**Advanced Telecommunications: Assignment 1**

A Web Proxy Server

*Claire Cassidy (16325301)*

Documentation Contents

[Problem Statement: 1](#_Toc66474000)

[Program Setup: 1](#_Toc66474001)

[Program Design: 3](#_Toc66474002)

[Implementation Details: 4](#_Toc66474003)

[Measuring Cache Performance: 5](#_Toc66474004)

[Personal Reflection: 9](#_Toc66474005)

## Problem Statement:

A Web Proxy Server is a server that acts as a middleman between a client and a real third-party server. The web proxy intercepts all outgoing HTTP(S) and WebSocket requests from the client, and fulfils those requests on the client’s behalf. This can either be achieved by contacting the third-party server directly to obtain the required resource(s), or by delivering an in-date cached copy of the resource that has been obtained by the server on a recent request for the same resource. In doing so, the proxy server can reduce response time for the client and bandwidth on the local network by negating the need to establish a connection with the third-party server and transfer the requested resources from a potentially distant source. This can result in increased performance of the network overall. Furthermore, the proxy can also be configured to dynamically block resources, including whole websites or specific resources on those websites, making the proxy a more general tool for a network administrator to manage network access.

## Program Setup:

Once downloaded, the program can be run by compiling the source files and running the Launcher class.

It is recommended to run this program by configuring it as an IntelliJ project and interacting with the program via the built-in terminal, since some terminals such as the generic Windows cmd terminal don’t support multi-coloured printing. Powershell appears to show some colours but doesn’t disambiguate between normal console logs and stack traces and may be more confusing to read.

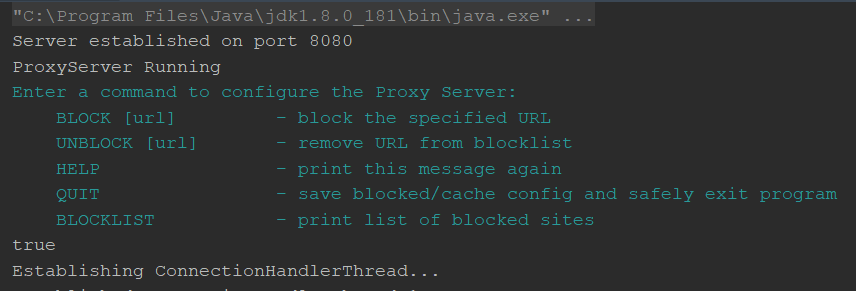


Figure 1: Example of coloured output from IntelliJ terminal

By default, the program is configured to use localhost port 8080. The program was tested in Firefox 86.0.1, due to its ease configuring it to use a proxy at the desired port. This can be done by going to *Options >* type “proxy” in the options searchbar > Settings… > Manual Proxy Configuration and entering the following:

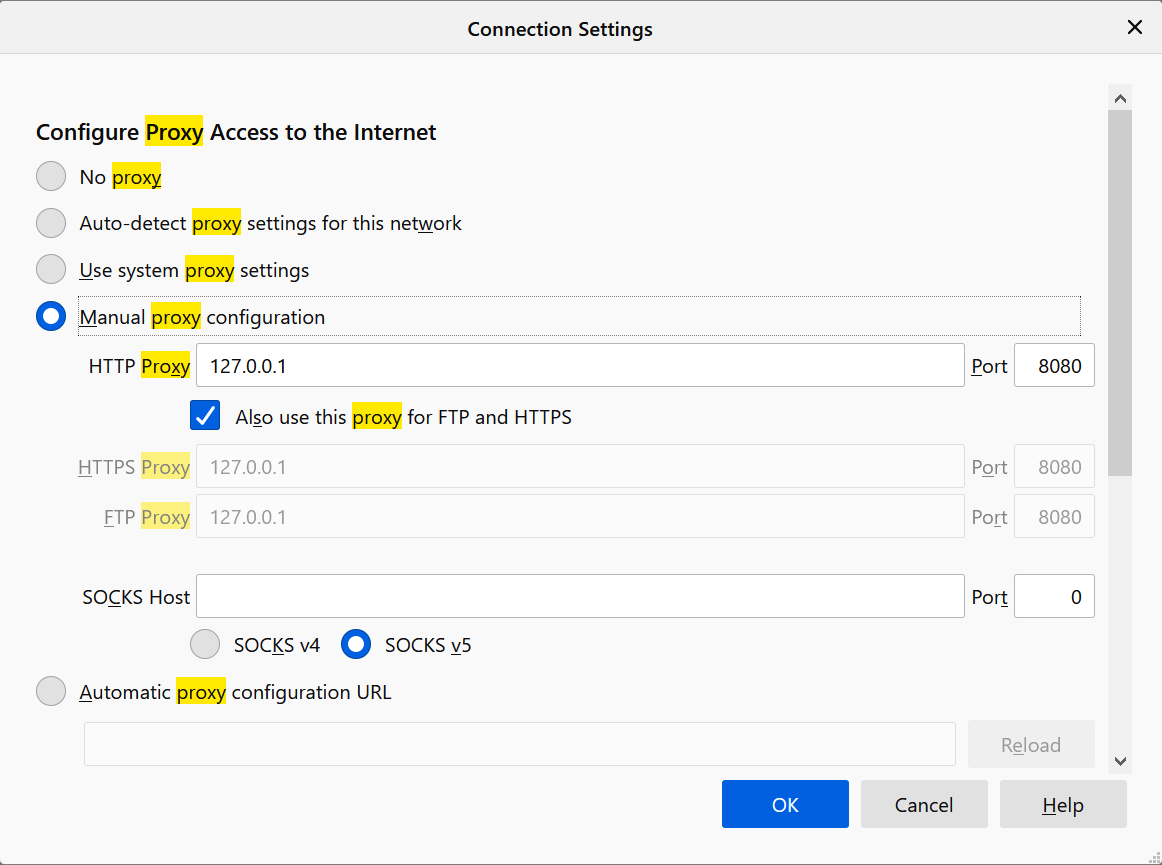


Figure 2: Configuring Firefox to use the Proxy on port 8080

Once this is done, all requests will automatically be routed through the program.

## Program Design:

I will now discuss the precise specifications given in the assignment and how my implementation tackles them at a high level. Subsequent sections will discuss each of the classes and their code in detail.

1. *Respond to HTTP and HTTPs requests and display each request on a management console. It should forward the request to the web server and relay the response in the browser.*

The project contains a MangementConsole class, which runs on a separate thread and exists solely to handle configuration input from the user. It polls stdin for input, parses it in relation to a set of recognised commands, such as BLOCK; UNBLOCK; QUIT; etc., and performs the appropriate action to satisfy the user’s request. To disambiguate between messages logged by the worker threads and communication between the end user and the management console, messages originating from the management console are printed in a separate colour.

As the entry and exit point for the program, the Management Console also has jurisdiction over creating and maintaining the persistent data that the proxy server needs to operate; namely the Cache directory and the file containing the blocked sites. These aspects will be discussed in further detail shortly.

The Management Console is booted up before the proxy server itself is started, so that the files are ensured to be created and appropriate data is loaded before operation begins. Once this is complete, the Management console creates a ProxyServer thread, which can now begin servicing requests. The ManagementConsole remains listening until it receives the QUIT command, at which point it will signal the blockedSites instance object to be written to the blockedsites.txt file. **For this reason, it is recommended that the program is only terminated after the QUIT command has been entered**, to ensure persistent features work as intended.

The ProxyServer thread actually doesn’t do much of the hard work; its job is just to accept incoming requests for resources from the browser and spawn a new ConnectionHandlerThread to service each request, passing it the socket on which the connection was made. Communication between the proxy and the client then takes place on a per-request basis, with each request being serviced in its entirety by its own ConnectionHandlerThread. HTTP and HTTPS requests alike are passed to the same kind of worker thread, although the method for dealing with HTTPS requests is slightly different, as will be discussed later. The other function of the ProxyServer is to respond to shutdown requests from the ManagementConsole, which are received on receipt of a QUIT command from the user.

The ConnectionHandlerThread checks the method type of the request and prints out whether it’s a HTTP or HTTPS request to the console, as well as the requested resource URL and the socket information for the connection to the client. The ConnectionHandlerThread then decides how to service the request based on whether the resource falls under a blocked resource and whether or not it is cached already. The details of this will be discussed later. Given that the resource is not blocked, the thread either delivers the cached version of the resource or performs a HTTP(S) request to the endpoint, and forwards the reply back to the client. The result of this can be seen in the user’s web browser as required.

1. *Handle Websocket connections.*

Due to time restrictions and lack of library support, I was unable to complete this portion of the problem statement. However, a form of full-duplex communication similar to that adopted by WebSockets can be seen in the way the program handles HTTPs requests (discussed in a later section), which may form part of the solution. What would remain would be to expand the ConnectionHandlerThread class to parse requests using the **ws://** and **wss://** headers.

1. *Dynamically block selected URLs via the management console.*

As described above, the ManagementConsole is launched at start-up and listens for BLOCK/UNBLOCK requests from the user at the console. This can be used to configure blocked sites on the fly, and block and unblock a resource in the same session without needing to reload the program. Each ConnectionHandlerThread consults a dynamically-updated arraylist containing the URLs of blocked resources before serving a request. The changes to the blocklist are persistent and written to/loaded from the blocked.txt file in the res folder to preserve the list between sessions.

1. *Efficiently cache HTTP requests locally and thus save bandwidth. You must gather timing and bandwidth data to prove the efficiency of your proxy.*

The res folder contains a subfolder called cache which contains each of the cached HTTP resources the proxy has access to. When a URL is fetched from a remote endpoint, this will mean the file wasn’t found in the cache. The fetched HTTP resource will then automatically be written to the cache as it is being sent back to the client, using an encoding scheme that preserves the original URL of the resource. When a new request comes in, the URL will be converted into this encoded form, checked against the filenames in the cache folder, and be retrieved and sent back to the user if it exists.

The cache files also take into account the expiry dates given in the HTTP response headers supplied by the endpoint, where they exist. Cache files are written with the expiry date/time on the first like, followed by the actual resource on the second line. Thus, once we find a match, we can also check the expiry date against the current time to ensure the user is getting a fresh copy of the resource they requested.

Timing and bandwidth data will be provided in the section *Measuring Cache Performance*.

1. *Handle multiple requests simultaneously using a threaded server.*

As discussed above, each HTTP(S) request spawns its own ConnectionHandlerThread, which deals with a single HTTP(S) request in its entirety and ensures that many requests can be serviced simultaneously.

## Implementation Details:

Having given an overview of my implementation in response to the problem statement, I elect in this section to go into further details regarding my implantation on a class-by-class basis.

### Launcher.java

This class contains the main method for the program and simply creates the ManagementConsole thread and leaves it to do the rest of the work

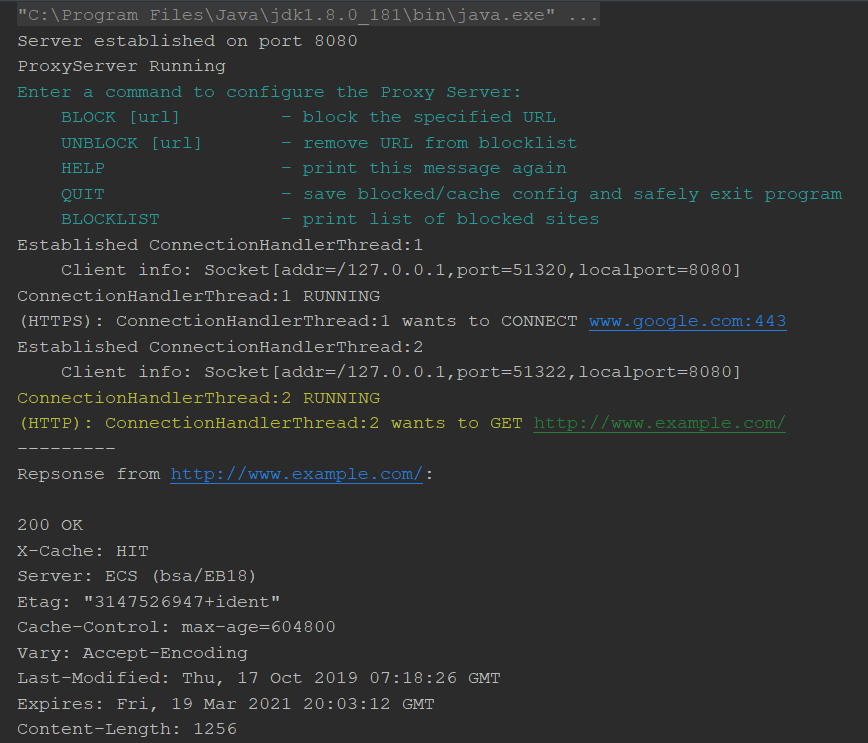
```

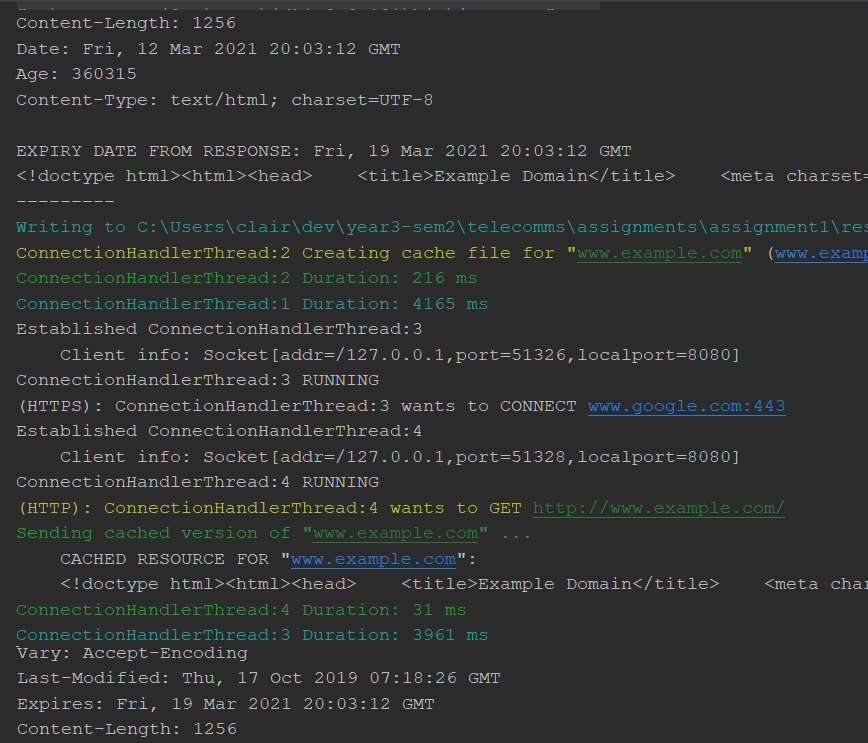
df

```

## Measuring Cache Performance:

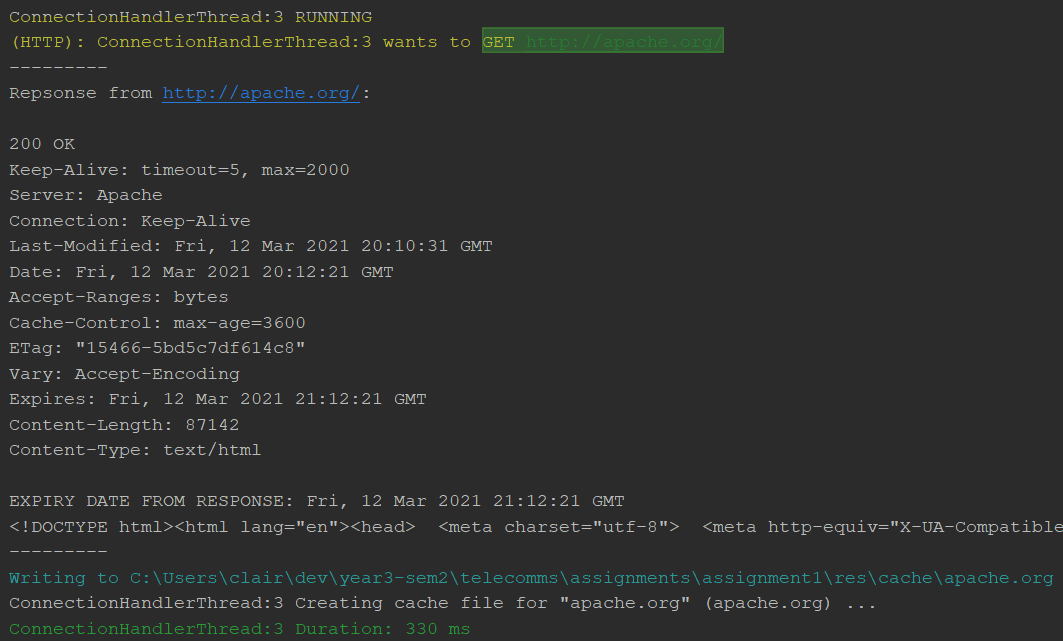
By using a fresh cache and attempting to access [www.example.com](http://www.example.com), then immediately sending another request for the same resource, I received the following timing information (see highlighted output):

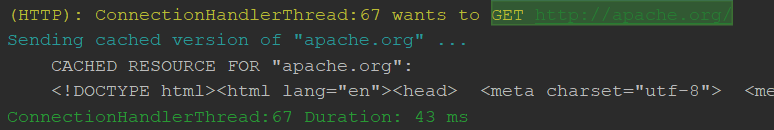


**

As the log output shows, the first attempt at fetching the resource required 216 ms to complete to fetch from the source and deliver to the client. Once the resource had been cached, it took only 31 ms.

I tried this again with another (mostly) HTTP site, apache.org. To get the base webpage without caching took 330 ms, whereas with caching it took only 43 ms.





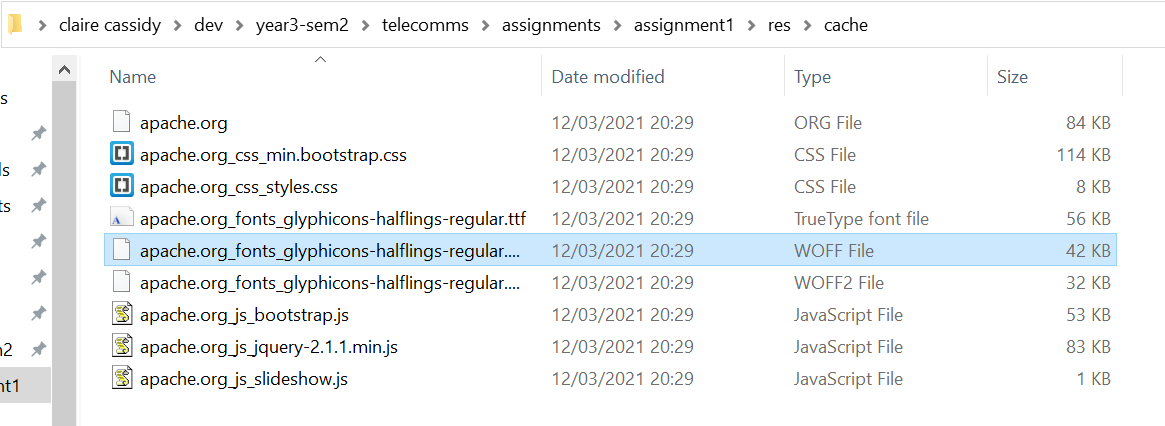
This method of timing cannot take into account the time to transmit to the client, however. When taking timing information from the browser itself (ensuring caching in the browser is turned off), caching also proved to achieve significant speedups. For example, see the before and after for the following files that are cached when visiting apache.org:

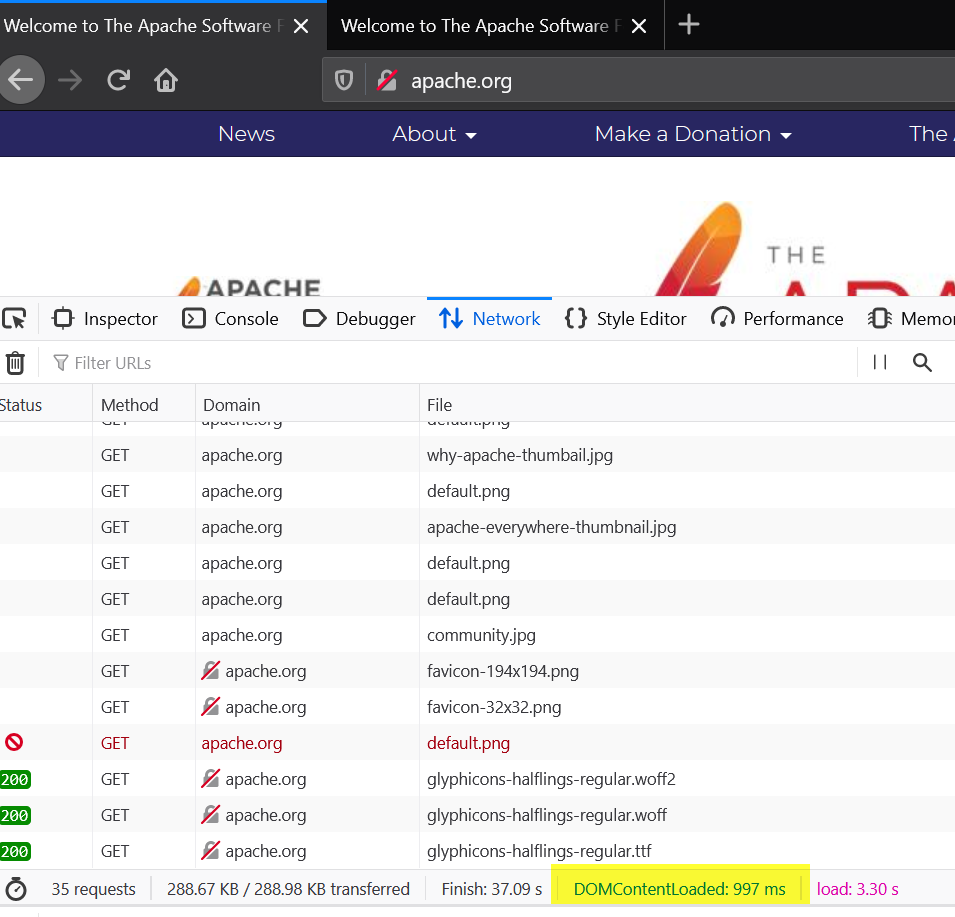
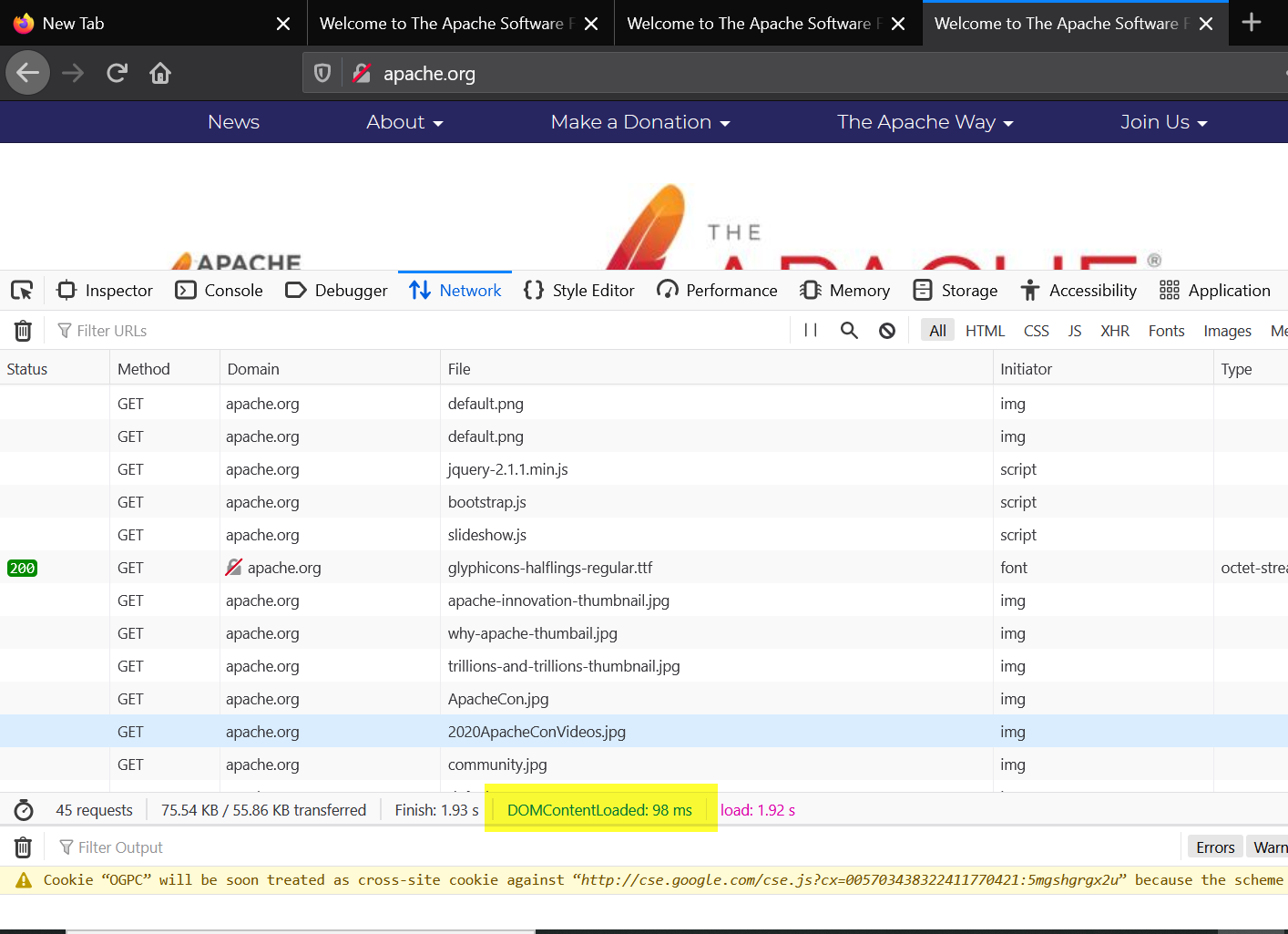
Before caching:



After caching:





Overall, there is a significant speedup in loading the DOM after caching: 

*Note: by default, the timing code has been commented out in the source files to prevent cluttering the output. You may uncomment them to verify timing information.*

## Personal Reflection:

I chose to write the project in Java, which in hindsight made the experience much more challenging than I anticipated. There weren’t a lot of resources online to help with the construction of a web proxy from scratch, at least one as sufficiently complex as is required for this project. If I could start over, I more than likely would’ve chosen a language like JS or Python. However, in the end I believe I managed to cover almost all aspects of the project specification.

// TODO won't stop until it receives one more request which is annoying; write in docs that its safe to just terminate the program after quit regardless of if it exits itself.

// can't load the cache dynamically.

// On startup, run through expiry dates and delete expired files.

// 1. check the directory for a filename that matches the url

// 2. if match -> open file -> get first line -> check cur date time against expiry date

// -> if expiry date OK -> get rest of file and serve

// -> else -> delete file from cache -> forward request as normal -> save response to cache

// 3. if no match -> forward request as normal -> save response to cache