash+2, slash);
63              fileName = fullURL.substring(fullURL.lastIndexOf("/")+1);
64              if (firstline.contains(".txt")) {
65                  downloadFile = true;
66              }
67          }
68          else{
69              // Additional check to detect CONNECT requests, these have to be forwarded to a different port
70              if (requestType.equals("CONNECT")) {
71                  host = request.substring(request.indexOf(" ")+1, request.indexOf(":"));
72                  connectSkip = true;
73              }
74              else{
75                  int dslash = request.indexOf("//");
76                  int slash = request.indexOf("/", dslash+2);
77                  host = request.substring(dslash+2, slash);
78              }
79          }
80          // The request is not an recognized automated one, print relevant output
81          if (!connectSkip && !skipSteps) {
82              System.out.println("Retrieved request from Firefox:\n");
83              System.out.println(request);
84          }
```

Listing 3: Unnecessary Request Checks

In Listing 3, the section from lines 50 to 55 checks if the *firstLine* contains the address of some known automatic Mozilla requests and sets the *skipSteps* flag accordingly. From lines 56 to 67, host and filename information is extracted using String manipulation. From lines 68 to 79, the CONNECT requests are handled with the initialization of flag *connectSkip* to recognise the CONNECT request further down the line. Finally, between lines 80 and 84, the relevant output is given to the terminal for unskipped requests.

```
87          // Connect to the requested remote server
88          Socket remoteSocket;
89          if (requestType.equals("CONNECT")) {
90              // forward connect requests to the correct port
91              remoteSocket = new Socket(host, CONNECT_PORT);
92              connectSkip = true;
93          }
94          else{
95              remoteSocket = new Socket(host, HTTP_PORT);
96          }
97          InputStream remoteIn = remoteSocket.getInputStream();
98          OutputStream remoteOut = remoteSocket.getOutputStream();
```

Listing 4: Establish Connection to Requested Server

Since the relevant hostname was extracted as shown in Listing 3, the next step is to establish the connection to the requested remote server. As seen in Listing 4, from lines 87 to 98, the Socket object for the requested remote server is declared and initialized with the name *remoteSocket*. In line 92, the *connectSkip* flag is initialized once again as a fail-safe, and in lines 97-98, InputStream and Output Stream objects are created to give requests to the server and receive responses, similar to the *clientSocket* from Listing 2.

```
100         // Forward the HTTP message to the remote server
101         remoteOut.write(buffer, 0, readBytesNum);
102         remoteOut.flush();
103
104         // Read response from the remote server
105         byte[] buffer2 = new byte[4096]; // 4096 bytes is the response length that can be read in a single time
106         int readBytesNum2 = remoteIn.read(buffer2);
107         String response;
108         // Check for edge cases and change variables accordingly
109         if (readBytesNum2 == -1) {
110             response = "";
111             skipSteps = true;
112         }
113         else{
114             response = new String(buffer2, 0, readBytesNum2);
115         }
```

Listing 5: Pass on Client's Request and Get Server's Response

The code section in Listing 5 starts with the client's request message being sent to the remote server in lines 100 to 102. Then in lines 104 to 115, the remote server's response is read, similar to how the client's request was read in Listing 2. The differing part of reading the server's response can be seen in lines 108 to 115, where no response message sent by the server edge case is checked using the variable *readBytesNum*2.

```java
117         // Get ready to process server response
118         int contentLengthInt = 0;
119         String responseToken = "";
120         int responseCodeInt = 0;
121         // Recognized automated request check
122         if (!skipSteps) {
123             try {
124                 int contentLengthLineIndex = response.indexOf("Content-Length");
125                 String contentLengthLine = response.substring(contentLengthLineIndex);
126                 String contentLength = contentLengthLine.substring(contentLengthLine.indexOf(" ")+1, contentLengthLine.indexOf("\n"));
127                 contentLengthInt = Integer.parseInt(contentLength.strip());
128             } catch (Exception e) {
129                 System.err.println("Content Length field doesn't exist");
130             }
131             int firstSpaceServer = response.indexOf(" ");
132             int firstlineEndServer = response.indexOf("\n");
133             responseToken = response.substring(firstSpaceServer+1, firstlineEndServer);
134             String responseCode = responseToken.substring(0, responseToken.indexOf(" "));
135             //String responseWord = responseToken.substring(responseToken.indexOf(" ")+1);
136             responseCodeInt = Integer.parseInt(responseCode);
137         }
```

Listing 6: Process Server's Response

The code snippet in Listing 6 starts with useful variable initializations through lines 117 to 120. Afterward, through line 122 to line 137, the information from the server's response message is extracted using String manipulation. The three important variables acquired from the response are the *contentLengthInt* and *responseCodeInt* integers and the *responseToken* string.

```java
139         // Declare output file object and create it if required conditions are met
140         FileOutputStream fileOut = null;
141         if (downloadFile && responseCodeInt == 200) {
142             fileOut = new FileOutputStream(fileName);
143         }
144         String responseString = new String(buffer2, StandardCharsets.UTF_8); // response String is only to get index to seperate header from the message
145         int fileBoundaryIndex = responseString.indexOf("\r\n\r\n") + 4; // start index of file body
```

Listing 7: Declare Output File and Do Other Necessary Preperations

In Listing 7, the output file is declared and created if the output file writing conditions are met, which can be seen in lines 139 to 143. Between lines 144 to 145, the server response stored in the byte array *buffer* is copied to a String object to find the starting index of the HTTP file body.

```java
147         if (!skipSteps) {
148             // If the response message is longer than 4090 bytes, process it multiple times
149             if (contentLengthInt > 4090) {
150                 // Forward remote server's response's first section to client
151                 clientOut.write(buffer2, 0, readBytesNum2);
152                 clientOut.flush();
153                 if (downloadFile && responseCodeInt == 200) {
154                     // Write the contents of the response body to output file
155                     fileOut.write(buffer2, fileBoundaryIndex, readBytesNum2 - fileBoundaryIndex);
156                 }
157                 // while condition to keep going until response is exhausted
158                 while (readBytesNum2 != -1) {
159                     readBytesNum2 = remoteIn.read(buffer2);
160                     if (readBytesNum2 != -1) {
161                         response = new String(buffer2, 0, readBytesNum2);
162                         // Forward remote server's response to client
163                         clientOut.write(buffer2, 0, readBytesNum2);
164                         clientOut.flush();
165                         if (downloadFile && responseCodeInt == 200) {
166                             // Write remaining contents of the response body to output file
167                             fileOut.write(buffer2, 0, readBytesNum2);
168                         }
169                     }
170                 }
171             }
172             else{
173                 // Forward remote server's response to client
174                 clientOut.write(buffer2, 0, readBytesNum2);
175                 clientOut.flush();
176                 if (downloadFile && responseCodeInt == 200) {
```

```
177                         // Write the contents of the response body to output file
178                         fileOut.write(buffer2, fileBoundaryIndex, readBytesNum2 - fileBoundaryIndex);
179                     }
180                 }
181             }
```

Listing 8: Processing Necessary Request's Server Response

Inside Listing 8, line 147 checks if the response from the server belongs to a necessary request, one that requires further operations, or to an unnecessary one, which doesn't require any extra operation. Through lines 148 to 171, the large HTTP responses are processed. The HTTP responses with content-length bigger than 4090 bytes are considered large and are read in multiple iterations. The if statement in line 149 checks for the content length of the response message and, through lines 150 to 156, completes the first iteration of the process. Lines 151 and 152 deliver the server's response to the client via the previously created OutputStream object. Between lines 153 to 156, the first section of the response body is written to the output file if the conditions are met. The same process is repeated between lines 157 and 170 inside a while loop. The while loop condition on line 158 will exit when the response body is exhausted. The section of code from line 172 to line 180 contains the same response-sending and file-writing logic, this time for small sized HTTP messages. Finally, the necessary request's response condition statement is closed in line 181.

```
182             else{
183                 // Process the automated requests that are not CONNECT requests, (connect requests crashed the code)
184                 if (!connectSkip) {
185                     // Forward remote server's response to client
186                     clientOut.write(buffer2, 0, readBytesNum2);
187                     clientOut.flush();
188                 }
189             }
```

Listing 9: Processing Unnecessary Request's Server Response

The code displayed in Listing 9 handles the unnecessary requests' responses. If the response from the server isn't sent to a CONNECT request, the server's response is transmitted to the client; otherwise, it is dropped to avoid crashing.

```
191             // Output message logic from variables acquired earlier
192             if (responseCodeInt != 0 && !skipSteps && !connectSkip) {
193                 if (responseCodeInt == 200) {
194                     System.out.printf("Downloading file ’%s’...\n", fileName);
195                     System.out.printf("Retrieved: %s\n", responseToken);
196                     System.out.println("Saving file...");
197                 }
198                 else if (responseCodeInt == 304) {
199                     System.out.printf("Retrieved: %s\n", responseToken);
200                     System.out.printf("No changes made to file ’%s’.\n", fileName);
201                 }
202                 else if (responseCodeInt == 404) {
203                     System.out.printf("Retrieved: %s\n", responseToken);
204                     System.out.println("The requested URL was not found on this server.");
205                 }
206                 else{
207                     System.out.printf("Retrieved: %s\n", responseToken);
208                 }
209             }
```

Listing 10: Output Conditions

The code section viewable in Listing 10 demonstrates the logical structure leading up to different output messages to the console. The logical structure utilizes the *responseToken*, *fileName* strings, and *responseCodeInt* integer to produce relevant output to the user.

```
210             // Transfer is complete, close the connection to the remote host
211             remoteSocket.close();
212             // If file was created, close it
213             if (fileOut != null) {
214                 fileOut.close();
215             }
216             // Print appropriate automated request outputs and user prompt
217             if (connectSkip | skipSteps) {
218                 System.out.println("Processed an automatic Mozilla request.");
219             }
220             System.out.println("\nAWAITING NEW ACTION\n");
221         }
```

Listing 11: Closing the Connection to Remote Server - End of While Loop

Listing 11 shows the cleanup done before the end of the while loop. The Socket object created to connect to the requested remote server is closed in line 211. In lines 213 to 215, the output file is closed if it was written into. Through lines 217 to 221, some final relevant output is given to the user, signalling the program is ready to receive another package.

```
222         // the code will never reach this point at its current form, but this line gets rid of warnings
223         serverSocket.close();
224     }
225 }
```

Listing 12: Final Section of the Code

Listing 12 displays the end of the entire code body.