2014

COMP 8005 – Assignment 2

Multi-thread, Select & E-poll Server

Shan Bains & Jivanjot Brar
BTECH - COMP 8005
2/21/2014

Abstract

The overall objective of this assignment is to create three servers, a traditional multi-threaded server, a select server and an e-poll server developed in your language of choice. For this assignment we chose to develop these three servers using the C programming language. Each server is able to handle multiple connections from clients. Testing of each server involved the use of many clients connected using multiple machines with the goal of overloading the server with sustained connections. During each test both the client and server store statistics related to the connection including IP addresses, data sent, data received, number of requests made ,etc. When the connection is closed from the client the relevant data is printed to a comma separated file (.csv) for analysis. When the connection to the server is closed a final report is printed on the screen. The traditional server was able to handle few thousand connections; however they were not long sustained connections. Where else select and epoll were easily able to surpass 10K connections with connection being sustained for random amount of time between 1 – 20 seconds. Select was able to sustain 18,000 connections with 400MB of total data received by the server and still managed to stay running. Epoll on the other hand sustained 20,024 connections without the server crashing with more than 800MB of total data received by the server.

Objective

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

Assignment Requirements

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 e-poll (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 have to 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 have to maintain their own statistics.

Constraints

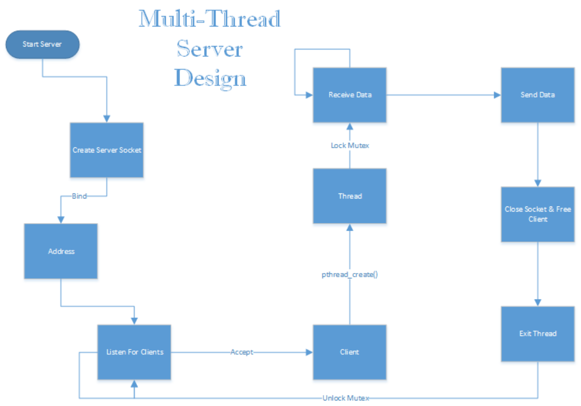
- The server will maintain a list of all connected clients (host names) and store the list together with the number of requests generated by each client and the amount of data transferred to each client.
- Each client will also maintain a record of how many requests it made to the server, the amount 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.

Experimental Environment

This experiment took place using a single server running an Intel Core i5 – 2400 CPU @3.10GHz with 4 cores. The server machine is equipped with 8GB of RAM and running on fedora 19. Experiments were performed using multiple client machines equipped with the same hardware and software as the server machine to ensure consistency between the results attained. The server will accept data packets of size 1024 bytes for communication between client and server. Each client will send data packets of size 1024 bytes to the server and the server will then respond by echoing back the received data to the client. The experiment will take place with multiple machines connecting to the same server with multiple clients running on the same machine. Each client will send data packets of size 1024 bytes to the server and stay alive between a random time intervals between 1 – 6 seconds inclusively.

Multi-threaded Server 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 different threads while the entire application is isolated to only fully utilize one CPU core/process. 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 E-poll. The multi-thread 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 by locking the mutex from outside processes or threads then start receiving data and when the data is received and stored into the appropriate structures it is sent back to the client as a confirmation.



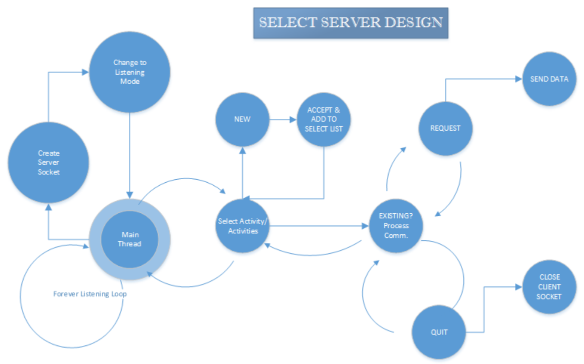
TCP Client for Multi-Thread Server

The Client used for testing the multi-threaded sever is designed using multiple processes to connect with the server simultaneously. The main client forks new child processes and performs an execl command to program the child to connect to the server for specified number of seconds ranging between 1 to 20 seconds. The child process 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 the child. The main process after forking waits for the child processes to finish before exiting itself. The figure below shows the design of the client.

Select Server Design

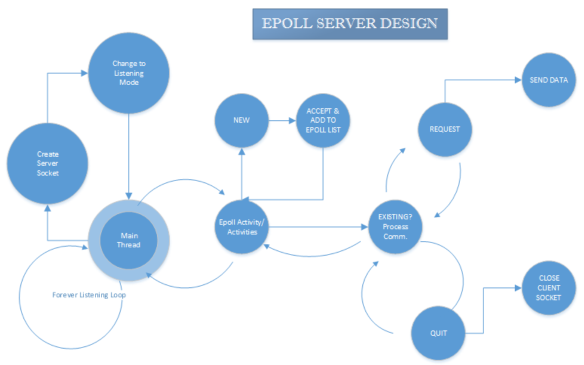
This server, instead of traditional multi-threaded approach, uses a system call called select. Select call is implemented at the kernel level. Select function takes the list of socket descriptors and monitors them for any activity. Select is a blocking call, and therefore it puts the process or thread 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 is designed to use a single thread used as a client manager. The client manager thread 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.



E-poll Server Design

E-poll server is designed in the similar manner as the select server and works in the similar manner as the select server. Figure below displays the design of the e-poll server.



The minor difference between the two calls is in the e-poll syntax itself. For e-poll, first we need to create an e-poll descriptor, and then initialize the e-poll struct with wanted events. Afterwards we need to add all the descriptors to the e-poll 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. E-poll similar to select returns number corresponding to the active descriptors and then we need to iterate through the list and handle the activity on the descriptors.

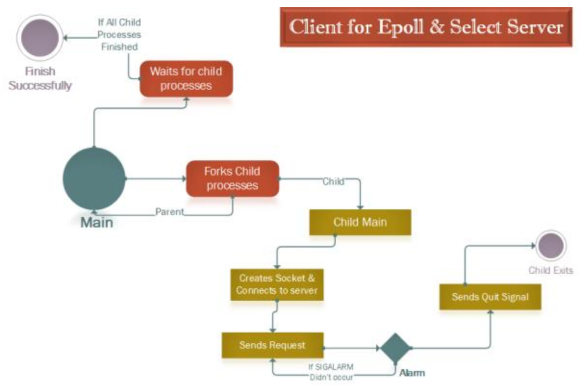
Advantages of using e-poll over select

E-poll has some significant advantages over *select/poll* both in terms of performance and functionality.

- *E-poll* 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 e-poll queue with *epoll_wait*(), something you cannot do with *select/poll*.

TCP Client for Select and E-poll Server

Client used for testing the select and e-poll sever is designed using multiple processes to connect with the server simultaneously. The main client forks new child processes and performs and execl command to program the child to connect to the server for specified number of seconds ranging between 1 to 20 seconds. The child process goes into an infinite loop for that many seconds and keeps sending the server data request. When the time expires, the client receives a SIGALARM event which turns off the forever loop, in which case it sends a quit signal to the server and closes the sockets. The main process after forking waits for the child processes to finish before exiting itself. The figure below shows the design of the client.

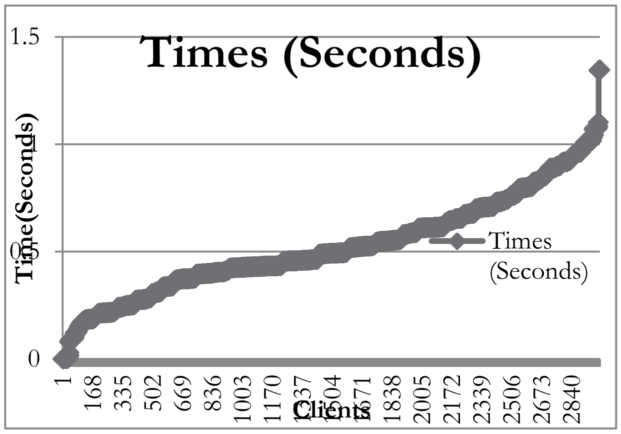


Results

Multi-threaded Server Performance

After completing our experiment and running tests against our multi-threaded server we were able to achieve acceptable performance with over 3100 sustained connections. The multi-threaded server was able to handle up to 3100 sustained connections using our TCP client that was sending and receiving 255 bytes of data with each request. The test was performed using two machines that were both running our TCP client that sent 255 bytes of data to the server. The two machines simultaneously sent and received data from the multi-threaded server that was developed in C. The results based upon the number of clients connected vs. time can be seen in the graph below.

Figure 1 - Client run time vs. number of connected clients



The multi-threaded server was setup to handle multiple client connections by allowing clients to connect multiple times on the same socket. Our TCP client was only designed to make one connection at a time so the client 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.

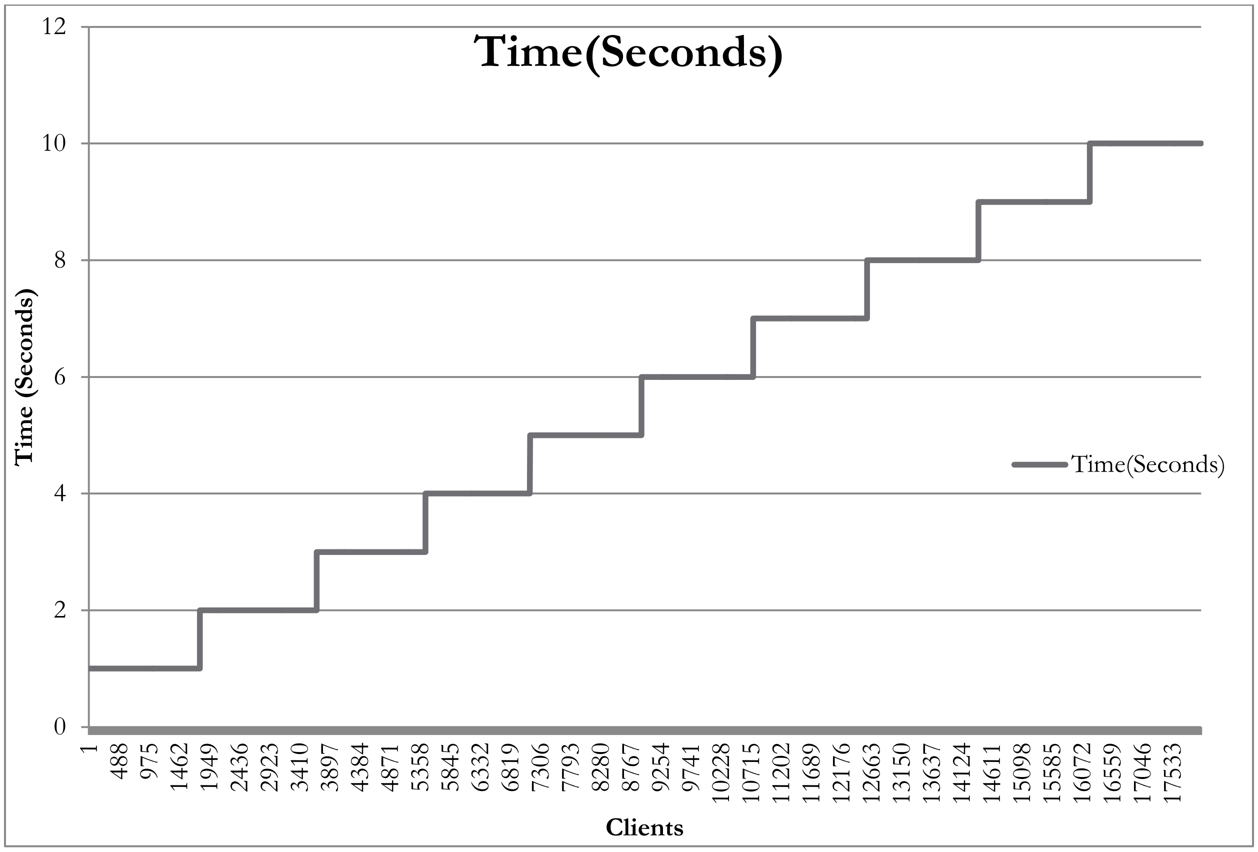
Select Server Performance

After completing our experiment and running tests against our select server we were able to achieve excellent performance by sustaining a total of 18,000 connections. The select server was able to handle up to

18,000 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 TCP client that sends and receives 1024 bytes of data from the server. The two machines were able to simultaneously send and receive data from the select server that was developed in C.

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

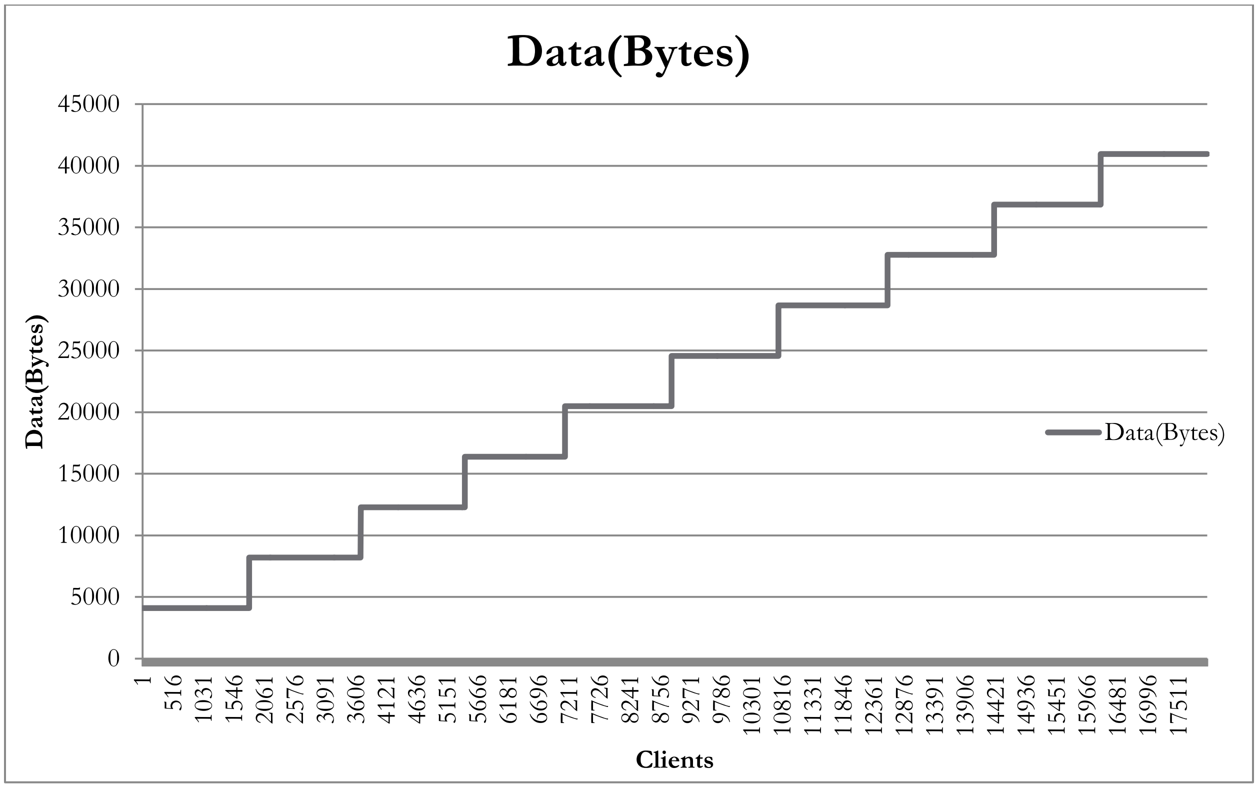
Figure 2 – Client run time vs. number of connected clients



The graph above has been sorted by client run time so it will naturally display a stair case type of design. 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 seconds inclusively. 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 select server.

The results based upon the number of data (bytes) received vs. the number of clients connected can be seen in the graph below.

Figure 3 – Data Received in bytes vs. number of connected clients



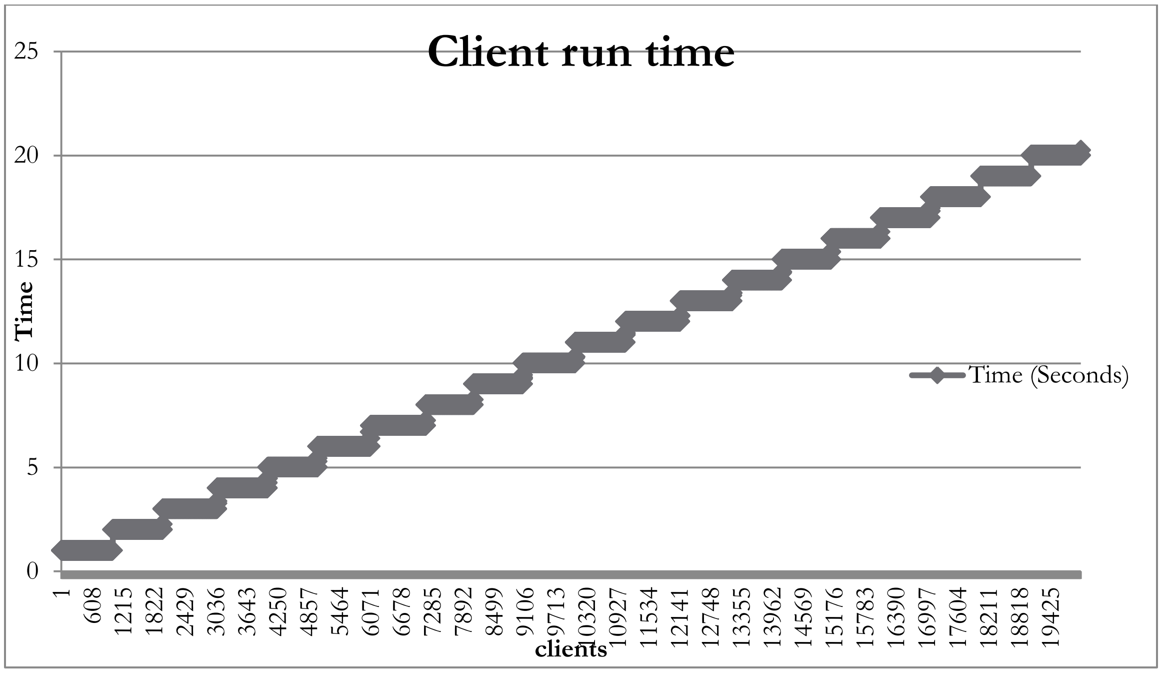
The graph above has been sorted by the number of bytes received so it will naturally have a stair case type of design. By analyzing the graph above you can conclude that all the connected clients transferred between 1024 bytes – 85000 bytes depending on how long the client was connected to the server. Each client sends a set amount of data (1024 bytes) to the server with each request and then the server sends the same amount of data back to the client. The amount of data varies for each client based on how long the sustained connection took place. Each client would hold a connection for a random amount of time which is between 1 – 10 seconds depending on the number that was generated through the client script. The data varies because as each client holds a longer connection more data is sent and received from the select server. The client would send data at a rate of once every 0.025 seconds, when the data is sent to the server the amount of data sent would be increased.

E-Poll Server Performance

After completing our experiment and running tests against our E-Poll server we were able to achieve excellent performance by sustaining a total of 18,000 connections. The E-Poll server was able to handle up to 18,000 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 TCP client that sends and receives 1024 bytes of data from the server. The two machines were able to simultaneously send and receive data from the E-Poll server that was developed in C.

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

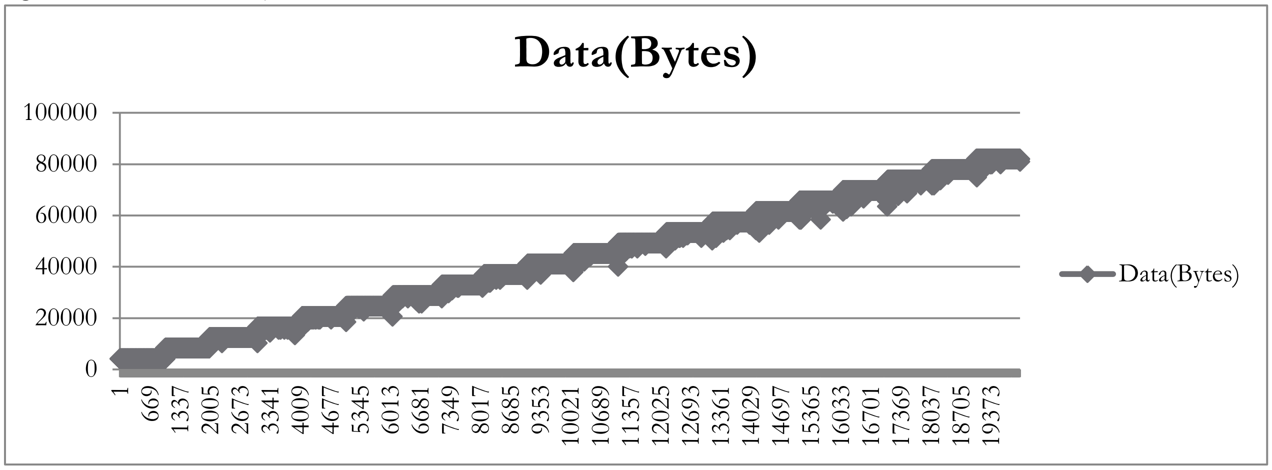
Figure 4 – Client run time vs. number of connected clients



The graph above has been sorted by client run time so it will naturally display a stair case type of design. By analyzing the graph you can conclude that all of the clients were connected to the server for a random time period between 1 – 20 seconds inclusively. 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 e-poll server.

The results based upon the number of data (bytes) received vs. the number of clients connected can be seen in the graph below,

Figure 5 – Data Received in bytes vs. number of connected clients



The graph above has been sorted by the number of bytes received so it will naturally have a stair case type of design. By analyzing the graph above you can conclude that all the connected clients transferred between 1024 bytes – 85000 bytes depending on how long the client was connected to the server. Each client sends a set amount of data (1024 bytes) to the server with each request and then the server sends the same amount of data back to the client. The amount of data varies for each client based on how long the sustained connection took place. Each client would hold a connection for a random amount of time which is between 1 – 20 seconds depending on the number that was generated through the client script. The data varies because as each client holds a longer connection more data is sent and received from the e-poll server. The client would send data at a rate of once every 0.025 seconds, when the data is sent to the server the amount of data sent would be increased.