Zackery Leman & Chris Lu

C++Server

We wrote a linux based C++Server using a multi-threaded structure. The main server thread establishes a listening socket and continuously loops to accept incoming connection requests. Upon receiving a proper connection request, the server creates a new thread specifically for that connection that accepts GET request(s) from the client, depending on whether the client is making HTTP/1.0 or HTTP/1.1 requests. The new thread waits for a request from the client. If the thread does not receive a request in a given time period that is equivalent to 60 seconds divided by the number of active connections, then the thread will cut the connection to the client by simply exiting. When the thread does receive a request, it iterates through the request for the request type, file path, and HTTP type to ensure that the request is a valid request. If the request is valid, the thread proceeds to look for the file by adding the file path given onto the root path. The thread then either writes back to the client through the pipe either the complete contents of the file, a 404 error if the file path does not exist, or a 403 error if the server does not have permission to access the file. In the case of a HTTP/1.0 request, the thread exits after handling the request. For a HTTP/1.1 request, the thread maintains the connection to listen for future requests and resets the timeout.

We ran into a few issues when constructing this server; The issues were ultimately simple to fix. The first issue was that we allocated too little space for the buffer that the read function fills when receiving a GET request. This resulted in multiple reads, with the subsequent reads existing as fragments of the initial get request. These fragments were thus not in a proper GET request format so we were failing to handle them appropriately. The second issue was that we could not get the write function to actually write data back to the client when running from a Linux machine, though it was working when running on an OSX machine. We ultimately were unable to resolve this issue.

The server is not able to parse every kind of file, such as a ".htaccess" file. If our server could handle such a file type, then upon the client sending a GET request for a specific file, the server would the check to see if the IP address of the client was on the whitelist specified by the htaccess file. If so then the server would receive and send the requested file back immediately to the client. If not, then the server would immediately write back to the client asking for a username and password. The server would then block until it receives a message from the client. Upon receiving the message, it would parse the message for the username and password and look in the according ".htpasswd" file to see if such a pairing exists. If it does, then the thread will return the contents of the file. If not, the thread writes back a 403 error.

If our server was allowed access to outside of Bowdoin's campus, the performance of HTTP/1.0 and HTTP/1.1 would vary more. 1.0 would be favored over 1.1 when dealing with large files during times of high traffic. Parallelism would enable 1.0 requests to be transferred simultaneously whereas with 1.1 requests, each file must wait until the file before it is completely sent. In addition, if bandwidth is high, it would be more advantageous to use 1.0 since being able to send multiple files at once would allow clients to completely saturate the communication pipes while using 1.1 would not necessarily saturate the communication pipes if the files and requests are not big enough. Also, if latency were relatively large, then using 1.0 would enable clients to send multiple requests without having to wait for requests ahead of it to process. Conversely, when traffic is high and the majority of the files are small, it would be more beneficial to use 1.1 since the overhead of TCP teardown and rebuild would outweigh the benefits of simultaneous transferal.