Project 1 - Web Proxy Server Report

Student Name: Darragh Murray

Student Number: 18320728

Introduction

The purpose of this project is to create a web proxy server with the following given specifications:

The objective of the exercise is to implement a Web Proxy Server. A Web proxy is a local server, which fetches items from the Web on behalf of a Web client instead of the client fetching them directly. This allows for caching of pages and access control.

The program should be able to:

1. Respond to **HTTP & HTTPS** requests and should **display each request** on a management console. It should forward the request to the Web server and relay the response to the browser.
2. Handle **Websocket** connections.
3. Dynamically **block selected URLs** via the management console.
4. Efficiently **cache** HTTP requests locally and thus save bandwidth. You must gather **timing and bandwidth** data to prove the efficiency of your proxy.
5. Handle multiple requests simultaneously by implementing a **threaded server**.

The program can be written in a programming language of your choice. However, you must ensure that you do not overuse any API or Library functionality that implements the majority of the work for you.

Implementation

I implemented this project using Javascript and NodeJS. I choose NodeJS because I was familiar with it and many libraries exist that allow for easier and more efficient completion of projects. I will now describe how I implemented each of the five bullet point requirements listed above. You will need to install node to run this project. Also, it is required to run the following two commands before running the server:

npm install node-cache

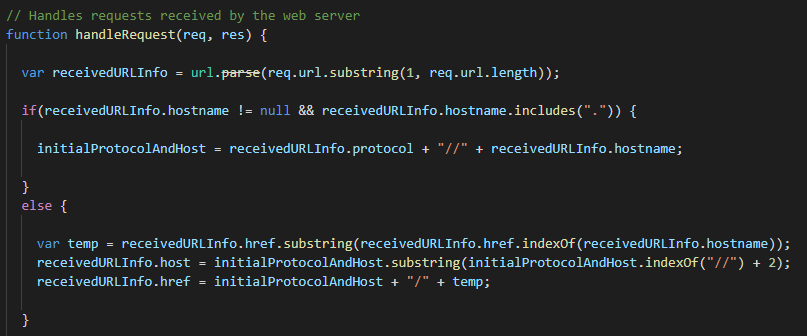
npm install ws

**HTTP & HTTPS Requests**

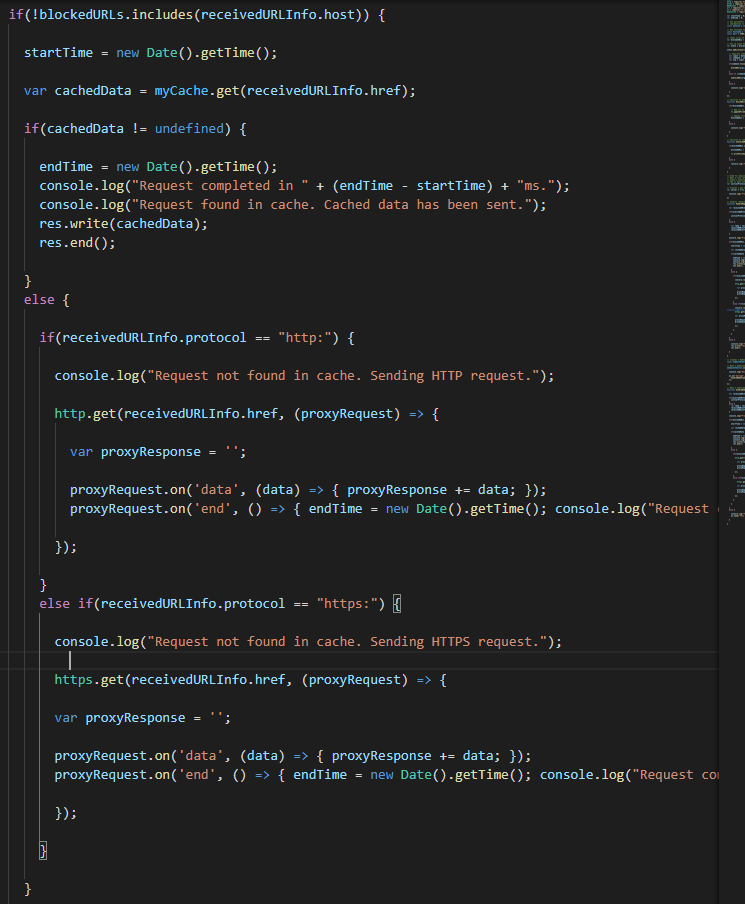
My web proxy, upon receiving a request, calls a function handleRequest.

Firstly, this function breaks up the request, which is passed through the parameters, into various URL parts using the url Javascript library.

Next, the code contains an if-else statement which modifies the value of a variable initialProtocolAndHost. The purpose of this variable is to handle the case where a file references other locally stored files but without the host or any protocol e.g. a html file referencing a picture stored locally as image1.png. When a request has been detected without a protocol or host, the last received request host and protocol is appended onto the front of the request.

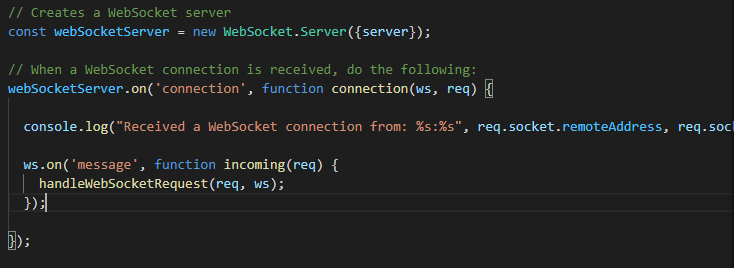
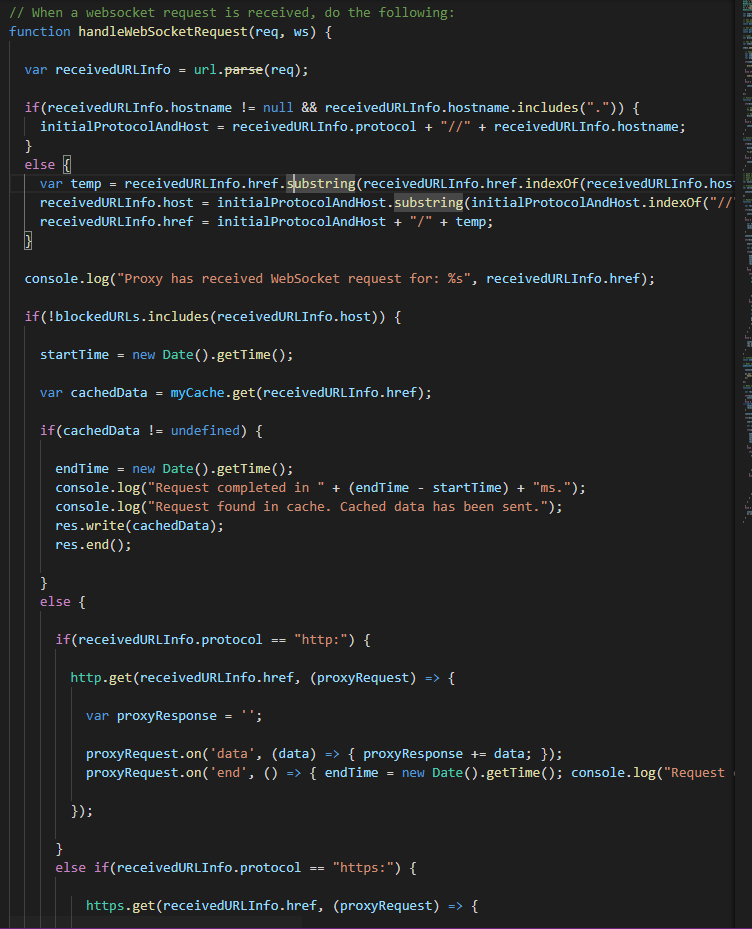


Then, the code will check if the host is blocked, if it is then a simple message will be printed stating so, if not, then the code will check if the request has been cached, it the request is found in the cache, the cached data is sent, if not, then an if-else statements checks what the protocol is and sends the appropriate HTTP or HTTPS get request. The response is then sent back to the source of the request.



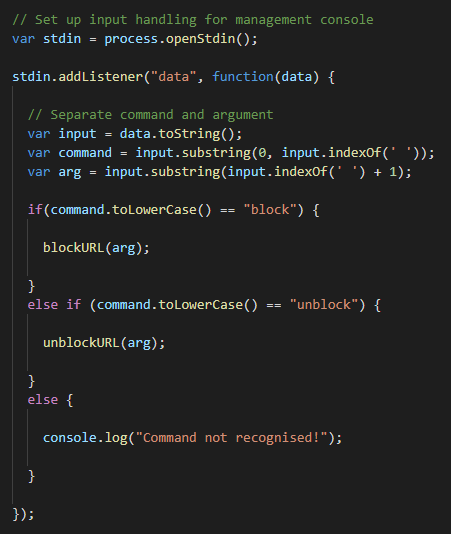
**WebSocket Connections**

My proxy is fully equipped to handle WebSocket connections and maintain all functionality that is available through standard requests. I use a Javascript library called ws to handle websocket connections.

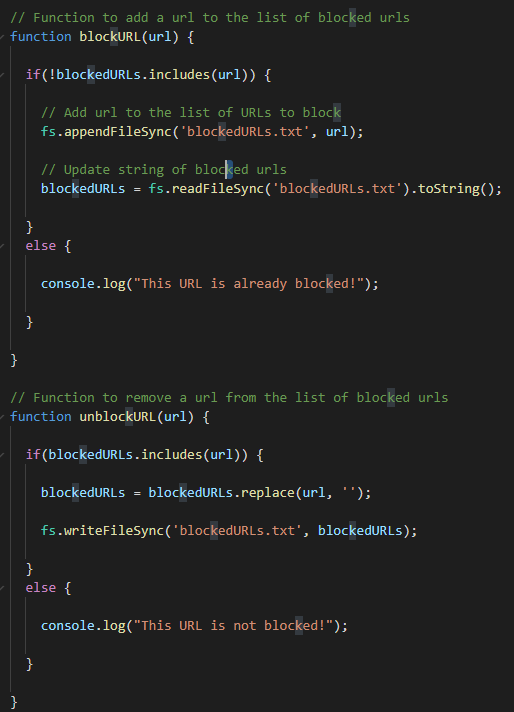
I then use a function I created called handleWebSocketRequest, which does all the same things as the function handleRequest, except at the end it sends to data through the websocket.

**Dynamic Blocking**

Users can block select hosts via the command line for the proxy.



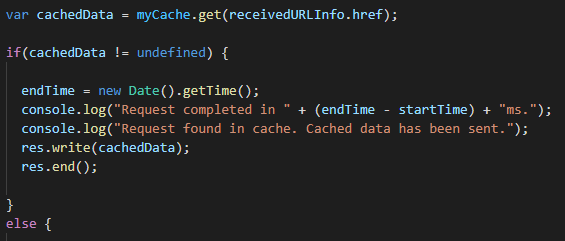
Typing ‘block url’ will block whatever url is specified by triggering the blockURL function, and ‘unblock url’ will unblock whatever url is specified by triggering the unblockURL function, given that it is indeed blocked.



The way the blocking system works is by writing to a file called ‘blockedURLs.txt’ and keeping an array within the code. This allows for the remembrance of blocked URL’s even if the proxy is shut down and restarted at a later date.

**Cache and Timings**

I implemented cache with the help of the node-cache library. The cache stores key-value pairs, with the key being the requested URL and the data being the data that was received when a request was made to the specified key. When a user requests, the cache checks if the request is stored in the cache, if it isn’t then http(s) request is sent and the data that is received is stored in the cache for the specified amount of time.



The way the timings work is that I have two global variables startTime and endTime. Whenever a request is made startTime is set equal to the current time in seconds since January 1st 1970, and then when the data is finally received, the endTime is set equal to current time in seconds since January 1st 1970, when I subtract startTime from endTime we get the time elapsed since the request was made. The console will print out the time it took for each request, therefore we can see the difference in time when a request needs to be made vs when a request is found in the cache.