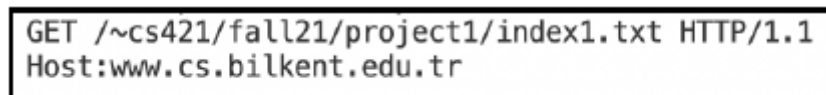


## CS 421 – Computer Networks

### Programming Assignment 2

Firstly, we have parsed the command line arguments as index file and connection count. Then, the splitLink function was called which basically splits the index file into server name and directory. By using that information, we have created the GET request message calling createGETrequestMessage function, its structure can be seen in Figure 1.



```
GET /~cs421/fall21/project1/index1.txt HTTP/1.1
Host:www.cs.bilkent.edu.tr
```

**Figure 1:** *HTTP GET message structure*

After creating the GET request message, we have called prepareSocket function. Inside this function, we firstly created a socket and connect to that socket by using the server name and server port which is 80 by default for HTTP. After connecting, we sent the encoded GET request message. Finally, we have returned that socket at the end of prepareSocket function. After sending the GET request message for the index file, the clientSocket received a GET response message. We received 4096 bytes from the client socket in each iteration and decode those bytes to store them inside a string. Finally, we have closed the socket.

In order to send HEAD request, we have called prepareSocket function again. We did not use the previously created socket since HTTP uses TCP and TCP uses a unique socket per connection. After creating the socket, we have sent the HEAD request message for the index file and again the HEAD response message is received as explained in the previous paragraph.

If the HEAD response message returns a message other than “200 OK”, we stop the execution of the program. If the HEAD response message includes “200 OK”, we called getBody function to extract file links from the GET response and return those links inside an array.

In order to download the content of the files, we called download\_files function and provided the links and connection count as parameters. The connection count represents the number of threads working concurrently to download the files. For each link, we firstly get the server name and directory by using splitLink function. We have again created GET and HEAD request messages and created two sockets to send each request separately. We received 4096 bytes in each iteration as explained before and stored them inside a string after decoding them.

If the HEAD response message does not return “200 OK”, we printed the message “The file ... is not found” and execution continued with the next file. If the response includes “200 OK”, we extract the length of the file from HEAD response by using getBodySizeChar function. The length of the file is stored inside the HEAD response as follows: Content-Length:<file\_length> and we accessed to that information. If the length is zero, we printed the message “The file ... is not found” message and execution continued with the next file again.

Burak Yetiştiren 21802608

Işık Özsoy 21703160

We were given the parallel connection count at the beginning, so we have used threads to download the files in parallel. We assign each thread an interval. For example, if we have a file of 9 bytes and two threads, the first thread will download bytes from 0 to 4 and the second thread will download bytes from 5 to 8. If the file length is divisible by the thread count, we calculated the start and end bytes to be downloaded for each thread as follows:

$$\text{Start\_byte} = i * \text{length} // \text{number\_of\_threads}$$
$$\text{End\_byte} = (i + 1) * \text{length} // \text{number\_of\_threads} - 1 \quad (\text{where } i \text{ is the thread index})$$

If the file length is not divisible by the thread count, we have followed the calculations below:

$$\text{Start\_byte} = i * (\text{length} // \text{number\_of\_threads} + 1)$$
$$\text{End\_byte} = (i + 1) * \text{length} // \text{number\_of\_threads} - 1 \quad (\text{where } i \text{ is the thread index})$$

If the end\_byte is larger than the length of the file, we set end\_byte to the index of last character in the file. Before, assigning a thread, we check if the start\_index is not greater than the file length.

After handling those corner cases, we give threads a function named downloader\_thread to run and parameters which are the link and start and end bytes. Moreover, in order to be able to access the return value of the function, which is downloaded bytes in our cases, we have used a dictionary data type which is a mutable object. For example, if there are three threads, the content of dictionary will be as follows:

```
{“0”: “<downloaded_bytes_by_thread0>”, “1”: “<downloaded_bytes_by_thread1>”, “2”:  
“<downloaded_bytes_by_thread2>”}
```

In order to download the file content, each thread creates two sockets to send GET and HEAD request messages. After sending the requests, they receive and decode bytes and store the GET and HEAD response inside strings. By using GET and HEAD response messages, the content of the files is extracted from GET response message and saved into dictionary. Then we merged the content downloaded by each thread and save them into a txt file.