CSU24031 Project 1
A Web Proxy Server

Specification

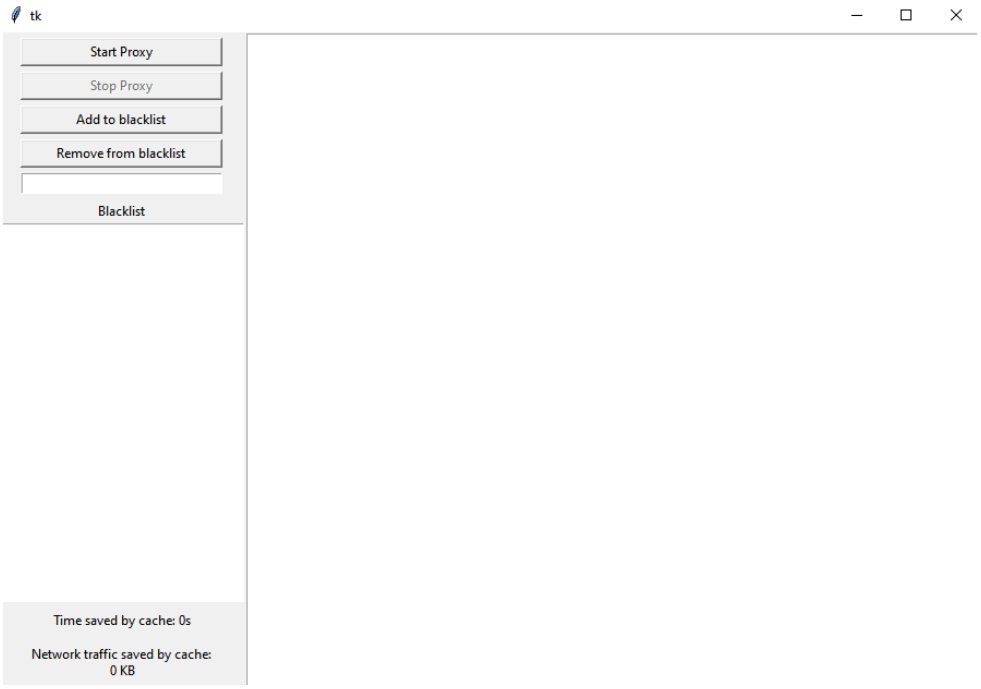
The objective of the exercise is to implement a Web Proxy Server. A Web proxy is a local server, which fetches items from the Web on behalf of a Web client instead of the client fetching them directly. This allows for caching of pages and access control.

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**.

Overview

I built my implementation using Python and the Tkinter library to build a GUI. An example of which can be seen below:



The GUI allows you to start/stop the proxy, blacklist and allow URLs, display each request made to the proxy as well as giving time and network statistics improved by the cache.

The proxy itself broadcasts from the port specified in the config, which I had set to port 12345. The program uses the python socket library to receive and send requests from the client and the server.

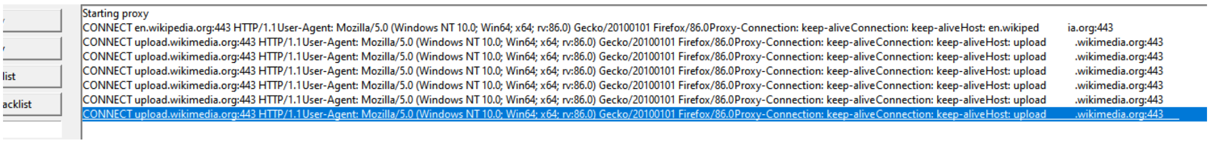
Requests

The proxy server first takes the request from the client and filters it to get the request itself, the destination server's web address and port number. Then depending on the type of request it will do different things.

HTTPS Requests:

These connections are determined by the CONNECT request made by the client. If a CONNECT request is made, the proxy send back a status 200 "OK" message and pipes any further data streams from the client and server directly to each other. As each request is encrypted, we cannot edit or handle the data without breaking the cert. Therefore we cannot manually handle these requests.

These requests are printed to the GUI.



HTTP Requests:

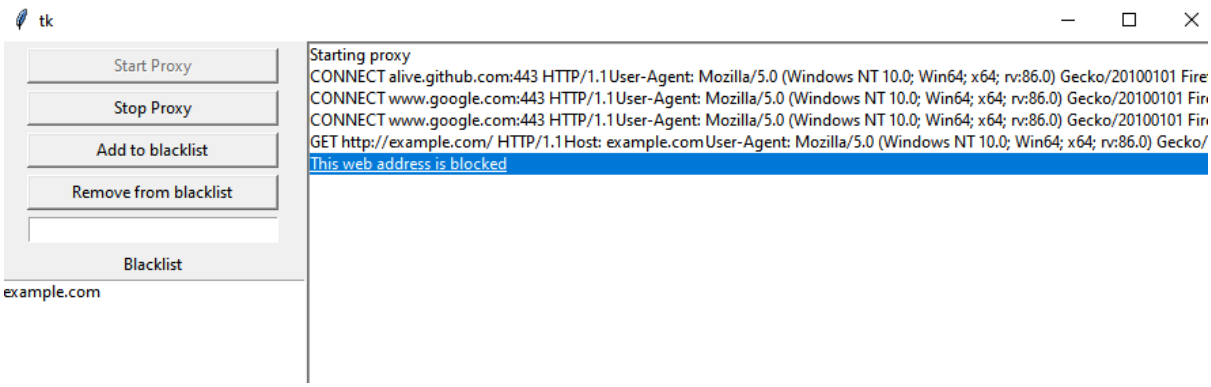
First the HTTP request is parsed to find the destination webserver address and port. The cache is then checked for any prior requests to this address. If the page was already previously cached, the proxy fetches the data from the cache and sends it to the client.

If the request hasn't been cached, the proxy created another socket and connects to the webserver at the given address and port. The proxy then sends the request to the destination server, caches the response, and then sends it to the client.

The proxy also supports WebSocket connections. If a WebSocket connection is requested, the proxy pipes all data streams directly between the client and server, allowing full duplex data streams between the two.

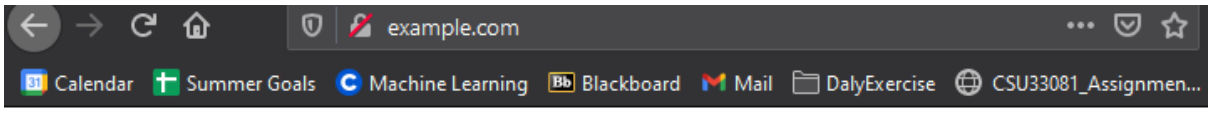
URL Blacklisting

The proxy supports the dynamic blacklisting of URLs. The GUI allows a user to input a URL which blocks all data streams between it and the client.



For HTTPS connections, all data between the two is blocked.

For HTTP the proxy sends back a small HTML file stating that this URL has been blocked by the proxy:



This website is blocked by the proxy

Users can also de-blacklist a URL by selecting it from the blacklist and clicking remove.

The blacklist works by parsing every request and checking the destination webserver vs the blacklist. If the blacklist contains the webserver, the proxy returns stating that this URL is blocked.

Caching

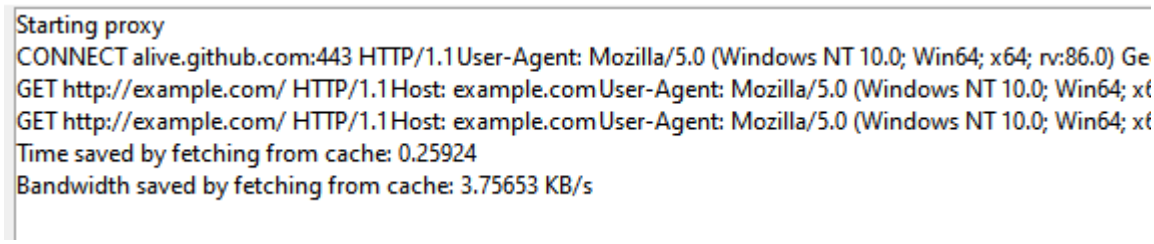
The proxy also supports the caching of HTTP requests. When a webserver is connected to for the first time, any response is saved in a dictionary with the URL as the key. The response is cached along with the time the request took to arrive and the Unix time that it was cached at.

The Unix time is used to check if the cached response it out of data and invalid.

When the website is accessed for the second time, if the cached response is valid, the proxy accesses the cache and sends back the result.

A cached response is invalid if the current time is greater than the time the data was cached plus the TTL stated in the config.

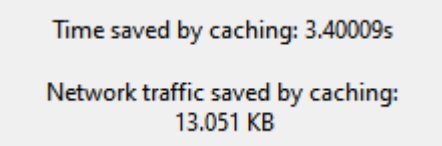
Here is an example gotten by loading the site ‘example.com’ twice.



Here we see that accessing the cache took 0.259 seconds less than directly accessing the server.

The bandwidth saved is calculated by dividing the size of the response payload by the time it was on the network. Here we reduced the network load by 3.76 KB/s.

The proxy also keeps a running total of the total time saved by caching and the total network traffic reduced.



As HTTPS requests are encrypted, no HTTPS request can be cached.

Threading

The python threading library allows you to thread functions.

When the proxy connects to a client, it spins up a new thread to handle the request. This allows the proxy to connect to up to 100 clients at the same time.

With this all the requirements are met.

Code:

GUI.py

```
import tkinter as tk
import threading
from proxy import Proxy


class GUI:

    config = {
        'HOST_NAME': '0.0.0.0',
        'BIND_PORT': 12345,
        'MAX_REQUEST_LENGTH': 8192,
        'CONNECTION_TIMEOUT': 15,
        'TTL': 10080
    }

    def __init__(self):
```

```
        self.proxyStarted = False
        self.m = tk.Tk()
        self.m.geometry('900x600')
        self.currentBlacklistUrl = tk.StringVar()

        self.timeSaved = tk.StringVar()
        self.timeSaved.set('Time saved by cache: 0s')
        self.bandwidthSaved = tk.StringVar()
        self.bandwidthSaved.set('Network traffic saved by cache:\n 0 KB')

        self.proxy = Proxy(self.config, self.timeSaved, self.bandwidthSaved)

        self.leftScreenSide = tk.Frame(self.m)

        self.startProxyButton = tk.Button(
            self.leftScreenSide, text='Start Proxy', width='25', command=lambd
a: self.startProxy(), state=tk.NORMAL)
        self.startProxyButton.pack(side='top', padx='5', pady='5')

        self.stopProxyButton = tk.Button(
            self.leftScreenSide, text='Stop Proxy', width='25', command=lambda
: self.stopProxy(), state=tk.DISABLED)
        self.stopProxyButton.pack(side='top', padx='5')

        self.addToBlacklistButton = tk.Button(
            self.leftScreenSide, text='Add to blacklist', width='25', command=
lambda: self.addToBlacklist())
        self.addToBlacklistButton.pack(side='top', padx='5', pady='5')

        self.removeFromBlacklistButton = tk.Button(
            self.leftScreenSide, text='Remove from blacklist', width='25', com
mand=lambda: self.removeFromBlacklist())
        self.removeFromBlacklistButton.pack(side='top', padx='5')

        self.blacklistEntry = tk.Entry(
            self.leftScreenSide, width='30', textvariable=self.currentBlacklis
tUrl)
        self.blacklistEntry.pack(side='top', padx='5', pady='5')

        self.blacklistText = tk.Label(
            self.leftScreenSide, width='30', text='Blacklist')
        self.blacklistText.pack(side='top', padx='5')

        self.leftScreenSide.pack(side='left', fill=tk.Y)

        self.blacklistBox = tk.Listbox(
            self.leftScreenSide, width=10, selectmode=tk.SINGLE)
```

```
        self.blacklistBox.pack(fill='both', expand=1, side='top')

        self.cacheTimeSavedText = tk.Label(
            self.leftScreenSide, width='30', textvariable=self.timeSaved)
        self.cacheTimeSavedText.pack(side='top', padx='5', pady='5')

        self.cacheBandwidthSavedText = tk.Label(
            self.leftScreenSide, width='30', textvariable=self.bandwidthSaved)
        self.cacheBandwidthSavedText.pack(side='bottom', padx='5', pady='5')

        self.proxyResponseBox = tk.Listbox(self.m, width=10)
        self.proxyResponseBox.pack(fill='both', expand=2, side='right')

        self.m.mainloop()

    def startProxy(self):
        if (not self.proxyStarted):
            self.proxy.initialise()
            self.proxyStarted = True
            self.startProxyButton['state'] = tk.DISABLED
            self.stopProxyButton['state'] = tk.NORMAL
            self.proxyResponseBox.insert(tk.END, 'Starting proxy')
            process = threading.Thread(
                target=lambda: self.proxy.listenForClients(self.proxyResponseB
ox))
            process.start()

    def stopProxy(self):
        if (self.proxyStarted):
            self.proxyStarted = False
            self.startProxyButton['state'] = tk.NORMAL
            self.stopProxyButton['state'] = tk.DISABLED
            self.proxyResponseBox.insert(tk.END, 'Shutting down proxy')
            self.proxy.shutdown()

    def addToBlacklist(self):
        url = self.currentBlacklistUrl.get()
        self.proxy.addToBlacklist(url)
        self.blacklistBox.insert(tk.END, url)
        self.blacklistEntry.delete(0, 'end')

    def removeFromBlacklist(self):

        index = self.blacklistBox.curselection()
        if (index != ()):
            url = self.blacklistBox.get(index[0])
            self.proxy.removeFromBlacklist(url)
            self.blacklistBox.delete(index)
```

```
def main():
    GUI()


if __name__ == '__main__':
    main()
```

Proxy.py

```
import signal
import socket
import threading
import sys
import tkinter as tk
import ssl
import time


class Proxy:

    timeSaved = 0.0
    bandwidthSaved = 0

    blockedHTML = """
    <html>
        <head>
        </head>

        <body>
            <h1>This website is blocked by the proxy</h1>
        </body>

    </html>

    """

    def __init__(self, config, timeText, bandwidthText):
        self.timeText = timeText
        self.bandwidthText = bandwidthText
        self.config = config
        self.blacklist = set()
```

```
        self.cache = {}

    def initialise(self):

        # Shutdown on Ctrl+C
        signal.signal(signal.SIGINT, self.shutdown)

        # Create a TCP socket
        self.serverSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

        # Re-use the socket
        self.serverSocket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1
)

        # Bind socket to a public host, and a port
        self.serverSocket.bind(
            (self.config['HOST_NAME'], self.config['BIND_PORT']))

        self.serverSocket.listen(100)  # Server Socket
        self.__clients = {}

    def listenForClients(self, textBox):
        self.textBox = textBox
        while True:
            # Establish the connection
            (clientSocket, clientAddress) = self.serverSocket.accept()

            d = threading.Thread(name=self.getClientName(clientAddress),
                                 target=self.proxyThread, args=(clientSocket,
clientAddress))

            d.setDaemon(True)
            d.start()
        self.shutdown(0, 0)

    def proxyThread(self, clientSocket, clientAddress):
        # Get the request from the browser
        request = clientSocket.recv(self.config['MAX_REQUEST_LENGTH'])
        startTime = time.time()

        try:
            # Parse the first line
            firstLine = request.decode().split('\n')[0]
            if (firstLine != ''):
                self.textBox.insert(tk.END, request.decode())
                url = firstLine.split(' ')[1]

                # Find pos of ://
```

```
                httpsPos = url.find('://')

                if (httpsPos == -1):
                    temp = url
                else:
                    temp = url[(httpsPos+3):]

                # Find port position
                portPos = temp.find(':')

                # Find end of web server
                webserverPos = temp.find('/')

                if (webserverPos == -1):
                    webserverPos = len(temp)

                webserver = ''

                port = -1
                if (portPos == -1 or webserverPos < portPos):
                    # Default port
                    port = 80
                    webserver = temp[:webserverPos]

                else:
                    # Specific port
                    port = int((temp[(portPos+1):])[:webserverPos-portPos-1])
                    webserver = temp[:portPos]

                if (webserver in self.blacklist):
                    self.textBox.insert(tk.END, "This web address is blocked")
                    clientSocket.sendall(self.blockedHTML.encode())

                    clientSocket.close()
                    return

                cache = self.getFromCache(firstLine)
                if (cache is not None):
                    bandwidth = 0
                    # Sending all the packets from the cache
                    for data in cache[1]:
                        bandwidth += len(data)
                        clientSocket.sendall(data)

                    endTime = time.time()
                    timeElapsed = endTime-startTime
                    currentTimeSaved = cache[2] - timeElapsed
```

```
                    self.timeSaved += currentTimeSaved
                    self.bandwidthSaved += bandwidth

                    self.textBox.insert(
                        tk.END, 'Time saved by fetching from cache: {:.5f}'.fo
rmat(currentTimeSaved))
                    self.textBox.insert(
                        tk.END, 'Bandwidth saved by fetching from cache: {:.5f
} KB/s'.format(
                            bandwidth / (cache[2] * 1024)))

                    self.timeText.set(
                        'Time saved by caching: {:.5f}s'.format(self.timeSaved
))
                    self.bandwidthText.set(
                        'Network traffic saved by caching:\n {:.3f} KB'.format
(self.bandwidthSaved / 1024))
                    clientSocket.close()
                    return

                try:

                    # Connect to destination server
                    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                    s.settimeout(self.config['CONNECTION_TIMEOUT'])

                    if ("CONNECT" in firstLine):

                        s.connect((webserver, port))
                        # Setting up http tunneling reply
                        reply = "HTTP/1.0 200 Connection established\r\n"
                        reply += "Proxy-agent: JohnsProxy\r\n"
                        reply += "\r\n"

                        clientSocket.sendall(reply.encode())

                        # Setting up websocket connection
                        clientSocket.setblocking(0)
                        s.setblocking(0)

                        while True:
                            try:
                                # Recieving new request
                                newRequest = clientSocket.recv(
                                    self.config['MAX_REQUEST_LENGTH'])
                                # Sending request to the server
                                s.sendall(newRequest)
                            except socket.error as err:
```

```
                                pass
                            try:
                                # Recieving reply from the server
                                reply = s.recv(
                                    self.config['MAX_REQUEST_LENGTH'])

                                # Sending reply to the client
                                clientSocket.sendall(reply)
                            except socket.error as err:
                                pass

                    else:
                        s.connect((webserver, port))

                        clientSocket.setblocking(0)
                        s.setblocking(0)

                        s.sendall(request)

                        cache = []

                        while True:
                            try:
                                # Recieve data from web serer
                                data = s.recv(
                                    self.config['MAX_REQUEST_LENGTH'])
                                # Checking if there is more data to come
                                if (len(data) == self.config['MAX_REQUEST_LENG
TH']):
                                    cache.append(data)
                                    clientSocket.send(data)
                                # Otherwise cache the response
                                elif (len(data) > 0):
                                    cache.append(data)
                                    clientSocket.send(data)
                                    endTime = time.time()
                                    # Cache the response along with the time i
t took to fetch and the url
                                    self.addToCache(
                                        firstLine, cache, (endTime - startTime
))
                                else:

                                    break
                            except:
                                pass

                        s.close()
```

```
                clientSocket.close()
            except socket.error as errorMsg:
                print('ERROR: ' + str(errorMsg))
                if s:
                    s.close()
                if clientSocket:
                    clientSocket.close()
        else:
            clientSocket.close()
    except:
        pass

def shutdown(self):
    # Close existing server
    self.serverSocket.close()

def getClientName(self, clientAddress):
    return str(clientAddress)

def addToBlacklist(self, url):
    self.blacklist.add(url)

def removeFromBlacklist(self, url):
    self.blacklist.remove(url)

def getFromCache(self, request):
    # If the request is HTTPS, there is no cached response
    if ('CONNECT' in request):
        return None

    # Check if the URL is already cached
    cached = self.cache.get(request)
    #
    if (cached is not None):
        # Check if the cached response has expired
        if (cached[0] + self.config['TTL'] > int(time.time())):
            return cached
    return None

def addToCache(self, request, data, requestTime):
    try:
        self.cache[request] = (
            int(time.time()),
            data,
            requestTime
        )
    except:
        print('something went wrong')
```