CS3031 Advanced Telecommunications
Project I

Séamus Woods
15317173

19/02/2019

0.1 Specification

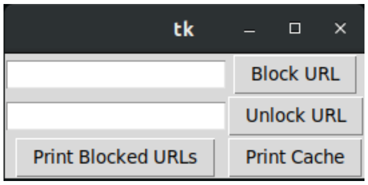
The objective of the excercise is to implement a Web Proxy Server. A Web proxy is a local server, which fetches items from the Web on behalf of a Web client instead of the client fetching them directly. This allows for caching of pages and access control.

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 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.

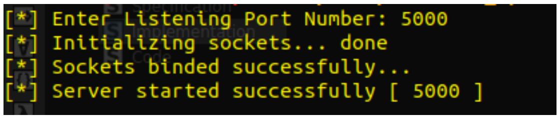
0.2 Implementation

The easiest way to explain my design and implementation is just by talking through the execution of the code below. I have successfully managed to implement all features listed above.
When the program is run, it will first start a thread on the tkinter function. Tkinter is what I used to dynamically block selected URLs(3).



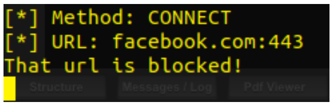
As seen from the image, the user can dynamically enter URLs to be blocked, unblocked, can view the currently blocked URLs and print the URLs we currently have cached. This can all be done as the proxy is running(3).

The program then prompts the user to enter a port to have the proxy listen on. Once the user enters the port we initiate a socket, bind it to the port and start listening for incoming connections.



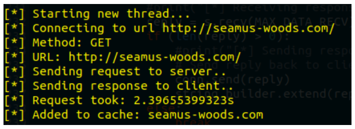
```
[*] Enter Listening Port Number: 5000
[*] Initializing sockets... done
[*] Sockets binded successfully...
[*] Server started successfully [ 5000 ]
```

Any connections we receive, we start a new thread on the proxy_thread function with that specific connection, allowing multiple connections(5).
In the proxy_thread function we begin to parse the clients request. After some parsing to get the webserver, port, method etc, we check if we already have this page cached. If we do, we simply send the cached response the the client and we're done, else we call the proxy_server function. (All of these requests are being timed and compared as part of the specification for caching(4)).
The first thing we do in our proxy_server function is check our blocked url dictionary. If the webserver the client is trying to access is currently blocked, we simply tell them that that url is blocked and close the connection(3).

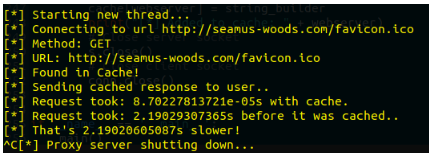


```
[*] Method: CONNECT
[*] URL: facebook.com:443
That url is blocked!
```

If the url is not blocked, we need to see if we are working with HTTP or HTTPS(1). If we are working with HTTPS the method should be CONNECT, and so we send the webserver a connection established. We then set up websocket connection(2) with the client and the webserver, which will persist until either of them end it. If we are working with HTTP, we don't need to worry about the CONNECT request, and just set up a websocket connection(2). As soon as we get the complete request, we store it in the cache, since we know it isn't there yet.



```
[*] Starting new thread...
[*] Connecting to url http://seamus-woods.com/
[*] Method: GET
[*] URL: http://seamus-woods.com/
[*] Sending request to server..
[*] Sending response to client..
[*] Request took: 2.39655399323s
[*] Added to cache: seamus-woods.com
```



```
[*] Starting new thread...
[*] Connecting to url http://seamus-woods.com/favicon.ico
[*] Method: GET
[*] URL: http://seamus-woods.com/favicon.ico
[*] Found in Cache!
[*] Sending cached response to user..
[*] Request took: 8.70227813721e-05s with cache.
[*] Request took: 2.19029307365s before it was cached..
[*] That's 2.19020605087s slower!
^C[*] Proxy server shutting down...
```

The code can be seen below.

0.3 Code

```
#! /usr/bin/env python
import os, sys, thread, socket, time
import Tkinter as tk
from Tkinter import *

# CONSTANTS
# How many pending connection will the queue hold?
BACKLOG = 200
# Max number of bytes to receive at once?
MAX_DATA_RECV = 4096
# Set true if you want to see debug messages.
DEBUG = True
# Dict to store the blocked URLs
blocked = {}
# Dict to act as a cache, stores responses.
cache = {}
# Dict to store time of response before caching.
timings = {}

# Tkinter function.. Used to dynamicall block URLs.
# Also used to display the current blocked URLs and the cache.
def tkinter():
        # Create block and unblock entries..
        console = tk.Tk()
        block = Entry(console)
        block.grid(row=0,column=0)
        unblock = Entry(console)
        unblock.grid(row=1, column=0)

        # Function for blocking urls.. basically take whats in the entry cell and put it into
        # the dict..
        def block_url():
                ret = block.get()
                temp = blocked.get(ret)
                if temp is None:
                        blocked[ret] = 1
                        print("[*] Successfully blocked: " + ret)
                else:
                        print("[*] This website is already blocked..")
        # Creating a button to call the block_url function..
        block_button = Button(console, text="Block URL", command=block_url)
        block_button.grid(row=0, column=1)

        # Function for unblocking urls.. basically tkaes whats in the entry cell and removes it
        # from the blocked dict if it exists..
        def unblock_url():
                ret = unblock.get()
                temp = blocked.get(ret)
                if temp is None:
                        print("[*] Url is not blocked: " + ret)
                else:
                        blocked.pop(ret)
                        print("[*] Successfully unblocked: " + ret)
        # Creating a button to call the unblock_url function..
        unblock_button = Button(console, text="Unlock URL", command=unblock_url)
        unblock_button.grid(row=1, column=1)

        # Function to print all currently blocked urls..
        def print_blocked():
                print(blocked)
        print_blocked = Button(console, text="Print Blocked URLs", command=print_blocked)
        print_blocked.grid(row=3, column=0)

        # Function to print all currently cached pages..
        def print_cache():
                for key, value in cache.iteritems():
                        print key
        print_blocked = Button(console, text="Print Cache", command=print_cache)
        print_blocked.grid(row=3, column=1)

        # Could add other functionality here :D

        mainloop()

# MAIN PROGRAM
def main():
        # Run a thread of our tkinter function..
        thread.start_new_thread(tkinter,())

```

```
    try:
        # Ask user what port they'd like to run the proxy on..
        listening_port = int(raw_input("[*] Enter Listening Port Number: "))
    except KeyboardInterrupt:
        # Handling keyboard interrupt.. looks nicer..
        print("\n[*] User Requested An Interrupt")
        print("[*] Application Exiting...")
        sys.exit()
    try:
        # Ininitiate socket
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        # Bind socket for listen
        s.bind(('', listening_port))
        # Start listening for incoming connections
        s.listen(BACKLOG)
        print("[*] Initializing sockets... done")
        print("[*] Sockets binded successfully...")
        print("[*] Server started successfully [ %d ]\n" % (listening_port))
    except Exception, e:
        print("[*] Unable to initalize socket...")
        sys.exit(2)

    while True:
        try:
            # Accept connection from client browser
            conn, client_addr = s.accept()
            # Receive client data
            data = conn.recv(MAX_DATA_RECV)
            # Start a thread
            thread.start_new_thread(proxy_thread, (conn, data, client_addr))
        except KeyboardInterrupt:
            s.close()
            print("[*] Proxy server shutting down...")
            sys.exit(1)
    s.close()



def proxy_thread(conn, data, client_addr):
    print("")
    print("[*] Starting new thread...")
    try:
        # Parsing the request..
        first_line = data.split('\n')[0]
        url = first_line.split(' ')[1]
        method = first_line.split(' ')[0]
        print("[*] Connecting to url " + url)
        print("[*] Method: " + method)
        if (DEBUG):
            print("[*] URL: " + url)

        # Find pos of ://
        http_pos = url.find("://")
        if (http_pos == -1):
            temp = url
        else:
            # Rest of url..
            temp = url[(http_pos+3):]
        # Finding port position if there is one..
        port_pos = temp.find(":")

        # Find end of web server
        webserver_pos = temp.find("/")
        if webserver_pos == -1:
            webserver_pos = len(temp)

        webserver = ""
        port = -1
        # Default port..
        if (port_pos == -1 or webserver_pos < port_pos):
            port = 80
            webserver = temp[:webserver_pos]
        # Specific port..
        else:
            port = int((temp[(port_pos+1):])[:webserver_pos-port_pos-1])
            webserver = temp[:port_pos]

        # Checking if we already have the response in our cache..
        t0 = time.time()
        x = cache.get(webserver)
        if x is not None:
            # If we do, don't bother with
```

```
                    proxy_server function and send the response on..
            print("[*] Found in Cache!")
            print("[*] Sending cached response to user..")
            conn.sendall(x)
            t1 = time.time()
            print("[*] Request took: " + str(t1-t0) + "s with cache.")
            print("[*] Request took: " + str(timings[webserver]) + "s before it was cached..")
            print("[*] That's " + str(timings[webserver]-(t1-t0)) + "s slower!")
        else:
            # If we don't, continue..
            proxy_server(webserver, port, conn, client_addr, data, method)
    except Exception, e:
        pass


def proxy_server(webserver, port, conn, client_addr, data, method):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # Initiating socket..

    # Checking our blocked dict to check if the URL the user is trying to connect to
    # is blocked..
    for key, value in blocked.iteritems():
        if key in webserver and value is 1:
            print("That url is blocked!")
            conn.close()
            return

    # If the method is CONNECT, we know this is HTTPS.
    if method == "CONNECT":
        try:
            # Connect to the webserver..
            s.connect((webserver, port))
            reply = "HTTP/1.0 200 Connection established\r\n"
            reply += "Proxy-agent: Pyx\r\n"
```

```
                reply += "\r\n"
                print("[*] Sending connection established to server..")
                conn.sendall(reply.encode())
            except socket.error as err:
                print(err)
                return
            conn.setblocking(0)
            s.setblocking(0)
            # Bidirectional messages here.. (Websocket connection)
            print("[*] Websocket connection set up..")
            while True:
                try:
                    #print("[*] Receiving request from client..")
                    request = conn.recv(MAX_DATA_RECV)
                    #print("[*] Sending request to server..")
                    s.sendall(request)
                except socket.error as err:
                    pass
                try:
                    #print("[*] Receiving reply from server..")
                    reply = s.recv(MAX_DATA_RECV)
                    #print("[*] Sending reply to client..")
                    conn.sendall(reply)
                except socket.error as err:
                    pass
            print("[*] Sending response to client..")
        # Else we know this is HTTP.
        else:
            # String builder to build response for our cache.
            t0 = time.time()
            string_builder = bytearray("", 'utf-8')
            s.connect((webserver, port))
            print("[*] Sending request to server..")
            s.send(data)
            s.settimeout(2)
            try:
                while True:
```

```
                #print("[*] Receiving response from server..")
                reply = s.recv(MAX_DATA_RECV)
                if (len(reply) > 0):
                    #print("[*] Sending response to client..")
                    # Send reply back to client
                    conn.send(reply)
                    string_builder.extend(reply)
                else:
                    break
        except socket.error:
            pass
        print("[*] Sending response to client..")
        t1 = time.time()
        print("[*] Request took: " + str(t1-t0) + "s")
        timings[webserver] = t1-t0
        # After response is complete, we can store this in cache.
        cache[webserver] = string_builder
        print("[*] Added to cache: " + webserver)
        # Close server socket
        s.close()
        # Close client socket
        conn.close()


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