Casey Brooks

CSCE 313 – 504

Machine Problem 5

For this machine problem, we were to extend what we did in the previous several problems by establishing the RequestChannel across a network connection. We were to use the POSIX sockets APIs to create a new RequestChannel class and dataserver process, which would have the same functionality as they did in MP3 and MP4 with respect to the client.

The key to implementing this machine problem was to fully understand the sockets APIs and know how to deal with the file descriptors, and know how the connection process differed from the client and the server. It was simple enough to get the client NetworkRequestChannel made, as the connection simply sets the given object to a specific file descriptor, which is read from and written to with no special code necessary. The server side, however, was a bit more tricky, as there was quite a bit more to be done with the connections coming in to the server. As such, I had trouble figuring out how to package everything necessary for the server side connections into the NetworkRequestChannel class, and so I opted to leave it all in the dataserver program, rather than have code scattered seemingly arbitrarily throughout several files. I could not figure out how to make a clean API for the server, and so I left it alone.

I also had a strange issue with the timing of the program being run, and I am unsure of whether it is simply an artefact of the way I had designed the server and client programs, or if there is an issue that I am unaware of that should be fixed. I based my client program off my MP3 code because it turned out to be more robust than MP4, but in the testing of MP5 I found that increasing the number of threads does not alter the speed of execution, although the program is correctly executing in multithreaded fashion.

The dataserver program was designed with heavy use of Beej’s Network Programming guide. I worked through the guide so that at each step I would understand what each system call does and how to use it in my code. In the end, I had tried multiplexing the server by spawning each newly accepted socket into a new thread, but this had the problem that one thread could actively communicate with the client, and so since there was activity on the one file descriptor, new ones were queued and weren’t accepted until the client had already finished everything. To remedy this, I copied the entirety of the section on the select system call, since I still had trouble understanding it after MP4, and began to step through it slowly, tweaking it to suit the needs of the dataserver program, and changing all the comments to show that I fully understand how it works, considering that if I had slugged through it from scratch I would have ended up with the exact same code, but probably less correct, and ultimately unfinished by the time this was due.

So in terms of the issue with more client threads described above, I really cannot imagine why it does not increase the speed of execution. In my novice understanding of how to manage sockets on the server side, one client can communicate with the socket at the given port at any time, but this can be tricked by having multiple file descriptors reading from the same port. This is where select is necessary, but in my code it is clear that multiple file descriptors are being written to, and they are being multiplexed correctly. So I am led to believe that this is simply the mechanics of how socket connections work, and if I wanted to write this code more correctly I would need the client connect to a different port which connects to a different process. But apart from the timing issue, the program works otherwise exactly as it should, handling data from multiple clients at once and producing correct results in all cases.

When I ran tests with one thread on the client with 10,000 data requests, it took about 110 seconds to finish the program. When I ran this same program with three clients connecting to the same dataserver, it took all of them about 300 seconds to finish, as the server could not handle the requests any faster, so in total each client ran as slowly as if it had the requests for all the processes. In other words, all three clients finished at roughly the same time, because on the server end all the requests were being processed together, with each client effectively receiving one third of the execution time of the server. So in the end, I am led to believe that the server simply cannot handle requests any faster than it can, no matter how efficiently threaded the clients are running. The select() method of multiplexing is simply a slow method of monitoring multiple file descriptors, and it does the job, just not very well.