**Group Members**

- Ang Ze Yu (A0187094U)

- Choo Xing Yu (A0202132E)

**External Libraries Used**

The main “external” library we used, **unrelated to network programming**, is OpenMP. As mentioned in the README.md, this is “external” (note the quotes), because it is included in the gcc / g++ compilers itself just by using the -fopenmp flag.

Specifically, we are using it only as a **threadpool abstraction / implementation** (step 4 of “main thread” section below), whereby after *accepting* a connection in the main thread, the main thread spawns a child task to forward data to and fro the client / destination in the child thread.

OpenMP is used by specifying certain constructs (e.g., #pragma omp task { … } sends a task to the threadpool).

There are only **3 lines of functional OpenMP** in our submission (all in the main() function), all commented and explained in code.

**Code description and design decisions**

Since this is a multi-threaded program, let’s break down how the main thread / child threads will behave in the program.

Main Thread

The main thread is responsible for:

1. Initialisation:
   1. Parsing inputs: port\_number, TELEMETRY ENABLED and blacklist file name.
   2. Initializing blacklist into a vector.
2. Administrating the main **epoll** loop. On each iteration, it listens for 2 types of “events”:
   1. **Listens** on the socket with its port specified by user, any thereby **accepting** (only this) new connections (if any)
      1. Uses a thread to handle the rest of the processing after accepting this connection (see **task 1** under child threads section).
   2. Gets **notified** (no processing done) of newdata from **existing connections** (if any)
      1. Signals a thread to start forwarding the data for this connection (see **task 2**)

Epoll is a linux API / facility for waiting on and “demultiplexing” a set of file descriptors (in this case the socket descriptors) simultaneously, resolving once any of said descriptors is ready for I/O.

Child Threads

Child threads are implicitly initialized by the Main thread (OpenMP) into a thread pool.

They handle **2 tasks:**

Task 1: Handling new connections

When the Main thread accepts a new connection, the child thread will be allocated with the connection and the child thread will call the function handle\_new\_client. Handle\_new\_client will be responsible for

1. Reading initial data from the client\_sock\_fd passed by Main thread.
2. Parsing the http header read from client\_sock\_fd to obtain
   1. Url
   2. Port number (if any), otherwise it is 443 by default
3. Checking if the connection being made is in the blacklist per the rules specified in the assignment pdf.
   1. if it is, prevent the connection from being made. (403 bad request)
4. Make a DNS query for the url in the http header, retrieving the respective IP address
5. Make the connection (open the socket) to the destination
6. Send a SSL greeting back to the client
7. **Immediately continue with task2 below**

Task 2: Forwarding **new data** in an existing connection

This task per a child thread is performed under the following 2 circumstances:

* The child thread finished setting up the proxy connection (task 1)
* The main thread detects new data via epoll for an existing connection

1. Loop
   1. The child thread epolls the (original) **client** socket descriptor and **destination** socket descriptor for data
      1. **If timed-out** (meaning no more data to send ***for now***), exit the loop
         1. Important: the connection is not closed! The client might want to persist it. This time out is very very short (refer to code), it is only to prevent a thread from blocking on epoll and doing nothing for too long. (resource wastage)
   2. If the client wants to send data, forward it to the destination
   3. If the destination wants to send data, forward it to the client
   4. If either side wants to **close the connection**, then do so (close both sockets, and perform some cleanup)

Blacklist

The blacklist file is parsed into a c++ vector (arraylist) and prior to each connection there’ll be a check to see if the connection’s domain is in the blacklist.

How the check works is it’ll loop through every element in the vector and do a substring search on whether the input’s domain contains any of the blacklist’s domain. if it does the thread will be blocked from making the connection and an error message will be printed out, else it’ll allow the connection to be made.

**One minor assumption**

We tested both on localhost and on the xcne server (but using browser from our systems).

For xcne server, we tested with **forticlient VPN**, and did not encounter any access issues as such.

We are assuming there are no xcne specific authentication / authorization details we need to know of.

**While running the telemetry, observe and explain the difference between HTTP/1.0**

**and HTTP/1.1.**

|  |  |
| --- | --- |
| http1\_logs.txt (HTTP/1.0) | http1.1\_logs.txt (HTTP/1.1) |
| Hostname: sg.yahoo.com, Size: 212014 bytes, Time: 0.683 sec | Hostname: sg.yahoo.com, Size: 302062 bytes, Time: 55.480 sec |
| Hostname: sg.yahoo.com, Size: 7638 bytes, Time: 0.045 sec |
| Hostname: sg.yahoo.com, Size: 89044 bytes, Time: 0.127 sec |

Table 1: sg.yahoo.com HTTP/1.0 VS HTTP/1.1

There are several differences between HTTP/1.0 and HTTP/1.1 as observed in Table 1. Namely:

1. There are 3 connections made when using HTTP/1.0 while there’s only 1 when using HTTP/1.1.
2. The time of each connection is short when using HTTP/1.0 (< 1 sec) while the duration of the connection for HTTP/1.1 is significantly longer (55.48sec).

Reason for both observation:

HTTP/1.1 maintains persistence connection by keeping the connection alive, and hence bundles requests/responses under the same connection while its within TTL (time to live) or until the connection is terminated.

Whereas HTTP/1.0 does not maintain persistent connection and hence it’ll terminate connections after each requests/response. This also means that HTTP/1.0 requires a brand-new connection to be made for every request/response.