CS3031: Project I

Conor McCauley - 17323203

February 24, 2020

1 Implementation

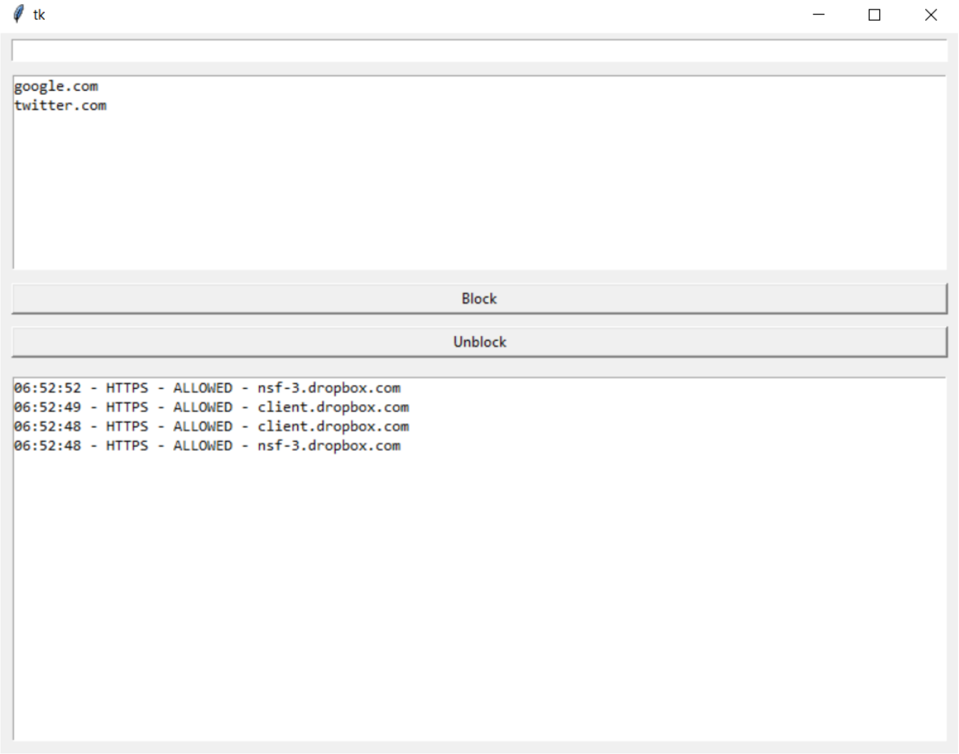
Upon being started, the web proxy server starts a *Tkinter* thread to generate and maintain the graphical user interface (GUI). The server then begins listening to port 8080 for requests. Whenever a request is received a new thread is created to handle it. This allows the server to handle multiple concurrent requests.

The server reads the raw request data into a *ProxyRequest* object which extracts the URL, port, method and other important data from the request. The server ensures that the request is not being sent to a blacklisted URL and then passes the request on to the appropriate handler method: HTTP or HTTPS.

The handler methods relay data between the client and remote server as needed. The HTTPS handler maintains a bidirectional *WebSocket* connection. The HTTP handler utilises caching to increase the efficiency of repeated requests.

1.1 Graphical User Interface

The web proxy server uses Python's *Tkinter* library to implement the GUI. It allows the user to block or unblock specific URLs while the application is running. It also displays a detailed log of all the requests that were made.

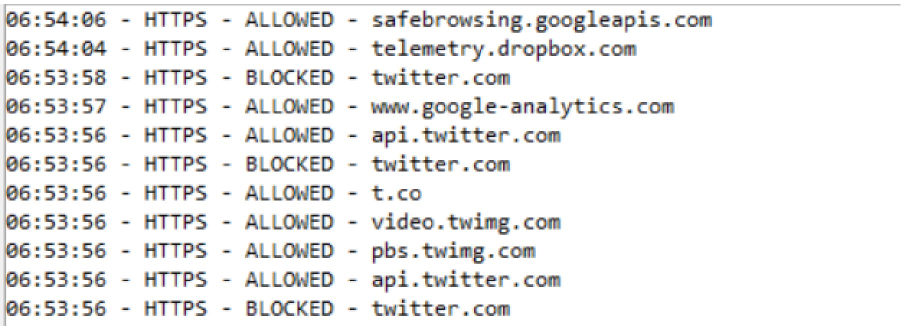


```
google.com
twitter.com
```

```
06:52:52 - HTTPS - ALLOWED - nsf-3.dropbox.com
06:52:49 - HTTPS - ALLOWED - client.dropbox.com
06:52:48 - HTTPS - ALLOWED - client.dropbox.com
06:52:48 - HTTPS - ALLOWED - nsf-3.dropbox.com
```

1.2 URL Blocking

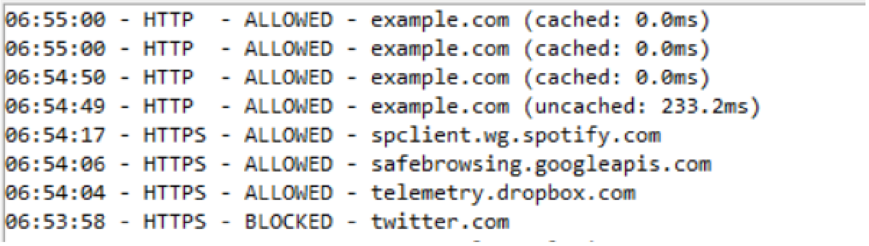
Whenever a request is made the destination URL is checked against the user's blacklist - if the URL is contained in the list the request is stopped.



```
06:54:06 - HTTPS - ALLOWED - safebrowsing.googleapis.com
06:54:04 - HTTPS - ALLOWED - telemetry.dropbox.com
06:53:58 - HTTPS - BLOCKED - twitter.com
06:53:57 - HTTPS - ALLOWED - www.google-analytics.com
06:53:56 - HTTPS - ALLOWED - api.twitter.com
06:53:56 - HTTPS - BLOCKED - twitter.com
06:53:56 - HTTPS - ALLOWED - t.co
06:53:56 - HTTPS - ALLOWED - video.twimg.com
06:53:56 - HTTPS - ALLOWED - pbs.twimg.com
06:53:56 - HTTPS - ALLOWED - api.twitter.com
06:53:56 - HTTPS - BLOCKED - twitter.com
```

1.3 Response Caching

The responses of all HTTP requests are stored in a dictionary and if a subsequent request is made to the same URL the stored response is immediately returned.



```
06:55:00 - HTTP  - ALLOWED - example.com (cached: 0.0ms)
06:55:00 - HTTP  - ALLOWED - example.com (cached: 0.0ms)
06:54:50 - HTTP  - ALLOWED - example.com (cached: 0.0ms)
06:54:49 - HTTP  - ALLOWED - example.com (uncached: 233.2ms)
06:54:17 - HTTPS - ALLOWED - spclient.wg.spotify.com
06:54:06 - HTTPS - ALLOWED - safebrowsing.googleapis.com
06:54:04 - HTTPS - ALLOWED - telemetry.dropbox.com
06:53:58 - HTTPS - BLOCKED - twitter.com
```

2 Code Listing

```
# Imports
import socket
import tkinter
from datetime import datetime
from enum import Enum
from queue import Queue
from threading import Thread
from tkinter import *

# Constants
SOCKET_PORT = 8080          # the port to bind the socket to
SOCKET_BACKLOG = 100        # the max length of the sockets
    backlog
DATA_LENGTH = 16384         # the max amount of data to receive
     at once
HTTP_PORT = 80              # the port to bind HTTP requests to
HTTPS_PORT = 443            # the port to bind HTTPS requests
    to
HTTPS_CONNECT = ”HTTP/1.1␣200␣Connection␣established\r\n\r\
    n”  # the response message for HTTPS connections

# Variables
blacklist = set()           # list of blocked URLs
cache = {}                  # dictionary of cached responses
```

```
req_queue = Queue()      # queue containing requests that
   need to be displayed


# Entry method
def main():

    # Start a thread for the UI
    Thread(target=init_ui, args=()).start()

    # Bind a socket to the port and start listening
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM
       )
    sock.bind(('', SOCKET_PORT))
    sock.listen(SOCKET_BACKLOG)

    # Continuously accept requests and handle them in new
       threads
    try:
        while True:
            client_conn, client_addr = sock.accept()
            Thread(target=request_thread, args=(client_conn
               , )).start()
    finally:
        sock.close()  # dispose of the socket if the
           program exits


# Set up the UI
def init_ui():

    ui = tkinter.Tk()
    ui.geometry("800x600")

    input_box = Entry(ui)
    input_box.pack(fill=X, padx=10, pady=5)
    block_list = Listbox(ui, font=("Consolas", 10))
    block_list.pack(fill=X, padx=10, pady=5)

    def block():
        url = input_box.get()
        if url not in blacklist:
            blacklist.add(url)
            block_list.insert(END, url)
        input_box.delete(0, END)

    def unblock():
        i = block_list.curselection()[0]
        url = block_list.get(i)
        block_list.delete(i)
        if url in blacklist:
            blacklist.remove(url)

    block_button = Button(ui, text=”Block”, command=block)
    block_button.pack(fill=X, padx=10, pady=5)

    unblock_button = Button(ui, text=”Unblock”, command=
       unblock)
    unblock_button.pack(fill=X, padx=10, pady=5)

    connection_box = Listbox(ui, font=(”Consolas”, 10))
    connection_box.pack(fill=BOTH, padx=10, pady=10, expand
       =1)

    while True:

        # Add all items in the display queue to the
           connection box
        while not req_queue.empty():
            connection_box.insert(0, req_queue.get())

        ui.update_idletasks()
        ui.update()


# Log a request to the display queue
def log_request(request):
    req_queue.put(str(request))
```

```
# Handle a request
def request_thread(client_conn):

    data = client_conn.recv(DATA_LENGTH)
    request = ProxyRequest(data)

    request.is_blocked = request.url in blacklist

    if request.is_blocked:
        log_request(request)  # if the request is blocked just log it and exit
    else:
        if request.request_type == ProxyRequestType.Http:
            handle_http_request(client_conn, request)
            log_request(request)  # log the request after handling it if it's HTTP
        else:
            log_request(request)  # log the request before handling it if it's HTTPS
            handle_https_request(client_conn, request)


# Handle a HTTP request
def handle_http_request(client_conn, request):

    request.is_cached = request.url in cache

    if request.is_cached:
        client_conn.sendall(cache[request.url])  # forward cached response to client
    else:

        server_conn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        server_conn.connect((request.url, request.port))
        server_conn.send(request.data)
        server_conn.settimeout(1)  # set a timeout otherwise the request will hang indefinitely

        response = bytearray()
        # Receive the entire response and catch the
           subsequent timeout
        try:
            while True:
                segment = server_conn.recv(DATA_LENGTH)
                if len(segment):
                    client_conn.send(segment)  # pass
                       response on to the client
                    response.extend(segment)
                else:
                    break
        except socket.error:
            pass

        server_conn.close()
        client_conn.close()

        cache[request.url] = response  # cache the response

    request.end()  # stop the timer


# Handle a HTTPS request
def handle_https_request(client_conn, request):

    server_conn = socket.socket(socket.AF_INET, socket.
       SOCK_STREAM)
    server_conn.connect((request.url, request.port))
    client_conn.sendall(HTTPS_CONNECT.encode())  # send the
        HTTPS connection message

    # Disable blocking
    server_conn.setblocking(False)
    client_conn.setblocking(False)

    # Continuously relay data between the client and the
       server
    while True:
        # Client to server
        try:
            data = client_conn.recv(DATA_LENGTH)
            server_conn.sendall(data)
        except socket.error:
            pass
        # Server to client
        try:
            data = server_conn.recv(DATA_LENGTH)
            client_conn.sendall(data)
        except socket.error:
            pass


# Basic request object (url, port, data, etc.)
class ProxyRequest:

    def __init__(self, data):

        self.data = data
        data_string = str(data)

        self.is_blocked = False
        self.is_cached = False

        line = data_string.splitlines()[0]
        method = line.split('␣')[0]
        raw_url = line.split('␣')[1]

        # The request is HTTPS if he method is 'CONNECT'
        self.request_type = ProxyRequestType.Https if "CONNECT" in method.upper() else ProxyRequestType.Http

        # Parse the URL
        self.url, self.port = self.parse_url(raw_url)

        # Initialise the timestamps
        self.start_date = datetime.now().strftime("%H:%M:%S")
        self.start_timestamp = datetime.now().microsecond
        self.end_timestamp = self.start_timestamp

    def __str__(self):

        req_type = "HTTP␣" if self.request_type ==
           ProxyRequestType.Http else "HTTPS"
        blocked = "BLOCKED" if self.is_blocked else "
           ALLOWED"
        duration = round(abs(self.end_timestamp - self.
           start_timestamp) / 1000, 2)
        cache_text = f"(cached:␣{duration}ms)" if self.
           is_cached else f"(uncached:␣{duration}ms)"

        if self.request_type == ProxyRequestType.Http and
           not self.is_blocked:
            return f"{self.start_date}␣-␣{req_type}␣-␣{
               blocked}␣-␣{self.url}␣{cache_text}"
        else:
            return f"{self.start_date}␣-␣{req_type}␣-␣{
               blocked}␣-␣{self.url}"

    def end(self):
        self.end_timestamp = datetime.now().microsecond

    # Extract the proper URL and port from the raw URL
       string
    def parse_url(self, url):

        if url.startswith("https://"):
            url = url[8:]
        elif url.startswith("http://"):
            url = url[7:]

        port_start_i = url.find(":")
        port_end_i = url.find("/")
        if port_end_i == -1:
            port_end_i = len(url)

        if port_start_i == -1 or port_end_i < port_start_i:
            port = 80
            url = url[:port_end_i]
        else:
            start_i, end_i = port_start_i + 1, port_end_i -
                port_start_i - 1
            port = int(url[start_i:][:end_i])
            url = url[:port_start_i]

        return url, port


class ProxyRequestType(Enum):
    Http = 1
    Https = 2


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