# Network Programming
# Assignment #2

## Design Decisions

**Date Submitted:**
**April 24, 2019**

**Vedant Patwary (2016A7PS0031P)**
**Kunal Jain (2016A7PS0022P)**

### Part 1

Deliverables included: multithread.c, eventdriven.c, client.c, makefile

How to run:

$ make

a. Event-driven server

$ ./eventdriven

b. Threaded server

$ ./threadedserver

c. Client

$ ./client <N> <M> <T>

Constraints:

1. Maximum size of message can be 50.
2. Client can have a name of maximum length 40.

Working of the code:

- Run any of the server (./eventdriven or ./threadedserver)
- Then it will prompt the user to enter a port number.
- Enter any free port number and continue.
- To use the server,
    - **Telnet:**
        - Use the command telnet <ip> <port>
        - JOIN <name>
        - LIST
        - UMSG <dest. Name> <msg> : press enter to send
        - BMSG <msg> : press enter to send
        - LEAV
    - **Client:**
        - Run ./client <N> <M> <T> : ex. $ ./client 10 10 20
        - The client will print all the tasks it is doing
        - The client will also print the time taken for it to complete the execution
- To exit the server, send SIGINT (Ctrl + C). The server closed successfully.
- To run the server again, the user can use the same port again.

Implementation:

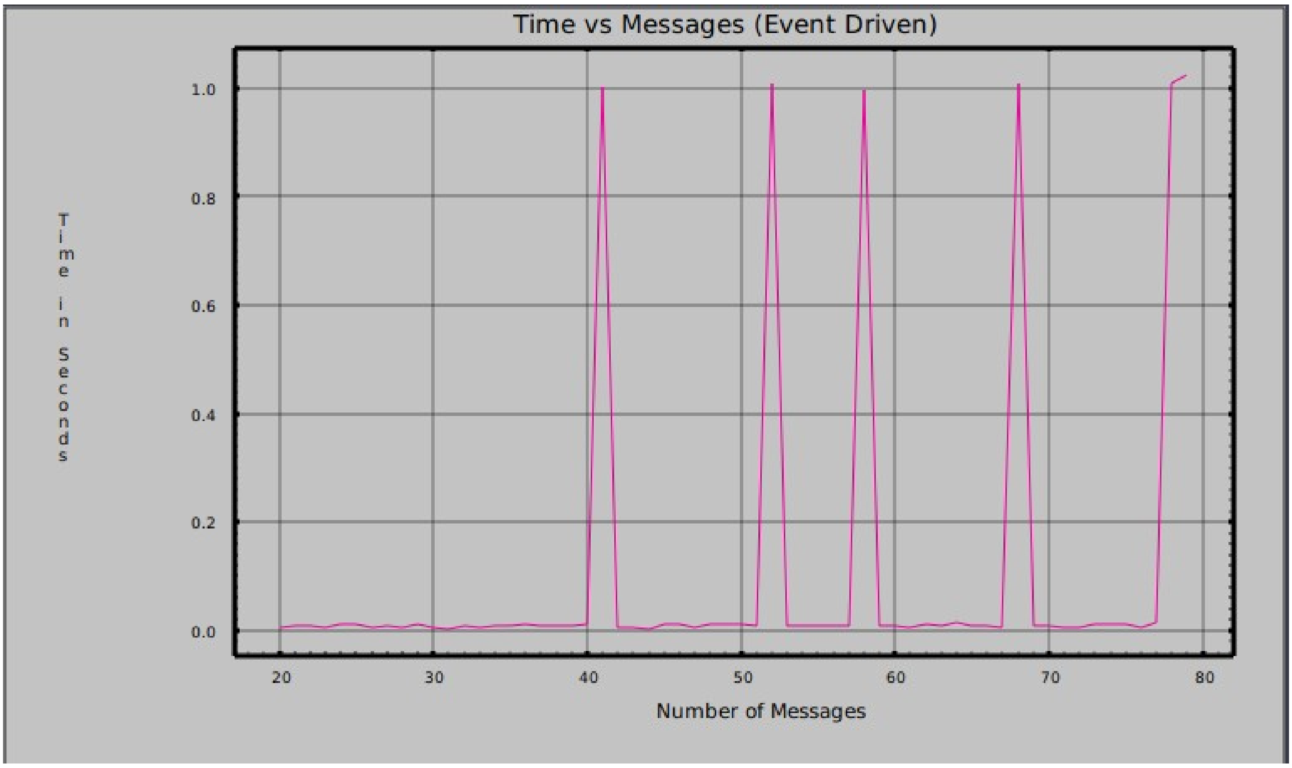
- **Event-driven:**
    - A listening socket is created to accept connections. *epoll_wait* waits for a client to connect to this socket.
    - When listening fd is readable, accept a client into *connfd*.
    - Make the *connfd* non blocking.
    - Add this *connfd* to epoll's read interest list.
    - Whenever client sends some data, this *connfd* becomes readable along with all the other connfds (of other clients, if they send some data at the same time).
    - Iterate over the ready list to find out those *connfds* that are ready to be sent to the *reading* event.
    - Once all the fds are queued into the *reading* event, we will process all the queued events at once. Now these events are queued in the *processing* event queue.
    - All the events queued in the above mentioned *processing* queue are now processed and if they're
        - JOIN: Inserting the details of the new client (*connfd* and *client name*) into a globally declared linked list.
        - LIST: Iterates over this linked list and stores list of names of all the clients online.
        - UMSG <dest name> <msg> : Finds the fd of the client with name *dest name* and stores the *msg* and *connfd*.
        - BMSG <msg> : Iterates over all the clients in the global linked list and stores their fds alongwith the *msg*.
        - LEAV: Deletes the entry for the source client from the global linked list.
    - and after storing, it queues the packet to be sent in the *writing* queue.
    - All the events queued in *writing* event queue are sent to the desired clients.
- **Threaded**
    - Same data structure is used from event-driven server.
    - We are using the child-thread-accept model.
    - A listening socket is created and *select* waits on this fd to be readable.
    - Whenever this fd is readable, a new client initiated a connection, so a new thread is made.
    - This thread accepts the connection into a *connfd* and uses this fd to communicate with the client.
    - A while(1) loop is used to read the data, process it and write back to client.
    - A conceptual problem arises when accessing the shared global linked list. The server should allow multiple clients to send UMSG, BMSG and LIST but only one client at a time is allowed to use JOIN and LEAV (as they are changing the linked list). This problem is an example of ***reader-writer*** problem.

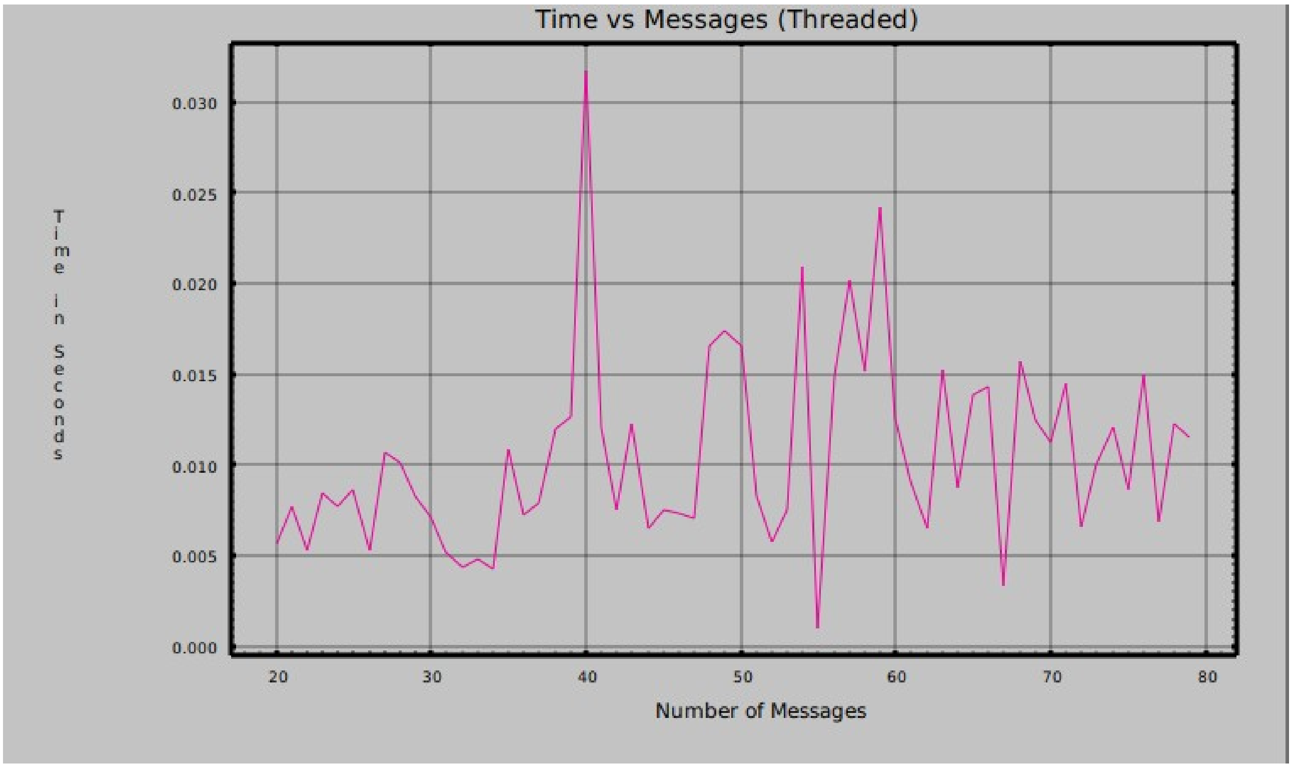
- To solve this, we used 3 shared variables:
    - *writing:* is one if someone is writing (using JOIN/LEAV)
    - *writers:* number of threads waiting to change and currently changing the linked list
    - *reading:* number of threads currently reading (using BMSG, UMSG, LIST)
- 1 mutex:
    - *mutex:* to introduce exclusion and synchronization between the shared variables.
- 1 conditional variable:
    - *cond:* to wake the waiting threads
- these all variables and mutexes are used to implement the solution to the problem-
    - The writer waits until there is no thread *reading* or *writing* currently
    - The reader waits if there are any *writers* present

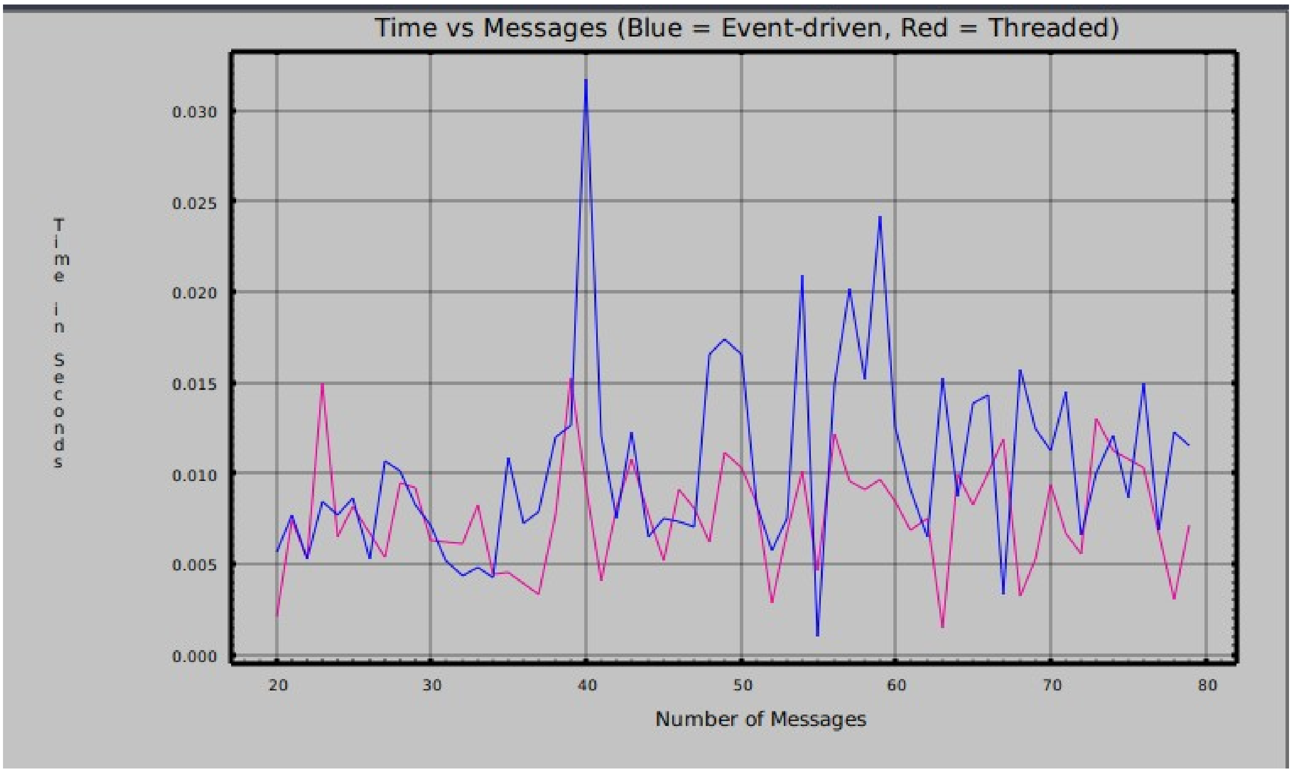
- **Client**
    - Takes N, M, T as CLA
    - Takes port number and then IP address as input.
    - Uses <u>pre-forking</u> to create N processes.
    - A variable maintains a count of children who have spawned to communicate with server. This variable is changed via SIGCHILD signal in parent process.
    - Whenever there is space to spawn, the process connects to the server and starts to communicate
    - send JOIN message alongwith the client number.
    - sends M number of UMSGs to itself
    - send LEAV and then terminates
    - prints the time taken to do all the tasks.

## Comparative Analysis:

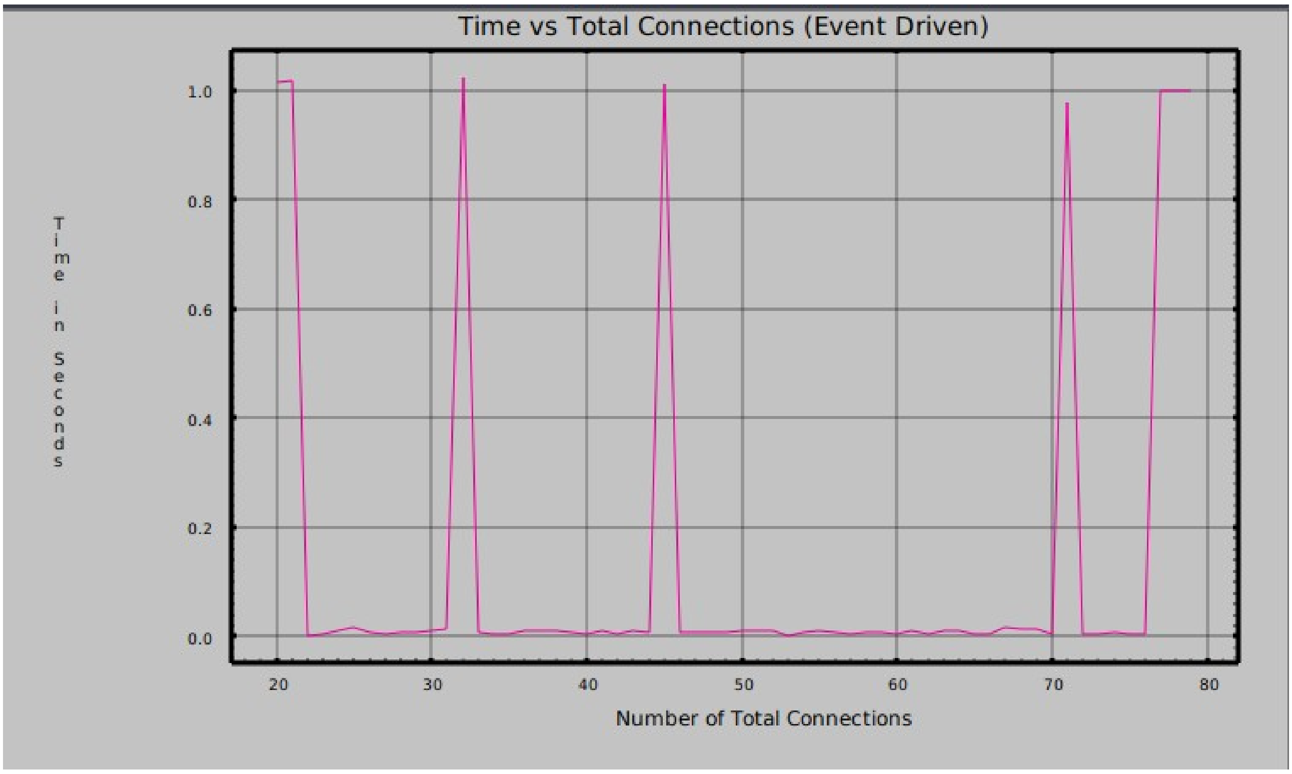
### Time vs Number of Messages

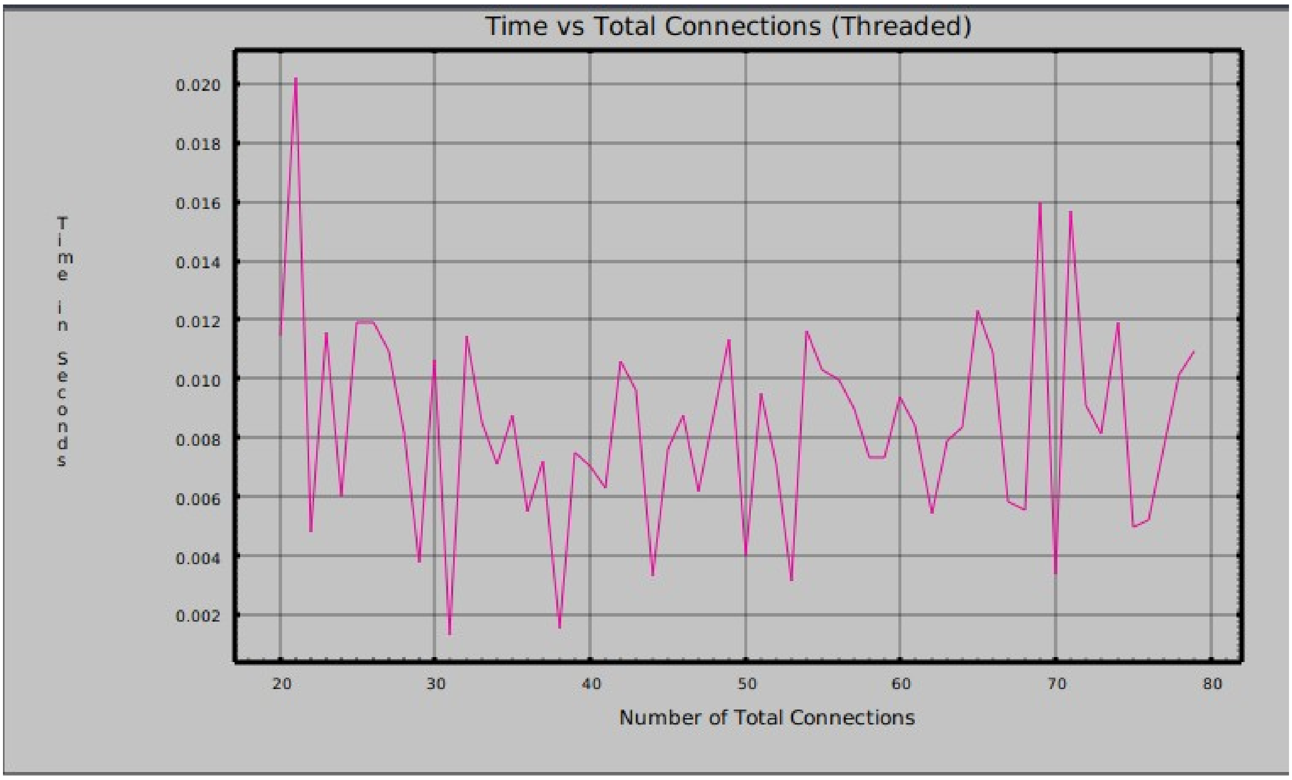


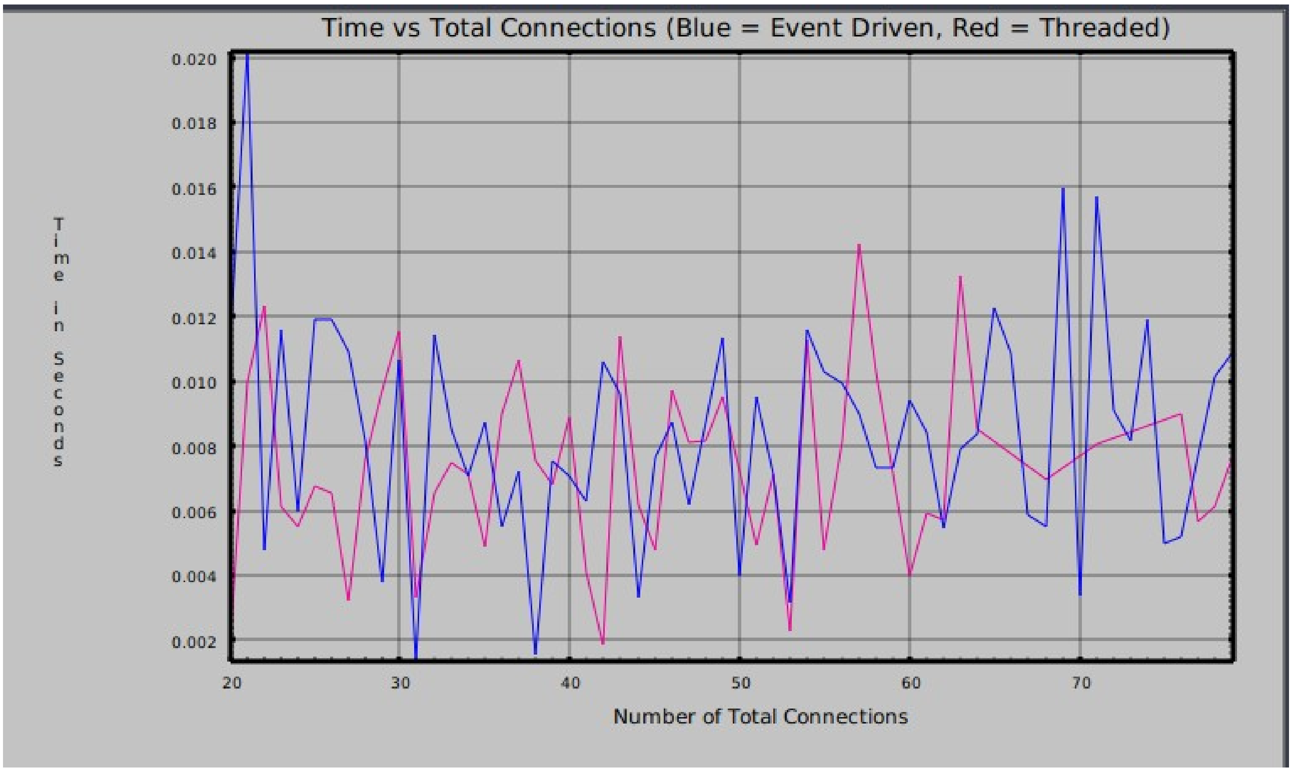




Time vs Messages (Blue = Event-driven, Red = Threaded)
Time in Seconds
0.000
0.005
0.010
0.015
0.020
0.025
0.030
20
30
40
50
60
70
80
Number of Messages

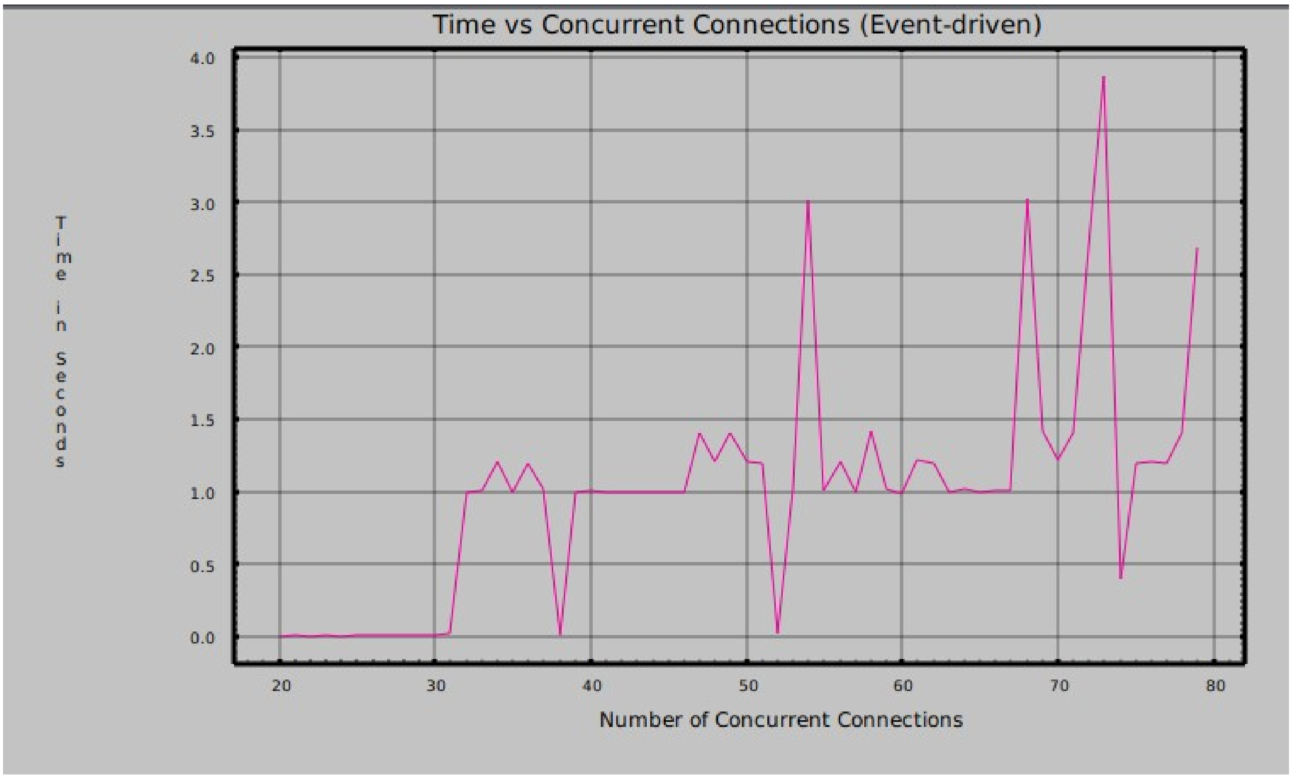
<u>Time vs Total Connections</u>

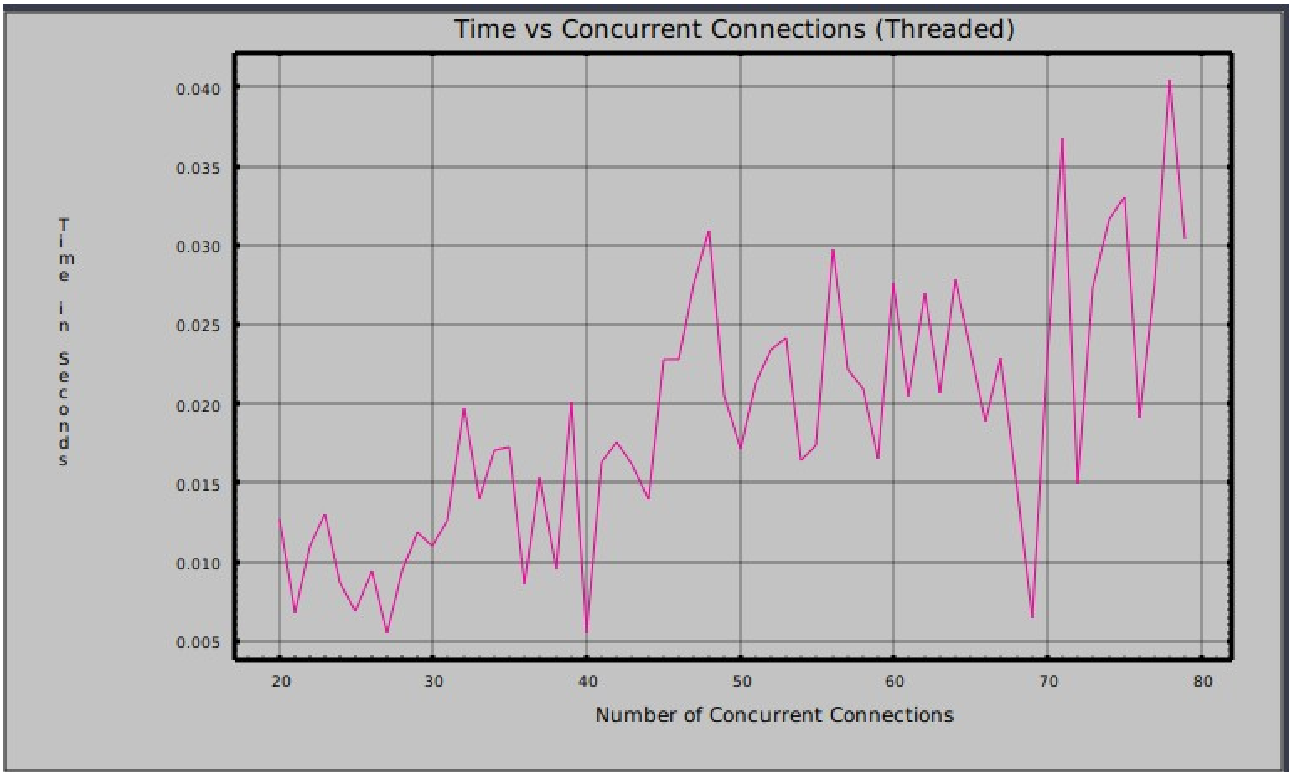


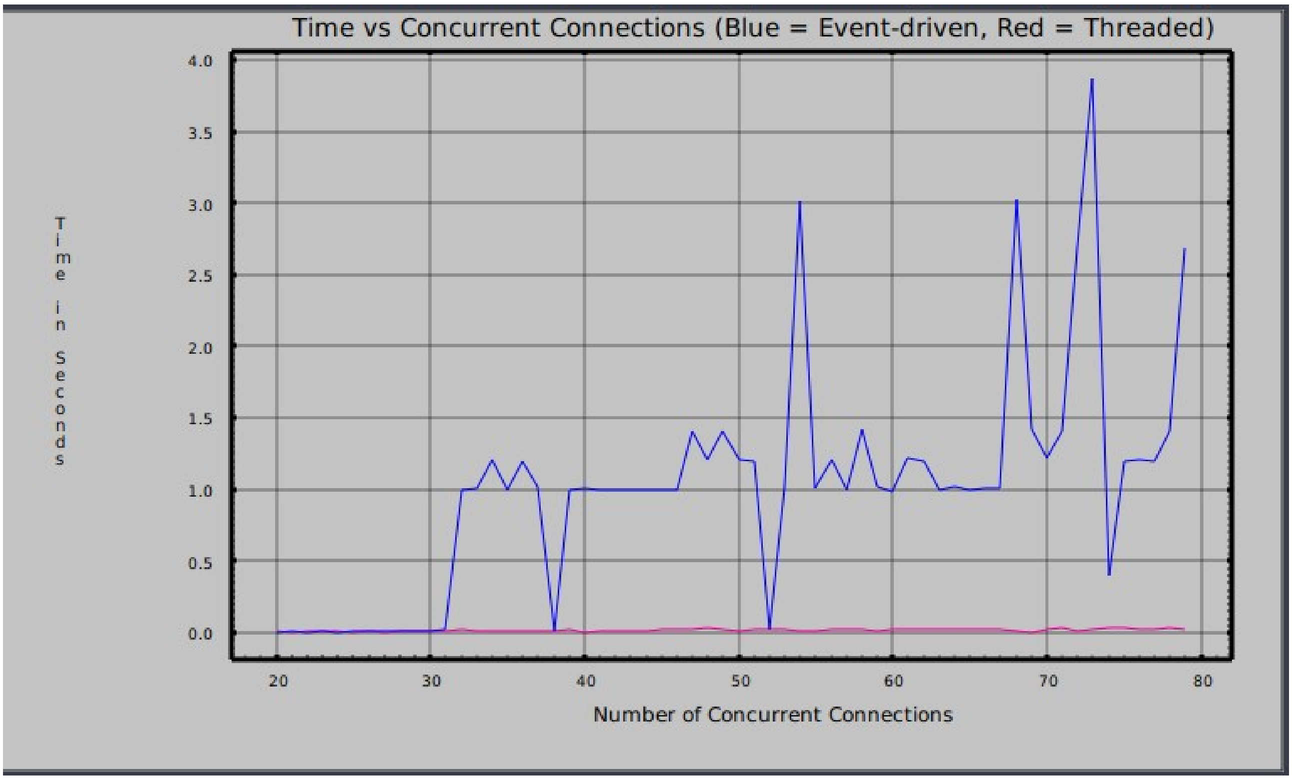




Time vs Total Connections (Blue = Event Driven, Red = Threaded)
Time in Seconds
0.020
0.018
0.016
0.014
0.012
0.010
0.008
0.006
0.004
0.002
20
30
40
50
60
70
Number of Total Connections

# Time vs Concurrent Connections






<u>Conclusion:</u>

It is evident from the comparision given above that changing the number of concurrent connections has most impact on the time taken to serve the clients. Also, threaded server seems to out-perform event-driven server in case 1 and 3 (not in total connections where both give similar result).

---X---