COMP 8005 – Scalable Server

Shane Spoor

Mat Siwoski

Objective

To compare the scalability and performance of the multi-threaded, select and ePoll based client-server implementations.

Details

The goal of this assignment is to design and implement three separate servers:

1. A multi-threaded, traditional server
2. A select (level-triggered) multiplexed server
3. An ePoll (edge-triggered) asynchronous server

Each server will be designed to handle multiple connections and transfer a specified amount of data to the connected client. Each server must be designed to handle multiple connections and transfer a specified amount of data to the connected client. A simple echo client must be designed and implemented with the ability to send variable length text strings to the server and the number of times to send the strings will be a user-specified value. Each client will maintain the connection for varying time durations, depending on how much data and iterations. This will be done to keep increasing the load on the server until its performance degrades quite significantly. This will be done to measure how many (scalability) connections the server can handle, and how fast (performance) it can deliver the data back to the clients. Each client and server will maintain their own statistics.

Constraints

- The server will maintain a list of all connected clients (host names) and store the list together with the number of requests generated by each client and the amount of data transferred to each client.
- Each client will also maintain a record of how many requests it made to the server, the amount of data sent to server, and the amount of time it took for the server to respond (i.e., echo the data back).
- All the data and findings must be in a properly formatted technical report. Make extensive use of tables and graphs to support your findings and conclusions.

Application Information

Compilation

The following must be done to compile and execute the application:

```
dnf install cmake
```

After installing the application to a folder, in the terminal move to application folder COMP8005-Assn2, and run the following:

```
mkdir build

cmake .. -G ”UNIX Makefiles”

make
```

Running the Client

If running the client, within the build folder, move to the client folder and run

```
./client -i [IP_ADDRESS] -p [PORT] -m [#_REQUESTS]-n [#_CONNECTIONS] -s [MSG_SIZE]
```

The following parameters can be set:

-i - The ip to connect to.
-p - The port to connect on
-m - The number of requests to send
-n - The number of threads to make
-s - The size of the message that will be sent to the server

Running the Server

If running the server, within the build folder, move to the server folder and run

```
./server -s [thread|select|epoll] //dependent on the server that you want to execute
```

The following parameters can be set:

-s - The type of server to run (Thread, Select or epoll.

Note

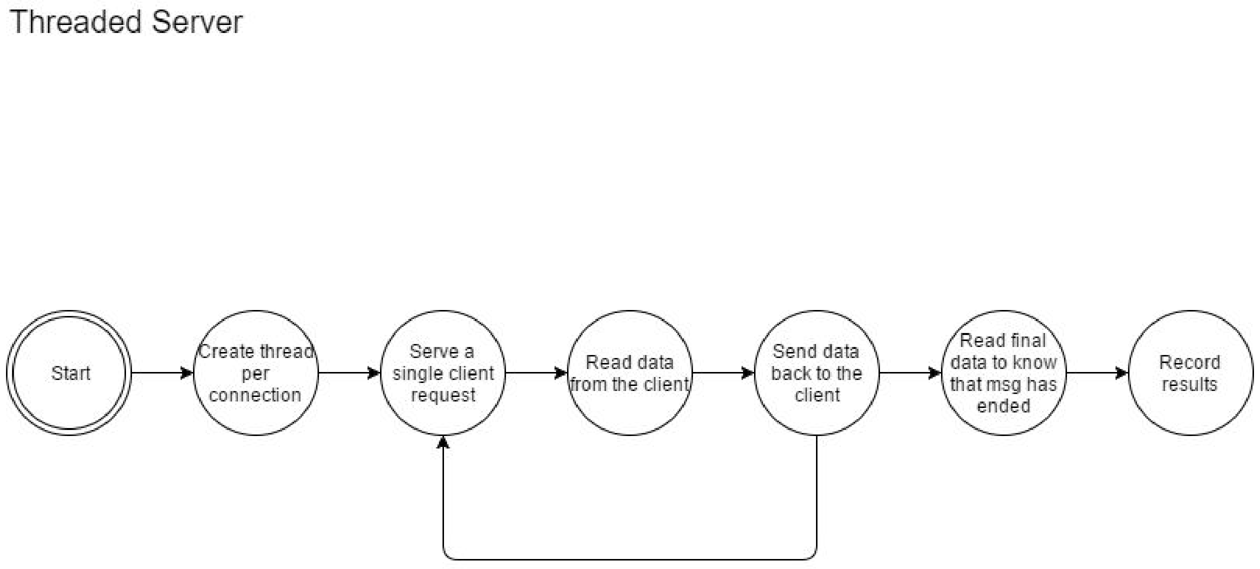
In the event that the client or server are getting the error “Too many open files” in the same terminal that is running the application, execute:

```
ulimit -n 65535    //this must be run on the terminal that the server/client is being executed on
```

Design

Multi-Threaded Design

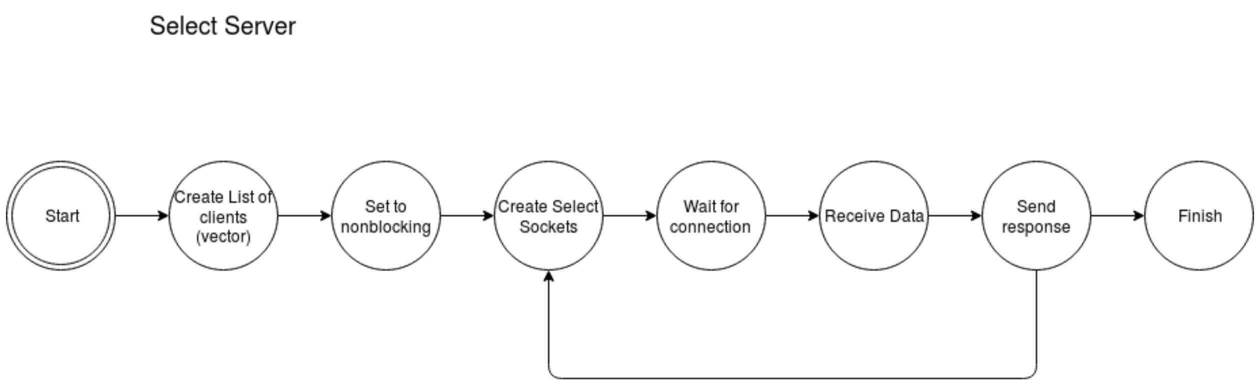
The multi-threaded server creates and binds a socket for use by multiple clients. The server will accept multiple new clients by creating new threads for each client to utilize. Each client will run on its thread. This will have a dramatic impact on performance in terms of the number of sustained connections that can be held by the multi-threaded server when compared to other higher end applications using Select or ePoll. The multi-threaded server will start by creating a socket, setting socket options then binding a name to the socket. The server will then wait to receive data from the connected client and when the data is received and stored into the appropriate structures, it is sent back to the client as a confirmation.


Threaded Server
Start
Create thread per connection
Serve a single client request
Read data from the client
Send data back to the client
Read final data to know that msg has ended
Record results

Select Server Design

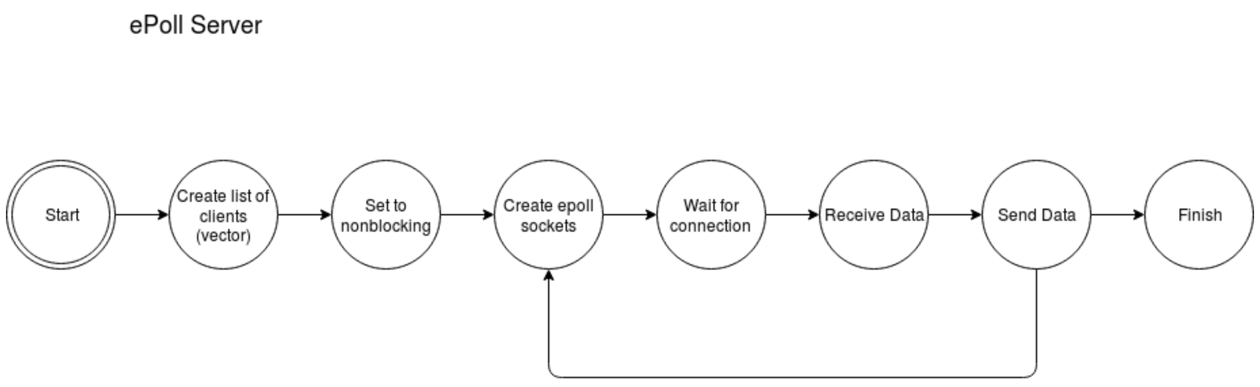
This server, instead of traditional multithreaded approach, uses a system call called select. Select call is implemented at the kernel level. Select function takes the list of socket descriptors and monitors them for any activity. Select is a blocking call, therefore, it puts the process or thread in a waiting mode until it detects any activity on any of the descriptors it's monitoring. If the select function detects any activity on any of the descriptors, it unblocks the process or a thread and returns a number corresponding to number of descriptors with activity on them.

The Select server initializes a server socket and creates a select descriptor used for all subsequent select calls. Select call monitors all the socket descriptors added to the select list. The client manager then uses select() call to monitor activity on all the descriptors. As soon as the select detects any activity on any of the descriptors it unblocks and returns a number corresponding to number of descriptors that have any activity. If the number returned by select is greater than zero, the client manager then checks if the activity is on the listening socket. If the activity is on the listening socket, it then accepts the incoming connect request and then adds the new socket descriptor to the list monitored by select. If the activity is on a client socket, the manager then handles and processes the client request and acts accordingly. If the server receives a request for the client, it then sends data to the client and performs the same activity as many times it receives a request from the client. If the server receives a quit signal from the client it then closes a socket and removes the client socket from the list. Figure below displays the design of the select server.



ePoll Server Design

The ePoll server is designed and works in similar manner as the select server. Figure below displays the design of the ePoll server.



The minor difference between the two calls is in the ePoll syntax itself. For ePoll, first we need to create an ePoll descriptor, and then initialize the ePoll struct with wanted events. Afterwards we need to add all the descriptors to the ePoll monitoring list using epoll_ctl(). Once the descriptors are added to the list, we use epoll_wait() to wait for any events on the descriptors in the monitoring list. ePoll is similar to select in that it returns a number corresponding to the active descriptors. All that needs to occur is to iterate through and handle the activity on the descriptors.

Advantages of using ePoll over select

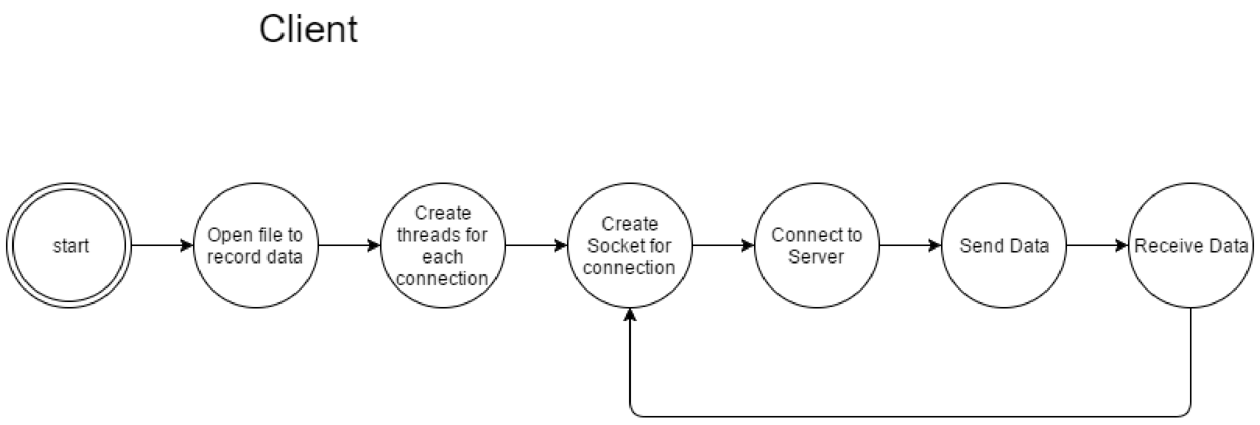
ePoll has some significant advantages over select/multi-threaded both in terms of performance and functionality.

- ePoll returns only the list of descriptors which triggered the events. No need to iterate through 10,000 descriptors to find that one which triggered the event
- You can add sockets or remove them from monitoring anytime, even if another thread is in the epoll_wait function. You can even modify the descriptor events.
- It is possible to have the multiple threads waiting on the same ePoll queue with

epoll_wait(), something you cannot do with select/poll.

Client Design

The Client used for testing the servers is designed using multiple processes to connect with the server simultaneously. The main client creates a new socket and connects to the server and then sends data over to the server and receives the same amount of data back before exiting.



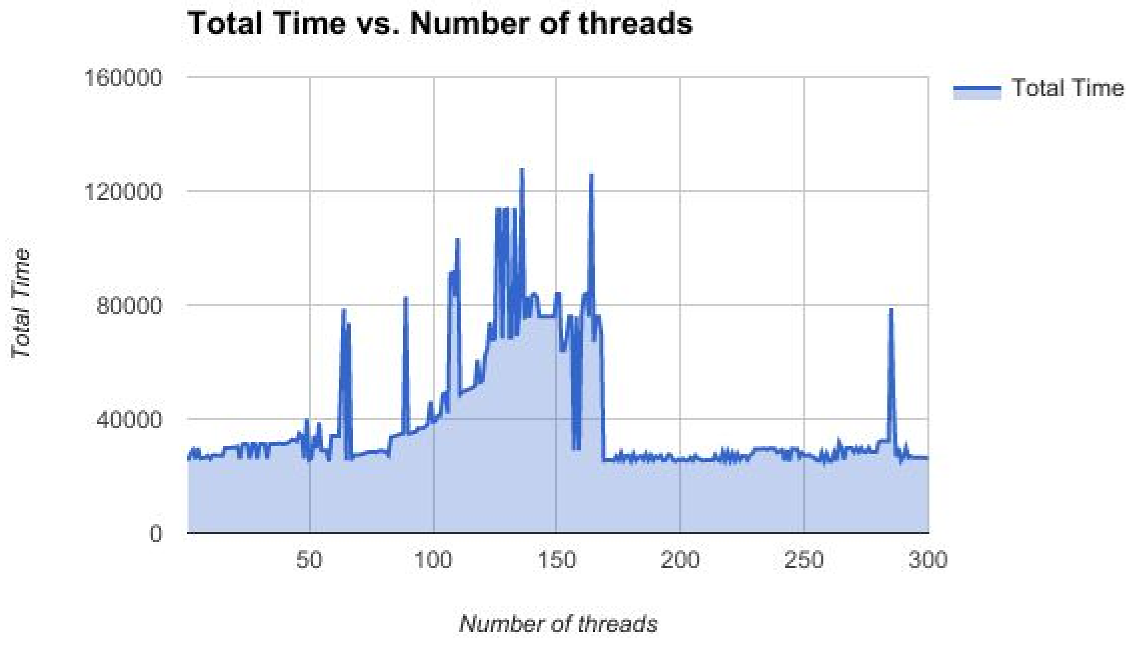
Results

These are the following results for the multi-threaded, select and ePoll servers. In the end, as was expected, the ePoll server handled 131066 connections, outperforming the select and multi-threaded servers.

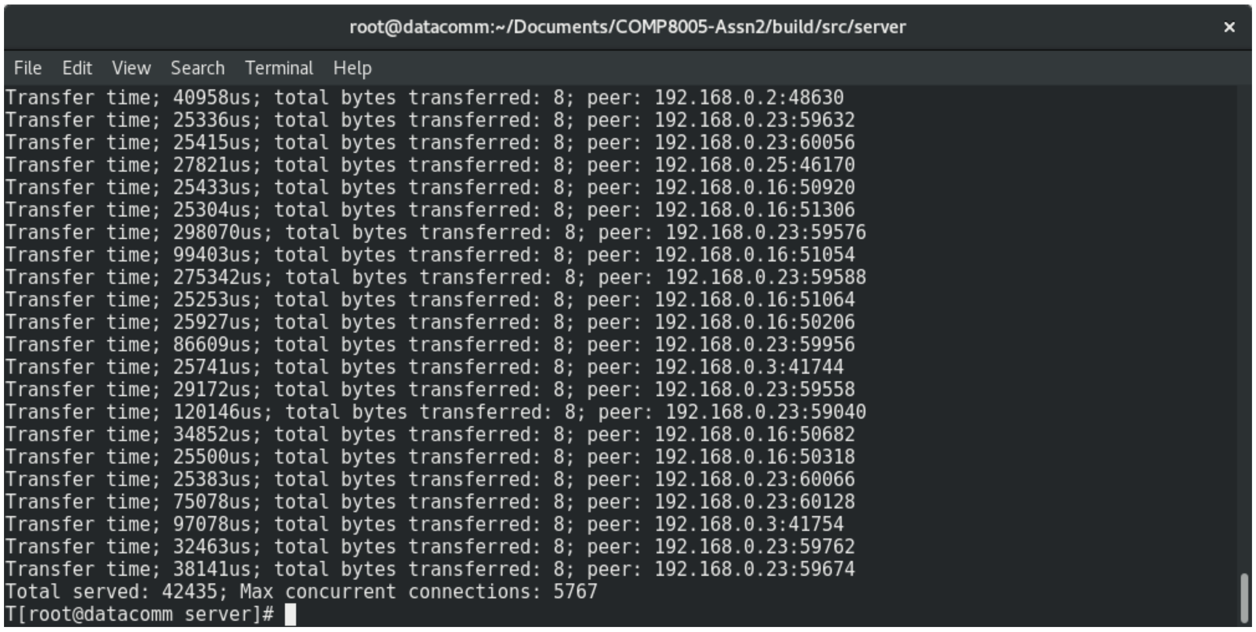
Multi-Threaded Performance

After completing our experiment and running tests against our multi-threaded server, we achieved acceptable performance with over 5767 sustained connections. The multi-threaded server handled up to 5767 sustained connections using our TCP client that was sending and receiving 1024 bytes of data with each request. The test was performed using two machines that were both running our echo client that sent 1024 bytes of data to the server.

The results based upon the number of clients connected vs. time can be seen in the graph below.



The multi-threaded server was setup to handle multiple client connections by allowing clients to connect multiple times on the same socket. Our client was designed to act as an echo client, in which it would send data to the server then receive a response and close the connection. The server was designed to handle multiple connections and stay open after sending and receiving responses, the server was designed to stay open until it was either overloaded with connections or a quit signal was received. The total amount of connections served was 42435 before the the server crashed.

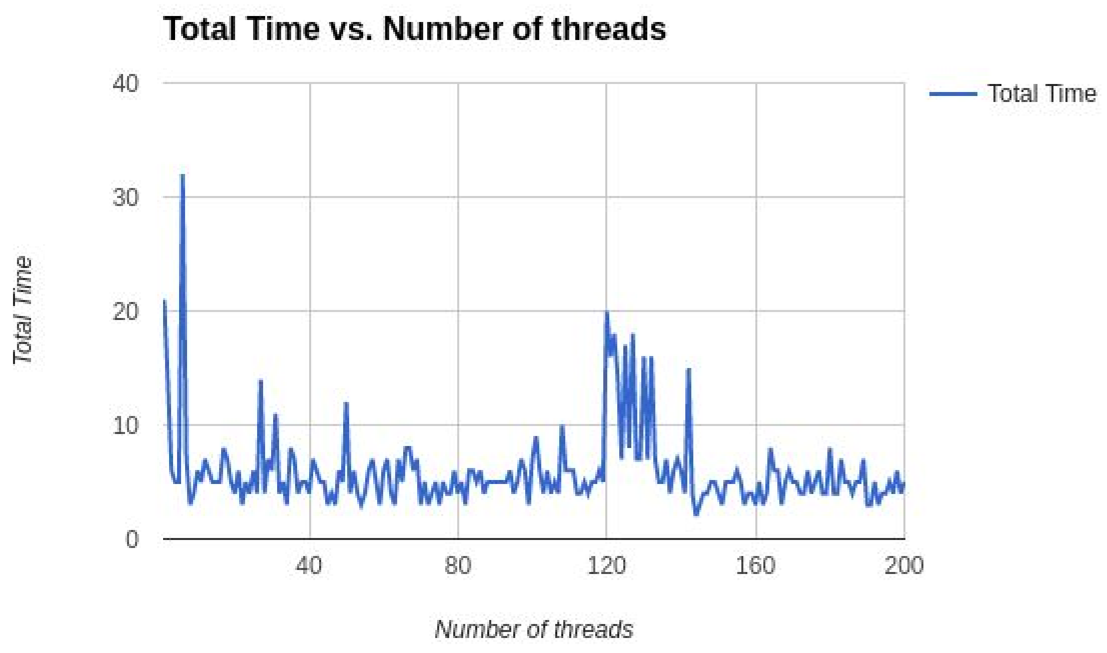

root@datacomm:~/Documents/COMP8005-Assn2/build/src/server
File Edit View Search Terminal Help

```
Transfer time; 40958us; total bytes transferred: 8; peer: 192.168.0.2:48630
Transfer time; 25336us; total bytes transferred: 8; peer: 192.168.0.23:59632
Transfer time; 25415us; total bytes transferred: 8; peer: 192.168.0.23:60056
Transfer time; 27821us; total bytes transferred: 8; peer: 192.168.0.25:46170
Transfer time; 25433us; total bytes transferred: 8; peer: 192.168.0.16:50920
Transfer time; 25304us; total bytes transferred: 8; peer: 192.168.0.16:51306
Transfer time; 298070us; total bytes transferred: 8; peer: 192.168.0.23:59576
Transfer time; 99403us; total bytes transferred: 8; peer: 192.168.0.16:51054
Transfer time; 275342us; total bytes transferred: 8; peer: 192.168.0.23:59588
Transfer time; 25253us; total bytes transferred: 8; peer: 192.168.0.16:51064
Transfer time; 25927us; total bytes transferred: 8; peer: 192.168.0.16:50206
Transfer time; 86609us; total bytes transferred: 8; peer: 192.168.0.23:59956
Transfer time; 25741us; total bytes transferred: 8; peer: 192.168.0.3:41744
Transfer time; 29172us; total bytes transferred: 8; peer: 192.168.0.23:59558
Transfer time; 120146us; total bytes transferred: 8; peer: 192.168.0.23:59040
Transfer time; 34852us; total bytes transferred: 8; peer: 192.168.0.16:50682
Transfer time; 25500us; total bytes transferred: 8; peer: 192.168.0.16:50318
Transfer time; 25383us; total bytes transferred: 8; peer: 192.168.0.23:60066
Transfer time; 75078us; total bytes transferred: 8; peer: 192.168.0.23:60128
Transfer time; 97078us; total bytes transferred: 8; peer: 192.168.0.3:41754
Transfer time; 32463us; total bytes transferred: 8; peer: 192.168.0.23:59762
Transfer time; 38141us; total bytes transferred: 8; peer: 192.168.0.23:59674
Total served: 42435; Max concurrent connections: 5767
T[root@datacomm server]# 
```

Select Server Performance

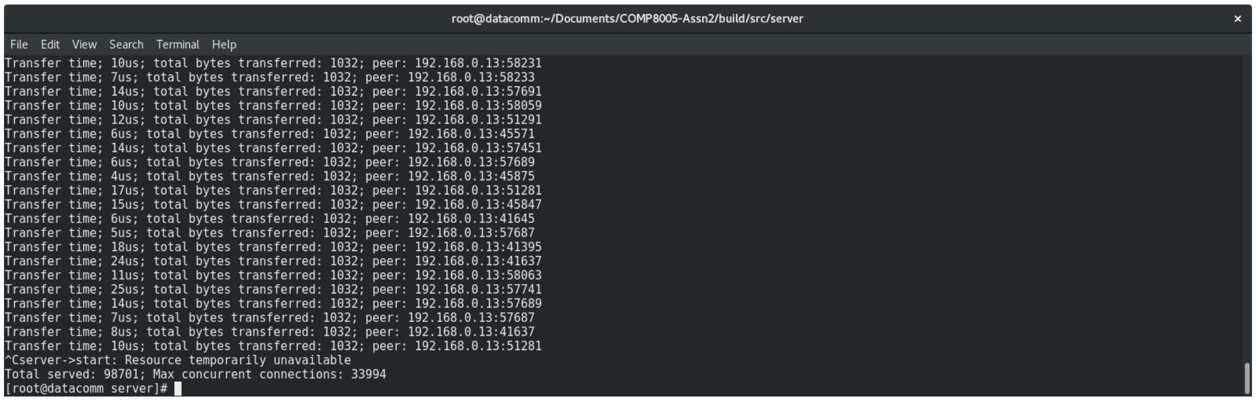
After completing our experiment and running tests against our select server, we achieved excellent performance as compared to the multi-threaded server. The select server handled up to 33994 sustained connections using our client that was sending and receiving 1024 bytes of data with each request. The test was performed using two machines and using our client which would send and receive 1024 bytes of data from the server.

The results based upon the number of clients connected vs. time can be seen in the graph below,



By analyzing the graph you can conclude that all of the clients were connected to the server for a random time period between 1 – 10 microseconds; however there are random spikes in which the total time to send is greater than 30 microseconds. The large spikes are most likely caused by the AIO_WRITE as although it's an asynchronous write, so many connections are occurring that it is still having issues printing fast enough to the file. This was tested on only 3 machines sending data to the client. Each client was designed to send data of 1024 bytes per request to the server. The client was designed to send these requests to the server for specified amount of time. Each client would then hold a connection for the specified amount of time while constantly sending and receiving data of 1024 bytes to and from the select server.

The total number of connections that were served was 98701 connections before the server crashed.

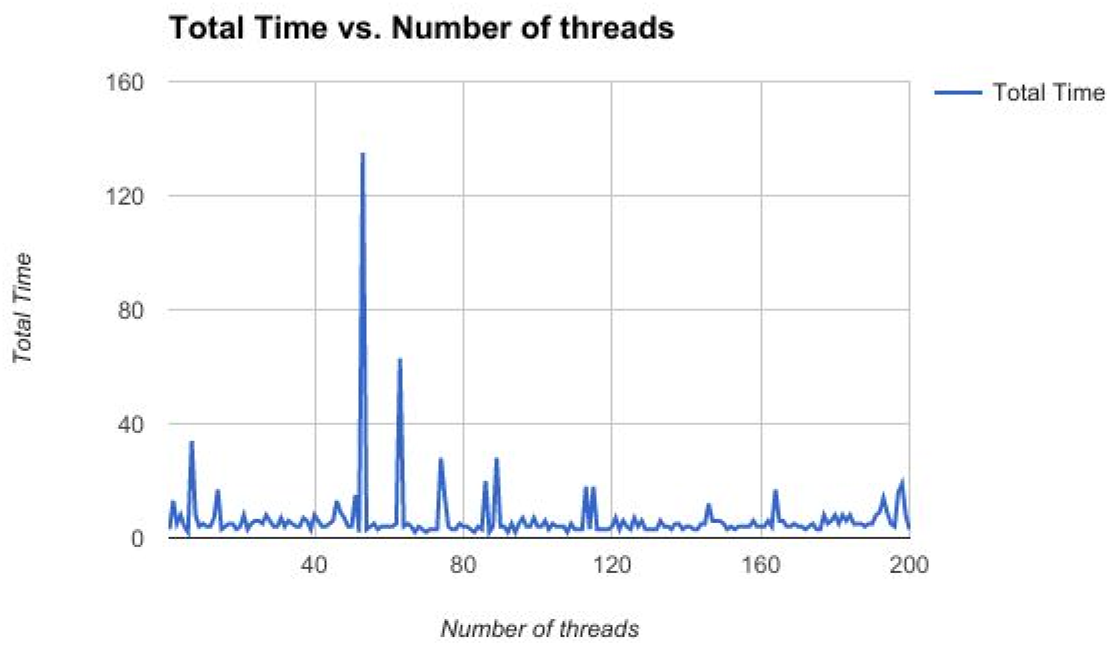

root@datacomm:~/Documents/COMP8005-Assn2/build/src/server
File Edit View Search Terminal Help

```
Transfer time; 10us; total bytes transferred: 1032; peer: 192.168.0.13:58231
Transfer time; 7us; total bytes transferred: 1032; peer: 192.168.0.13:58233
Transfer time; 14us; total bytes transferred: 1032; peer: 192.168.0.13:57691
Transfer time; 10us; total bytes transferred: 1032; peer: 192.168.0.13:58059
Transfer time; 12us; total bytes transferred: 1032; peer: 192.168.0.13:51291
Transfer time; 6us; total bytes transferred: 1032; peer: 192.168.0.13:45571
Transfer time; 14us; total bytes transferred: 1032; peer: 192.168.0.13:57451
Transfer time; 6us; total bytes transferred: 1032; peer: 192.168.0.13:57689
Transfer time; 4us; total bytes transferred: 1032; peer: 192.168.0.13:45875
Transfer time; 17us; total bytes transferred: 1032; peer: 192.168.0.13:51281
Transfer time; 15us; total bytes transferred: 1032; peer: 192.168.0.13:45847
Transfer time; 6us; total bytes transferred: 1032; peer: 192.168.0.13:41645
Transfer time; 5us; total bytes transferred: 1032; peer: 192.168.0.13:57687
Transfer time; 18us; total bytes transferred: 1032; peer: 192.168.0.13:41395
Transfer time; 24us; total bytes transferred: 1032; peer: 192.168.0.13:41637
Transfer time; 11us; total bytes transferred: 1032; peer: 192.168.0.13:58063
Transfer time; 25us; total bytes transferred: 1032; peer: 192.168.0.13:57741
Transfer time; 14us; total bytes transferred: 1032; peer: 192.168.0.13:57689
Transfer time; 7us; total bytes transferred: 1032; peer: 192.168.0.13:57687
Transfer time; 8us; total bytes transferred: 1032; peer: 192.168.0.13:41637
Transfer time; 10us; total bytes transferred: 1032; peer: 192.168.0.13:51281
^Cserver->start: Resource temporarily unavailable
Total served: 98701; Max concurrent connections: 33994
[root@datacomm server]# 
```

Epoll Server Performance

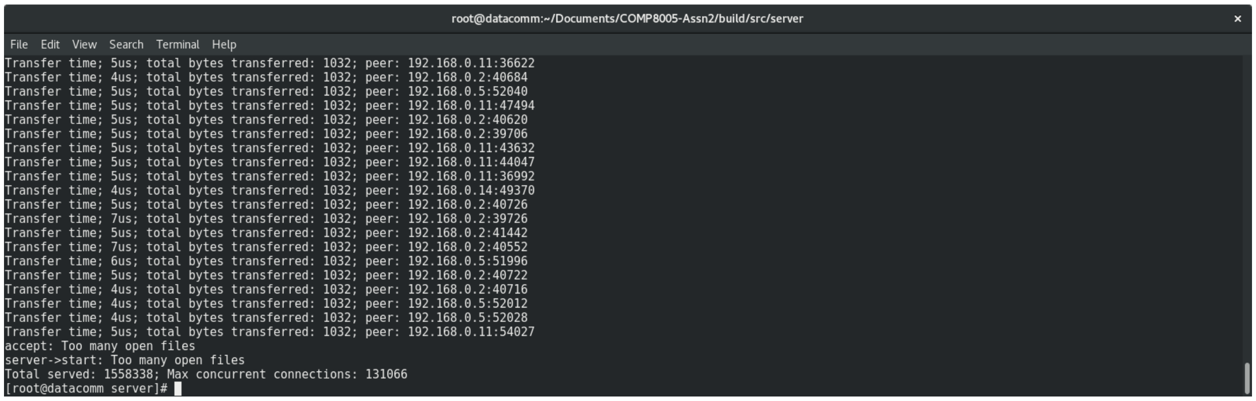
After completing our experiment and running tests against our ePoll server, we achieved excellent performance by sustaining a total of 131066 connections, beating out the select and multi-threaded server. The E-Poll server handled up to 131066 sustained connections using our client that was sending and receiving 1024 bytes of data with each request.

The results based upon the number of clients connected vs. time can be seen in the graph below.



By analyzing the graph you can conclude that all of the clients were connected to the server for a random time period between 1 – 10 microseconds; however there are random spikes in which the total time to send is greater than 120 microseconds. This was with 10 machines sending 1000 clients to the server. The large spikes are most likely caused by the AIO_WRITE as although it’s an asynchronous write, so many connections are occurring that it is still having issues printing fast enough to the file. Each client was designed to send data of 1024 bytes per request to the server. The client was designed to send these requests to the server for specified amount of time depending on a random number that is generated through the client script. Each client would then hold a connection for the specified amount of time while constantly sending and receiving data of 1024 bytes to and from the ePoll server.

The total number of connections that occurred was 1558338 before the server crashed.


root@datacomm:~/Documents/COMP8005-Assn2/build/src/server
File Edit View Search Terminal Help

```
Transfer time; 5us; total bytes transferred: 1032; peer: 192.168.0.11:36622
Transfer time; 4us; total bytes transferred: 1032; peer: 192.168.0.2:40684
Transfer time; 5us; total bytes transferred: 1032; peer: 192.168.0.5:52040
Transfer time; 5us; total bytes transferred: 1032; peer: 192.168.0.11:47494
Transfer time; 5us; total bytes transferred: 1032; peer: 192.168.0.2:40620
Transfer time; 5us; total bytes transferred: 1032; peer: 192.168.0.2:39706
Transfer time; 5us; total bytes transferred: 1032; peer: 192.168.0.11:43632
Transfer time; 5us; total bytes transferred: 1032; peer: 192.168.0.11:44047
Transfer time; 5us; total bytes transferred: 1032; peer: 192.168.0.11:36992
Transfer time; 4us; total bytes transferred: 1032; peer: 192.168.0.14:49370
Transfer time; 5us; total bytes transferred: 1032; peer: 192.168.0.2:40726
Transfer time; 7us; total bytes transferred: 1032; peer: 192.168.0.2:39726
Transfer time; 5us; total bytes transferred: 1032; peer: 192.168.0.2:41442
Transfer time; 7us; total bytes transferred: 1032; peer: 192.168.0.2:40552
Transfer time; 6us; total bytes transferred: 1032; peer: 192.168.0.5:51996
Transfer time; 5us; total bytes transferred: 1032; peer: 192.168.0.2:40722
Transfer time; 4us; total bytes transferred: 1032; peer: 192.168.0.2:40716
Transfer time; 4us; total bytes transferred: 1032; peer: 192.168.0.5:52012
Transfer time; 4us; total bytes transferred: 1032; peer: 192.168.0.5:52028
Transfer time; 5us; total bytes transferred: 1032; peer: 192.168.0.11:54027
accept: Too many open files
server->start: Too many open files
Total served: 1558338; Max concurrent connections: 131066
[root@datacomm server]#
```

Conclusion

Our final conclusions held up with what was expected regarding scalable servers. The ePoll server outperformed the select and multi-threaded server by sustaining 131066 connections. The select server was in the middle in regards to performance with 33994 connections and finally, the multi-threaded server had the least performance with 5767 connections. These results were expected and consistent through multiple tests.