Select/Multithreaded/Epoll Server Analysis

COMP 8005 - Assignment #2

Submitted by
Filip Gutica (A00781910)
Gary Khoo (A00564204)

Introduction

The continuing growth and development of Software as a Service platforms such as Google, Facebook and Twitter require servers to handle requests from an ever increasing number of computing devices which results in a large concurrent load on the server. In the earlier days of the internet, servers traditionally handled these requests by assigning a thread to each request, however this approach is constrained by the CPU/memory resources available on the server. In this report, we examine the scalability and performance of the traditional multithreaded server along with newer architectures based on the select and epoll system calls.

Design Methodology

To study the characteristics of these servers, we implemented different versions of an echo server using select, multithreading and epoll. A scalable client was also written in order to stress test our servers and gather data on the response time for each message sent to the server. We present the design of our servers and client in more detail in the following sections below.

Multithreaded Server

The multithreaded server works by spawning a thread every time a client connects to the server and was written using version 2.7.11 of Python. Figure 1 below details the server's operation at a high level.

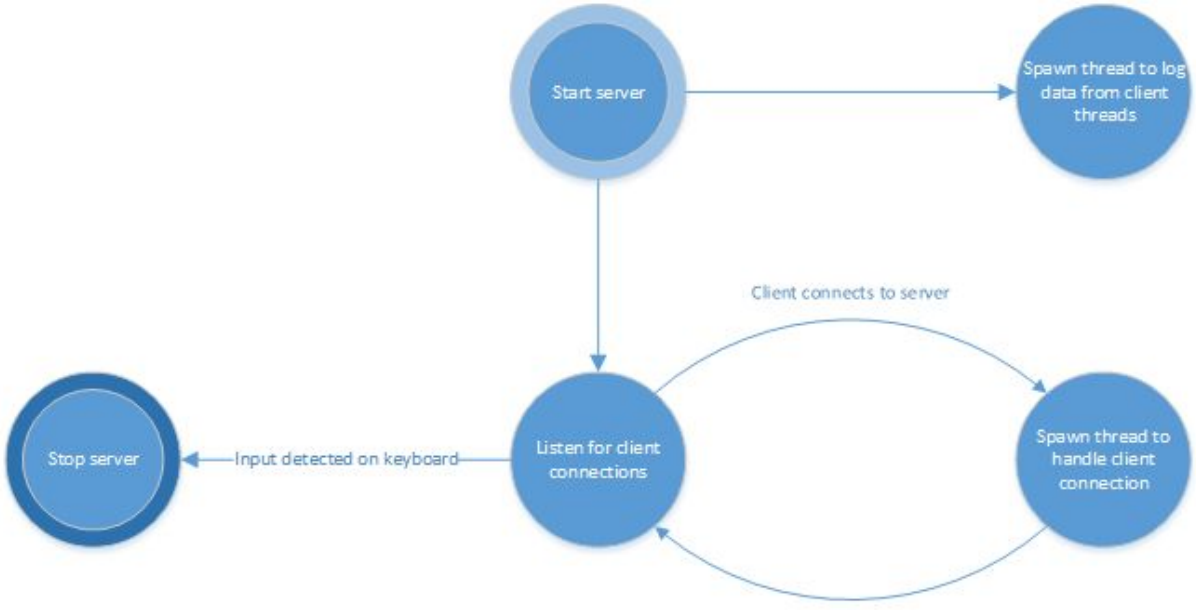


Figure 1: Multithreaded Server State Diagram

Each thread spawned by the server handles the job of receiving messages from the specific client and echoing them back. The thread does not terminate until it detects that the client has disconnected by reading an empty string from the network buffer. Figure 2 demonstrates the state machine behavior of a client thread spawned by the multithreaded server.

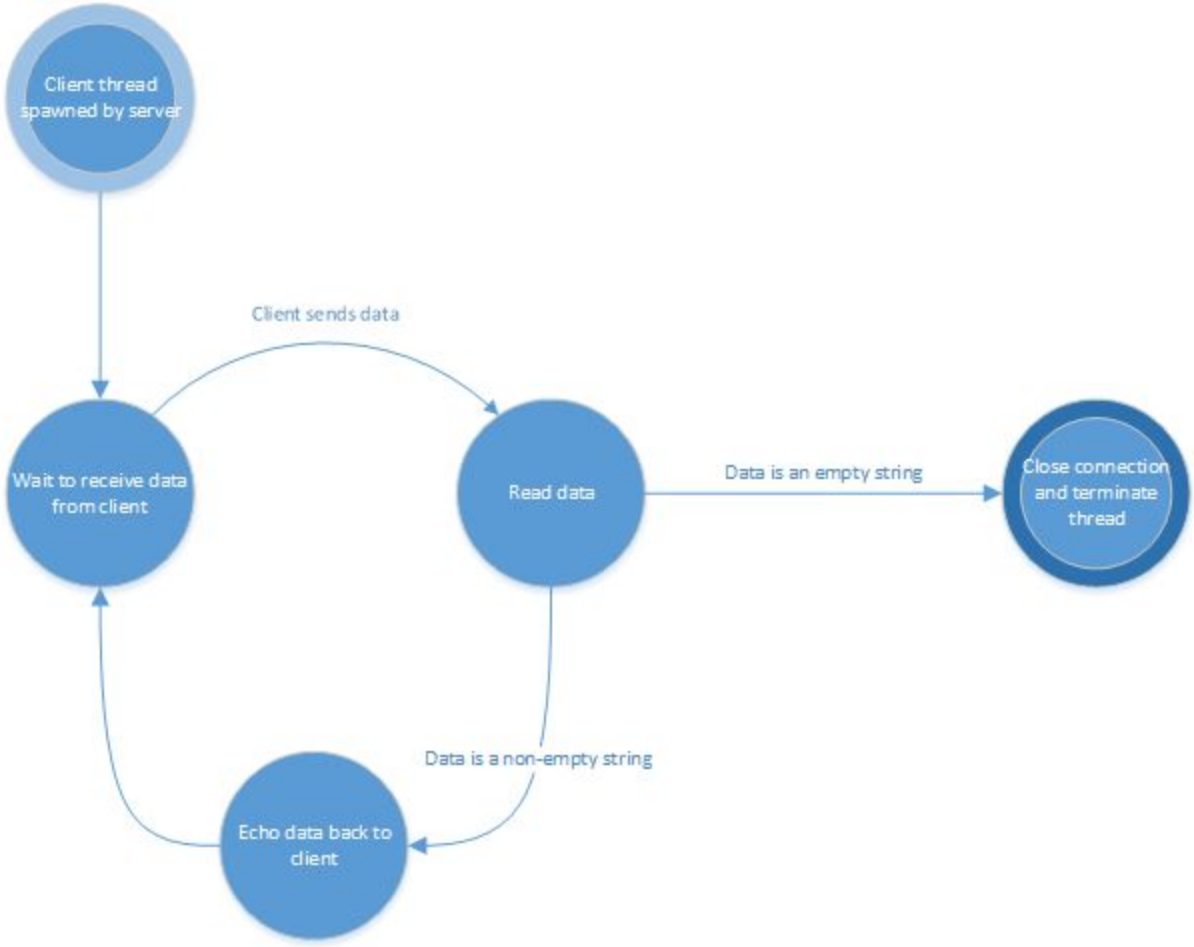


Figure 2: State Machine Diagram for Client Thread

The process of assigning one thread to handle one connection is simple, however we expect the multithreaded server to eventually fail once the thread limit from a process is reached [1].

Select Server

Instead of spawning a thread to handle each client connection, the select server monitors a list of descriptors for read/write/exception activity and notifies us via the select() call whenever any activity takes place. For the purposes of our echo server, we only need to monitor our sockets for read activity. The select server was also written using version 2.7.11 of Python. A diagram of the server's operation at a high level is detailed below.

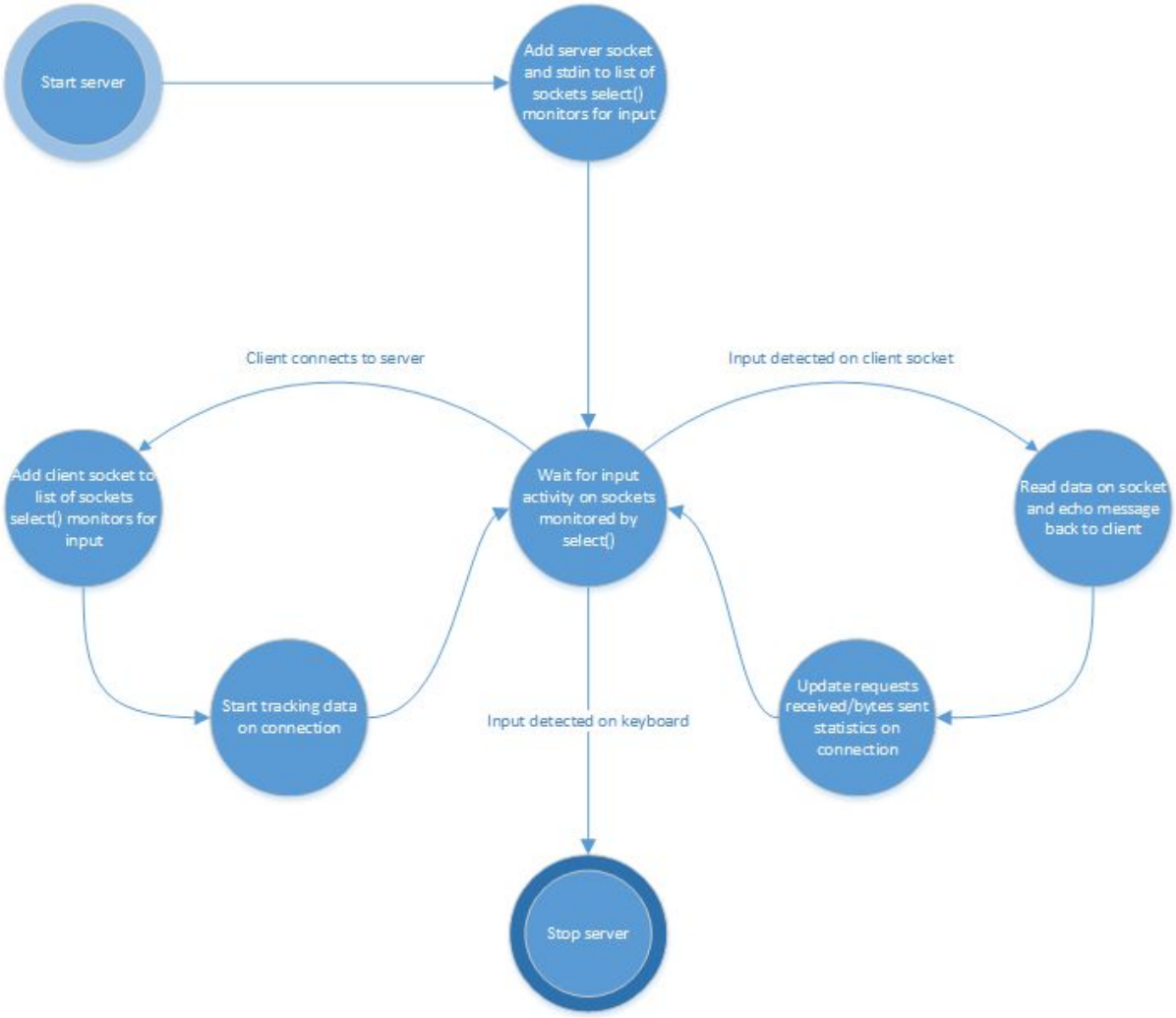


Figure 3: Select Server State Diagram

Epoll Server

Our Epoll server works by having one main thread with its own epoll event queue that only accepts new connections. We then have N worker threads, each with their own epoll event queue that check for the EPOLLIN and EPOLLRDHUP events in their event queues and respond accordingly. The main thread just passes on new file descriptors evenly to each worker thread to be processed.

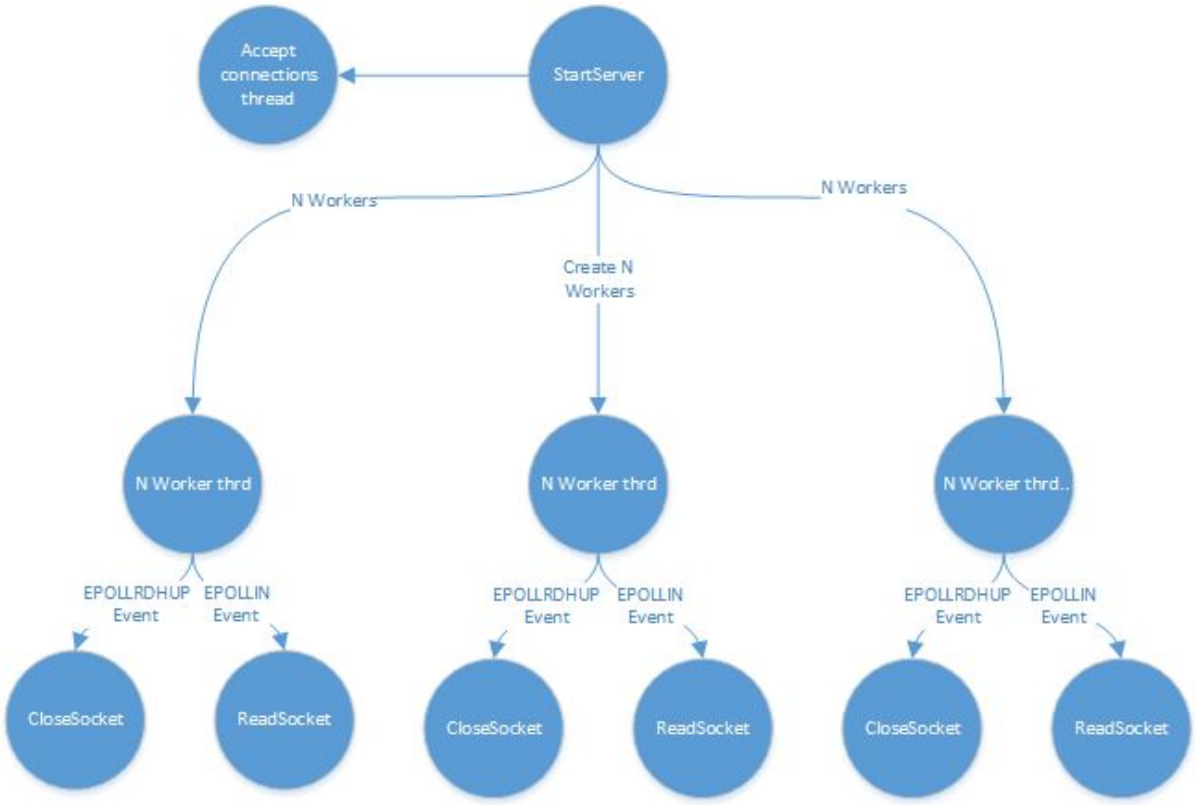


Figure 4: Epoll Server State Diagram

Client

To stress test our servers, an epoll-based client was written in Python. The client allows us to specify a message to send to the server, the number of times to send the message and the number of connections to load the server with. The efficiency of epoll allows one PC to generate up to 28000 connections (the ephemeral port limit on Linux) without running into CPU/memory utilization issues as we would with a traditional threaded client implementation. The high level operation of our epoll client is presented in the state diagram below

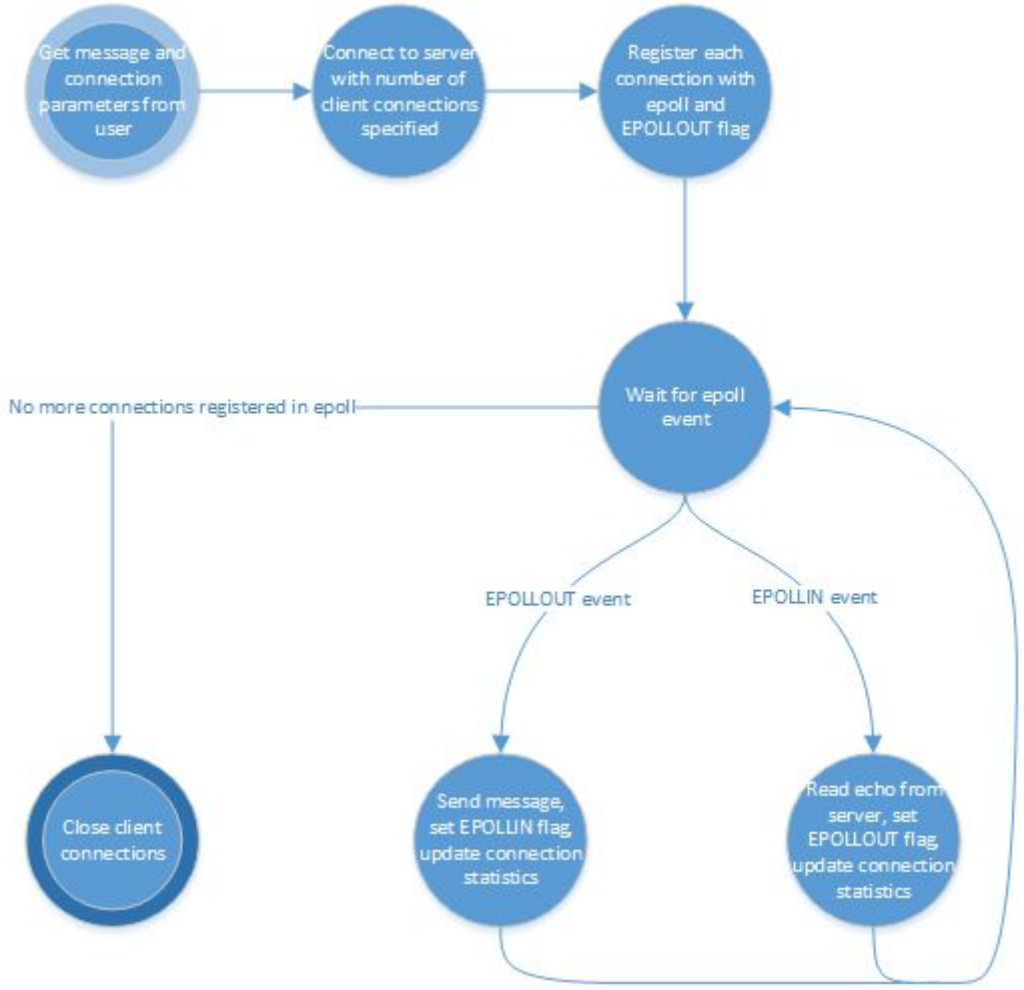


Figure 5: Epoll Client State Diagram

Testing Methodology

To test the limits of each server, we used our epoll client to establish an increasing number of connections until we stopped getting timely responses from the server, or observed a server-side/client-side error during our test run.

The following systems were configured and utilized in our test environment:

Server: Fedora 23 desktop with Intel Core i5 4670K and 8 GB RAM
Client #1: Fedora 23 VM with Intel Core i5 2500K and 4 GB RAM
Client #2: Fedora 23 desktop with Intel Core2Duo E8400 and 4 GB RAM

The network used to test the server was configured as follows in the diagram below:

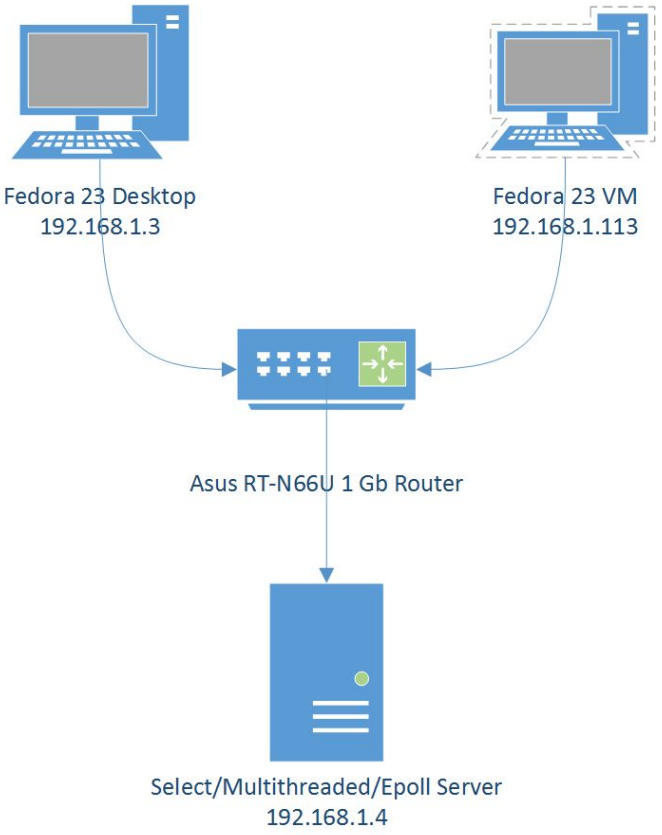


Figure 6: Network Diagram Used to Test Server

Each of the servers are started using the following commands (bolded) in a Linux terminal window:

python server_mt.py (multithreaded server)
python server_select.py (select server)
./server (epoll server)

On the computers that will connect to the servers, run the following command to increase the file descriptor limit on Linux. This will allow the epoll client to use the entire range of ephemeral ports for connecting to the server (which is about 28000 connections on Linux):

ulimit -n 1000000

After increasing the file descriptor limit, you will have to modify the HOSTNAME variable in the client_epoll.py file to match theh IP address that the server is running on. The clients can then be generated by running the epoll client with the following command:

python client_epoll.py

The epoll client will prompt you to provide a message to send to the server, the number of times to send it per connection and finally the number of connections it should make to the server. For our tests, we used the following values for each connection:

Message to send = **Gary's spamming**
Number of times to send message = **100**

After providing this information, the clients are generated, the messages sent and the server echos analyzed for their response time. Upon successful completion, the epoll client generates a .csv file with response time statistics for each client.

Test Cases

With the goal of determining the scalability and performance of the respective server architectures, we use our epoll client to increase the number of connections to each server until it performance degrades or the server fails outright. With this in mind, we used the following test cases on each server:

Test Case	Scenario	Expected Behavior	Actual Results	Status
1	1 client connecting to select server	No errors generated by server or client, client generates log file with data for connection	No errors observed on either side, log file generated successfully	Passed, see Results (1)
2	1000 clients connecting to select server	No errors generated by server or client, client generates log file with data for connection	No errors observed on either side, log file generated successfully	Passed, see Results (2)
3	2000 clients connecting to select server	No errors generated by server or client, client generates log file with data for connection	Error observed on server side	Failed, see Results (3)
4	1 client connecting to multithreaded	No errors generated by server or client, client generates log file with	No errors observed on either side, log file generated successfully	Passed, see Results (4)

	server	data for connection		
5	10000 clients connecting to multithreaded server	No errors generated by server or client, client generates log file with data for connection	No errors observed on either side, log file generated successfully	Passed, see Results (5)
6	30000 clients connecting to multithreaded server	No errors generated by server or client, client generates log file with data for connection	No errors observed on either side, log file generated successfully	Passed, see Results (6)
7	35000 clients connecting to multithreaded server	No errors generated by server or client, client generates log file with data for connection	Error observed on server side	Failed, see Results (7)
8	1 client connecting to epoll server	No errors generated by server or client, client generates log file with data for connection	No errors observed on either side, log file generated successfully	Passed, see Results (8)
9	10000 clients connecting to epoll server	No errors generated by server or client, client generates log file with data for connection	No errors observed on either side, log file generated successfully	Passed, see Results (9)
10	40000 clients connecting to epoll server	No errors generated by server or client, client generates log file with data for connection	No errors observed on either side, log file generated successfully	Passed, see Results (10)
11	45000 clients connecting to epoll server	No errors generated by server or client, client generates log file with data for connection	No errors observed on either side, log file generated successfully	Passed, see Results (11)
12	100000 clients connecting to epoll server	No errors generated by server or client, client generates log file with data for connection	No errors observed on either side, log file generated successfully	Passed, see Results (11)

Test Case 1 - 1 client connecting to select server

Running our epoll client with the default values and making one connection to the select server, we find that it finishes successfully without any error.

```
[root@Irisviel comp8005]# python client_epoll.py
message to send: Gary's spamming
number of times to send message: 100
number of clients to simulate: 1
[root@Irisviel comp8005]#
```

After running the test, the log file **client data - 1.csv** is also generated successfully and is included in the **data\select** folder of our submission. A preview of the log file is displayed below:

1	requests made	data sent (bytes)	average response time (sec)	session duration (sec)
2	100	1500	0.000201845	0.022844076
3				
4				

Test Case 2 - 1000 clients connecting to select server

Again running our epoll client with default values and making 1000 connections to the select server, we find that it finishes successfully without any error.

```
[root@Irisviel comp8005]# python client_epoll.py
message to send: Gary's spamming
number of times to send message: 100
number of clients to simulate: 1000
[root@Irisviel comp8005]#
```

After running the test, the log file **client data - 1000.csv** is also generated successfully and is included in the **data\select** folder of our submission. A preview of the log file is displayed below:

1	requests made	data sent (bytes)	average response time (sec)	session duration (sec)
999	100	1500	0.00958863	1.851197004
1000	100	1500	0.009585505	1.851024866
1001	100	1500	0.009546309	1.850842953

Test Case 3 - 2000 clients connecting to select server

When running our epoll client with default values and attempting 2000 connections to the select server, we find that the server terminates unexpectedly with the following error thrown:

```
[root@octavia servers]# python server_select.py
Server listening for connections on port 8005, press enter to shut down the serv
er
Traceback (most recent call last):
  File "server_select.py", line 105, in <module>
    startServer()
  File "server_select.py", line 59, in startServer
    inputReady, outputReady, errorReady = select.select(descriptorSet, [], [])
ValueError: filedescriptor out of range in select()
[root@octavia servers]#
```

The cause of this termination is due to the limit of 1024 file descriptors on the select system call. From the manpage of select:

> *An fd_set is a fixed size buffer. Executing FD_CLR() or FD_SET() with a value of fd that is negative or is equal to or larger than FD_SETSIZE will result in undefined behavior. Moreover, POSIX requires fd to be a valid file descriptor.*

Because FD_SETSIZE is set to 1024, attempting to monitor more than this number of sockets on a select server will lead to abnormal behavior as defined above.

Test Case 4 - 1 client connecting to multithreaded server

Running our epoll client with the default values and making one connection to the multithreaded server, we find that it finishes successfully without any error.

```
[root@Irisviel comp8005]# python client_epoll.py
message to send: Gary's spamming
number of times to send message: 100
number of clients to simulate: 1
[root@Irisviel comp8005]#
```

After running the test, the log file **client data - 1.csv** is also generated successfully and is included in the **data\mt** folder of our submission. A preview of the log file is displayed below:

1	requests made	data sent (bytes)	average response time (sec)	session duration (sec)
2	100	1500	0.000201845	0.022844076
3				

Test Case 5 - 10000 clients connecting to multithreaded server

When running our epoll client with the default values and making 10000 connections to the multithreaded server, we find that it finishes successfully without any error.

```
[root@Irisviel comp8005]# python client_epoll.py
message to send: Gary's spamming
number of times to send message: 100
number of clients to simulate: 10000
[root@Irisviel comp8005]#
```

After running the test, the log file **client data - 10000.csv** is also generated successfully and is included in the **data\mt** folder of our submission. A preview of the log file is displayed below:

1	requests made	data sent (bytes)	average response time (sec)	session duration (sec)
9999	100	1500	0.066057935	22.35024905
10000	100	1500	0.063443379	22.34981084
10001	100	1500	0.064776871	22.34972906

Test Case 6 - 30000 clients connecting to multithreaded server

Increasing the number of clients further to 30000, we find that the clients and server still finish successfully without any error. To load the server with this many connections, we ran our epoll client on two PCs to send 15000 connections each to the server.

```
[root@Irisviel comp8005]# python client_epoll.py
message to send: Gary's spamming
number of times to send message: 100
number of clients to simulate: 15000
[root@Irisviel comp8005]#
```

```
[root@localhost comp8005]# python client_epoll.py
message to send: Gary's spamming
number of times to send message: 100
number of clients to simulate: 15000
[root@localhost comp8005]#
```

After running the test, two log files are generated successfully and are included in the **data\mt\30000** folder of our submission. A preview of the log file is displayed below:

1	requests made	data sent (bytes)	average response time (sec)	session duration (sec)
29999	100	1500	0.349767077	58.29747915
30000	100	1500	0.346556573	58.28698587
30001	100	1500	0.349633324	58.27857995

Test Case 7 - 35000 clients connecting to multithreaded server

When using both our client computers to hit the multithreaded server with 35000 combined connections, the server terminated prematurely with the error message below:

```
^C[root@octavia servers]# python server_mt.py
Server listening for connections on port 8005, press CTRL + C to shut down the s
erver
Traceback (most recent call last):
  File "server_mt.py", line 82, in <module>
    startServer()
  File "server_mt.py", line 72, in startServer
    clientThread.start()
  File "/usr/local/lib/python2.7/threading.py", line 736, in start
    _start_new_thread(self.__bootstrap, ())
thread.error: can't start new thread
```

Based on the error message, we see that the server has exhausted the number of threads it is able to spawn to handle the incoming connections.

Test Case 8 - 1 client connecting to epoll server

Running our epoll client with the default values and making one connection to the epoll server, we find that it finishes successfully without any error.

```
[root@Irisviel comp8005]# python client_epoll.py
message to send: Gary's spamming
number of times to send message: 100
number of clients to simulate: 1
[root@Irisviel comp8005]#
```

After running the test, the log file **client data - 1.csv** is also generated successfully and is included in the **data\epoll** folder of our submission. A preview of the log file is displayed below:

1	requests made	data sent (bytes)	average response time (sec)	session duration (sec)
2	100	1500	0.000194433	0.022692919

Test Case 9 - 10000 clients connecting to epoll server

When running our epoll client with the default values and making 10000 connections to the epoll server, we find that it finishes successfully without any error.

```
[root@Irisviel comp8005]# python client_epoll.py
message to send: Gary's spamming
number of times to send message: 100
number of clients to simulate: 10000
[root@Irisviel comp8005]#
```

After running the test, the log file **client data - 10000.csv** is also generated successfully and is included in the **data\epoll** folder of our submission. A preview of the log file is displayed below:

1	requests made	data sent (bytes)	average response time (sec)	session duration (sec)
9999	100	1500	0.063514004	22.06160212
10000	100	1500	0.063748257	22.06145692
10001	100	1500	0.063904495	22.0612762

Test Case 10 - 40000 clients connecting to epoll server

Increasing the number of clients further to 40000, we find that the clients and server still finish successfully without any error. To load the server with this many connections, we ran our epoll client on two PCs to send 20000 connections each to the server.

```
[root@Irisviel comp8005]# python client_epoll.py
message to send: Gary's spamming
number of times to send message: 100
number of clients to simulate: 20000
[root@Irisviel comp8005]#
```

```
[root@localhost comp8005]# python client_epoll.py
message to send: Gary's spamming
number of times to send message: 100
number of clients to simulate: 20000
[root@localhost comp8005]#
```

After running the test, two log files are generated successfully and are included in the **data\epoll\40000** folder of our submission. A preview of the log file is displayed below:

1	requests made	data sent (bytes)	average response time (sec)	session duration (sec)
39999	100	1500	0.336204674	75.59377694
40000	100	1500	0.337494998	75.22705197
40001	100	1500	0.34675379	75.22671199

Test Case 11 - 45000 clients connecting to epoll server

Again increasing the load to 45000, we find that the epoll server still remains functional. At this point we are unable to load the server any further without more machines that we can use to run our epoll client on.

```
[root@Irisviel comp8005]# python client_epoll.py
message to send: Gary's spamming
number of times to send message: 100
number of clients to simulate: 22500
[root@Irisviel comp8005]#
```

```
[root@localhost comp8005]# python client_epoll.py
message to send: Gary's spamming
number of times to send message: 100
number of clients to simulate: 22500
[root@localhost comp8005]#
```

After running the test, two log files are generated successfully and are included in the **data\epoll\45000** folder of our submission. A preview of the log file is displayed below:

1	requests made	data sent (bytes)	average response time (sec)	session duration (sec)
44999	100	1500	0.239156041	51.15369487
45000	100	1500	0.2390803	51.15345097
45001	100	1500	0.23903815	51.15231395

Test Case 12 - 100000 clients connecting to epoll server

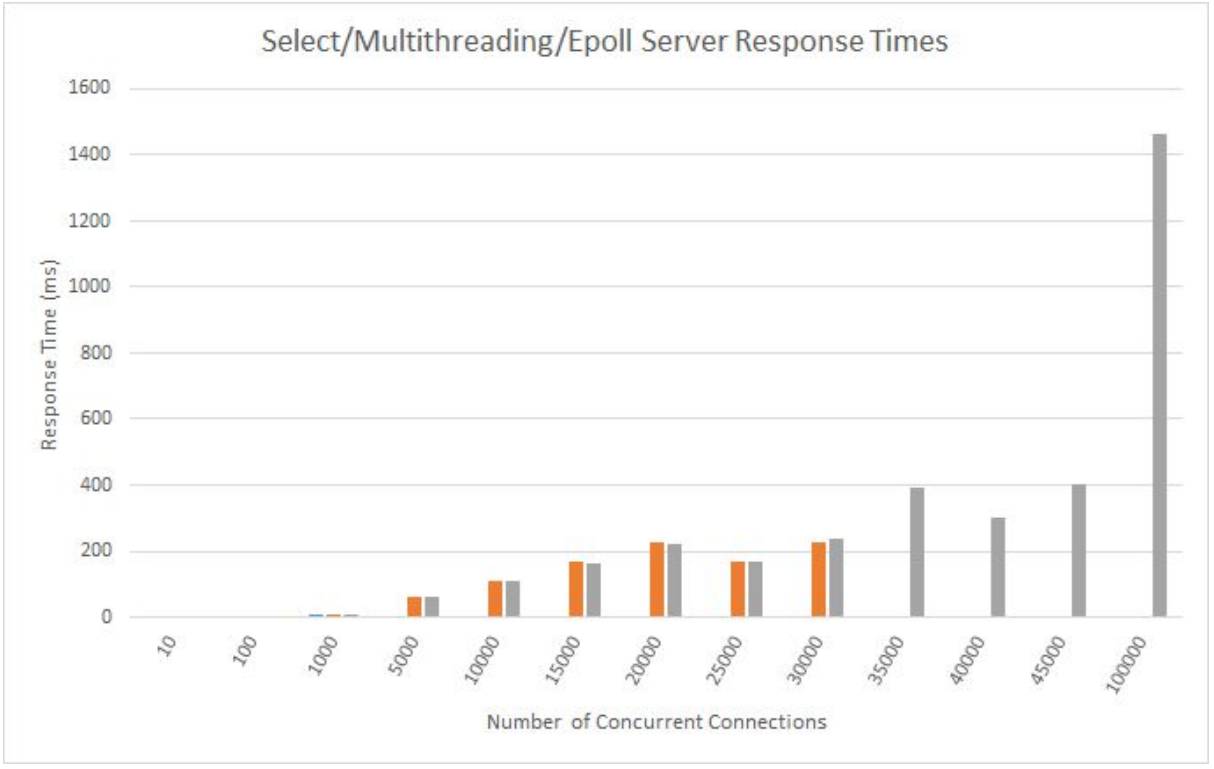
We were able to perform this test case during our demo with Aman thanks to the additional hardware available in the DataComm lab. Using four client computers with 25000 connections each, the epoll server was able to successfully process all connections without any errors.

Results

An analysis of the resulting log files from our tests gives the following average response time for each client and server scenario:

	Server Response Times (ms)		
Clients	Select	Multithreading	Epoll
1	0.185034275	0.201845169	0.194432735
10	0.231821775	0.223065138	0.213132381
100	0.913751769	0.893004775	0.918962336
1000	9.416260757	9.008485031	9.239154434
5000	Server error	61.57107982	60.87135549
10000	Server error	111.0927481	109.9073103
15000	Server error	168.8930116	163.146669
20000	Server error	225.193129	219.1963537
25000	Server error	168.2334311	170.1910665
30000	Server error	227.7007857	235.0338752
35000	Server error	Server error	390.2573484
40000	Server error	Server error	298.7459723
45000	Server error	Server error	403.5949744
100000	Server error	Server error	1461.298685

For comparison, these results are presented in bar graph format below:



Conclusion

After our tests it is clear that the most robust and scalable method of server design and implementation is the non-blocking, edge-triggered epoll method. The reason for this is that unlike the model of creating a thread for every client, with epoll, a single thread can handle 100's and even 1000's of clients with very little to no delay. This means that we can save a lot of CPU time since we do not have to switch between threads/processes as much.

Using this model we can just create N worker threads, depending on the architecture of the machine we are running the server on. For example, we can very easily increase the number of worker threads if the machine has more cores, thus giving the epoll server very much scalability.

When assigning a thread for every client, the performance can rapidly deteriorate the more clients connect as you can find yourself in the situation that you have 1000's of threads all doing I/O simultaneously completely pinning all the cores on your machine.

With select, it seems that we hit a physical limitation of FD_SETSIZE rendering the select model completely useless for scalable server design where you would want to be able to manage tens or even hundreds of thousand simultaneous connections.

Our results and conclusion all point to epoll being the best method for scalable server development. There are many real world services that are using epoll "under the hood" networking. A great example of a comparison between a service that does not use epoll and one that does is Nginx vs Apache. Apache uses the one-thread per client model while Nginx uses non-blocking asynchronous I/O on a single thread.
According to this article:
https://blog.webfaction.com/2008/12/a-little-holiday-present-10000-reqssec-with-nginx-2/
Nginx performance drops much slower as there are more concurrent connections, while Apache's performance drops much faster as there are more concurrent connections. These findings are in line with what we have noticed through our tests and further reinforce that non-blocking event driven i/o is the best model for scalable server design.

References

1) https://dustycodes.wordpress.com/2012/02/09/increasing-number-of-threads-per-process/
2) http://stackoverflow.com/questions/20915738/libevent-epoll-number-of-worker-threads
3) http://www.kegel.com/c10k.html
4) http://linux.die.net/man/4/epoll
5) http://man7.org/linux/man-pages/man2/epoll_ctl.2.html