Overview

For the fourth practical we had to build a web proxy server to keep a subnet secure. The proxy server should accept a regular HTTP request on a port from clients and then forward it on to target machine in HTTPS. Then it should read the response and transmit the data back to the original client. Therefore, the client can talk HTTP to the proxy, and the proxy talks HTTPS to the target server, thus securing all connections, despite clients only using HTTP.

Design – Basic

For this practical I decided to make the proxy multi-threaded allowing it to simultaneously handle requests thus improving the speed at which webpages can be transferred and displayed.
For the basic requirement, I made two classes, the first class "ProxyServer" contains the main method and is used to start the proxy server. This class is the responsible for spawning a new thread per request. When the program is run a new "ServerSocket" object is created, in a try with resources block, next the "ServerSoket" is bound to port, and finally an infinite while loop is entered. In the while loop a new "ProxyThread" object is initiated. Here the accept method is called on the "ServerSocket" object which listens for a connection to be made to this socket and accepts it. The method blocks until a connection is made and when a connection is made a new "Socket" is created which is taken in by the "ProxyThread" constructor finally the "start" method is called which starts processing the request on a new thread.

The next class I created is the "ProxyThread" class which is where the core logic of the program resides. This class extends the "Thread" class and overrides the "run" method. When a new request is received, the constructor takes in the "Socket" object and stores it, then by default the run method will be called, by the "ProxyServer" class.

Once in the run method a new "DataOutputStream" object and "BufferedReader" object is created. The "BufferedReader" object is used to receive the request from the client which will contain the URL for the target server. To get the URL I have created a method "getURL" which takes in the "BufferedReader" object. A while loop is then used to read in each line of the request header and each line is split up into individual token by a "StringTokenizer" object. Because the first line of the request is the line which contains the URL for the target machine I've used a simple if statement to check the current line of the received request. If the current line is the first line, it's split up by spaces and each section is stored in an array. By default, the second element in the first line will be the desired URL which is saved as a string. Finally, the rest of the header is parsed and the URL is returned. If the URL returned starts with "http://" this is replaced with "https://" then a new instance of the "HttpURLConnection" class is created and set to null.
Once this is done I decided to create a method which checks if the target machine supports HTTPS, called "pingURL". This method takes in the URL string and tries to connect the target machine using a new "HttpsURLConnection" object. If the "HttpsURLConnection" object responds with a code between 200 and 400, within a given time limit, it means the target machine supports HTTPS and the function returns true, otherwise it returns false.
As an extension, I decided to allow the proxy to open a connection to target machine even when it doesn't support HTTPS, so if the "pingURL" function returns false, the "https://" at the start of the URL is then replaced with "http://". Therefor the proxy will by default always first try to first communicate to the target machine in HTTPS, however if this fails, the proxy rolls over to HTTP meaning all webpages are available to a user of the proxy. The reason I decided to do this was because there are still a lot of websites which don't support HTTPS or at least there are elements on the webpage that don't support HTTPS. Initially such websites or elements on websites would not load and therefore deemed most website unreachable and unusable but by allowing HTTP the wide majority of connections are secured but connections to URLS for .css and .js files are now allowed and mean website build properly.

Next a method "createHTTPConnection" is called which takes in the URL string. Here a new URL object is created using the input string, then a new "HttpURLConnection" is created and is set equal to a "URLConnection" object which is created by calling the "openConnection" method on the URL object. The response code of the "HttpURLConnection" object is checked to see if the target machine has moved, if so the new URL of the target machine is found and the "HttpURLConnection" is updated. Finally, the "HttpURLConnection" is returned and is passed into the "writeData" method along with the "DataOutputStream" object. Here an "InputStream" object is created which is used to receive the data from the HttpURLConnection, a new array of bytes is also created. Next an integer variable "index" is set equal to the result of the "read" method which reads up to a set number of bytes from the input stream into the byte array and then returns the total number of bytes read into the buffer, or -1 if there is no more data. Once the bytes have been read in and the index variable has been set a while is entered which, using the "DataOutputStream" object, writes the contents of the byte array to client then the index variable is again set to the result of the read method which will read in the next chunk of bytes from the input stream. The loop breaks when "index" is set equal to -1, signifying all the data has been read in.

Design – Extension

For this practical I decided to implement a cashing system which in theory would improve performance by storing commonly-accessed sites meaning the proxy would not need to contact the target server if a page already existed in its cache. However, because many websites change and update, each page stored should only be stored for a certain time before it expires and is removed. Moreover, to stop the cache consuming a lot of the space, there should limit to the number of sites stored and so even if a stored item hasn't expired but is not regularly accessed it should also be removed. To do this I have opted to use the apache common collection contain the "LRUMap" which, removes the least used entries from a fixed sized map. As a guide on how to create a thread safe caching system I used a tutorial located at http://crunchify.com/how-to-create-a-simple-in-memory-cache-in-java-lightweight-cache/.
To implement the cashing system, I have created two classes: "CachedPage" and "CacheManager". The "CachedPage" is used to represent a webpage stored in cache has two attributes: "lastAccessed" which in a long and stores the time the site was last accessed and "page" which is a byte array containing the data to make the webpage. The "CacheManager" is then the class used to manipulate the cache. This class contains a constructor which takes in a long "timeToLive", that is the default value all pages stored expire after, a long "timeInterval", which is the time interval between checking for expired items and finally an integer "maxItems" which is the maximum number of webpages that can be stored in the cache. When the constructor is called, a new thread is created, which contains an infinite loop, in the loop the "Thread.sleep" method is called with the "timeInterval" parameter, then the method "cleanup" is called. This means a thread is run in the background for the duration the program is being run and maintains the cache store.

The next methods in this class are the “put” and “get” methods, which are both synchronized, meaning they can be accessed by one thread at a time and will stop concurrent modification of the cache. Therefore, for example, if the same webpage was accessed twice at the same time, only one copy of the webpage would be stored because only one thread can access each method at any given time. The put method maps the page URL to a “CachedPage” object. The get method uses a page URL to find items stored in the cache. The final method is the “cleanUp” method which is also synchronized. Upon being called the current time is recorded and a new Array List, “deleteKey”, is created which is used to store all the deleted items. Next a “MapIterator” object is created which is used to iterate over the contents of the “LRUMap”, the reason for doing this is to because you can’t loop over a map and delete items in the list while doing so i.e. (concurrent modification). A check is then preformed on each “CachedPage” item in the iterator, which checks if its “timeToLive” value plus the “lastAccsessed” value is less than the current time, if so its expired and it’s added to the Array List, “deleteKey”. Finally, when the iterator completes, it loops through the Array List, “deleteKey” and removes all the entries from the “LRUMap”.

To implement the cache system, I first create a new static instance of the “CacheManager” class when the “ProxyThread” class is initated. The “CacheManager” object is set up with the following attributes: TimeToLive = 300 seconds, TimerInterval = 100 seconds and maxItems = 10000. In the “writeData” method I have had to implement some logic to only cache unseen websites, to do this I added a new Boolean parameter to the method, “cached”. In the method, I now create a new “ByteArrayOutputStream” which is used to store all the bytes read in from the “inputStream”. If the Boolean value “fromCache” is set true and the website dosent already exists in cache, the “put” method from the “CacheManager” class is called; where the URL is set as the key and “ByteArrayOutputStream” object is converted to a byte array and stored as the “CachedPage” attribute. In the run method, I have also added an if statement, which checks if the current URL exists in the cache, if so the “writeMethod” is called directly which the “CachedPage” object found.

For my final extension, I decided to create a new proxy which can handle HTTPS requests as well as HTTP requests. I did this because I noticed when testing, if a user tries to send a HTTPS request through the proxy it fails because the proxy is verified. This means that sites that force HTTPS e.g. Twitter, Google and Facebook would not work correctly. Moreover, I discovered that when the proxy was located on an external machine from the client, the traffic sent between the client and the proxy could still be intercepted because the client can only talk HTTP to the proxy, this therefore was a major security flaw of my original proxy. To fix this I, need to go use a technology which was lower down the TCP/IP layering model which was sockets. Using sockets has the added advantage that the “URLConnection” classes uses sockets internally, thus sockets should a little bit faster as “URLConnection” class add overhead to the sockets. I have used a guide for reference on stack over flow to complete this extension which can be found here: https://stackoverflow.com/questions/16351413/java-https-proxy-using-https-proxyport-and-https-proxyhost.

To implement this extension, I have created two more classes: “ProxyServer” and “ProxyThread”. The “ProxyServer” class is very like the original “ProxyServer” class however this class also extends the Thread class and overrides the “run” method used to create new instances of the “ProxyThread” class. The “ProxyThread”, like the original, also extends the “Thread” class and overrides the “run” method.

In the “ProxyThread”, class the first method that gets called from the “run” method is the “readLine” method which is used to parse the header and it works by first creating a “ByteArrayOutputStream”, next a while loop is entered which terminates when the integer variable “next” is equal to -1. Upon each loop the “read” method is used to read the next byte of data from the client’s socket, input stream. In the while loop there is an if statement which checks if the current byte of data signifies a new line, if so the loop breaks as the URL is in the first line, otherwise the byte is written to the “ByteArrayOutputStream” object. One the while loop has been broken, the function returns the “ByteArrayOutputStream” object as a string using the "ISO-8859-1" encoding and all the new line and space characters are removed.

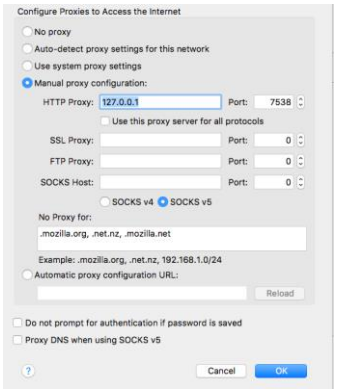
Next I have decided to the use the “Pattern” and “Matcher” classes to check if the string returned by the “readLine” matches an expected pattern. The pattern for a HTTPS request comes in a format: "CONNECT someurl:443 HTTP/1.1” and the pattern for the regular HTTP request comes in the format “(HTTP Request Method) someurl HTTP/1.1”. Therefore, I can check if the client sent a regular HTTP request or a HTTPS method or neither. If the client sent a regular HTTP request, the URL from the request is extracted and “http://” is replaced with “https://”. Finally, a header is sent back to the proxy using a “OutputStreamWriter” object. The header contains a 302-redirect message and the location of the new URL with the “https://” added and the “OutputStreamWriter” is closed. The client socket will then receive this redirect and send back a HTTPS request, which will then match the HTTPS pattern.

Once a valid HTTPS request has been received an attempt to open a port to the target machine on port 443 is made, if successful a new “OutputStreamWriter” object is used to send a header containing a 200 Connection Successful to the client port. Once completed the “forwardData” method is called which takes in the socket to the target machine.
Upon being called the “forwardData” method creates a new thread, which calls the “writeData” method and hands in the socket to target machine as the input socket and the client socket as the output socket. The “writeData” method is the same as in the original “ProxyThread” class. Next the “readData” method is called which reads in the next byte from the client’s socket input stream, if there is any more bytes available which don’t signify a new line, the “wirteData” is called again this time the clients socket is used as the input socket and the target socket is used as the output stream. The reason for this is to allow data to be tunneled from the target socket to the client socket on one thread and simultaneously on another thread have data tunneled client socket back the target socket, in doing so a SSL handshake can take place and the proxy is verified. Finally, when the all the data has been received by both sides the ports are closed.

Source: https://stackoverflow.com/questions/16351413/java-https-proxy-using-https-proxyport-and-https-proxyhost
The reason I didn’t include a cashing system in this extension is because the data in encrypted end to end and so the data received by the cant be read or reused.

Testing- Basic

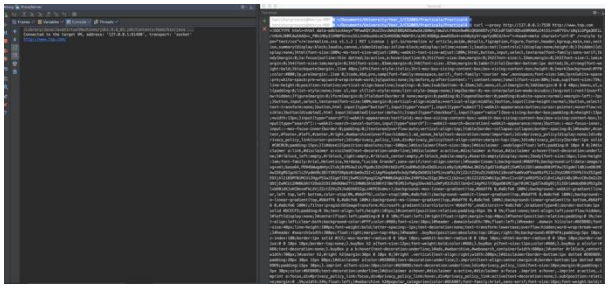
I've set up my proxy to run on port 7538 and to test my proxy I've used curl with the command: curl --proxy http://127.0.0.1:7538 http://www.example.com and Firefox with the proxy setting set up as follows:

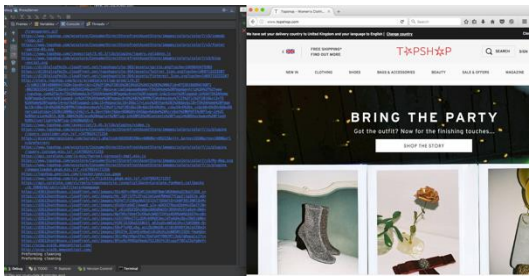


Test Description	Input	Expected Output	Actual Output	Comment
Connect to website that supports https but doesn't force it	www.topshop.com www.apple.com www.asos.com www.bigben.com www.chelseafc.com	The program will receive all the requests, make a https calls where possible for all sites and display the correct output pages should load properly and redirects should be followed	The program received all the requests and were all https calls where possible (otherwise http was used) Firefox and curl then displayed the correct data, redirects were also correctly followed	The program worked as expected

www.topshop.com

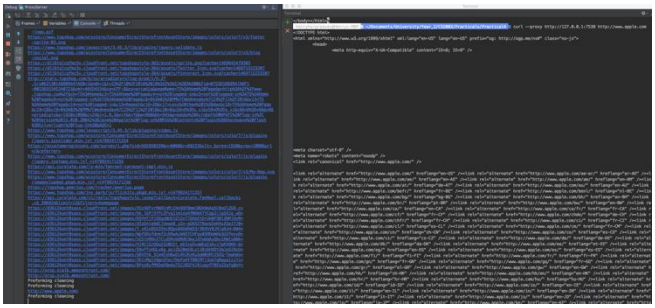
Curl **Firefox**

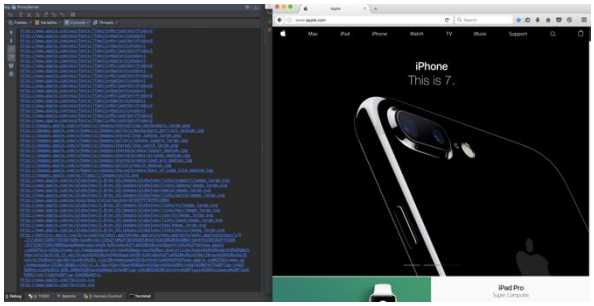




www.apple.com

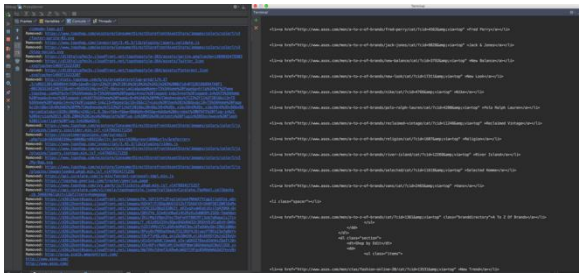
Curl **Firefox**

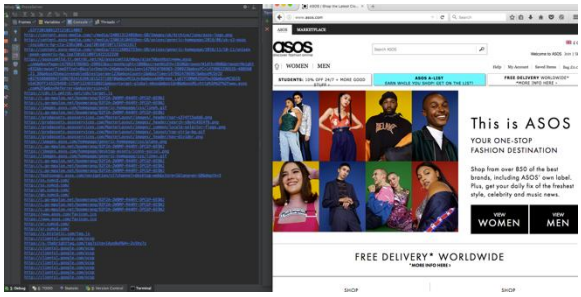




www.asos.com

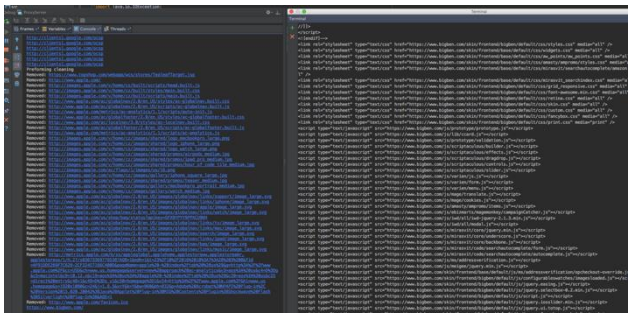
Curl Firefox

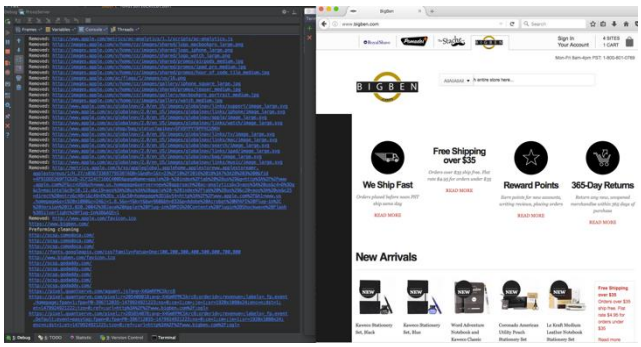




www.bigben.com

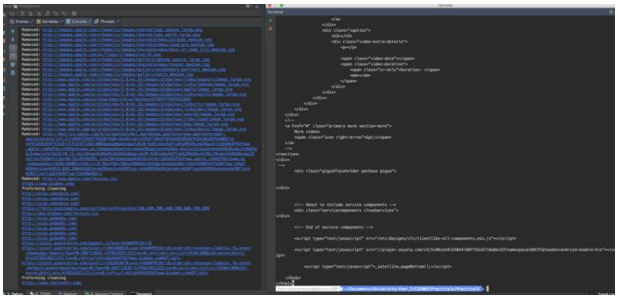
Curl Firefox

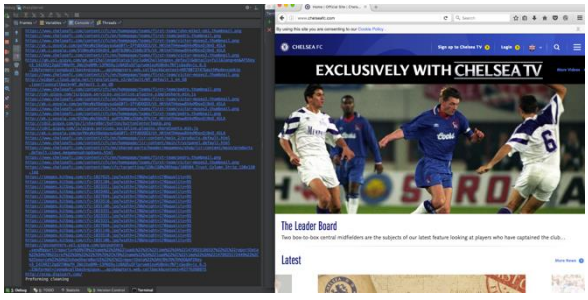




www.chelseafc.com

Curl Firefox



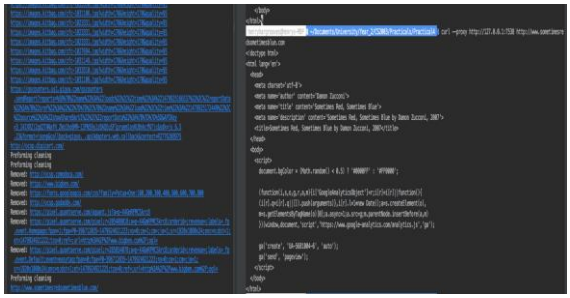


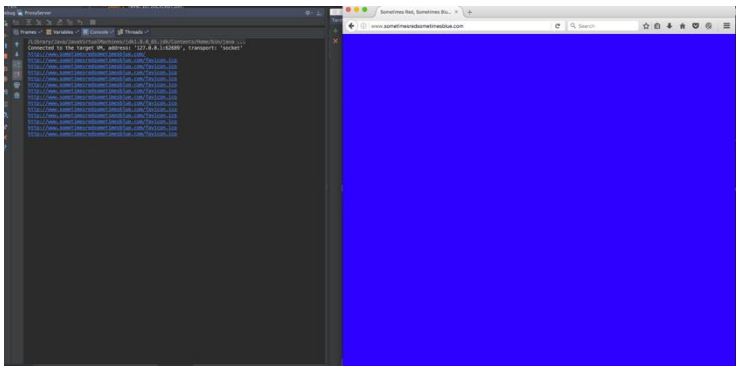
Testing

Test Description	Input	Expected Output	Actual Output	Comment
Connect to website that doesn't support https	www.sometimesred-sometimesblue.com www.telegraph.co.uk	The program will receive all the requests and make a http calls for all sites and display the correct output pages should load properly and redirects should be followed	The program received all the requests and made http calls Firefox and curl then displayed the correct data, redirects were also correctly followed	The program worked as expected

www.sometimesredsometimesblue.com

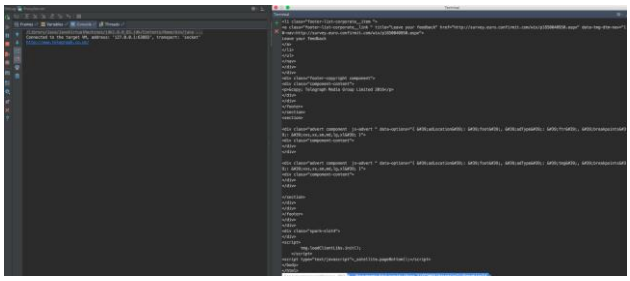
Curl Firefox

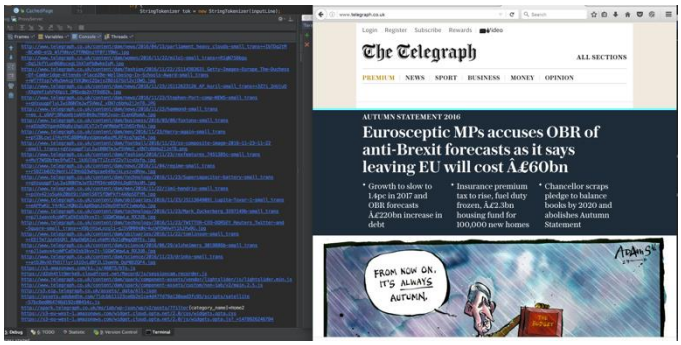




www.telegraph.co.uk

Curl Firefox





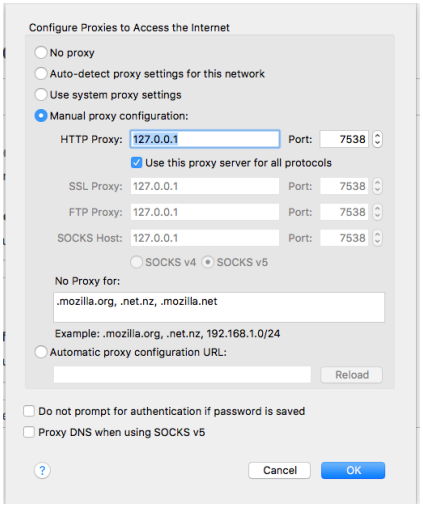
Test Description	Input	Expected Output	Actual Output	Comment
Connect to a website, then connect to the same website again.	www.topshop.com www.bing.com www.theguardian.com www.apple.com	The program will connect to the website the first time and the second time the website should be loaded from cache, the load time the second time should be quicker	The program connected to the website the first time and the second time the website was loaded from cache, the load time the second was also much quicker	The program worked as expected, however when testing websites that force HTTPS for example google.com a cached version cannot be loaded in because the client socket does not have the correct certificates.
Have items removed from cache automatically	Time to live = 10 seconds	Items should be removed from cache where the time to live is exceeded	When the time to live for a page exceeded it was automatically removed from cache	The program worked as expected
Have items removed from cache when cache limit is reached	Max = 10 items	Items should be removed from the cache which have the least frequency of uses	Items were removed from the cache which had the least frequency of uses	The program worked as expected

When testing with Firefox, cache and history were disabled to ensure fair results I've also totaled the results to load each element. I have noticed after

Test Description	From Target Curl (millisecond)	From cache Curl (millisecond)	From Target Fire Fox (millisecond)	From cache Fire Fox (millisecond)
www.topshop.com	878	1	52295	15
www.bing.com	409	1	14766	10
www.theguardian.com	1117	2	16651	10
www.apple.com	487	1	18011	3

Testing- Basic

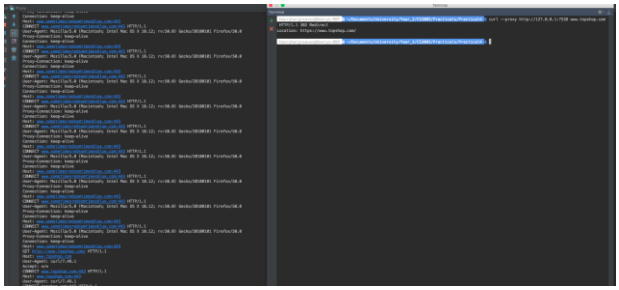
I've set up my proxy to run on port 7538 and to test my proxy I've used curl with the command: curl --proxy http://127.0.0.1:7538 http://www.example.com and Firefox with the proxy setting set up as follows:

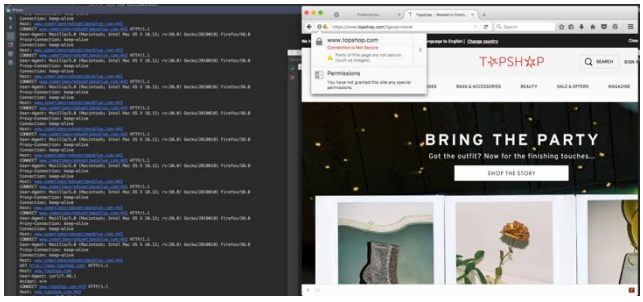


Test Description	Input	Expected Output	Actual Output	Comment
Connect to website that supports https but doesn't force it	www.topshop.com www.apple.com www.asos.com www.bigben.com www.chelseafc.com	The program will receive all the requests, make a https calls where possible for all sites and display the correct output pages should load properly and redirects should be followed	The program received all the requests and were all https calls where possible (otherwise http was used) Firefox and curl then displayed the correct data, redirects were also correctly followed	The program worked as expected

www.topshop.com

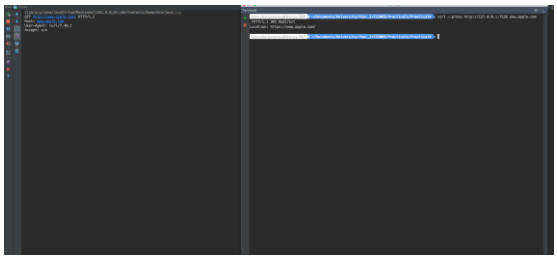
Curl **Firefox**



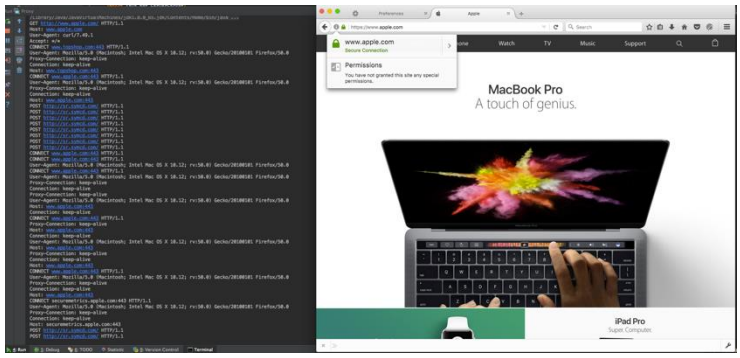


www.apple.com

Curl

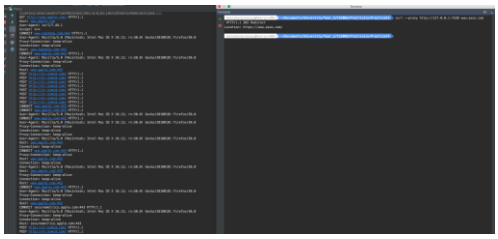


Firefox

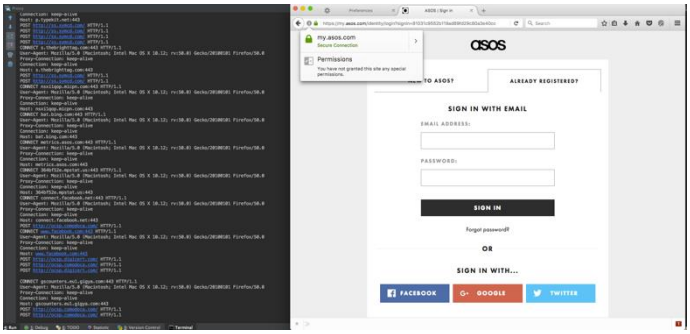


www.asos.com

Curl

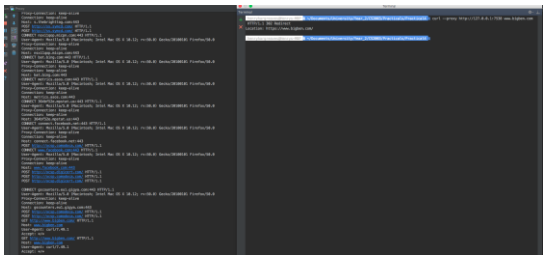


Firefox

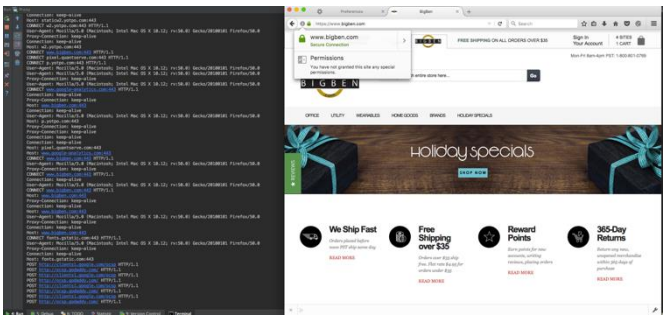


www.bigben.com

Curl

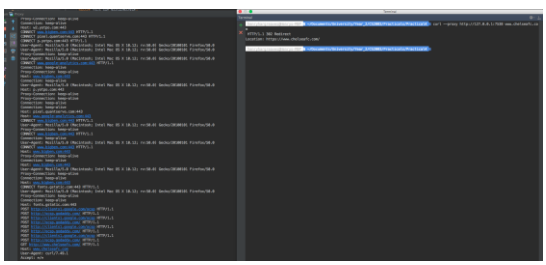


Firefox

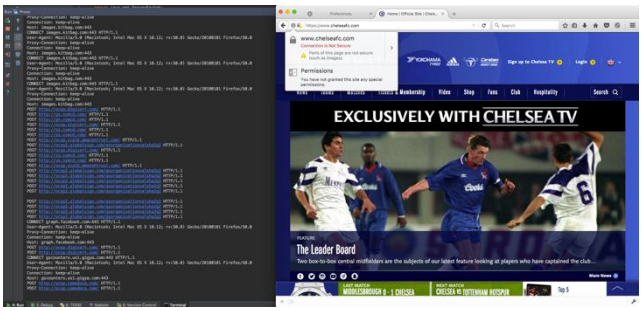


www.chelseafc.com

Curl



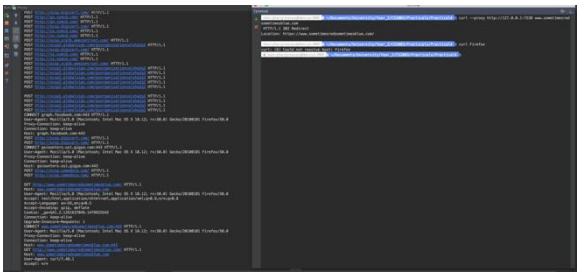
Firefox

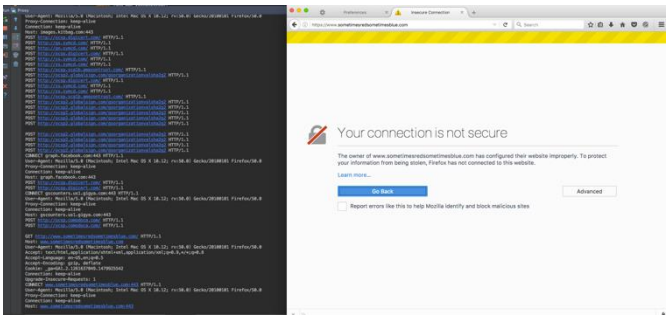


Test Description	Input	Expected Output	Actual Output	Comment
Connect to website that doesn't support https	www.sometimesred-sometimesblue.com www.telegraph.co.uk	The program will receive all the requests and wont processes it	The program received the requests and didn't processes it	The program worked as expected

www.sometimesredsometimesblue.com

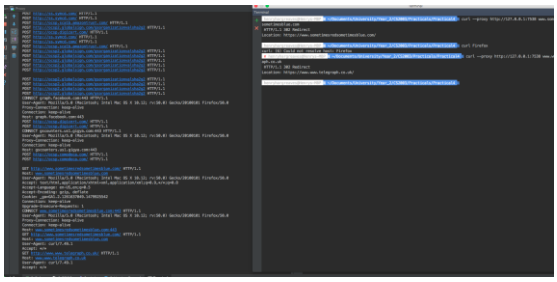
Curl Firefox

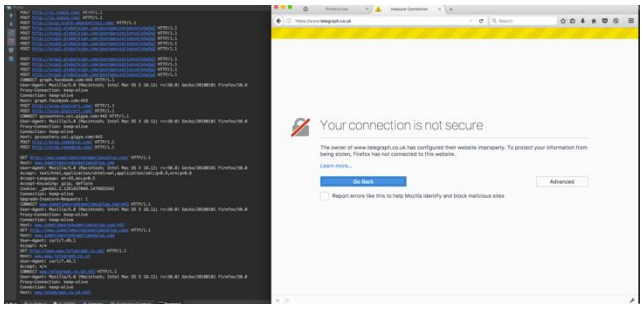




www.telegraph.co.uk

Curl Firefox





Evaluation

Although the screen shots aren't too clear, you can make out that when testing the basic proxy, despite all the requests coming in as HTTP requests, the program has processed this and has converted the URL's to HTTPS and then sent the requests as HTTPS requests where possible, with the HTTPS URL's being printed to the console. If HTTPS is not available, the URL has then been converted back to HTTP and the request send, again with the URL's being printed to the console. This allowed the basic proxy to load all the elements correctly and allow the website to be displayed without any elements missing.

After testing the cache I'm very pleased with the results as you can see there a huge difference from loading a page in from the cache compared to contacting the target server. Moreover, after testing the cache clean up system, it works efficiently and seamlessly.

When testing the extension as you can all the webpages load correctly, moreover with the curl tests, you can see the response from the proxy sends a redirect whenever a HTTP URL is input. Although curl, cannot follow these redirects, Firefox could and displayed the SSL version of the website being requested. From the screen shots, you can see that requests that are processed always come in a HTTPS format with the HTTP request method being set to CONNECT. Furthermore, Firefox correctly shows the connection is encrypted end to end and when a HTTP website is request which doesn't support HTTPS an error message is automatically displayed on Firefox which is a bonus.

Conclusion

In conclusion, I'm very happy with my end program, it can carry out the required tasks with ease and the added functionality makes the proxy very much more a usable program with practical applications. If however I had more time I would have liked to made some sort of caching system for the extension to improve its overall performance.

678h6mlx