CIS*3210 A3 Report

Nathan Starkman (ID: 1127811), (Daniel Mojzes ID: 1167271)

For this assignment, Daniel and I worked in a pair to complete this assignment. We needed to create a program that allowed clients to submit a file from the client side and send it in parts to the server side. We completed the task together, using Daniels Port Number - 50570, to complete the assignment. Using the TCPClient and TCPServer files submitted in class, we created two files to handle the file being sent to the server. There are two elements of this project that we were asked to document for the report: The Server Testing on Different Machines and the filename repeating issue.

1. Testing on Different Machines:

a. Local Tests

Our first set of tests was completed on the LocalHost port on the Campus Wifi. We did the test solely on Daniels Computer, sending the file from one terminal to the other. To complete all tests, we created a python file called test_runner.py which ran either the Local or Server version of the program 20 times each, outputting the time and file transfer rate.
NOTE: For testing, we used the wonderland.txt test file, and as a result, hardcoded the size of the file as 169855 kb.
Here are the screenshots from the Local Tests, including the Ping test, with the packet loss percentage, Round trip min, max, average, and standard deviation.

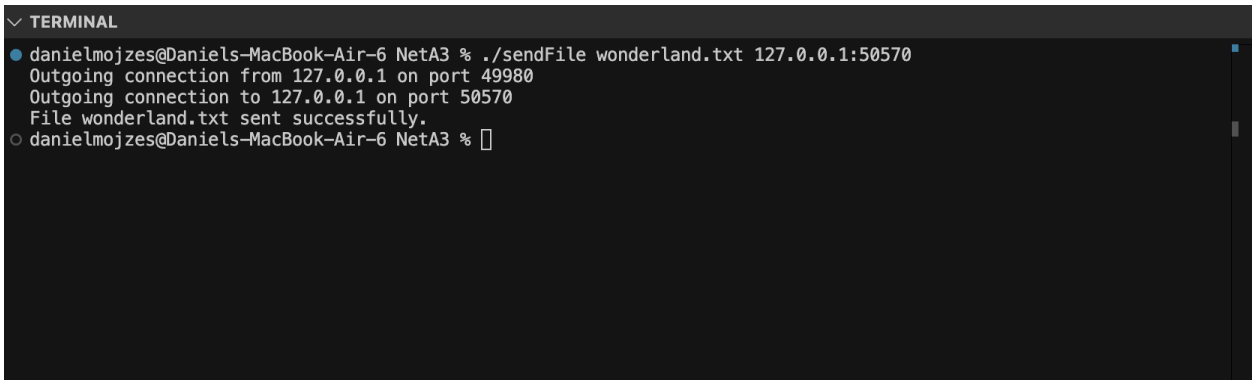
```
danielmojzes@Daniels-MacBook-Air-6 NetA3 % ping -c 20 127.0.0.1
PING 127.0.0.1 (127.0.0.1): 56 data bytes
64 bytes from 127.0.0.1: icmp_seq=0 ttl=64 time=0.185 ms
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.204 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.318 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.236 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.360 ms
64 bytes from 127.0.0.1: icmp_seq=5 ttl=64 time=0.308 ms
64 bytes from 127.0.0.1: icmp_seq=6 ttl=64 time=0.347 ms
64 bytes from 127.0.0.1: icmp_seq=7 ttl=64 time=0.354 ms
64 bytes from 127.0.0.1: icmp_seq=8 ttl=64 time=0.216 ms
64 bytes from 127.0.0.1: icmp_seq=9 ttl=64 time=0.481 ms
64 bytes from 127.0.0.1: icmp_seq=10 ttl=64 time=0.318 ms
64 bytes from 127.0.0.1: icmp_seq=11 ttl=64 time=0.325 ms
64 bytes from 127.0.0.1: icmp_seq=12 ttl=64 time=0.298 ms
64 bytes from 127.0.0.1: icmp_seq=13 ttl=64 time=0.205 ms
64 bytes from 127.0.0.1: icmp_seq=14 ttl=64 time=0.256 ms
64 bytes from 127.0.0.1: icmp_seq=15 ttl=64 time=0.289 ms
64 bytes from 127.0.0.1: icmp_seq=16 ttl=64 time=0.336 ms
64 bytes from 127.0.0.1: icmp_seq=17 ttl=64 time=0.343 ms
64 bytes from 127.0.0.1: icmp_seq=18 ttl=64 time=0.358 ms
64 bytes from 127.0.0.1: icmp_seq=19 ttl=64 time=0.140 ms

--- 127.0.0.1 ping statistics ---
20 packets transmitted, 20 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 0.140/0.294/0.481/0.077 ms
```

As you can see, the localhost ping data has no packet loss, with the following RTT stats:

- Rtt min - 0.140 ms
- Rtt avg - 0.294 ms
- Rtt max - 0.481 ms
- Rtt stddev - 0.077 ms

Next, we will show the output from running the files individually, (No Specified Buffer)



```
danielmojzes@Daniels-MacBook-Air-6 NetA3 % ./server 50570
Incoming connection from 127.0.0.1 on port 49980
fileName = wonderland(1).txt
Total size: 169855 bytes
Received from: 127.0.0.1
Buffer size: 4096 bytes
```

These are, in order, the input and output from the Client and Server files when ran on the local machine. They work as expected and return exactly what was asked, including an output file with a unique name (More on that later).

Our final Local test was running it 20 times, checking the overall speed averages and transfer rates of the local version. We ran the Python file mentioned above, which runs the local version. It got the following output:

```
danielmojzes@Daniels-MacBook-Air-6 NetA3 % python3 test_runner.py Local
Usage: ./sendFile <fileName> <IP-address:port> [bufSize]
Running The Makefile
Running iteration 1/20
Running iteration 2/20
Running iteration 3/20
Running iteration 4/20
Running iteration 5/20
Running iteration 6/20
Running iteration 7/20
Running iteration 8/20
Running iteration 9/20
Running iteration 10/20
Running iteration 11/20
Running iteration 12/20
Running iteration 13/20
Running iteration 14/20
Running iteration 15/20
Running iteration 16/20
Running iteration 17/20
Running iteration 18/20
Running iteration 19/20
Running iteration 20/20

Min time: 0.0034439563751220703
Average time: 0.00433039665222168
Max time: 0.007035255432128906

Min transfer rate: 9929277.006913379
Average transfer rate: 16709907.694179157
Max transfer rate: 20283357.973001037
```

This returns the Time and Transfer Rate average, minimum and maximum.

- Min Time - 0.0034439564

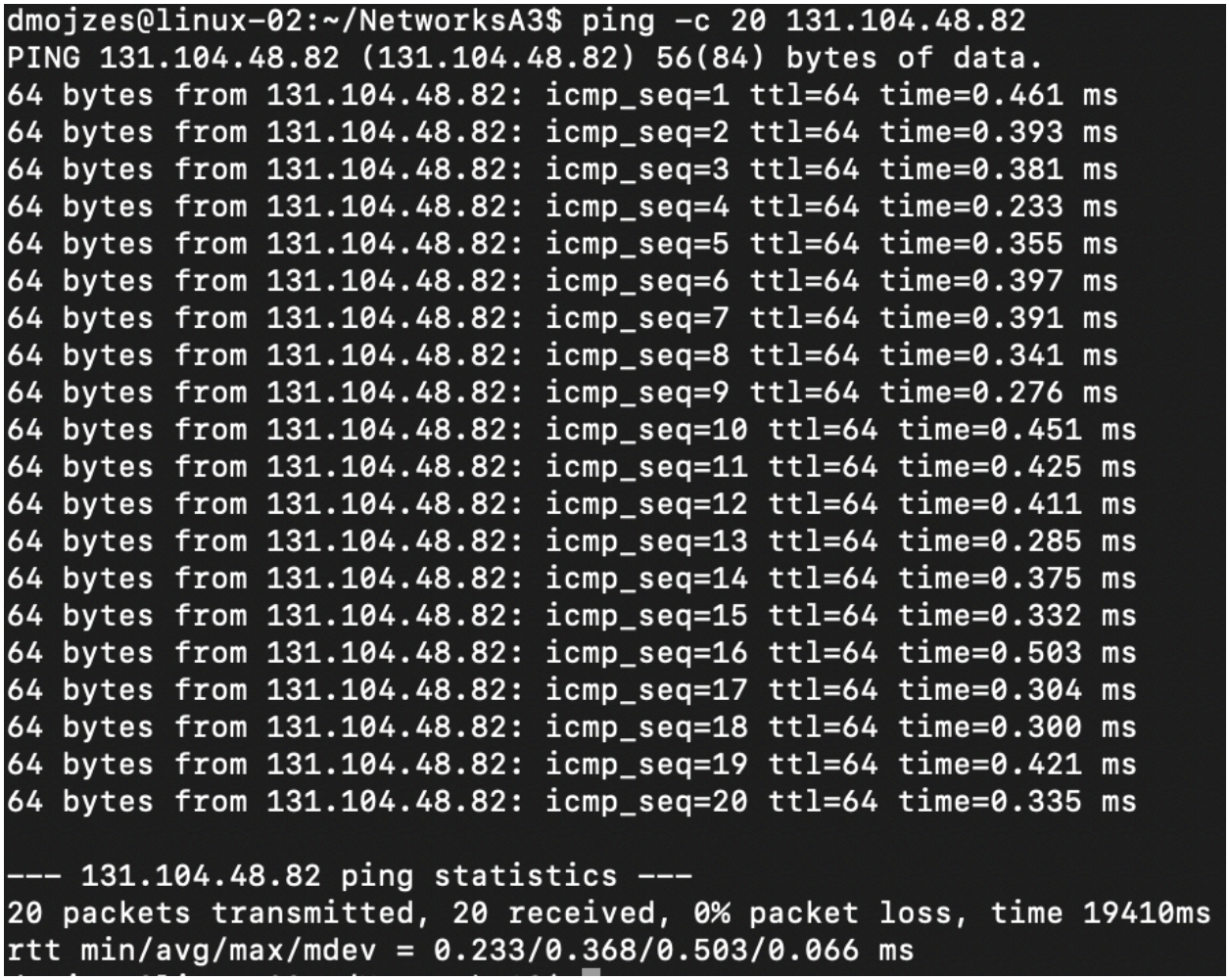
- Average Time - 0.00433039665
- Max Time - 0.0070352554321

- Min Transfer Rate - 9929277
- Average Transfer Rate - 16709907
- Max Transfer Rate - 20283357

This finishes the local tests.

b. Server Tests

Our Second set of tests was to ensure that the program worked over the school server. We first used ping to find the address of linux-01.socs.uoguelph.ca and found that it is 131.104.48.82.

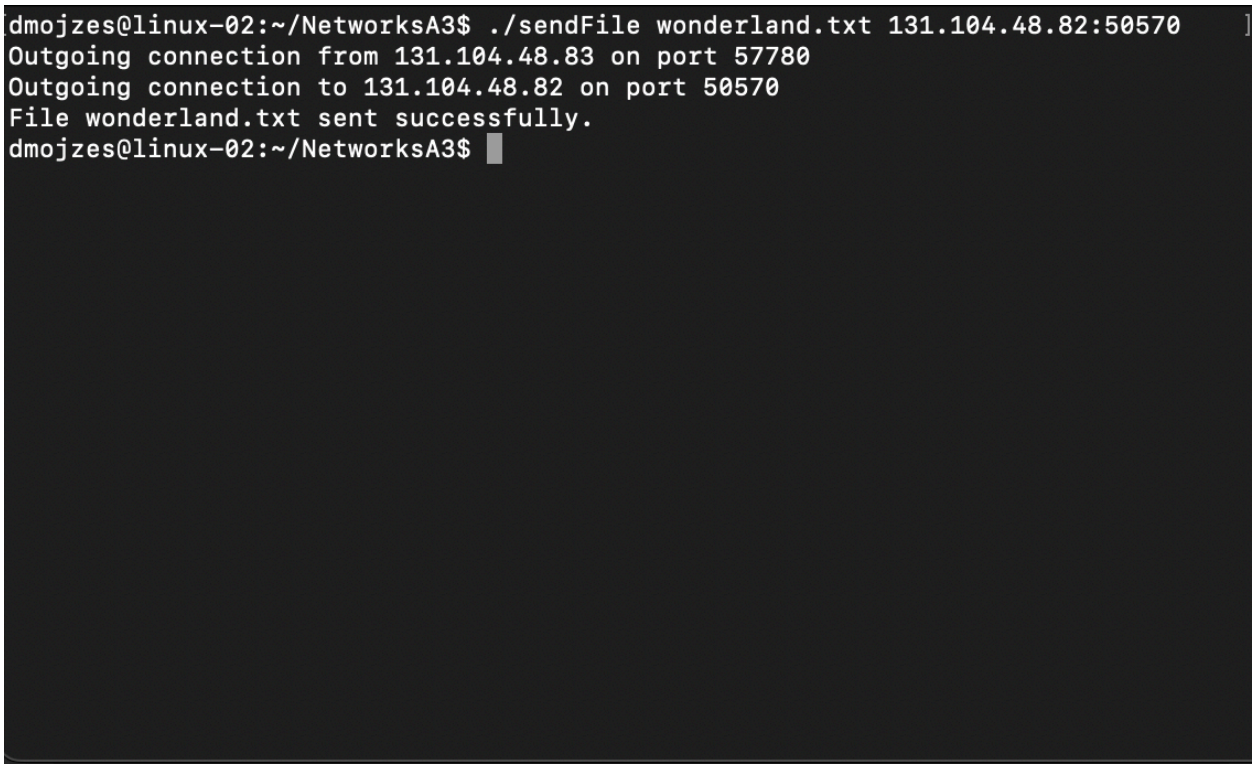


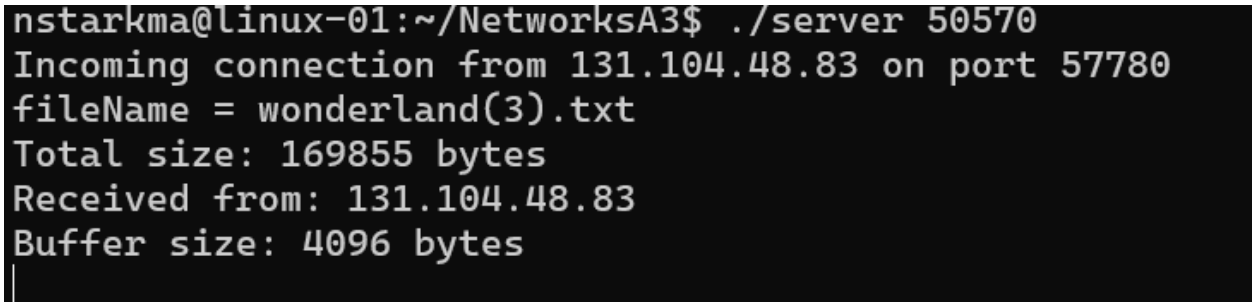
As you can see, the server ping data has no packet loss, with the following RTT stats:

- Rtt min - 0.233 ms
- Rtt avg - 0.368 ms
- Rtt max - 0.503 ms
- Rtt stddev - 0.066 ms

On average, the speed of the server is slightly slower than the localhost, but the RTT is generally closer to the average.

For the server, the client was ran on Daniels computer and the Server on Nathans. Nathan was on linus-01, while daniel was on 02. Here were the outputs for the single results:





These are, in order, the input and output from the Client and Server files when ran on the School Server. They work as expected and return exactly what was asked, including an output file with a unique name (More on that later).

Our final Server test was running it 20 times, checking the overall speed averages and transfer rates of the local version. We ran the Python file mentioned above, which runs the local version. It got the following output:

```
dmojzes@linux-02:~/NetworksA3$ make
gcc -Wall -c TCPclient.c
gcc -Wall -o sendFile TCPclient.o
gcc -Wall -c TCPserver.c
gcc -Wall -o server TCPserver.o
dmojzes@linux-02:~/NetworksA3$ python3 test_runner.py Server
Usage: ./sendFile <fileName> <IP-address:port> [bufSize]
Running The Makefile
Running iteration 1/20
Running iteration 2/20
Running iteration 3/20
Running iteration 4/20
Running iteration 5/20
Running iteration 6/20
Running iteration 7/20
Running iteration 8/20
Running iteration 9/20
Running iteration 10/20
Running iteration 11/20
Running iteration 12/20
Running iteration 13/20
Running iteration 14/20
Running iteration 15/20
Running iteration 16/20
Running iteration 17/20
Running iteration 18/20
Running iteration 19/20
Running iteration 20/20

Min time: 0.005000591278076172
Average time: 0.3163373351097107
Max time: 1.0516939163208008

Min transfer rate: 66421.41683616239
Average transfer rate: 5900638.2348012
Max transfer rate: 13969348.04615238
```

This returns the Time and Transfer Rate average, minimum and maximum.

- Min Time - 0.0050005912
- Average Time - 0.316337335
- Max Time - 1.051693916321

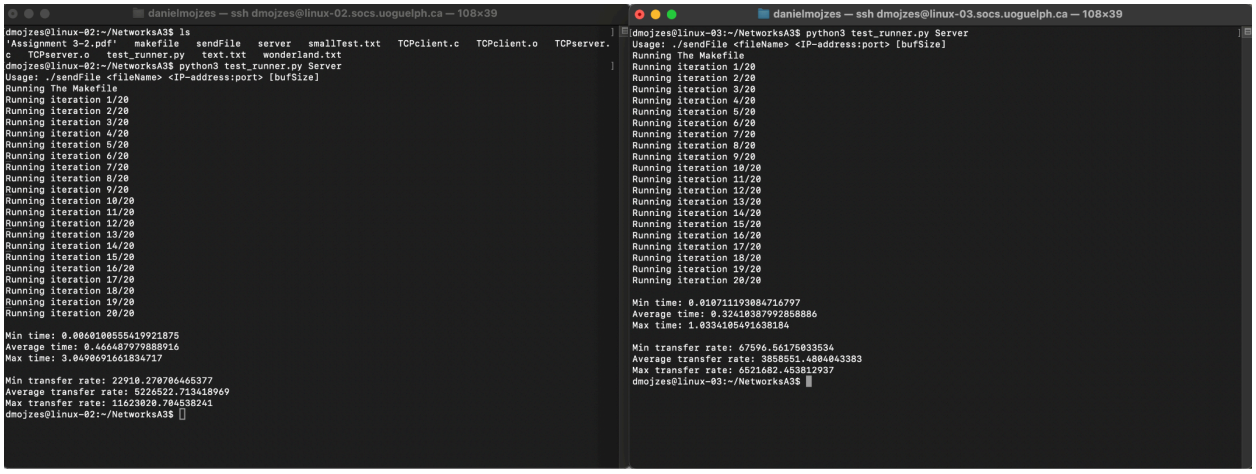
- Min Transfer Rate - 66421.416836
- Average Transfer Rate - 5900638
- Max Transfer Rate - 13969348.046

This ends the Server Tests.

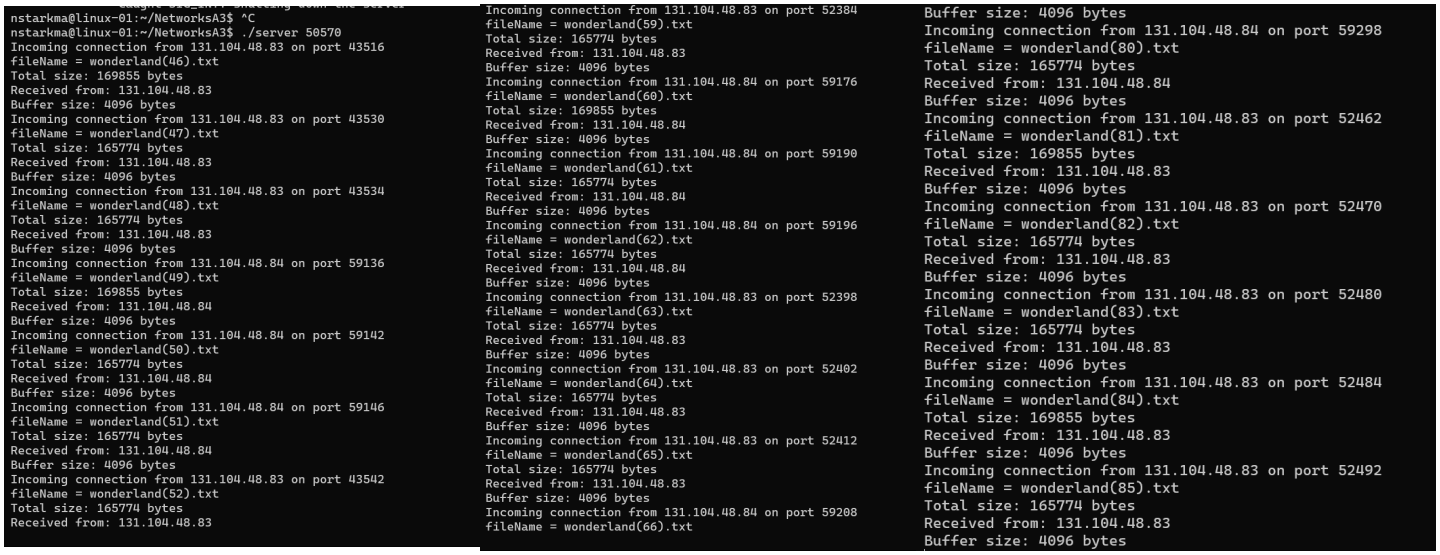
c. Dual Client Test

The final set of tests was that we wanted to ensure that files could be sent to the server at the same time to ensure that the server is able to receive multiple files at one time. We ran the SSH twice, and from Daniel's computer, ran the client in two windows. This resulted in the files all being received properly by the server, showing every output, its buffer, and its file length properly.

Here is the Client terminals:



Here are the Server Clients:



These show that all of the files are being created by the server properly, with every single clients file coming through to the server at the same time, regardless of them using the same server at the same time.

2. File Name Conflicts:

To handle the issue of File Name conflicts, we implemented the following. The client sends over the name of the file it is trying to transfer. Once the server gets the file and is ready to write it into another file, it checks if in the space where the server exists, that there is a file with the name of the received filename. If it exists, it adds a number in brackets ((1), (2), etc) to the end of the file. If that also exists, it keeps incrementing until it is unique. The server then writes the data into that new unique file. For example, with wonderland, if the files wonderland, wonderland(1), wonderland(2) and wonderland(3) exist in the servers folder, it will create a wonderland(4) file for output.