COMP 8005 Assignment 2 Report

Alex Zielinski - A00803488

03/05/2018

Contents

Report

Testing Explained

The data that is analysed during this report was retrieved by executing a series of identical tests on each server in order to capture useful data that can be used to compare server performance. The server ran on lab machine IP ***192.168.0.18*** and used port **7000** to bind to. The client program ran on lab machine IP ***192.168.0.20***. The client program was set to spawn 2000 threads, where each thread simulates a separate client. Each client then transmits packets of length 1000 bytes and reads the servers echo response. The client timeout was set to 20 seconds, so, once a client encounters a timeout, is ceases transmission, disconnects from the server and shuts down.

The command used to run the client is as follows:

```
./clt_thread 192.168.0.28 7000 2000
```

The servers were ran using the following commands respectively:

```
./srv_thread 7000
./srv_pol 7000
./srv_epoll 7000
```

At the end of each test the client and server create a log file containing some statistical data regarding each client transmission. The server log file tracks:

- *the client IP that connected*
- *when they connected*
- *the number of requests from the client the server processed*
- *the total number of bytes processed.*

The client log file tracks

- *the time of a client connection*
- *the number of requests sent*
- *the total number of bytes transmitted.*

The data in the log files was then inputted into Excel in order create charts and to retrieve averages, totals, maximums minimums.

<u>Multi-Threaded Server</u>

My multi-threaded server is a basic server and uses blocking sockets. Each client connection is accommodated within its own thread. This means that whenever a new client connects to the server, the server creates a new thread for that client connection, and that thread then facilitates all transmission events between the server and client. The lab machines are limited to creating 4900 threads and no more. Therefore, my multi-threaded server design already has a hard cap and is not able to accommodate more than 4900 client connections at one time (however threads are already being used by background OS related programs, so that hard cap number is actually lower).

Multi-Threaded Server Test Results	
Test Characteristics: Packet size: 1000 , Client transmission duration: 20 seconds	
Successfully accommodated 2000 clients	Yes
Avg num of requests processed (server side)	1226 requests
Avg server response time (client side)	18.45659 ms
Lowest num of requests processed (server side)	504 requests with 504 KB transmitted
Highest num of requests processed (server side)	16033 requests with 16.03 MB transmitted

An interesting observation after analysing the test results is that the range of processed requests by the multi-threaded server is quite large. Where the smallest number of processed requests by the server is 504 and the highest number is 16033. This means that in 20 seconds one client was able get 504 of its requests processed by the server, and in the exact same amount of time a different client was able to get triple the amount of its requests processed by the server. As a result, some clients are starved, and others are favored (even though no priority options are implemented). This discrepancy among client connections can be visually seen in ***Figure 1.0***. This goes to show that the multi-threaded model is not good at evening out resource allocation among thousands of connected clients. Another interesting observation is that only a few clients (seen at the top of the chart in ***figure 1.0***) surpassed 5000 requests whereas all other processed client requests fall around the threshold (marked by the red line in ***figure 1.0***). This means that the initial client connections are favored and hog up the server resources for a time being.

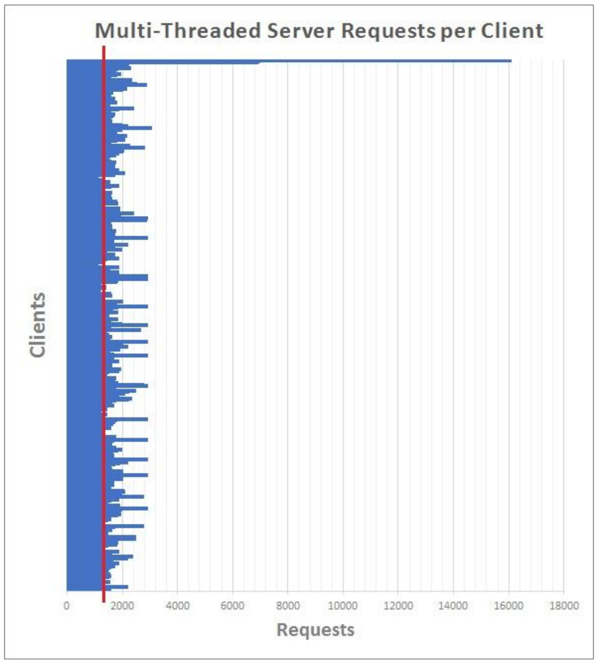


Figure 1.0 *a chart showing the number of client requests to the server per client*

Multiplexed Server (poll)

My multiplexed server makes use of the poll() API call and is not multi-threaded. Although my multiplexed server and multi-threaded server are both synchronous in nature, the multiplexed server is still more efficient as it is capable of handling more clients than the multi-threaded server (go to the **Stress Test** section to see the max number of clients each server can handle). Poll uses a non-blocking socket implementation and monitors multiple socket descriptors to see if an IO operation is possible on any of them. Poll is also level triggered, the CPU is notified of socket IO events a lot faster than in my multi-threaded server.

Multiplexed Server Test Results (poll)	
Test Characteristics: Packet size: 1000 , Client transmission duration: 20 seconds	
Successfully accommodated 2000 clients	Yes
Avg num of requests processed (server side)	1328 requests
Avg server response time (client side)	18.04135 ms
Lowest num of requests processed (server side)	480 requests with 480 KB transmitted
Highest num of requests processed (server side)	6185 requests with 6.18 MB transmitted

In comparison to the multi-threaded server the multiplexed server processed about 100 more client requests on average and was about 0.5 of a millisecond faster at responding to requests on average. This is a small increase in performance but an increase never-the-less. The biggest difference between the two servers however is in the range of its lowest number of requests processed and its highest number of requests processed. The multi-threaded server's low to high range was 504 – 16033 and the multiplexed server's low to high range is 480 – 6185. This is a considerably smaller range. However, by looking at ***Figure 1.1*** we can see that the multiplexed server is still favoring the initial first connections as the multi-threaded server did, thus creating a large spike at the top of the graph. That being said, the multiplexed server seems to be allocating its resources among clients a little bit more evenly (since the range is smaller) but we still get a jagged graph showing that some clients are being starved more than others.

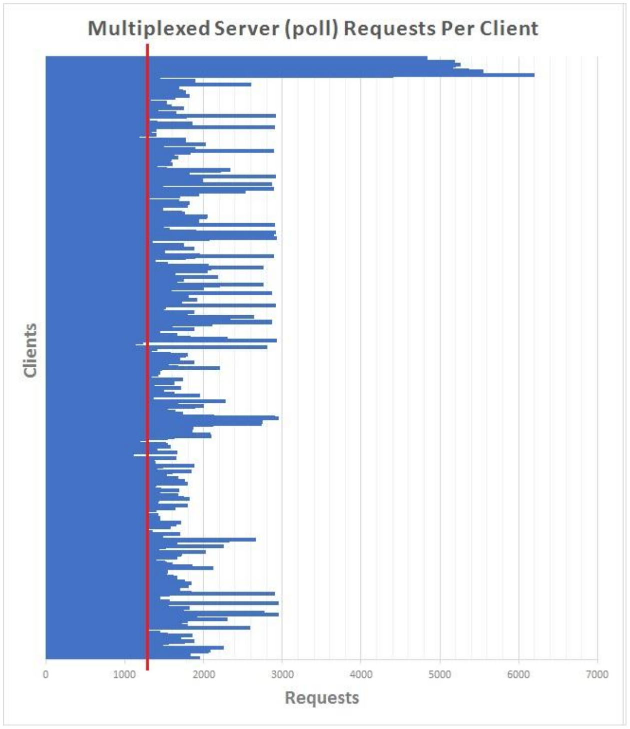


Figure 1.1 *a chart showing the number of client requests to the server per client*

Asynchronous Server (epoll)

My Asynchronous server makes use of the epoll() API call and is not multi-threaded. Epoll is similar to poll as it uses non-blocking sockets and monitors multiple socket descriptors to see if an IO operation is possible on any of them. However, epoll is edge-triggered, and this is where the performance boost comes in when compared to poll/select which are both level-triggered (more on this in the **Stress Test** section.

Multiplexed Server Test Results (poll)	
Test Characteristics: Packet size: 1000 , Client transmission duration: 20 seconds	
Successfully accommodated 2000 clients	Yes
Avg num of requests processed (server side)	1311 requests
Avg server response time (client side)	18.02421 ms
Lowest num of requests processed (server side)	521 requests with 521 KB transmitted
Highest num of requests processed (server side)	2927 requests with 2.93 MB transmitted

It was a bit surprising to me to see that the average number of requests and average server response time we roughly the same between the epoll and poll server. However, the difference with the epoll server can be seen in the its range of lowest number of requests processed and highest number of requests processed. As we went from a multi-threaded server (504 – 16033), to a multiplex server (480 – 6185), to an asynchronous server (521 – 2927), the low high range kept shrinking along the way. Epoll is a lot more efficient when it comes to allocating resource evenly among connect clients. Unlike the spikes we saw in the ***Figure 1.0*** and ***Figure 1.1***, illustrating that certain clients were hogging the server's resources, ***Figure 1.2*** doesn't contain these spikes. The issue of client starvation is less prevalent with epoll when analysing the aggregated logged data.

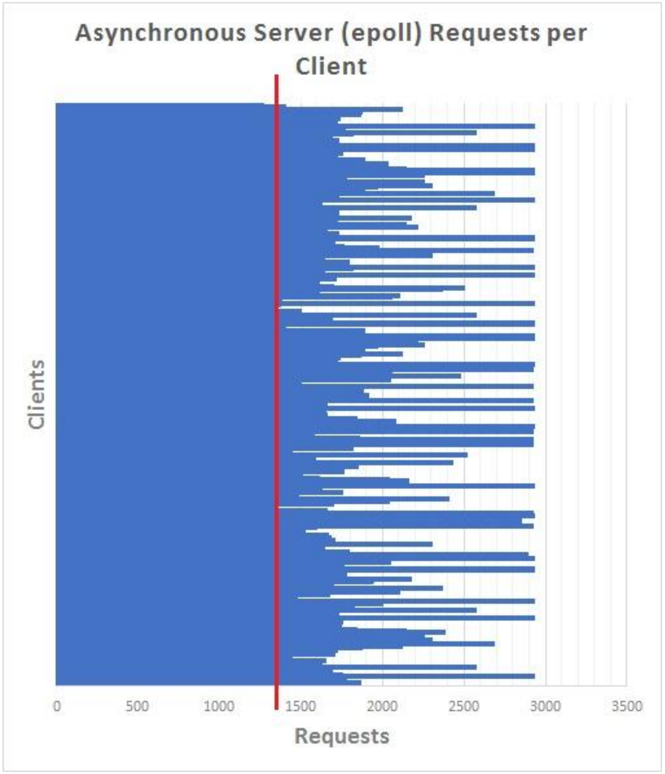


Figure 1.2 *a chart showing the number of client requests to the server per client*

Stress Test

Testing Explained

Next the breaking point of each server was tested. In order to do this, I started out with a small number of clients for each server where each client would send packets of 1000 bytes for a duration of 20 seconds. As each test passed I would ramp up the number of clients until the server could clearly not handle the traffic.

Multi-Threaded Server (max = 4000 clients)

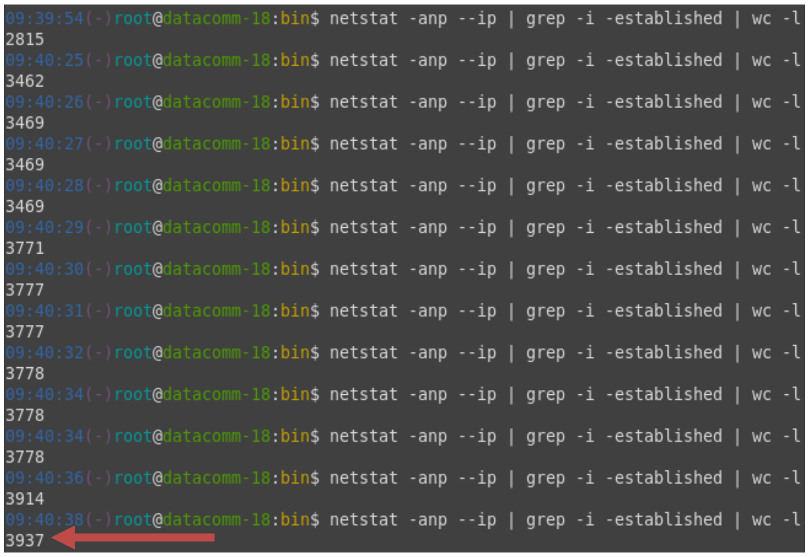
As mentioned earlier the multi-threaded server already has a hard limit of handling 4900 clients at one time as this is the max number of threads the lab machines can accommodate. This number is actually smaller however as background programs that we are unaware of are using threads.

The multi-threaded servers' max number of clients that it could handle before breaking was **4000** clients. If more than 4000 clients connected to the server then errors would start occurring as the server is not able to accommodate all the client requests. The test that concluded that the multi-threaded server could handle a max of 4000 clients consisted of the server running on one machine (IP: **192.168.0.18**) and the client program running on two machines (IP: **192.168.0.20**, **192.168.0.21**). The client programs then spawned 2000 clients each (resulting in 4000 client in total).

During the transmission session, the following netstat command was used to check the number of active socket descriptors:

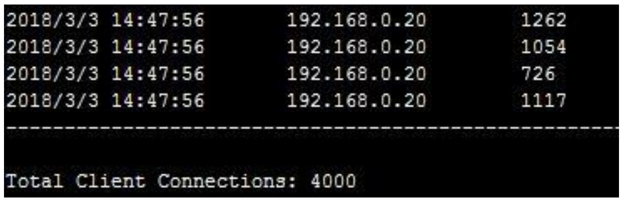
netstat -anp –ip | grep -I established | wc -l

The command was run until the number of socket descriptors started to decrease, thus revealing the max number of client connections my multi-threaded server could handle at one point which is **3937** (as seen in the screenshot below next to the red arrow).



```
09:39:54(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
2815
09:40:25(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
3462
09:40:26(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
3469
09:40:27(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
3469
09:40:28(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
3469
09:40:29(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
3771
09:40:30(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
3777
09:40:31(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
3777
09:40:32(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
3778
09:40:34(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
3778
09:40:34(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
3778
09:40:36(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
3914
09:40:38(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
3937
```

Although my multi-threaded server could only handle 3937 connections at one time it was able to accommodate all 4000 connections (some clients just had to wait until the server's resources were freed up) as the server's log file showed that 4000 clients were able to successfully connect to the server.



```
2018/3/3 14:47:56       192.168.0.20        1262
2018/3/3 14:47:56       192.168.0.20        1054
2018/3/3 14:47:56       192.168.0.20        726
2018/3/3 14:47:56       192.168.0.20        1117
--------------------------------------------------

Total Client Connections: 4000
```

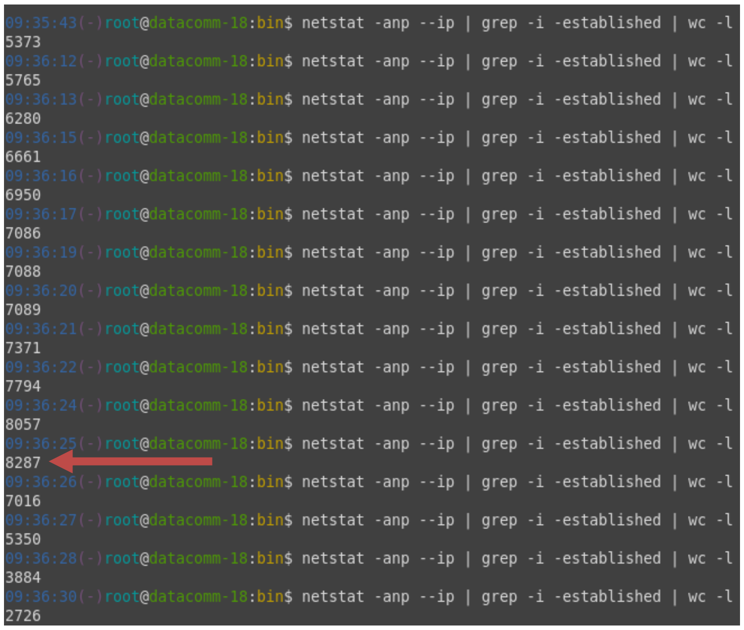
<u>Multiplexed Poll Server (max = 11,936 clients)</u>

The multiplexed servers' max number of clients that it could handle before breaking was **11,936** clients. If more than 11,936 clients connected to the server then errors would start occurring as the server is not able to accommodate all the client requests. The test that concluded that the multi-threaded server could handle a max of 11,936 clients consisted of the server running on one machine (IP: **192.168.0.18**) and the client program running on three machines (IP: **192.168.0.20**, **192.168.0.21**, **192.168.0.22**). The client programs then spawned 4000 clients each (resulting in 12000 clients in total).

During the transmission session, the following netstat command was used to check the number of active socket descriptors:

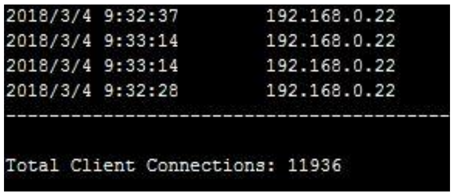
netstat -anp –ip | grep -I established | wc -l

The command was run until the number of socket descriptors started to decrease, thus revealing the max number of client connections my multi-threaded server could handle at one point which was **8287** (as seen in the screenshot below next to the red arrow).



```
09:35:43(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
5373
09:36:12(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
5765
09:36:13(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
6280
09:36:15(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
6661
09:36:16(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
6950
09:36:17(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
7086
09:36:19(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
7088
09:36:20(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
7089
09:36:21(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
7371
09:36:22(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
7794
09:36:24(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
8057
09:36:25(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
8287
09:36:26(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
7016
09:36:27(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
5350
09:36:28(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
3884
09:36:30(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
2726
```

Errors did start to occur as my multiplexed server had troubles handling 12000 client connections. However, after the transmission session finished the server's log files showed that 11,936 clients successfully connected.



```
2018/3/4 9:32:37        192.168.0.22
2018/3/4 9:33:14        192.168.0.22
2018/3/4 9:33:14        192.168.0.22
2018/3/4 9:32:28        192.168.0.22
----------------------------------------

Total Client Connections: 11936
```

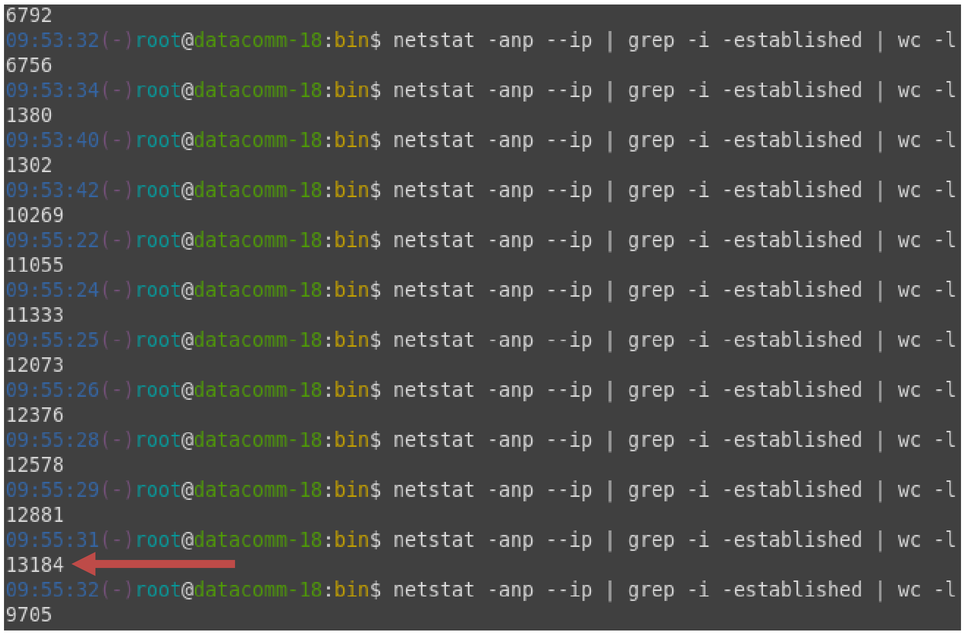
Asynchronous Epoll Server (max = 17,773 clients)

The asynchronous servers' max number of clients that it could handle before breaking was **17,773** clients. If more than 17,773 clients connected to the server then errors would start occurring as the server is not able to accommodate all the client requests. The test that concluded that the multi-threaded server could handle a max of 17,773 clients consisted of the server running on one machine (IP: **192.168.0.18**) and the client program running on five machines (IP: **192.168.0.16, 192.168.0.17, 192.168.0.20**, **192.168.0.21**, **192.168.0.22**). The client programs then spawned 4000 clients each (resulting in 20000 clients in total).

During the transmission session, the following netstat command was used to check the number of active socket descriptors:

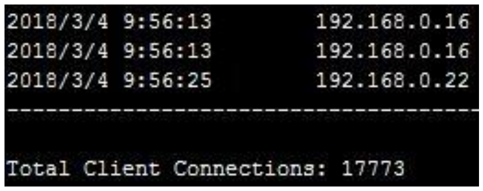
netstat -anp –ip | grep -I established | wc -l

The command was run until the number of socket descriptors started to decrease, thus revealing the max number of client connections my multi-threaded server could handle at one point which was **13,184** (as seen in the screenshot below next to the red arrow).



```
6792
09:53:32(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
6756
09:53:34(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
1380
09:53:40(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
1302
09:53:42(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
10269
09:55:22(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
11055
09:55:24(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
11333
09:55:25(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
12073
09:55:26(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
12376
09:55:28(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
12578
09:55:29(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
12881
09:55:31(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
13184
09:55:32(-)root@datacomm-18:bin$ netstat -anp --ip | grep -i -established | wc -l
9705
```

Errors did start to occur as my asynchronous server had troubles handling 20000 client connections. However, after the transmission session finished the server's log files showed that 17,773 clients successfully connected.



```
2018/3/4 9:56:13        192.168.0.16
2018/3/4 9:56:13        192.168.0.16
2018/3/4 9:56:25        192.168.0.22
------------------------------------

Total Client Connections: 17773
```

Conclusion

In the end the aggregated data showed that not only can epoll handle more client connections but that it is better at evening out the server's resources to the connected clients. This makes the issue of starvation that was encountered with the multi-threaded server and the poll server less prevalent in epoll. However, the question arises, what makes epoll so much more efficient? The answer is in the difference between level triggered and edge triggered? As computer architecture become more efficient, Poll and select could not handle the demanding nature of modern applications. Epoll was created to solve this issue and as a result is able to handle many more connections than a multi-threaded or a multiplexed server design. The performance boost is a result of epoll being level-triggered. The difference between edge triggered and level triggered interrupts is when the CPU catches the interrupt. For level triggered the processor samples for interrupts during certain times within each bus cycle. If the interrupt is not active when the processor samples for it then it will not be caught. If the interrupt becomes active right after the processor samples it then the CPU will not see it until the next bus cycle (which takes time). Now, for edge triggered interrupts the CPU is made aware of the interrupt right when it occurs, thus not having to waste time waiting for another bus cycle. This is why epoll can handle more connections than poll or select, because it can respond to socket IO events much faster.