Design Document for P1 HTTP Proxy

Partner A: Bryce Richardson

Partner B: Raquel Moreno

1. Socket Management

Describe 1) how many socket objects are used, 2) the variable name for each socket, and 3) the purpose of each socket object.

> We used three distinct socket objects in our code.

1. proxySocket is created at the start of the program, and it listens and waits for clients to connect.
2. An instance of clientSocket is declared any time the proxySocket accepts a new connection. A clientSocket receives requests and data from its respective client.
3. An instance of serverSocket is created between the proxy and the host defined by a client's request header. It receives data from the serverSocket.

   > While there is only one proxySocket, there can be as many clientSockets or serverSockets as there are clients.

2. Multi-threading

Overall Design

Describe how multithreading is used in your program. You must include 1) the number of threads you use including the main thread, 2) what each thread is for and doing, 3) any loops you have for each thread.

> In the main function, we declare one additional thread that handles the proxySocket while the main thread handles the keyboard and .
>
> The proxySocket creates a new thread any time a new client is accepted.
>
> Finally, two new threads are created when a client asks to CONNECT to a host: one that receives from the server and one that receives from the client.

Justification

Justify why your design is efficient and show evidence it can handle multiple client requests efficiently. Specify your testing scenario (how many requests were made, which websites were being used, etc)

We assert that our design is efficient based on the quickness of response and the large loads that it can manage.

Our proxy can load HTTP requests without issue. For example, we can load neverssl.com instantly. Doing so usually requires three requests:

- GET http://neverssl.com
- GET http://majesticgrandastoundingyawn.neverssl.com/online
- GET http://majesticgrandastoundingyawn.neverssl.com/favicon.ico

Where the sub-domain name is variable.

After this, the proxy normally makes multiple requests (usually ~7-10) to several different image sites. Such requests include:

- CONNECT contile-images.services.mozilla.com:443
- CONNECT img-getpocket.cdn.mozilla.net:443
- POST http://r3.o.lencr.org/

However, our proxy also works with much larger and more sophisticated servers. For example, we can load www.wikipedia.com near-instantly as well, despite being an HTTPS website.

This normally requires several requests (~5) of “CONNECT www.google.com:443” followed by:

- CONNECT wikipedia.com:443
- CONNECT www.wikipedia.org:443

After which the proxy is connected to Wikipedia.

In addition, if we search youtube.com, which has a large amount of dynamic content, we receive dozens of requests between the time that the browser connects and is fully loaded. These requests include:

- CONNECT yt3.ggpht.com:443
- CONNECT jnn-pa.googleapis.com:443
- CONNECT static.doubleclick.net:443
- CONNECT play.google.com:443
- POST http://ocsp.pki.goog/gts1c3

The final one of these, “POST http://ocsp.pki.goog/gts1c3”, is by far the most frequent. It often occurs immediately after any of the other requests, and likely represents the act of loading visual data retrieved from another server.

On Youtube, we can stream 4+ videos (including a live-stream) at 480px and browse Wikipedia or Reddit without any sort of noticeable delay or lag. Doing so caused dozens of requests to be made every minute to a very diverse group of websites.

Because of this, we feel confident that our design is efficient.

Noticeably, we discovered an issue where our browser cannot load “cnn.com”. After further investigation, this is because the browser makes a request to http://cnn.com, which defaults to port 80. However, if we search “https://cnn.com” instead, it loads properly, so we concluded this was not a problem with our design.

3. Streaming

Describe how streaming is implemented in your Proxy and the parameter (i.e. num of bytes) used for streaming. Justify the choice of your parameter.

We used 2048 bytes because it seemed like a reasonable upper limit for the size of a request header (the only data we would need to receive all at once). This parameter caused no problems during testing, so we decided to stick with it.

To stream, we simply pass whatever a socket receives using socket.recv() to another socket using socket.sendall(). The result was the appearance of a speedy and reliable streaming connection.

4. Data structures

In the cell below, list any notable data structures you used and justify the use of the data structure. Also specify how you handle synchronization for the data structure if there were any need for synchronization. If none, you can say "None".

Our only notable data structure is a Python dictionary struct that we utilized in our “handleRequests” method to write data to a .json file.

To ensure synchronization for these .json files, we used uuid.uuid1() to generate unique file names so that no two threads would ever try to write to the same file.

5. How shutdown is handled

Describe how you handled the shutdown gracefully.

Each of our threads are designed to close themselves automatically when they detect that one or all its sockets have been closed (either intentionally or due to error).

When the EOF is reached or Ctrl+D is entered into the command prompt, the main thread closes the proxySocket that accepts new clients. This allows the proxy to shut down quickly and gracefully by relinquishing its remaining resources.

6. Error handling

Describe how you handle unexpected errors by specifying 1) what kind of errors you may encounter 2) what you do for each error.

The majority of errors we could encounter occur when using socket.recv() and socket.sendall(). This may happen because of an unexpected failure or because the host or client has closed the connection.

To handle this, we surround these commands in try/except blocks wherever we expect them to appear.

7. Any libraries used

List any libraries that you used for the implementation and justify the usage.

- **json**: allows us to convert the python dictionary into a json object that we can write to our json files
- **os**: allows us to create the paths that our Log directory will be created as well as our json files
- **sys**: allows us to read data from command line to retrieve the arguments
- **threading**: allows us to create threads for the server and client
- **socket**: allows us to sockets for client, server, and proxy server to allow communication. Also allows us to use TCP constants like SOCK_STREAM
- **uuid**: allows us to generate a uuid for each .json file

8. Reflection

What was the most challenging part working on this project? Most fun part?

The most challenging part of this project was starting. Given the brief hints for piazza code and prior knowledge of TCP along with knowledge from P0P, it took us a while to figure out how to translate our thoughts into code. Beginning with a blank file challenged us to fully understand the spec.

The most fun part was running urls through Firefox and actually seeing the traffic pop up in our terminal. Getting to experiment with Firefox and search different URLs was rewarding because we knew that our code was working properly and as expected.

If you are to do this all over again, how would you do it differently?

We would attend office hours before getting started with code to ensure that we had the spec down. Although it is expected to have to constantly visit the spec, if we had a solid understanding of each part going in then it would have shortened our transition time between HTTP, logging CONNECT, etc.

Reflection on pair programming

Log of the amount of time spent driving and the amount of time spent working individually for each part (e.g., X drives 1 hour; Y drives 45 minutes; X works alone for 1 hour, etc.)

For Check point:

- 3/25: Bryce drives 1 hour, Raquel drives 1 hour
- 3/26: Bryce drives 1 hour, Raquel drives 1 hour
- 3/28: Raquel drives 1 hour, Bryce drives 1 hour

For Final submission:

- 3/29: Bryce drives 30 min, Raquel drives 30 min
- 3/30: Bryce and Raquel discuss code issues for 45 min, both agree need break
- 3/31: Raquel drives 1 hour, Bryce drives 1 hour

What went well/or not-so-well doing pair programming? What was your take away in this process?

> Our use of VSCode liveshare went really well as it allowed us to collaboratively code and share ideas. It was easy to follow along with what the other was doing and allowed each of us to try our own ideas when the other person's idea wouldn't work.
>
> Our take away from this process is that you can learn a lot from another person by watching them code. If you pay attention to what a person is coding and ask questions about what is going on, you can gain a better understanding of the programming language being used and also why something works as it does.

Submission Instruction

Remember to export to pdf and push it to your github team repo under the project root