



T.C.

MARMARA UNIVERSITY

FACULTY of ENGINEERING

COMPUTER ENGINEERING DEPARTMENT

CSE 4074 – Computer Network Term Project Report

Ömer Faruk Çakı - 150117821

Cem Güleç - 150117828

Lecturer: Assistant Prof. Ömer Korçak

16 January 2021

We described our program in detail below. We also completed all bonus parts available (3+15+5 = 23pts in total).

1. HTTP Server

We used Nodejs to write our HTTP server implementation and the proxy server. In our code we used **net** module which is Node.js's native TCP protocol implementation.

The first step is creating a TCP server and start listening on localhost for incoming connections. We used **net.createServer()** method to create a basic TCP server. Since Node.js is an event-driven programming language, most of native functions use events. So after creating our TCP server we attached our event listener for incoming connections with **on('connection')**. This event will fire when a new connection is established. Apart from that we started listening on given port at the very end of the code with **server.listen(PORT)**

Inside our connection event, we have a new listener for **'data'** events, this will fire after connection is established and the client sends any data, such as an HTTP request object. Our HTTP server implementation starts from here, first of all we should parse the incoming HTTP request.

In the following line of code we split incoming HTTP messages (if a valid HTTP format) into parts, which are method, url and protocol. Since our server only handles GET requests, we only handled first line and we do not have any checks for the body of the request.

```
const [method, url, protocol] = request.split(/\r\n|\n|\r/)[0].split(' ');
```

Then we checked the HTTP method of the incoming request and we sent response of 501 Not Implemented for valid HTTP methods other than GET. Also, 400 Bad Request for any other request which are not valid HTTP methods while printing useful logs to console during all operations.

For GET requests we had another parsing operation for the url to get length of the document requested. For the sizes between 100-20000 we generated responses of requested lengths by repeating 'a' character. After that we built our valid HTTP response by setting the body and headers accordingly. Then we write that response to socket as a response to the client with **socket.write(responseObject)**; Here, **socket** represents the current connection with the client. If the length of the requested document type was not an integer or it is less than 100 or greater than 20000 we respond with 400 Bad Request. After we are done with sending a response to the client we close the connection with **socket.end()**.

For the multithreading, we used a **cluster** native module of Node.js. The cluster module allows easy creation of child processes that all share server ports. We thought that it's a wise choice to create child processes according to our machine's CPU cores. So the program checks the CPU cores of the systems and creates new child processes according to that amount. For example if the PC that runs the code has 6 cores CPU it will create 6 processes by forking the main process. Created child process will run the same code we described above, all on the same port. During a high server load other processes will handle connections simultaneously.

As it can be seen on Figure 1, we printed information of all stages including request object, response object which is sent to client along with all connection states.

```
Windows PowerShell
A connection is established

[12744] Data received from client:
=====
GET /100 HTTP/1.1

{ method: 'GET', url: '/100', protocol: 'HTTP/1.1' }

[12744] Data sent to the client:
=====
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Content-Length: 100

<html><head></head><body>aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa</body></html>
Closing connection with the client
A connection is established

[12744] Data received from client:
=====
GET /favicon.ico HTTP/1.1

{ method: 'GET', url: '/favicon.ico', protocol: 'HTTP/1.1' }
Not a valid request

[12744] Data sent to the client:
=====
HTTP/1.1 400 Bad Request

Closing connection with the client
```

Figure 1: Informative logs printed by web server

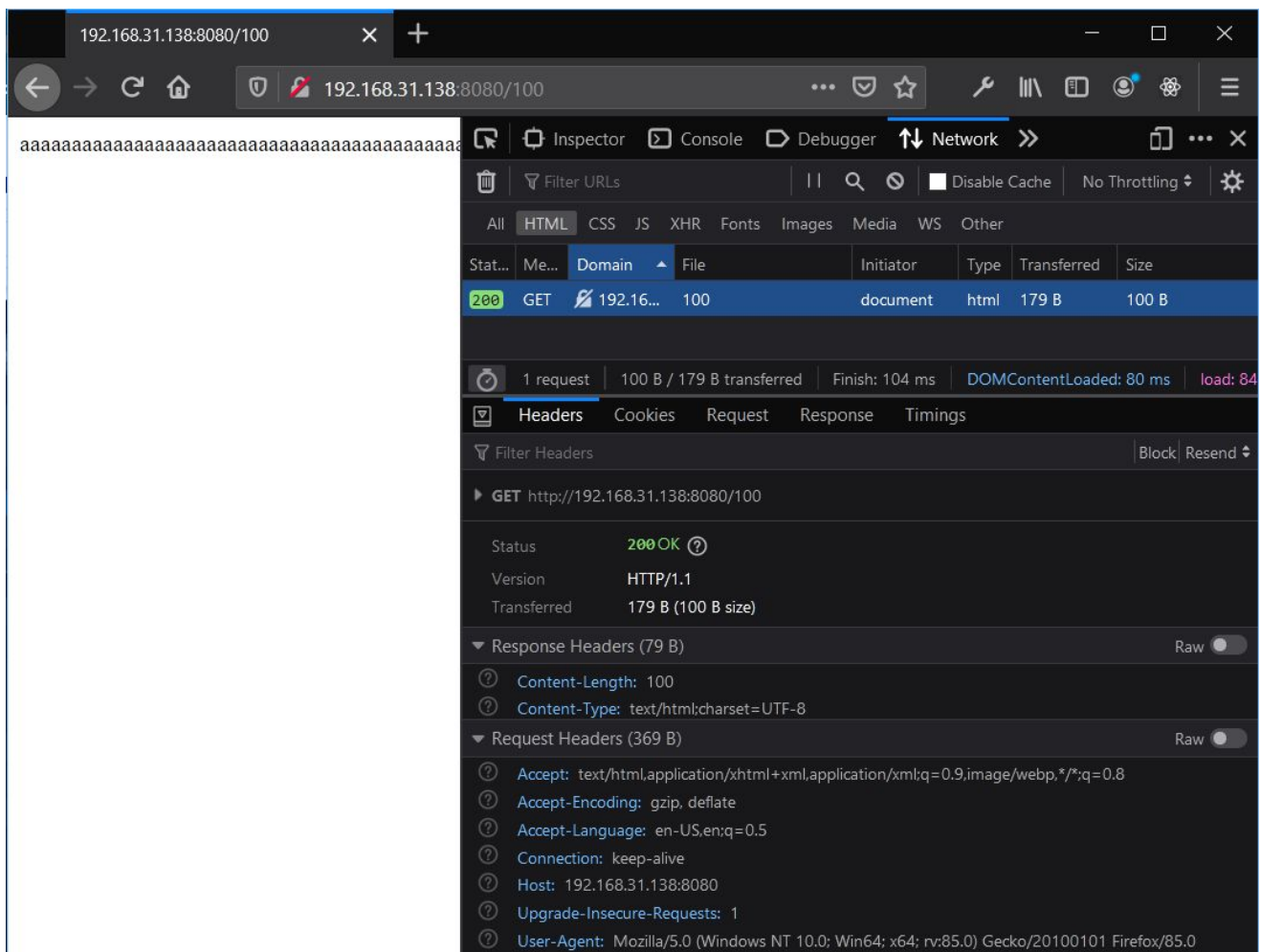
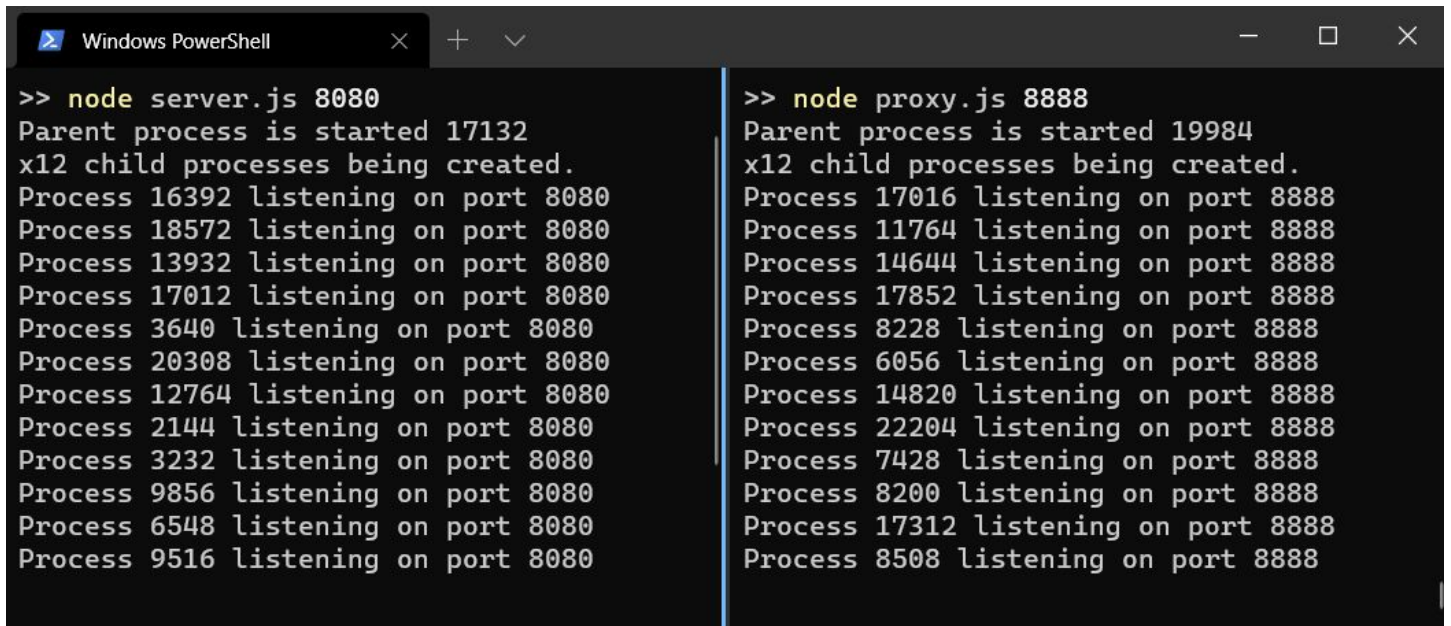


Figure 2: Same process in Figure 1 visualized in web browser

2. Proxy Server

Proxy server implementation is very similar to HTTP server implementation described above with few changes. It is again a multithreaded server using the same approach used on HTTP server.



```
>> node server.js 8080
Parent process is started 17132
x12 child processes being created.
Process 16392 listening on port 8080
Process 18572 listening on port 8080
Process 13932 listening on port 8080
Process 17012 listening on port 8080
Process 3640 listening on port 8080
Process 20308 listening on port 8080
Process 12764 listening on port 8080
Process 2144 listening on port 8080
Process 3232 listening on port 8080
Process 9856 listening on port 8080
Process 6548 listening on port 8080
Process 9516 listening on port 8080

>> node proxy.js 8888
Parent process is started 19984
x12 child processes being created.
Process 17016 listening on port 8888
Process 11764 listening on port 8888
Process 14644 listening on port 8888
Process 17852 listening on port 8888
Process 8228 listening on port 8888
Process 6056 listening on port 8888
Process 14820 listening on port 8888
Process 22204 listening on port 8888
Process 7428 listening on port 8888
Process 8200 listening on port 8888
Process 17312 listening on port 8888
Process 8508 listening on port 8888
```

Figure 3: Multithreaded web server(left) and proxy server(right) are started

We again wait for incoming TCP connections but handle only HTTP GET requests. So our proxy server parses the incoming request payload and checks whether it is a HTTP request and also the method is GET. Other requests which are not a part of this project is dropped. After that we parse the absolute url from request, we send a new request to destination server with relative url(including parameters), then we are returning the response obtained from server to the client.

```
A connection is established

[8508] Data received from client:
=====
GET http://192.168.31.138:8080/100 HTTP/1.1
Host: 192.168.31.138:8080
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:85.0) Gecko/20100101 Firefox/85.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1

{ method: 'GET', url: 'http://192.168.31.138:8080/100' }
Socket closed.
Connected to destination server
HTTP message is sent to the destination server.Data received from server:
=====
GET http://192.168.31.138:8080/100 HTTP/1.1
Host: 192.168.31.138:8080
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:85.0) Gecko/20100101 Firefox/85.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1

client destroyed
Connection with the client is closed
```

Figure 4: Informative logs printed by proxy server

In case Web Server is currently not running, the proxy server returns a Not Found error message with status code of 404.

Our proxy server is also directs to any web server. **(+3 Bonus Part)**. We run our proxy server on a linux remote machine(AWS EC2 instance) and we set the proxy server in Firefox like that:

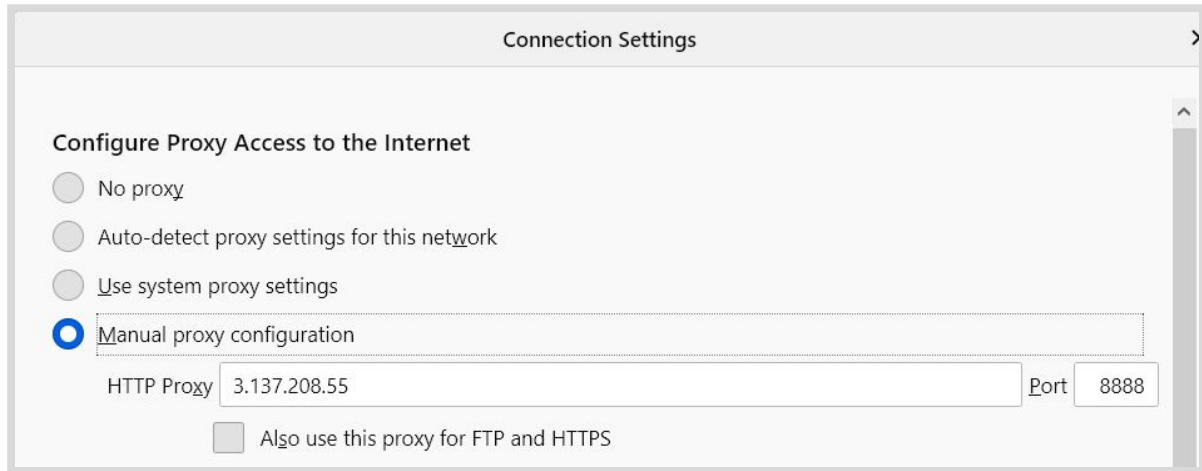
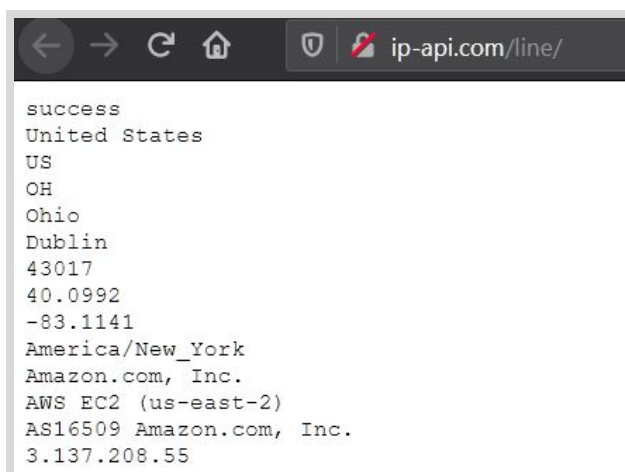


Figure 5: Example connection settings applied

3.137.208.55 is the IP address of our remote server and proxy is running on port 8888. In order to better show that it is working we found the website <http://ip-api.com/line/> serving over HTTP which shows information of the IP address. When we visit that url(with our PC) while proxy is enabled we receive this response:



This example shows that our proxy server works perfectly with any HTTP server to proxy HTTP GET requests.

For the requests to our HTTP server we check the absolute url to determine if it's a request to our local server or not. If it's a request to our server we parse the size from the url and check if it's greater than 9999. If so we respond with 414 Request-URI Too Long message.

If the client requests a relative URL (such as GET /500 HTTP/1.0) from the proxy, our proxy server will redirect it to our web server which is running on port 8080 by default.

Figure 6: Example output from the bonus part

We also implemented cache **(+15 Bonus Part)** including Conditional GET **(+5 Bonus Part)**.

Following snippet shows the caching process. **retrieve** function checks if the request is previously cached. **retrieve()** function is available in **cache.js** file. That function checks the **cache/** folder for the given number (for example 100 means an html content with 100 bytes, which is saved during a request to /100 before). If the requested object is cached before it returns the content, null otherwise. We also check the path to determine whether it's even length or odd length. We assumed even length requests were not modified as proposed in the project, so if the request is even length we don't retrieve from cache and instead perform a request to the destination server each time.


```

const cachedObject = retrieve(path);
if (cachedObject !== false && !isEven(path)) {
  socket.write(
    'HTTP/1.1 304 Not Modified\n' +
    `Date: ${new Date().toUTCString()}\n` +
    '\n' +
    cachedObject
  );
  socket.end();
  return;
}

```

When our proxy server receives a new valid request, we create a TCP client with `const client = new net.Socket()` to be used to send a request to the destination server to obtain a response. Using this client we connect to the destination server on the provided port and send a request to the relative url, and we directly return the received response from server to client. After getting the response we also save it to be used later by cache if the same request is occurred, we save it as a file in cache/ folder using `save({ name: path, stream: data })` function(it's available on cache.js).

Here are the objects that are saved in our cache. In the screenshot above, file names represent what we requested from server, relative URL or path in other words. We save response objects with corresponding names with their size and we easily see that sizes match with the response body when saved to disk.

```
λ ls -l .\cache
```

				size (bytes)					File name
total 23									
-rw-r--r--	1	omerf	197609	100	Jan	16	15:18	100	
-rw-r--r--	1	omerf	197609	10000	Dec	26	23:08	10000	
-rw-r--r--	1	omerf	197609	111	Dec	26	23:09	111	
-rw-r--r--	1	omerf	197609	499	Jan	16	15:20	499	
-rw-r--r--	1	omerf	197609	5000	Jan	6	01:43	5000	

Figure 7: Example objects saved in process of caching

3. Using ApacheBench Program

Concurrency level = 1

Time taken for tests (seconds)	29.203 seconds																									
Total transferred (bytes) and HTML transferred (bytes)	52431510 bytes 52428800 bytes																									
Time per request (ms)	2393.220 [ms]																									
Requests per second (#/sec)	0.42 [#/sec]																									
Transfer rate (Kbytes/sec)	1753.33 [Kbytes/sec]																									
Connection times	<div>Connection Times (ms)<table><tr><th></th><th>min</th><th>mean[+/-sd]</th><th>median</th><th>max</th></tr><tr><td>Connect:</td><td>55</td><td>58 2.0</td><td>57</td><td>61</td></tr><tr><td>Processing:</td><td>1311</td><td>2863 1376.3</td><td>2950</td><td>5045</td></tr><tr><td>Waiting:</td><td>56</td><td>57 1.9</td><td>57</td><td>62</td></tr><tr><td>Total:</td><td>1368</td><td>2920 1376.9</td><td>3005</td><td>5107</td></tr></table></div>		min	mean[+/-sd]	median	max	Connect:	55	58 2.0	57	61	Processing:	1311	2863 1376.3	2950	5045	Waiting:	56	57 1.9	57	62	Total:	1368	2920 1376.9	3005	5107
	min	mean[+/-sd]	median	max																						
Connect:	55	58 2.0	57	61																						
Processing:	1311	2863 1376.3	2950	5045																						
Waiting:	56	57 1.9	57	62																						
Total:	1368	2920 1376.9	3005	5107																						

Concurrency level = 5

Time taken for tests (seconds)	16.694 seconds																									
Total transferred (bytes) and HTML transferred (bytes)	52431510 bytes 52428800 bytes																									
Time per request (ms)	8346.823 [ms]																									
Requests per second (#/sec)	0.60 [#/sec]																									
Transfer rate (Kbytes/sec)	3067.19 [Kbytes/sec]																									
Connection times	<div>Connection Times (ms)<table><tr><th></th><th>min</th><th>mean[+/-sd]</th><th>median</th><th>max</th></tr><tr><td>Connect:</td><td>55</td><td>60 2.7</td><td>61</td><td>64</td></tr><tr><td>Processing:</td><td>4513</td><td>6771 1617.2</td><td>6938</td><td>9300</td></tr><tr><td>Waiting:</td><td>57</td><td>80 38.6</td><td>63</td><td>177</td></tr><tr><td>Total:</td><td>4568</td><td>6831 1619.0</td><td>7000</td><td>9362</td></tr></table></div>		min	mean[+/-sd]	median	max	Connect:	55	60 2.7	61	64	Processing:	4513	6771 1617.2	6938	9300	Waiting:	57	80 38.6	63	177	Total:	4568	6831 1619.0	7000	9362
	min	mean[+/-sd]	median	max																						
Connect:	55	60 2.7	61	64																						
Processing:	4513	6771 1617.2	6938	9300																						
Waiting:	57	80 38.6	63	177																						
Total:	4568	6831 1619.0	7000	9362																						

Concurrency level = 10

Time taken for tests (seconds)	15.514 seconds																									
Total transferred (bytes) and HTML transferred (bytes)	52431510 bytes 52428800 bytes																									
Time per request (ms)	15514.493 [ms]																									
Requests per second (#/sec)	0.64 [#/sec]																									
Transfer rate (Kbytes/sec)	3300.31 [Kbytes/sec]																									
Connection times	<div>Connection Times (ms)<table><tr><th></th><th>min</th><th>mean[+/-sd]</th><th>median</th><th>max</th></tr><tr><td>Connect:</td><td>59</td><td>76 14.9</td><td>85</td><td>95</td></tr><tr><td>Processing:</td><td>11338</td><td>13355 1179.5</td><td>13795</td><td>14693</td></tr><tr><td>Waiting:</td><td>76</td><td>285 179.6</td><td>300</td><td>580</td></tr><tr><td>Total:</td><td>11424</td><td>13431 1186.0</td><td>13884</td><td>14788</td></tr></table></div>		min	mean[+/-sd]	median	max	Connect:	59	76 14.9	85	95	Processing:	11338	13355 1179.5	13795	14693	Waiting:	76	285 179.6	300	580	Total:	11424	13431 1186.0	13884	14788
	min	mean[+/-sd]	median	max																						
Connect:	59	76 14.9	85	95																						
Processing:	11338	13355 1179.5	13795	14693																						
Waiting:	76	285 179.6	300	580																						
Total:	11424	13431 1186.0	13884	14788																						

Concurrency level = 1, with “Connection: Keep-Alive”

Time taken for tests (seconds)	16.930 seconds																														
Total transferred (bytes) and HTML transferred (bytes)	52431560 bytes 52428800 bytes																														
Time per request (ms)	1693.013 [ms]																														
Requests per second (#/sec)	0.59 [#/sec]																														
Transfer rate (Kbytes/sec)	3024.35 [Kbytes/sec]																														
Connection times	<table><tr><th colspan="5">Connection Times (ms)</th></tr><tr><th></th><th>min</th><th>mean[+/-sd]</th><th>median</th><th>max</th></tr><tr><td>Connect:</td><td>0</td><td>6 18.4</td><td>0</td><td>58</td></tr><tr><td>Processing:</td><td>920</td><td>1687 515.3</td><td>1548</td><td>2874</td></tr><tr><td>Waiting:</td><td>54</td><td>58 4.2</td><td>57</td><td>68</td></tr><tr><td>Total:</td><td>920</td><td>1693 530.3</td><td>1548</td><td>2933</td></tr></table>	Connection Times (ms)						min	mean[+/-sd]	median	max	Connect:	0	6 18.4	0	58	Processing:	920	1687 515.3	1548	2874	Waiting:	54	58 4.2	57	68	Total:	920	1693 530.3	1548	2933
Connection Times (ms)																															
	min	mean[+/-sd]	median	max																											
Connect:	0	6 18.4	0	58																											
Processing:	920	1687 515.3	1548	2874																											
Waiting:	54	58 4.2	57	68																											
Total:	920	1693 530.3	1548	2933																											

Concurrency level = 5, with "Connection: Keep-Alive"

Time taken for tests (seconds)	16.815 seconds																																				
Total transferred (bytes) and HTML transferred (bytes)	52431560 bytes 52428800 bytes																																				
Time per request (ms)	8407.616 [ms]																																				
Requests per second (#/sec)	0.59 [#/sec]																																				
Transfer rate (Kbytes/sec)	3045.02 [Kbytes/sec]																																				
Connection times	<table><tr><th colspan="6">Connection Times (ms)</th></tr><tr><th></th><th>min</th><th>mean</th><th>[+/-sd]</th><th>median</th><th>max</th></tr><tr><td>Connect:</td><td>0</td><td>30</td><td>31.5</td><td>57</td><td>65</td></tr><tr><td>Processing:</td><td>5280</td><td>6933</td><td>1422.9</td><td>6888</td><td>10524</td></tr><tr><td>Waiting:</td><td>55</td><td>78</td><td>42.2</td><td>61</td><td>183</td></tr><tr><td>Total:</td><td>5338</td><td>6963</td><td>1431.2</td><td>6945</td><td>10582</td></tr></table>	Connection Times (ms)							min	mean	[+/-sd]	median	max	Connect:	0	30	31.5	57	65	Processing:	5280	6933	1422.9	6888	10524	Waiting:	55	78	42.2	61	183	Total:	5338	6963	1431.2	6945	10582
Connection Times (ms)																																					
	min	mean	[+/-sd]	median	max																																
Connect:	0	30	31.5	57	65																																
Processing:	5280	6933	1422.9	6888	10524																																
Waiting:	55	78	42.2	61	183																																
Total:	5338	6963	1431.2	6945	10582																																

Concurrency level = 10, with "Connection: Keep-Alive"

Time taken for tests (seconds)	16.478 seconds																									
Total transferred (bytes) and HTML transferred (bytes)	52431560 bytes 52428800 bytes																									
Time per request (ms)	16478.320 [ms]																									
Requests per second (#/sec)	0.61 [#/sec]																									
Transfer rate (Kbytes/sec)	3107.28 [Kbytes/sec]																									
Connection times	<div>Connection Times (ms)<table><tr><th></th><th>min</th><th>mean[+/-sd]</th><th>median</th><th>max</th></tr><tr><td>Connect:</td><td>55</td><td>61 5.0</td><td>60</td><td>71</td></tr><tr><td>Processing:</td><td>11555</td><td>14420 1318.9</td><td>14659</td><td>15860</td></tr><tr><td>Waiting:</td><td>56</td><td>240 158.3</td><td>248</td><td>494</td></tr><tr><td>Total:</td><td>11611</td><td>14481 1320.1</td><td>14723</td><td>15918</td></tr></table></div>		min	mean[+/-sd]	median	max	Connect:	55	61 5.0	60	71	Processing:	11555	14420 1318.9	14659	15860	Waiting:	56	240 158.3	248	494	Total:	11611	14481 1320.1	14723	15918
	min	mean[+/-sd]	median	max																						
Connect:	55	61 5.0	60	71																						
Processing:	11555	14420 1318.9	14659	15860																						
Waiting:	56	240 158.3	248	494																						
Total:	11611	14481 1320.1	14723	15918																						

When we increase the concurrency level, request rate (request per second) is increased because multiple downloads are in progress while we increment concurrency, hence it affects requests rate.

Time per request is increased because while we use concurrency level one we are only downloading 1 file at a time with the maximum speed, on the other hand when multiple concurrent processes are running at the same time, there are multiple files downloading at a time. Because of that all concurrent downloading processes(hence requests) will slow down accordingly and completion of requests will take more time.

Transfer rate is increased from concurrency level 1 to 5 but not much from 5 to 10. It is because while we are using level=1 our internet connection speed is capable of more throughput and on level=5 we are using more

throughput. But when we used level=10 concurrency transfer rate did not increase that much because we were already using almost full of our available throughput.

Similarly, time taken for tests are not much different except the first evaluation. Because in all others we already reach the maximum bandwidth we have, hence it doesn't matter if we are downloading in parallel or not. But for the single concurrency cases; we can't reach the maximum throughput in first evaluation while we can with -k param. It is because using keep-alive connections we reduced the number of requests by keeping the connection active and doesn't require creating a new connection for each packet.

Total bytes transferred a bit higher with keep-alive connections. Additional headers transferred might be causing that, such as "Connection: Keep-alive" header and tokens that are being sent with each following request. Total HTML transferred is the same because that's the size of the file and not affected by headers etc.

Connections times vary between non keep-alive and keep-alive connections. Results are much better with keep-alive connections since connections stay active after established and there is no need for handshaking at each request.

4. Testing our server with ApacheBench

```
ab -n 1000 -c <concurrency_level> http://localhost:8080/100
```

Concurrency Level	Time per request (mean) (ms)	Time per request (mean, across all concurrent requests) (ms)	Requests per second (#/sec)
1	7.379	7.379	135.52
2	9.029	4.515	221.50
3	11.076	3.692	270.86
4	12.094	3.024	330.74
5	13.910	2.782	359.44
6	15.708	2.618	381.97
12	30.665	2.555	391.33
30	76.036	2.535	394.55
50	128.017	2.560	390.57
100	258.345	2.583	387.08

Since our server is multithreaded, when we increase the concurrency, the number of requests per second is also increased until some reasonable concurrency, because of that, the mean of time per request across all concurrent requests is also decreased. Looking at the results table and *Figure: 8-9*, we can say that our server is well performant with 12 concurrent requests at a time and increasing the concurrent request count has no obvious effect. We found this result satisfactory since our web server is also running with 12 threads.

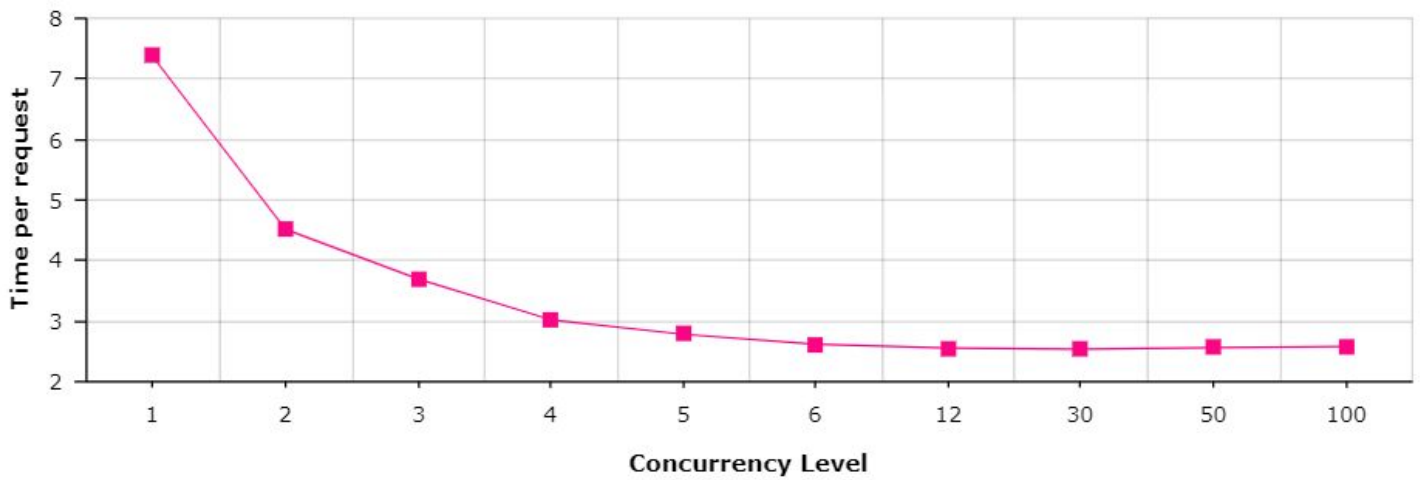


Figure 8: Line chart representation of time per request (mean, across all concurrent requests)

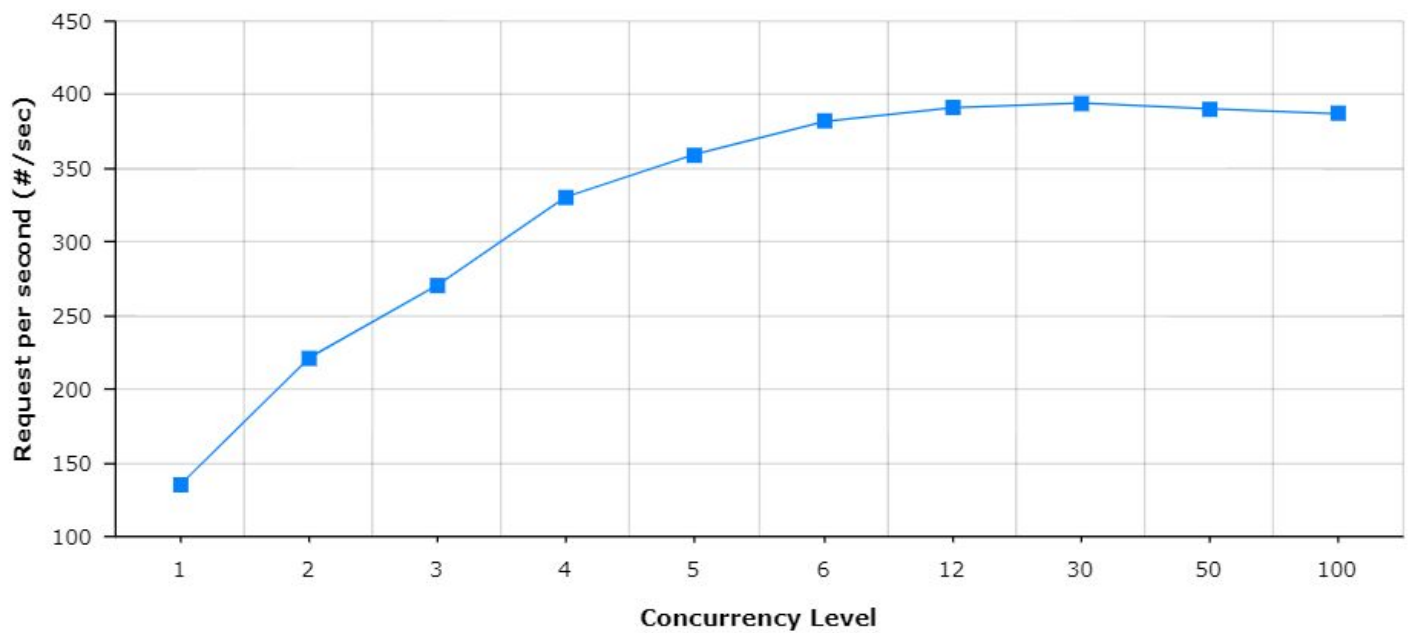


Figure 9: Line chart representation of request per second

Repeating same tests for proxy server:

ab -n 1000 -c <concurrency_level> -X 127.0.0.1:8888 <http://localhost:8080/100>

Concurrency Level	Time per request (mean)	Time per request (mean, across all concurrent requests)	Requests per second (#/sec)
1	8.820	8.820	113.38
2	14.174	7.087	141.10
3	17.915	5.972	167.45
4	20.656	5.164	193.65
5	24.636	4.927	202.95
6	32.981	5.497	181.92
12	60.612	5.051	197.98
30	138.213	4.607	217.06
50	252.607	5.052	197.94
100	381.747	3.817	261.95

Same comments regarding our web server are valid for proxy servers as well. Request rates are lower because there is an additional proxy server now.

Using the -k argument for ApacheBench has no effect since our HTTP web server has no supporting feature for keep-alive connections.