Programming Assignment 1

Ankur Saikia (A20445640)

Details:

Choice of protocol: TCP/IP

Most dominant overhead: Copying file from storage to memory in server & cost of copying from memory back to storage in client.

TCP needs a constant connection, hence if server fails or client fails we get broken pipe exception.

TCP needs proper boundaries between data packets while sending and reading data. Else we would be reading invalid data on the client side.

Findings:

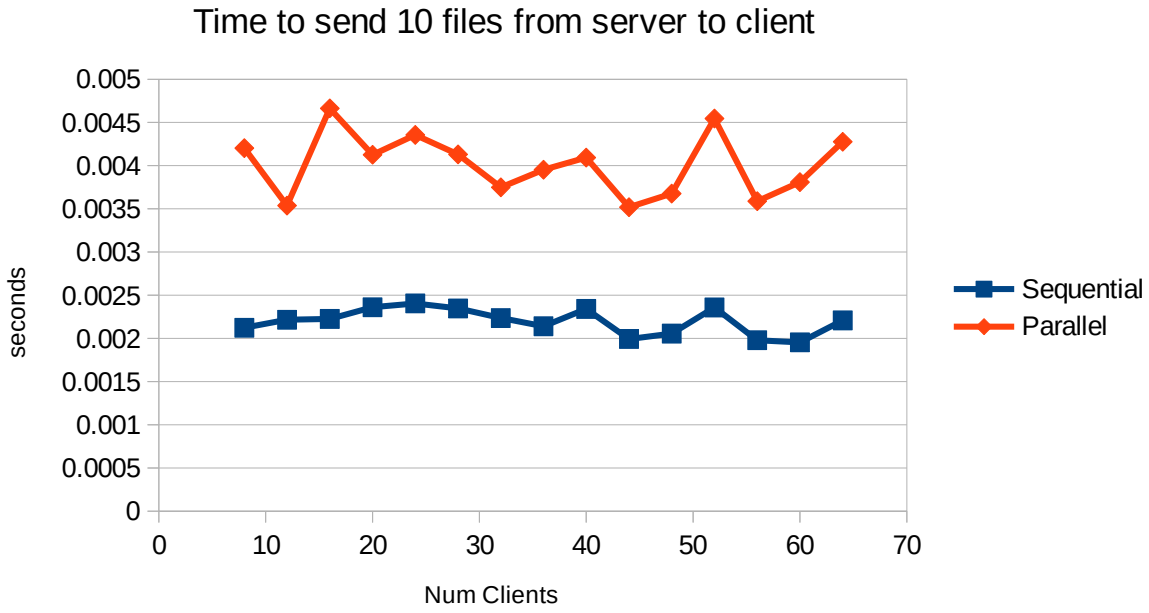
From the tests executed we find that sequential downloads are much more **robust** than parallel downloads. There were **less md5 mismatches** in sequential downloads than in parallel downloads.

Whenever we need parallel downloads, a lot of data is sent to client parallely and hence if there are more than a single file to download, we need to **send a header** in the packet. This header would determine which packet belongs to which file. It makes sense to implement multi threading in the server side to send multiple files parallely, but on the client side we have to be careful while receiving packets parallely, as there is a chance we might write a later part of data before writing the previous chunk of data to file.

Additionally, we use **thread mutex locks** in our client implementation for parallel downloads. This kind of implementation is prone to deadlocks which happens in a few scenarios and also is much harder to code than simple sequential downloads.

The server was able to handle all **64** clients parallely. Below are the average times to send **10** files from server to client. The **steps** column represent the number of clients connected at any moment. The sequential and parallel columns represent time required in seconds to send the 10 files to client.

Steps	Sequential	Parallel
8	0.002123480769231	0.004202615384615
12	0.002215972222222	0.003537638888889
16	0.002225265625	0.004663453125
20	0.00236075	0.00412625
24	0.002405066666667	0.0043562
28	0.00234690625	0.00413034375
32	0.002235392857143	0.003748392857143
36	0.002140725	0.00395185
40	0.002341613636364	0.004092795454545
44	0.001992958333333	0.003518541666667
48	0.00205575	0.0036765
52	0.002358089285714	0.00454525
56	0.001979083333333	0.003587583333333
60	0.00195475	0.0038086875
64	0.0022086875	0.004275916666667



From the above graph we see that for sequential transfer, the average time taken to transfer 10 files, remains the same, which is in the range of **0.002 to 0.0025 seconds**.

The transfer time increases when we transfer files parallely. This might be due to the fact that we create 10 threads for 10 files and hence it causes an overhead. The increased transfer time may also be due to the fact that server needs to send a header to the client while sending the file data so that client can differentiate between different file numbers. The transfer times for parallel downloads is not consistent and have a higher range of values ranging from **0.0035 to 0.0045 seconds**.

Additionally, the amount of md5 errors were also calculated and the total errors for sequential downloads was **0 files** out of **10** and it was **3.3 files out of 10 for parallel** downloads.

For sequential steps:

Overall average error: 0.0

Overall average time: 0.002196299431933807

For Parallel steps:

Overall average error: 3.386135114885115

Overall average time: 0.004014801241790616

The overall average represents the average of all the steps starting from 8 up to 64. The overall average error represents the average md5 mismatches in 1 run for all the steps starting from 8 to 64.