**Socket Programming Lab #4: HTTP Web Server Proxy**

This lab report will describe all added code to the skeleton program *proxy.py*. For purposes of readability, added code will be preceded by comments starting with 3 #'s. This report will be organized into 4 sections, with each covering a function within *proxy.py*: *parse_http_headers*, *forward_and_cache_response*, *forward_request*, and *proxyServer*.

***parse_http_headers*:**

This function is responsible for parsing the HTTP header. The skeleton code returns a 2-tuple consisting of the headline of the HTTP message and a list of the HTTP headers and its corresponding values. The code has been modified to return a 3-tuple, including a content field.

```
39  # Read an HTTP message from a socket file object and parse it
40  # sockf: Socket file object to read from
41  ### Returns: (headline: str, [(header: str, header_value: str), content: str])
42  def parse_http_headers(sockf):
43      # Read the first line from the HTTP message
44      # This will either be the Request Line (request) or the Status Line (response)
45      headline = interruptible_readline(sockf).decode().strip()
46
47      # Set up list for headers
48      headers = []
49      while True:
50          # Read a line at a time
51          header = interruptible_readline(sockf).decode()
52
53          # If it's the empty line '\r\n', it's the end of the header section
54          if len(header.rstrip('\r\n')) == 0:
55              break
56
57          # Partition header by colon
58          headerPartitions = header.partition(':')
59
60          # Skip if there's no colon
61          if headerPartitions[1] == '':
62              continue
63
64          headers.append((headerPartitions[0].strip(), headerPartitions[2].strip()))
65
66      ### Read contents if Content-Length header exists
67      content = ""
68
69      if any(['Content-Length' in header for header in headers]):
70          contentLength = [val for header, val in headers if header == 'Content-Length']
71          content += interruptible_read(sockf, int(contentLength[0])).decode()
72
73      ### Return a 3-tuple: content along with headline and headers
74      return(headline, headers, content)
```

On line 67, we initialize a *content* buffer variable to an empty string. This buffer variable will hold any data from the contents of a HTTP message. Unlike with the header section, there is no nothing to denote the end of the content section. The Content-Length header tells us how many bytes of data are in the contents of the HTTP message. Lines 69-70 check if there exists a Content-Length header from the list of headers previously parsed and store its value into a

*contentLength* variable. Line 71 passes the value of *contentLength* into the *interruptible_read* function call and concatenates the encoded byte-form data read from the HTTP message into the *content* buffer variable. Line 74 returns the 3-tuple consisting of the headline, a list of headers and the content. In the event that there is a HTTP message that does not have a content section (i.e., a HTTP GET request, or a HTTP response to a HTTP POST request), then the *content* variable will remain an empty string.

***forward_and_cache_response*:**

This function will receive the HTTP response from the server, cache the response, then forward the response to the client. On line 81, the parameters have been modified to also include the HTTP request method, as we will only want to cache responses to HTTP GET requests. Line 86 has been modified to also check if the HTTP request method is GET, if so, then a cache file will be created.

```
76  # Forward a server response to the client and save to cache
77  # sockf: Socket file object connected to server
78  # fileCachePath: Path to cache file
79  # clisockf: Socket file object connected to client
80  ### requestMethod: Request method of original client request
81  def forward_and_cache_response(sockf, fileCachePath, clisockf, requestMethod):
82      cachef = None
83
84      # Create the intermediate directories to the cache file
85      ### Only create a cache file if client request was HTTP GET
86      if fileCachePath is not None and requestMethod == "GET":
87          os.makedirs(os.path.dirname(fileCachePath), exist_ok=True)
88          # Open/create cache file
89          cachef = open(fileCachePath, 'w+b')
90
```

Line 94 has been modified to also store HTTP response content data into a *content* variable. If the original client request is an HTTP POST message, then we can expect *content* to be an empty string.

```
91      try:
92          # Read response from server
93          ### Read content from server response as well
94          statusLine, headers, content = parse_http_headers(sockf)
95          # Filter out the Connection header from the server
96          headers = [h for h in headers if h[0] != 'Connection']
97          # Replace with our own Connection header
98          # We will close all connections after sending the response.
99          # This is an inefficient,  single-threaded proxy!
100         headers.append(('Connection', 'close'))
101
```

On line 104, we create a *data* buffer variable that will store the contents of the HTTP response message. We initialize *data* to the status line concatenated with '\r\n'. In a HTTP message, '\r\n'

marks the end of a line. Lines 107-108 iterate through the list of headers and concatenate the header and its corresponding value to *data*. Line 111 concatenates '\r\n\' and the contents of the HTTP message. A line with just '\r\n' marks the end of the header section. Line 114 checks if the client HTTP request method was GET, if so, then it encodes and writes the contents of the *data* buffer variable into the cache file. Line 117 encodes *data* into byte form and forwards it to the client socket.

```
102         # Fill in start.
103         ### Initialize data buffer with response status line, each line ends with '\r\n'
104         data = statusLine + '\r\n'
105 
106         ### Add rest of response headers and their corresponding values to the data buffer
107         for header in headers:
108             data += header[0] + ": " + header[1] + '\r\n'
109 
110         ### Concatenate '\r\n' to data buffer to denote end of header section and content from the server response
111         data += '\r\n' + content
112 
113         ### Only write to cache file if original client request is HTTP GET
114         if requestMethod == 'GET': cachef.write(data.encode())
115 
116         ### Encode and forward response to client socket file object
117         clisockf.write(data.encode())
118         # Fill in end.
119 
120     except Exception as e:
121         print(e)
122     finally:
123         if cachef is not None:
124             cachef.close()
125 
```

***forward_request*:**

This function is responsible for forwarding the client request to the server. Line 133 has been modified to include the client request's HTTP message content. If there is no content section, such as is the case with a HTTP GET request, then an empty string can be passed. Line 141 initializes a *data* buffer variable that will store the contents of the HTTP response message with the client HTTP request headline with '\r\n'. In a HTTP message, '\r\n' marks the end of a line. Line 144-145 iterate through the list of headers and concatenate the header and its corresponding value to *data*. Line 148 concatenates '\r\n\' and the contents of the client HTTP request. A line with just '\r\n' marks the end of the header section. Line 151 encodes the HTTP request message into byte form and forwards it to the server socket.

```
126  # Forward a client request to a server
127  # sockf: Socket file object connected to server
128  # requestUri: The request URI to request from the server
129  # hostn: The Host header value to include in the forwarded request
130  # origRequestLine: The Request Line from the original client request
131  # origHeaders: The HTTP headers from the original client request
132  ### origContent: The Contents from the original client request
133  def forward_request(sockf, requestUri, hostn, origRequestLine, origHeaders, origContent):
134      # Filter out the original Host header and replace it with our own
135      headers = [h for h in origHeaders if h[0] != 'Host']
136      headers.append(('Host', hostn))
137      # Send request to the server
138
139      # Fill in start.
140      ### Initialize data buffer with headline, each line ends with '\r\n'
141      data = origRequestLine + '\r\n'
142
143      ### Add rest of headers and their corresponding values to the data buffer
144      for header in headers:
145          data += header[0] + ": " + header[1] + '\r\n'
146
147      ### Concatenate '\r\n' to data buffer to denote end of header section and content from the original client request
148      data += '\r\n' + origContent
149
150      ### Encode and write data to socket file object connected to server
151      sockf.write(data.encode())
152      # Fill in end.
153
```

***proxyServer*:**

This function serves as the proxy server. Line 162 binds the proxy server socket to the port number specified in the function parameters. Line 163 has the proxy server socket listen for any incoming TCP packets.

```
154  def proxyServer(port):
155      if os.path.isdir(cacheDir):
156          shutil.rmtree(cacheDir)
157      # Create a server socket, bind it to a port and start listening
158      tcpSerSock = socket(AF_INET, SOCK_STREAM)
159
160      # Fill in start.
161      ### Bind socket to specified port and listen for incoming TCP packets
162      tcpSerSock.bind(('', port))
163      tcpSerSock.listen()
164      # Fill in end.
165
166      tcpCliSock = None
```

Line 178 has been modified to also store the client HTTP request message content returned from the 3-tuple. This will be used when calling *forward_request* later on. Line 187 extracts the client HTTP request method from the HTTP request headline. This will be used when calling *forward_and_cache_response* later on.

```
167      try:
168          while 1:
169              # Start receiving data from the client
170              print('Ready to serve...')
171              tcpCliSock, addr = interruptible_accept(tcpSerSock)
172
173              print('Received a connection from:', addr)
174              cliSock_f = tcpCliSock.makefile('rwb', 0)
175
176              # Read and parse request from client
177              ### 3-tuple is returned: client request headline, client request headers, and client request content
178              requestLine, requestHeaders, requestContent = parse_http_headers(cliSock_f)
179
180              if len(requestLine) == 0:
181                  continue
182
183              # Extract the request URI from the given message
184              requestUri = requestLine.split()[1]
185
186              ### Extract request method from request line
187              requestMethod = requestLine.split()[0]
188
```

Line 202 defines a cache file path as cacheDir/{name of file queried for in client HTTP request}. Line 204 stores a boolean variable, checking whether or not the cache file path exists. Line 213-214 open and read from the cache file if the cache file path exists. Line 217 forwards the contents of the cache file to the client socket.

```
189             # if a scheme is included, split off the scheme, otherwise split off a leading slash
190             uri_parts = requestUri.partition('http://')
191             if uri_parts[1] == '':
192                 filename = requestUri.partition('/')[2]
193             else:
194                 filename = uri_parts[2]
195
196             print(f'filename: {filename}')
197
198             if len(filename) > 0:
199                 # Compute the path to the cache file from the request URI
200                 # Change for Part Three
201                 ### Define cached file path from requested file in original client request
202                 fileCachePath = 'cacheDir/' + filename.partition('/')[2]
203                 ### Check if cached file path exists
204                 cached = os.path.exists(fileCachePath)
205
206                 print(f'fileCachePath: {fileCachePath}')
207
208                 # Check whether the file exists in the cache
209                 if fileCachePath is not None and cached:
210                     # Read response from cache and transmit to client
211                     # Fill in start.
212                     ### Open and read contents from cached file into buffer
213                     cacheFile = open(fileCachePath, 'rb')
214                     cacheContent = cacheFile.read()
215
216                     ### Write contents from buffer into client socket file object
217                     cliSock_f.write(cacheContent)
218                     # Fill in end.
219                     print('Read from cache')
```

Line 224 creates a socket on the proxy server.

```
220                 else:
221                     # Create a socket on the ProxyServer
222                     # Fill in start.
223                     ### Create socket
224                     c = socket(AF_INET, SOCK_STREAM)
225                     # Fill in end.
226                     hostn = filename.partition('/')[0]
227                     print(f'hostn: {hostn}')
228
```

Line 223 extracts the server address and the server port number. Line 236 connects the socket created on the proxy server to the previously extracted server address and server port number. Line 241 calls the *forward_request* method, passing the contents of the client HTTP request as one of the parameters. Line 246 calls the *forward_and_cache_response* method, passing the client HTTP request method as one of the parameters.

```
229                     try:
230                         # Connect to the socket
231                         # Fill in start.
232                         ### Separate server address and server port number from hostn
233                         serverAddress = hostn.partition(':')
234 
235                         ### Connect to server socket
236                         c.connect((serverAddress[0], int(serverAddress[2])))
237                         # Fill in end.
238 
239                         # Create a temporary file on this socket and ask port 80 for the file requested by the client
240                         fileobj = c.makefile('rwb', 0)
241                         ### Pass client request content into parameters as well
242                         forward_request(fileobj, f'/{filename.partition("/")[2]}', hostn, requestLine, requestHeaders, requestContent)
243 
244                         # Read the response from the server, cache, and forward it to client
245                         ### Pass client request method into parameters as well
246                         forward_and_cache_response(fileobj, fileCachePath, cliSock_f, requestMethod)
247                     except Exception as e:
248                         print(e)
249                     finally:
250                         c.close()
251             tcpCliSock.close()
252     except KeyboardInterrupt:
253         pass
```

Line 258-259 close the connections to the server socket and the client socket.

```
254 
255     # Close the server socket and client socket
256     # Fill in start.
257     ### Close server socket and client socket
258     tcpSerSock.close()
259     tcpCliSock.close()
260     # Fill in end.
261     sys.exit()
```