Connor Kenny – 304437322
Justin Liu – 504487373
Anderson Huang – 104299133

CS 118 – Project 1 Report

1) High Level Design

In HTTP protocol, the client must be able to generate a request and transmit it to a remote host and the server must be able to respond to formatted requests and transmit files. In our design, we use HTTP abstraction, derived used classes, inheritance, and simple string parsing, to create HTTP messages that conform to the HTTP 1.0 standard. Because HTTP 1.0 is non-persistent, each client only opens one connection per session.

To actually do its job, we designed the client in the following manner. First, parse command line arguments. Then, proceed to resolve IP addresses using socket programming functions. After successfully establishing a connection to a remote host, the next step is to send and receive data. Using HTTP abstraction, the request is generated, and encoded into a byte stream, and sent to the server. After transmitting the request, the client finds the end of the header by moving an index forward constantly checking for “/r/n/r/n”. Then it processes the headers accordingly and moves onto the payload. It saves this into a file by continuously reading data into a buffer, and then the outfile. At the end of the program, the payload has been successfully saved into the outfile. This was checked for files up to 1GB, but should work for even bigger files if necessary.

The server starts by looking for command line arguments and assigning the first 3 to represent the hostname, port number, and current directory, respectively. Then it takes the hostname and resolves it to retrieve an IP address. After that, it creates a socket, binds it to the retrieved IP address, and then goes into a loop where it listens to and accepts any incoming connections on the socket. Upon receiving a connection, the server creates a new thread to parse the HttpRequest and retrieve the requested file, or reply with an error message if necessary. The thread encodes as much of the file as possible into a HttpResponse object and then sends it to the client. If there is more of the file that didn’t fit in the response, it is sent in chunks to the client. The client would not know that these are sent in two steps as it only receives the bytes. This was tested on files up to 1GB, but should work for even bigger files if necessary.

Extra Credit: We implemented both the timeout and the asynchronous server for extra credit. This means that we completed 3/5 of the extra credit - we did not do the persistent connections. For timeout on the client, we simply adjusted the socket timeout using setsockopt(...). For the server (we implemented it in our asynchronous server for your reference), we used a similar method and simply close the connection on timeout. In order to implement the asynchronous server, we first learned from the sample code given in the hints. Then, building from that, we simply added the code that was in our thread function from the regular server to the part of the asynchronous server where the file descriptor does not equal the listening file descriptor.

2) Problems and Solutions

a) We were able to download files, but could not download images. We were able to solve this problem by switching from a buffer of characters to a buffer of uint8_t. We thought that they should work the same because they are one byte, but clearly they did not. Once we changed the buffer to uint8_t, we were able to download images perfectly. This worked for both images in HTML files along with files like JPG's.

b) After researching how to send files back to the client, we originally planned to use the sendfile function, which read from a file descriptor into the socket descriptor. However, we eventually realized that we needed to copy the file into payload and encode that within the HttpResponse object instead. This required significant rewriting of the thread function within the server.

c) We also had a problem with a "length_error" when encoding our HTTP response. We still don't know why this occurred, but we were able to fix it by changing the way that we created the string and ByteBlob. A small change from creating a string using the constructor to appending individual characters one by one solved this entire problem.

d) We ran into the problem that we could not handle files over 100MB. This was because we were trying to store the entire file encoded into a vector. The problem was that the vectors can only hold a certain amount of data contiguously. In order to fix this, we split the file into chunks that were read and written right after each other. This way, the entire file is never actually stored in a single vector. This allowed us to test on files up to 1GB as was suggested on Piazza.

3) Additional Instructions

We did not use any extra libraries, which means there are no extra instructions in order to build and run our project.

4) How We Tested

Client - We tested the client by first downloading web pages and images from the internet. This was to make sure that we did not absolutely need the hostname and port number explicitly. Once this worked, we tested our client on our server by downloading text files and large images. The files were up to 1GB in size, but it should work for any size files as long as they fit on the computer.

Server - We tested the server initially by running a single instance of our client and downloading files. We tested both extremely small and large files in order to make sure that all the writing and reading on both sides worked correctly. We also tested our server with curl / wget to make sure

it was accessible from the outside. Finally, we ran numerous instances of our client and made roughly 20 requests for 30 MB files and they all worked correctly. This was important to prove to ourselves that the server could handle concurrent connections. The server also works with files up to 1GB in size, even with multiple clients running concurrently each requesting considerably sized files.

5) Contributions

Connor (304437322) - I mostly focused on the HTTP abstraction and the client for this project. We first implemented the HTTP abstraction and then started on both the client and the server at the same time. I aided both Justin and Anderson as they worked on the client and server, but I mostly worked with Anderson to make sure that the client worked perfectly, so that when the server was up and running, we could test it. I also went through after we had believed we finished the project to test it thoroughly. This resulted in many small changes that needed to be done in order to handle certain corner cases, like extremely large files.

Justin (504487373) - I worked primarily on the server code from start to finish. I expanded on the sample code given by the project spec and implemented command line arguments, multithreading, and the creation, parsing, and transmission of HttpRequest and HttpResponse objects within the server. I also cleaned up the HTTP implementation code and fixed several compilation issues.

Anderson (104299133) - I worked with Connor on the HTTP abstraction while fixing minor details relating to parsing requests, responses, and the like. I was also in charge of implementing the client, which included duties such as parsing command line arguments, encoding HTTP messages, resolving server IP addresses, sending and receiving data, and responding to HTTP response codes.