1. The implementation strategy mostly consisted of 2 large parts: the socket api calls, and the string parsing. I began with writing the client socket api calls as they were the simplest to familiarize myself a bit more with the socket api. Then I wrote the string parsing for the sserver portion as it was the most complex. Then I wrote the sserver socket api calls as they were a bit more complex, and then reused the string parsing for the client-side header reads. The concurrent connections were done using method a in the assignment description, using fork. I had a while loop with accept to grab the next connection in queue (automatically handled by accept()), and I had a queue size of 10, as was recommended in Beej (5-10 connections). If you were forked into a parent task, you aren't supposed to really do anything. If you were forked into a child task, then you would be delegated the work to process a client call. That way you could manage the branches easily without running into any weird interdependent hanging.

2. I tested the string parsing by writing it inside the client code, and then passing the file-to-read.txt buffer into the string parsing so that way I could test things such as case-insensitivity and switching around header values. I tested the connections mostly just by running them and having a lot of print statements that would tell me when things are progressing properly. I also tested the simultaneous functionality by running the command:

```
for file in file-to-send.txt binary smallboi.txt whitespace; do    ./sclient -p 1080 -s localhost < "$file"
& done
```

This essentially loops through my code and executes them. However, it's not executed sequentially due to the "&" symbol. This will make the code run them all in the background, so I knew my code was running concurrently.
Source for code:
https://unix.stackexchange.com/questions/103920/parallelize-a-bash-for-loop

each of the files listed included several different kinds of files, whether it was just a file that was consisted only of binary text, a file with lots of whitespace characters, or an equivalence case file with just plaintext.

3. Right now I think the biggest bug is dealing with fragmented responses. I didn't actually realize we had to account for this edge case until I went to submit the file and I saw that the server tested fragmented responses. I also couldn't find the test server files when I ssh'd into the portal under my own log-in, so I wasn't able to test this myself as I wasn't sure how to even test this. Basically, all of my string parsing is written assuming that the header is sent completely, so I'm like pretty sure that this doesn't work. The only way it might be saved is if the recv manages to wait until all the content length before sending the package off to string parsing, but I definitely did not intentionally account for this.

4. I mostly collaborated with Joshua as we discussed the implementation of the code. I also used the Beej sockets guide to help me through the socket api. I also used the stack exchange above for the parallelization.