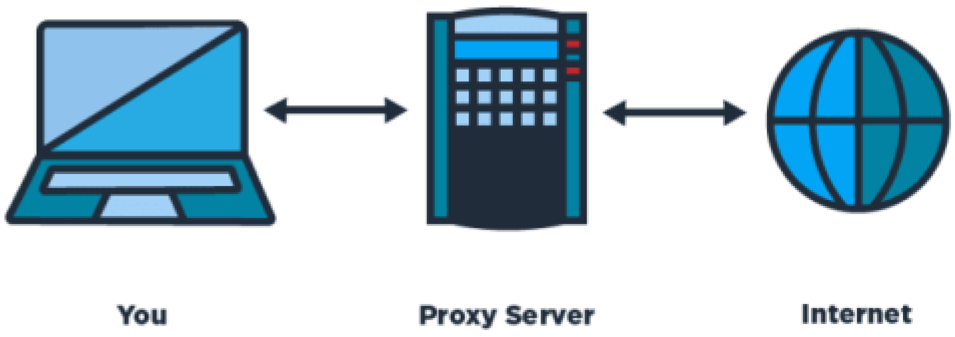


Proxy Server

A PROXY IMPLEMENTED IN PYTHON USING WEBSOCKETS

Rvail Naveed - 17321983 | Advanced Telecommunications | 24-02-2020

https://github.com/rvailnaveed/proxy-server

Introduction

The goal of this project was to implement a 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. The proxy should be able to handle HTTP as well as HTTPS requests, cache pages for faster access and offer dynamic blocking of URL's as a form of access control.

The technologies I used to accomplish this were:

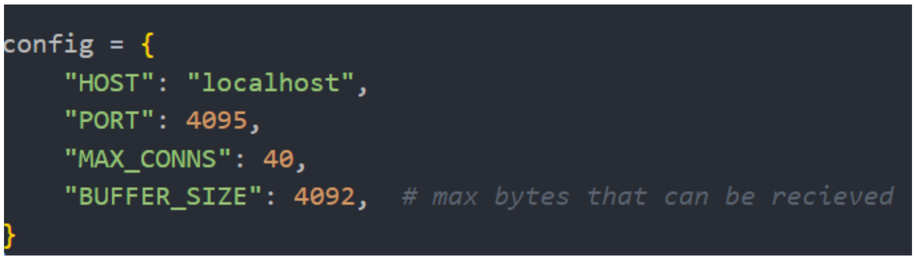
- **Python 3.7**
- The standard library's **sockets** module

Approach

I decided to use python as it is well supported and very easy to program in. You can have a working prototype in a very short amount of time.

Configuration

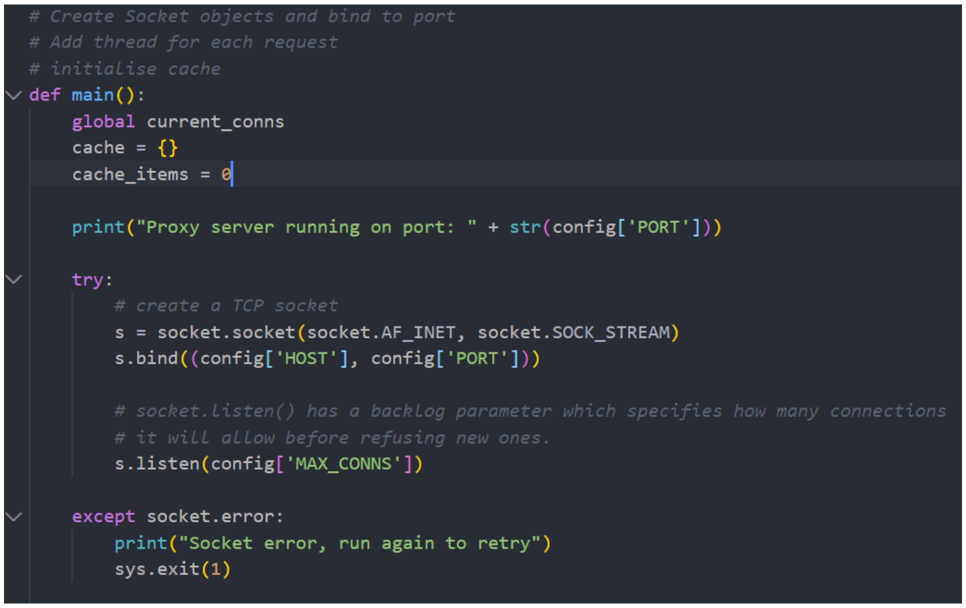
I made a configuration dictionary object which can hold the designated port number and the host. This is to configure the socket for later use to specify what port it should bind to and the max bytes that can be passed to and from the client/server. It also specifies a "**MAX_CONNS**" parameter which can be passed to the sockets *listen()* function as a backlog. This is the number of clients that the socket will accept before blocking and waiting for previous clients before accepting new ones.



```
config = {
    "HOST": "localhost",
    "PORT": 4095,
    "MAX_CONNS": 40,
    "BUFFER_SIZE": 4092,   # max bytes that can be recieved
}
```

Server

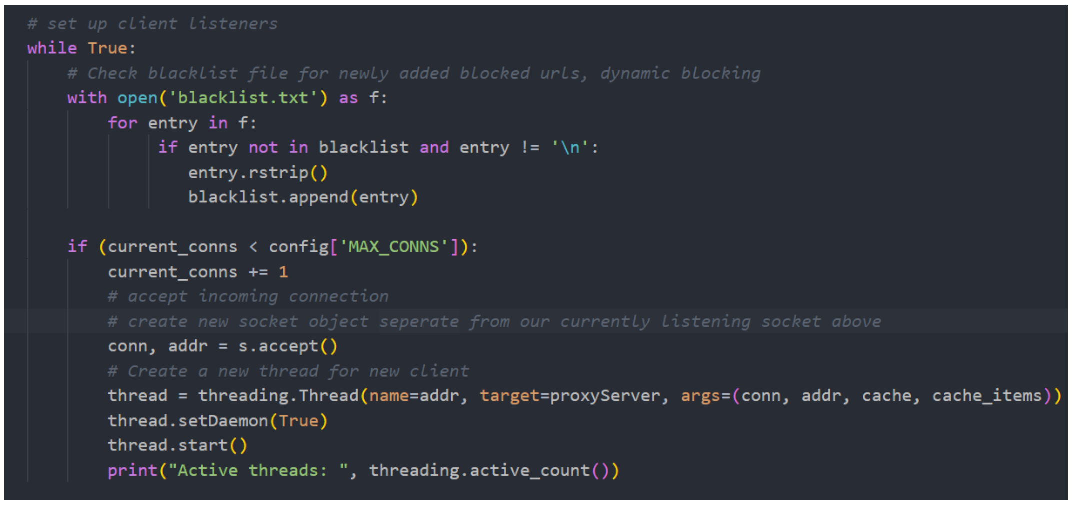
Using pythons socket module from the standard library, I set up a TCP socket that would listen to incoming connections to the socket and respond appropriately to them.



```python
# Create Socket objects and bind to port
# Add thread for each request
# initialise cache
def main():
    global current_conns
    cache = {}
    cache_items = 0

    print("Proxy server running on port: " + str(config['PORT']))

    try:
        # create a TCP socket
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.bind((config['HOST'], config['PORT']))

        # socket.listen() has a backlog parameter which specifies how many connections
        # it will allow before refusing new ones.
        s.listen(config['MAX_CONNS'])

    except socket.error:
        print("Socket error, run again to retry")
        sys.exit(1)
```

Threading

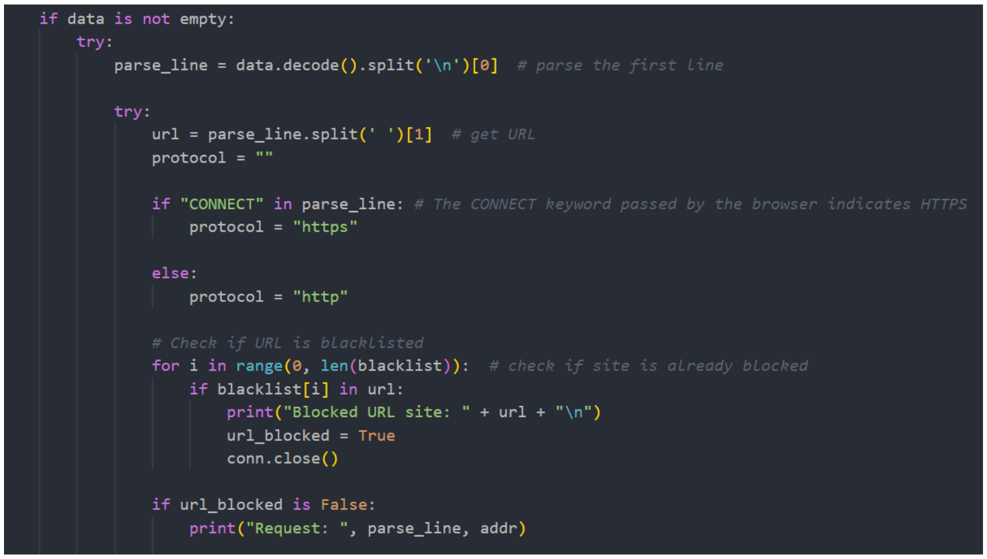
For each new request I created a new thread and passed it the proxyServer function. There can be as many threads as MAX_CONNS.



```python
# set up client listeners
while True:
    # Check blacklist file for newly added blocked urls, dynamic blocking
    with open('blacklist.txt') as f:
        for entry in f:
            if entry not in blacklist and entry != '\n':
                entry.rstrip()
                blacklist.append(entry)

    if (current_conns < config['MAX_CONNS']):
        current_conns += 1
        # accept incoming connection
        # create new socket object seperate from our currently listening socket above
        conn, addr = s.accept()
        # Create a new thread for new client
        thread = threading.Thread(name=addr, target=proxyServer, args=(conn, addr, cache, cache_items))
        thread.setDaemon(True)
        thread.start()
        print("Active threads: ", threading.active_count())
```

Handling requests

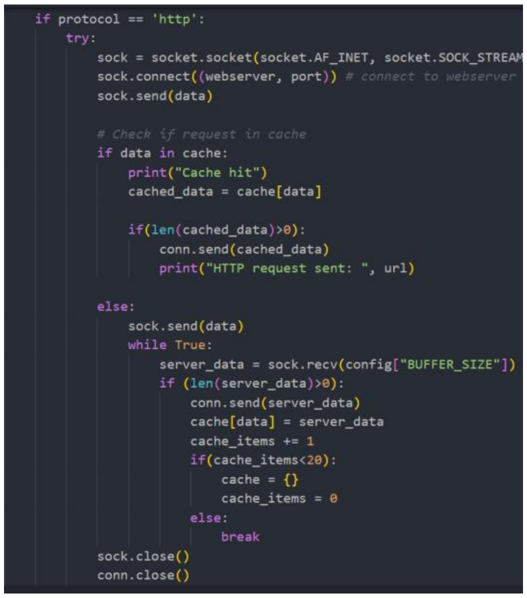
The data retrieved from the request is parsed to find if the protocol used by the website is either http or https. HTTPS is determined by it containing the “CONNECT” keyword. The URL is then checked to see if it is in the list of blacklisted URL’s, if so the connection between the client and webserver is closed.



```
if data is not empty:
    try:
        parse_line = data.decode().split('\n')[0]  # parse the first line

        try:
            url = parse_line.split(' ')[1]  # get URL
            protocol = ""

            if "CONNECT" in parse_line: # The CONNECT keyword passed by the browser indicates HTTPS
                protocol = "https"

            else:
                protocol = "http"

            # Check if URL is blacklisted
            for i in range(0, len(blacklist)):  # check if site is already blocked
                if blacklist[i] in url:
                    print("Blocked URL site: " + url + "\n")
                    url_blocked = True
                    conn.close()

            if url_blocked is False:
                print("Request: ", parse_line, addr)
```

HTTP

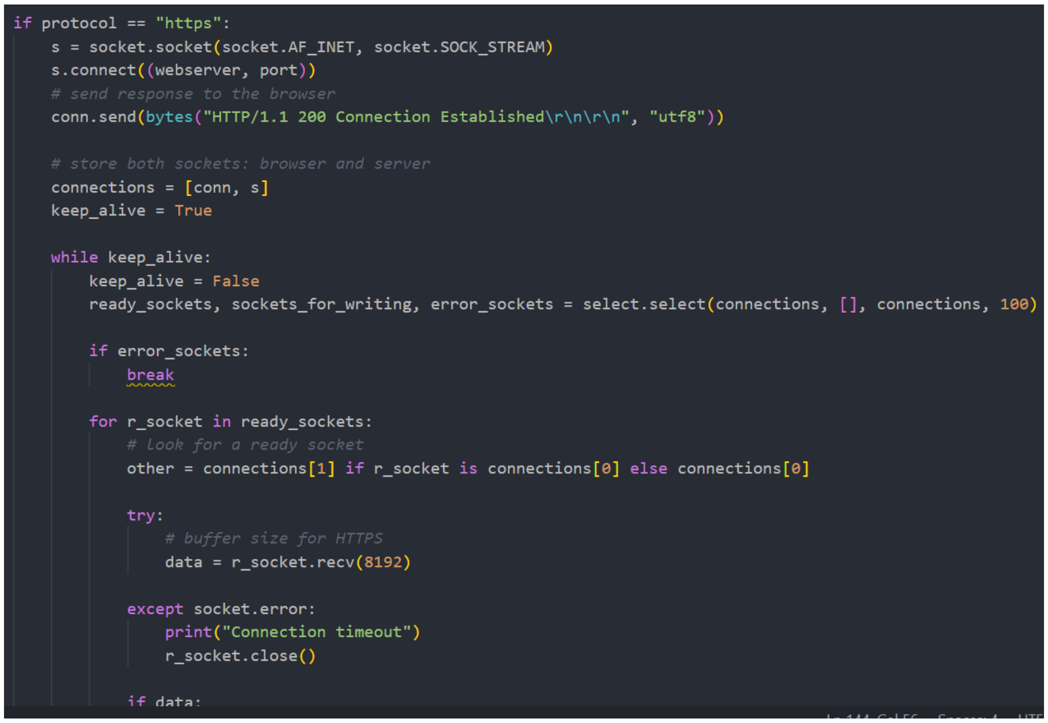
If the protocol is http, then a new socket object is created that connects to the webserver. Then the data being requested is checked against the cache to see if it is present. If so, then it is served from cache, otherwise the request is forwarded to the server by the proxy.



```
if protocol == 'http':
    try:
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM
        sock.connect((webserver, port)) # connect to webserver
        sock.send(data)

        # Check if request in cache
        if data in cache:
            print("Cache hit")
            cached_data = cache[data]

            if(len(cached_data)>0):
                conn.send(cached_data)
                print("HTTP request sent: ", url)

        else:
            sock.send(data)
            while True:
                server_data = sock.recv(config["BUFFER_SIZE"])
                if (len(server_data)>0):
                    conn.send(server_data)
                    cache[data] = server_data
                    cache_items += 1
                    if(cache_items<20):
                        cache = {}
                        cache_items = 0
                else:
                    break
        sock.close()
        conn.close()
```

HTTPS

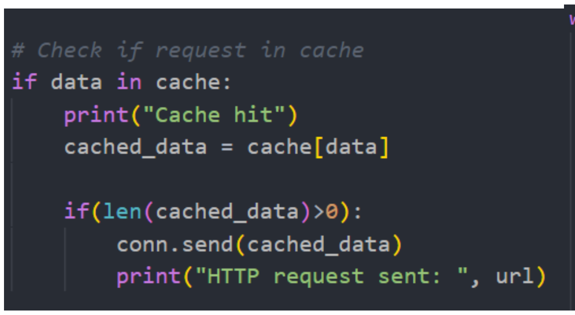
HTTPS requests are handled a bit differently. The connection must be kept alive between the browser and the server. The server will then perform a TLS handshake with the site being requested and the request will be processed. The buffer size is also increased for HTTPS sites,



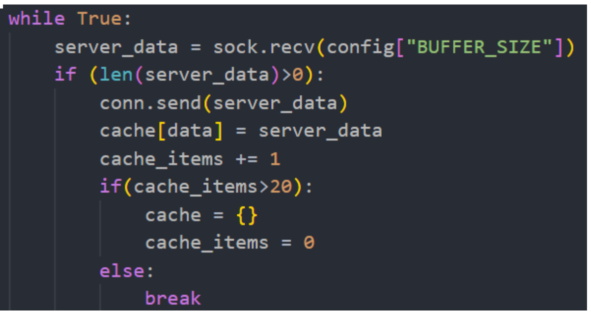
```python
if protocol == "https":
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((webserver, port))
    # send response to the browser
    conn.send(bytes("HTTP/1.1 200 Connection Established\r\n\r\n", "utf8"))

    # store both sockets: browser and server
    connections = [conn, s]
    keep_alive = True

    while keep_alive:
        keep_alive = False
        ready_sockets, sockets_for_writing, error_sockets = select.select(connections, [], connections, 100)

        if error_sockets:
            break

        for r_socket in ready_sockets:
            # look for a ready socket
            other = connections[1] if r_socket is connections[0] else connections[0]

            try:
                # buffer size for HTTPS
                data = r_socket.recv(8192)

            except socket.error:
                print("Connection timeout")
                r_socket.close()

            if data:
```

Caching

The cache was implemented using a dictionary. If a http request is sent, it is checked if that request exists in the dictionary, if not is added and the request is served from the webserver the first time. Any subsequent requests to the same URL will be served by the cache. If more than 20 entries exist in the cache, the cache is then flushed.



```python
# Check if request in cache
if data in cache:
    print("Cache hit")
    cached_data = cache[data]

    if(len(cached_data)>0):
        conn.send(cached_data)
        print("HTTP request sent: ", url)
```

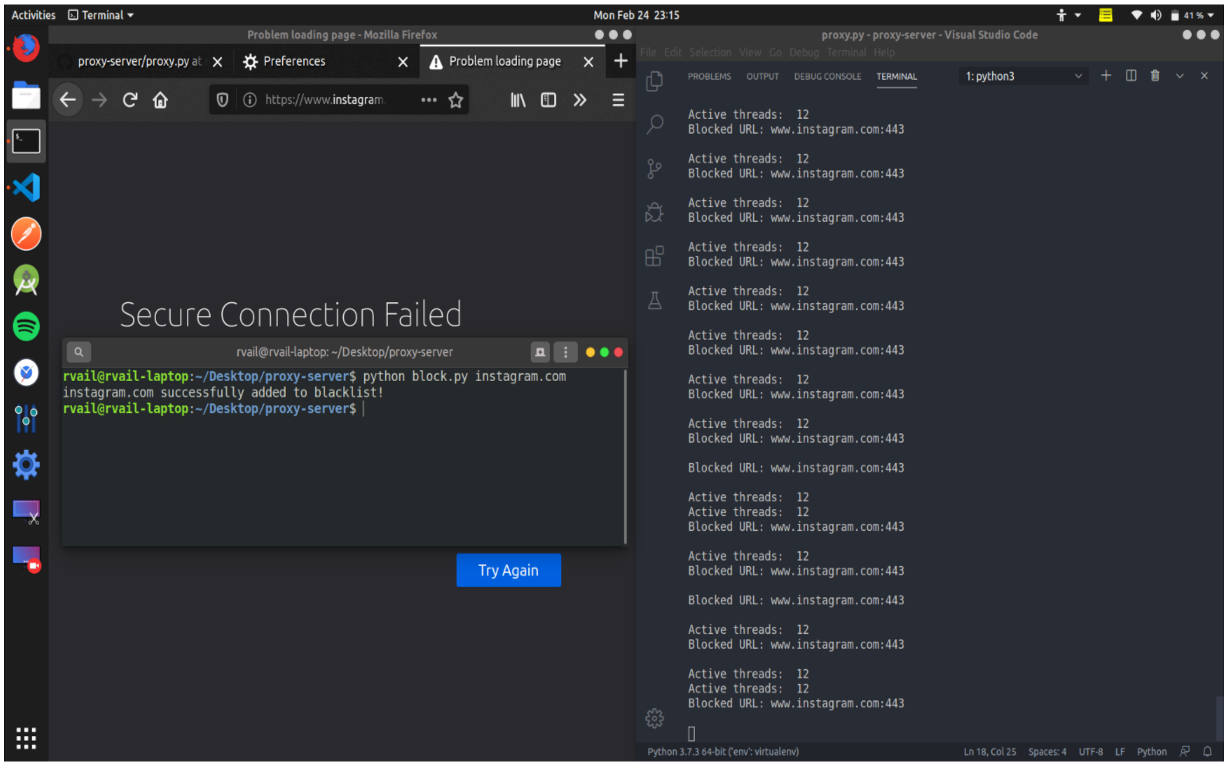


```python
while True:
    server_data = sock.recv(config["BUFFER_SIZE"])
    if (len(server_data)>0):
        conn.send(server_data)
        cache[data] = server_data
        cache_items += 1
        if(cache_items>20):
            cache = {}
            cache_items = 0
    else:
        break
```

Dynamic Blocking

Dynamic blocking is implemented with the file block.py. While the server is running, in a terminal you can run the command: *python block.py <url>* and the URL will then be added to the blacklist by the proxy. This works by writing the url to a file and the proxy checking the file for new entries every set interval.

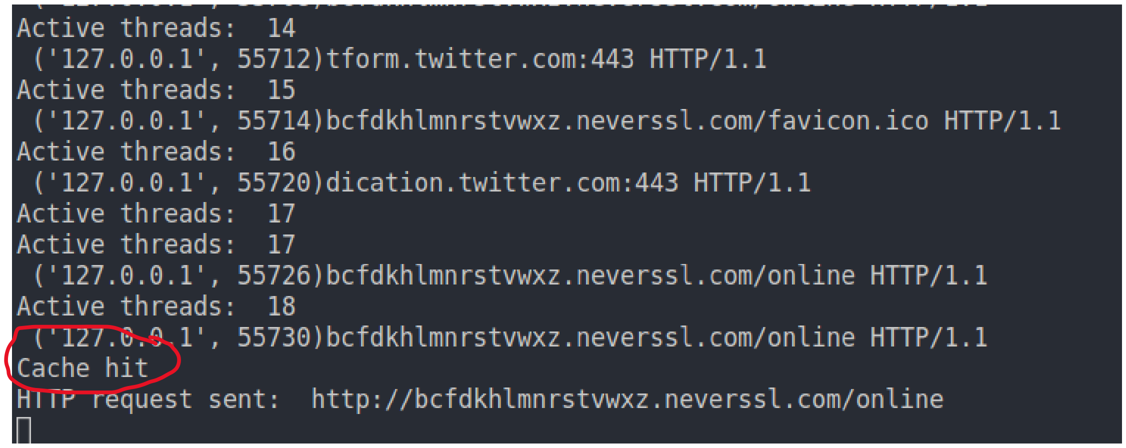
Proxy In-Action Screenshots:



```
rvail@rvail-laptop:~/Desktop/proxy-server$ python block.py instagram.com
instagram.com successfully added to blacklist!
rvail@rvail-laptop:~/Desktop/proxy-server$ |
```

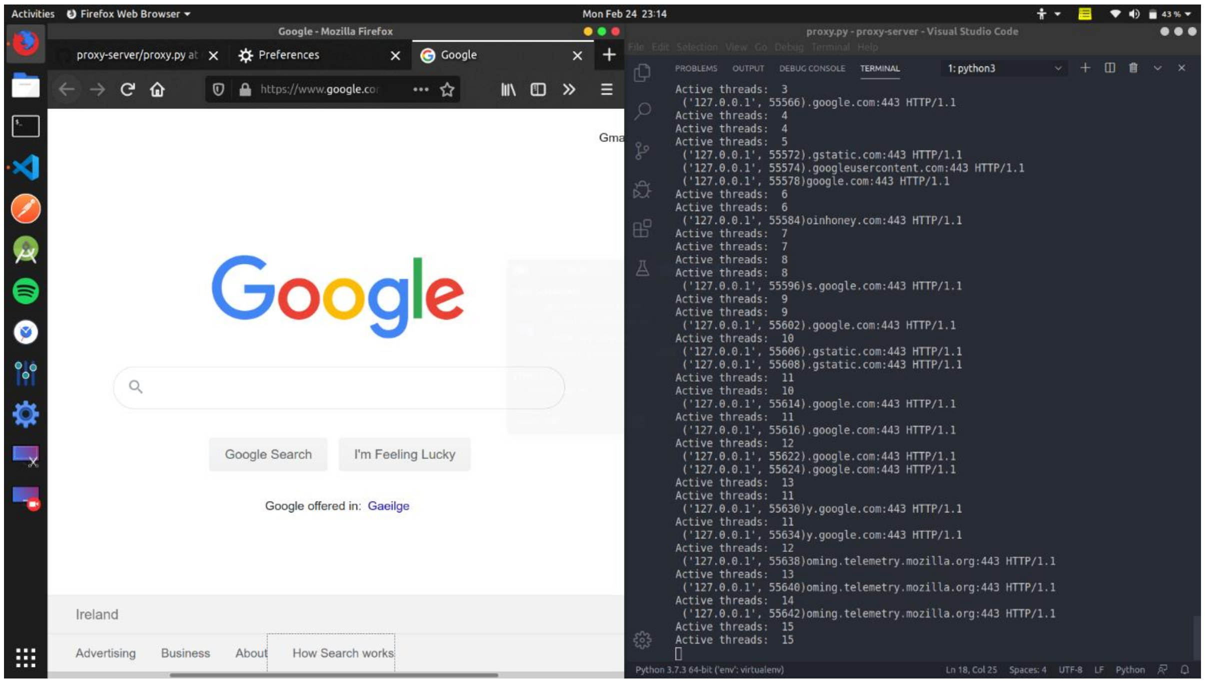
```
Active threads:  12
Blocked URL: www.instagram.com:443

Active threads:  12
Blocked URL: www.instagram.com:443

Active threads:  12
Blocked URL: www.instagram.com:443

Active threads:  12
Blocked URL: www.instagram.com:443

Active threads:  12
Blocked URL: www.instagram.com:443

Active threads:  12
Blocked URL: www.instagram.com:443

Active threads:  12
Blocked URL: www.instagram.com:443

Active threads:  12
Blocked URL: www.instagram.com:443

Blocked URL: www.instagram.com:443

Active threads:  12
Active threads:  12
Blocked URL: www.instagram.com:443

Active threads:  12
Blocked URL: www.instagram.com:443

Blocked URL: www.instagram.com:443

Active threads:  12
Blocked URL: www.instagram.com:443

Active threads:  12
Active threads:  12
Blocked URL: www.instagram.com:443

▯
```

Dynamic blocking



```
Active threads:  14
 ('127.0.0.1', 55712)tform.twitter.com:443 HTTP/1.1
Active threads:  15
 ('127.0.0.1', 55714)bcfdkhlmnrstvwxz.neverssl.com/favicon.ico HTTP/1.1
Active threads:  16
 ('127.0.0.1', 55720)dication.twitter.com:443 HTTP/1.1
Active threads:  17
Active threads:  17
 ('127.0.0.1', 55726)bcfdkhlmnrstvwxz.neverssl.com/online HTTP/1.1
Active threads:  18
 ('127.0.0.1', 55730)bcfdkhlmnrstvwxz.neverssl.com/online HTTP/1.1
Cache hit
HTTP request sent:  http://bcfdkhlmnrstvwxz.neverssl.com/online
▯
```

Cache hit



```
Active threads:  3
 ('127.0.0.1', 55566).google.com:443 HTTP/1.1
Active threads:  4
Active threads:  4
Active threads:  5
 ('127.0.0.1', 55572).gstatic.com:443 HTTP/1.1
 ('127.0.0.1', 55574).googleusercontent.com:443 HTTP/1.1
 ('127.0.0.1', 55578)google.com:443 HTTP/1.1
Active threads:  6
Active threads:  6
 ('127.0.0.1', 55584)oinhoney.com:443 HTTP/1.1
Active threads:  7
Active threads:  7
Active threads:  8
Active threads:  8
 ('127.0.0.1', 55596)s.google.com:443 HTTP/1.1
Active threads:  9
Active threads:  9
 ('127.0.0.1', 55602).google.com:443 HTTP/1.1
Active threads:  10
 ('127.0.0.1', 55606).gstatic.com:443 HTTP/1.1
 ('127.0.0.1', 55608).gstatic.com:443 HTTP/1.1
Active threads:  11
Active threads:  10
 ('127.0.0.1', 55614).google.com:443 HTTP/1.1
Active threads:  11
 ('127.0.0.1', 55616).google.com:443 HTTP/1.1
Active threads:  12
 ('127.0.0.1', 55622).google.com:443 HTTP/1.1
 ('127.0.0.1', 55624).google.com:443 HTTP/1.1
Active threads:  13
Active threads:  11
 ('127.0.0.1', 55630)y.google.com:443 HTTP/1.1
Active threads:  11
 ('127.0.0.1', 55634)y.google.com:443 HTTP/1.1
Active threads:  12
 ('127.0.0.1', 55638)oming.telemetry.mozilla.org:443 HTTP/1.1
Active threads:  13
 ('127.0.0.1', 55640)oming.telemetry.mozilla.org:443 HTTP/1.1
Active threads:  14
 ('127.0.0.1', 55642)oming.telemetry.mozilla.org:443 HTTP/1.1
Active threads:  15
Active threads:  15
```

Normal Execution