# 

# Concurrent Socket Server

---

Andrew Sullivan and Dylan Vance

CNT4504 Computer Networks

Professor Kelly

April 19th, 2024

---

**Introduction**

To begin, the concurrent socket server project is one meant to challenge students' understanding of client-server configurations through the utilization of a concurrent, multi-threaded server. The main goal of this project was to analyze the performance of a mult-threaded server that is handling multiple client requests concurrently. In this report, we will discuss the implementation and analysis of a concurrent server project, starting with our client-server configurations, then data collection and analysis, our conclusions, and finally the lessons learned through this process.

**Client-Server Setup and Configuration**

The client-server setup for this project involves the development of a multi-threaded server and client, each designed to handle concurrent operations efficiently. There were many design decisions made throughout this process. For instance, we chose to code the client and server in Java due to the socket library and the threading capabilities of Java that made for a more streamlined process. The basic operations of our code are as follows: the server listens for a client to connect and after a stable connection, prompts the user for a desired command. After, the client program creates multiple client threads, each responsible for sending a request to the server. Next the server spawns a new server instance for each client request in parallel in order to process the requests and send it back to the client. Finally, the client threads receive the server response and display the output to the user.

**Testing and Data Collection**

We collected elapsed time data on each operation by adding in code into the client application and the client thread class that would record and print out the total elapsed time, the average elapsed time, and the elapsed time of each individual thread, all recorded in milliseconds. The average elapsed time was calculated by dividing the total elapsed time by the number of requests. The average elapsed time is rounded down to the nearest integer because our code uses integer division on the calculation to save extra lines and to print cleaner looking data. This makes the average elapsed time a rough estimation to the nearest integer.

During the period when this data was recorded, there were five users on the class server, not including our own use of the server. It is unknown what effect these users had on our elapsed time calculations, but we assume that our data paints an accurate picture of the trends of the different server operations and that any outlying data points were caused by other users affecting the server, speeding up or slowing down the operations we ran.

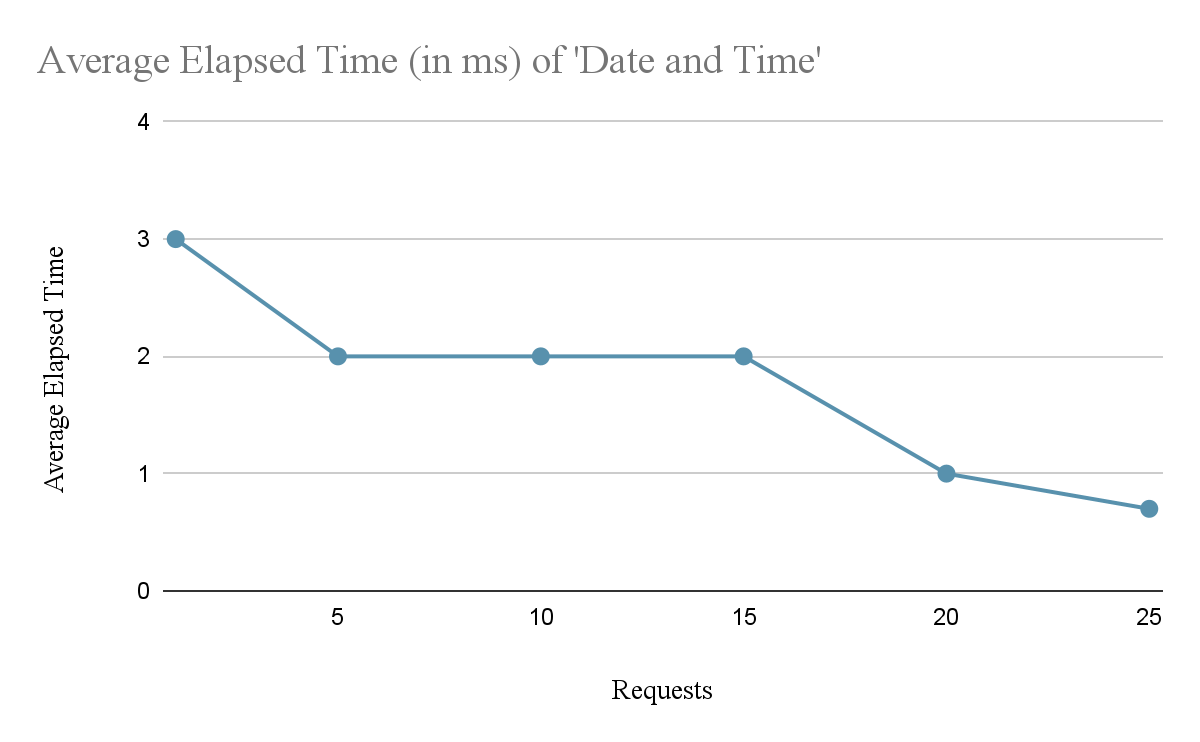
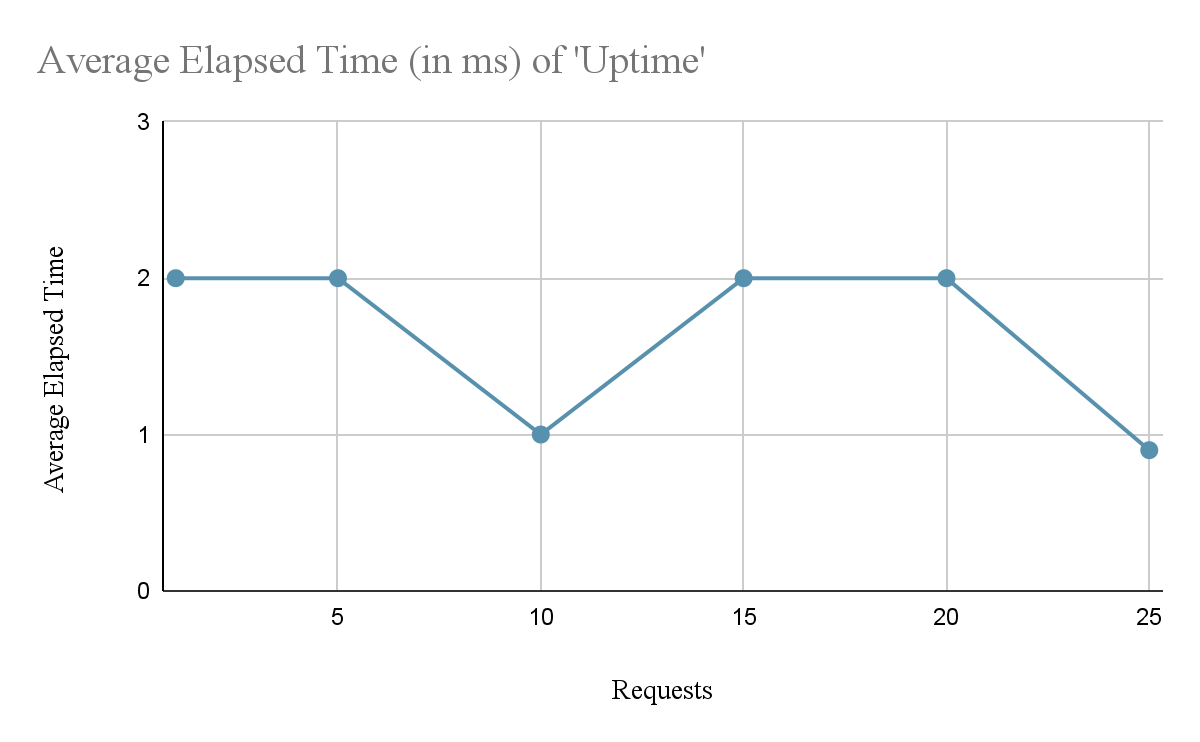
The average elapsed time of the six different server operations can be seen below in Figure 1; the total elapsed time of each operation can be seen below in Figure 2. Because of how we implemented the timer functionality into our code, the total elapsed time does consider the time it takes to print the operation’s output to the client’s terminal. For the ‘NetStat’ and ‘Running Processes’ operations, this had a clear effect on the total elapsed time because these two operations, ‘NetStat’ especially, print out a lot of lines. However, for the other operations, the time taken to print its output to the client has virtually no effect on the total elapsed time.

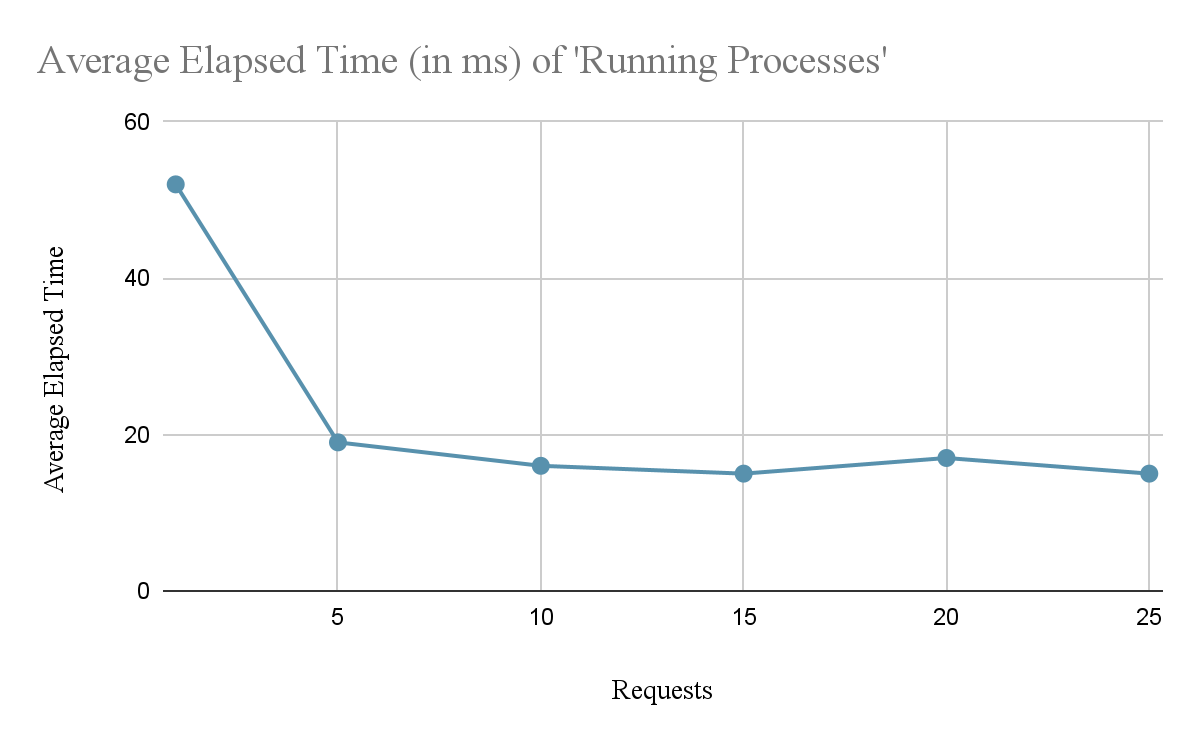
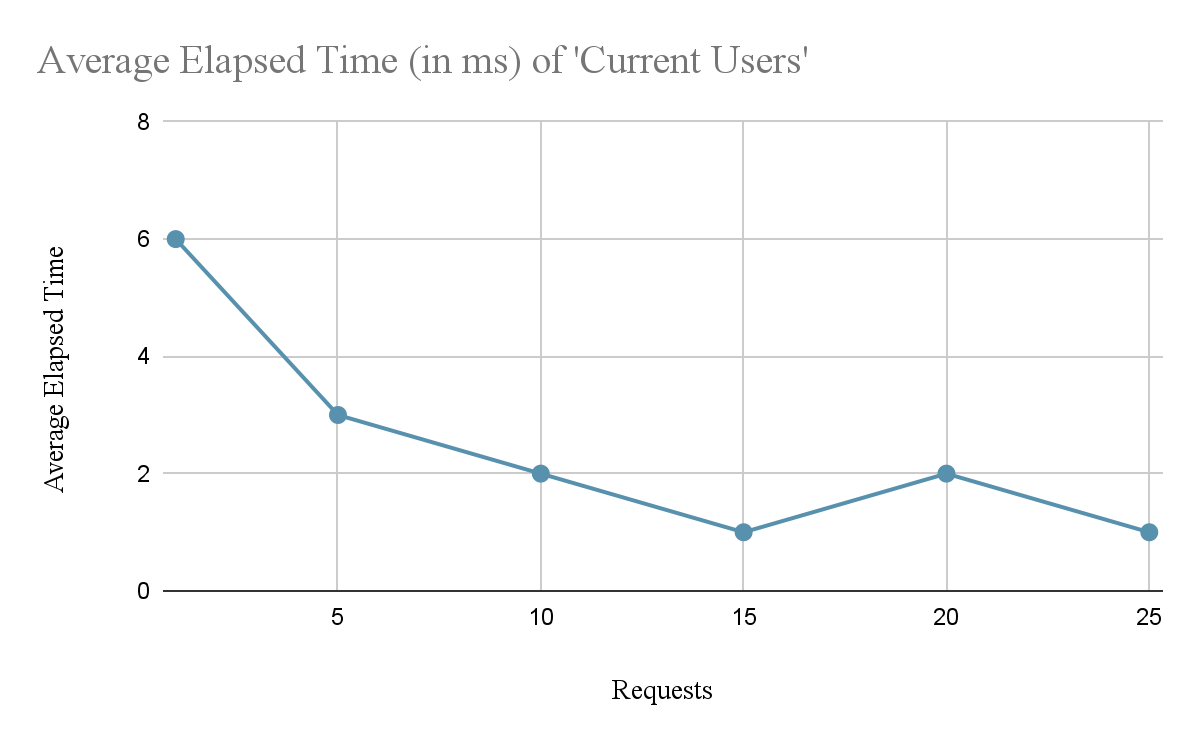
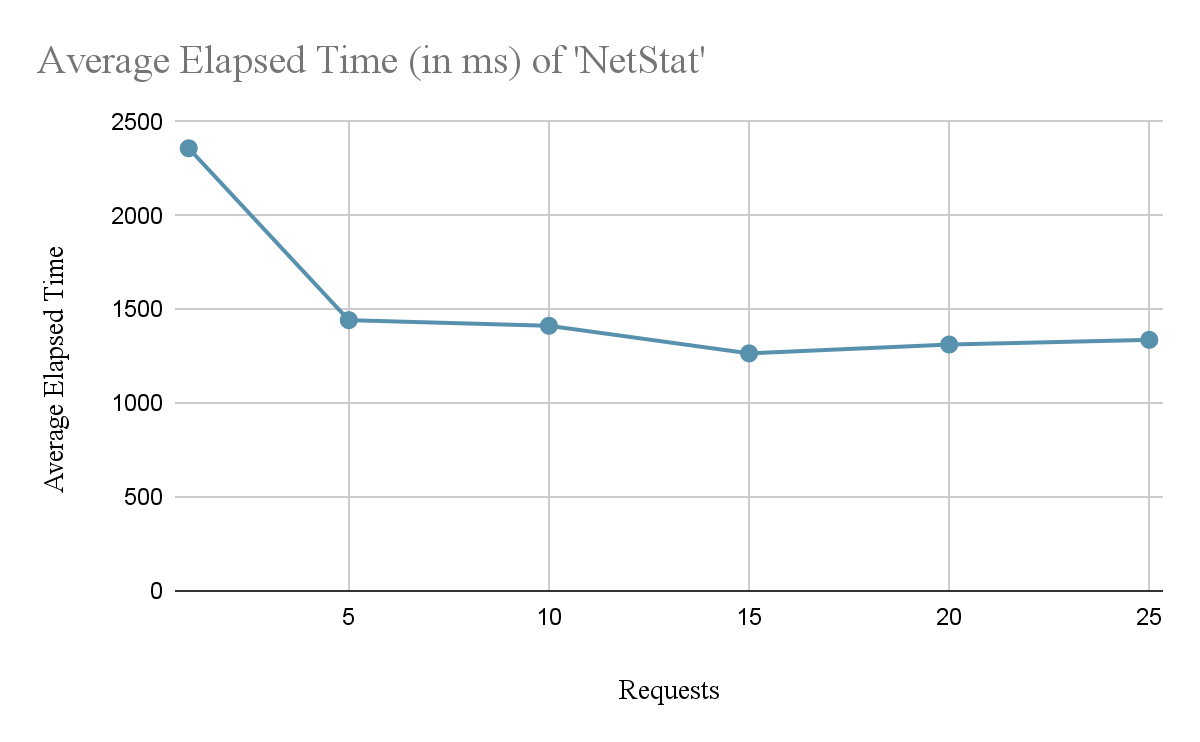
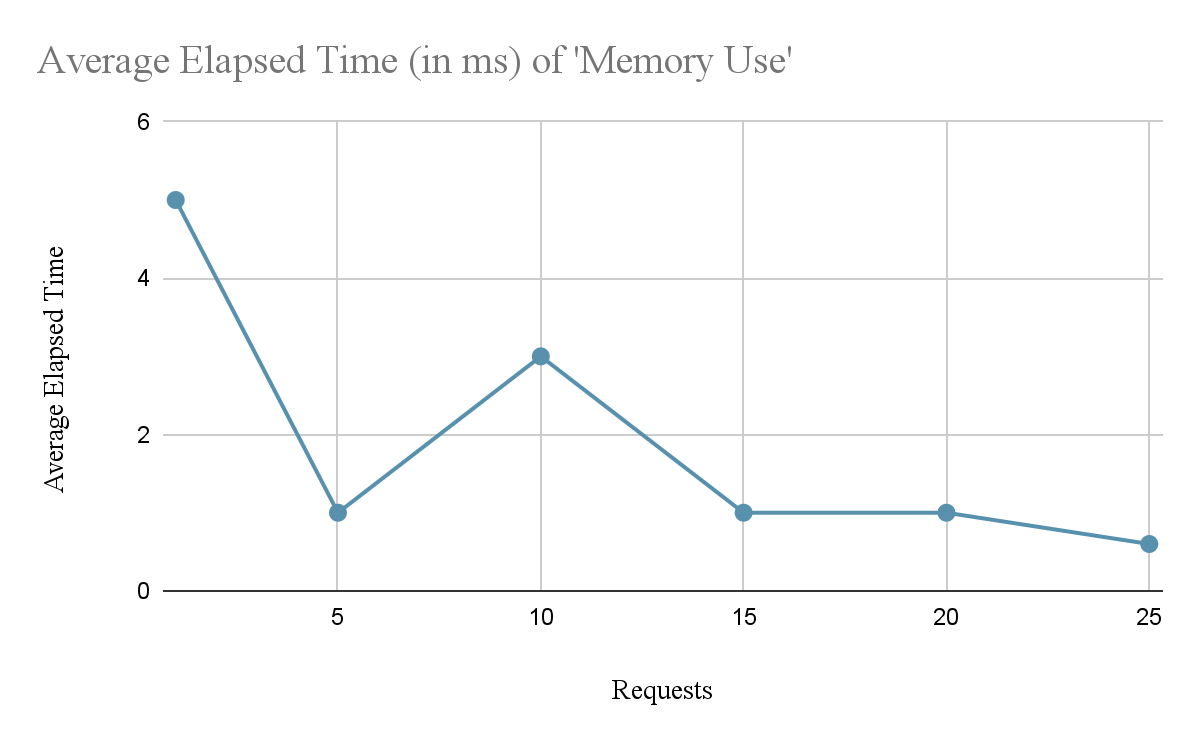
Figure 1; Average Elapsed Time (in ms.) for each Server Operation

|  | Date and Time | Uptime | Memory Use | NetStat | Current Users | Running Processes |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 2 | 5 | 2359 | 6 | 52 |
| 5 | 2 | 2 | 1 | 1443 | 3 | 19 |
| 10 | 2 | 1 | 3 | 1413 | 2 | 16 |
| 15 | 2 | 2 | 1 | 1266 | 1 | 15 |
| 20 | 1 | 2 | 1 | 1313 | 2 | 17 |
| 25 | <1 | <1 | <1 | 1338 | 1 | 15 |
| 100 | <1 | 10 | <1 | 1385 | 1 | 16 |

Figure 2; Total Elapsed Time (in ms.) for each Server Operation

|  | Date and Time | Uptime | Memory Use | NetStat | Current Users | Running Processes |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 2 | 5 | 2359 | 6 | 52 |
| 5 | 10 | 14 | 8 | 7218 | 15 | 98 |
| 10 | 26 | 18 | 38 | 14130 | 29 | 166 |
| 15 | 35 | 31 | 25 | 19001 | 24 | 232 |
| 20 | 37 | 48 | 34 | 26276 | 40 | 352 |
| 25 | 17 | 24 | 15 | 33468 | 48 | 387 |
| 100 | 67 | 1049 | 66 | 138509 | 108 | 1677 |





**Data Analysis**

One of the primary objectives of this project was to examine the performance of the server under increasing client loads. By varying the number of client requests and observing the corresponding turn-around times, we can assess the performance of the concurrent server under different stress tests. As the number of client requests increased the turn-around time for individual clients actually decreased. As well as with the average turn-around time, the increasing client requests seemingly decreased the average turn-around time. This effect could be due to a couple of possible factors. For one, if the load on the virtual machine was lighter than expected, a degradation in performance may not be seen. Another reason could be because of the number of users on the server changing between tests. In comparison with the iterative socket server, the average turn-around time for the concurrent socket can be seen to be lower as its ability to handle concurrent requests allows for more efficiency under heavy loads. An iterative server may be better suited for situations with lower loads or if client requests need to be handled sequentially. As for concurrent servers, it is better used for high loads and to handle multiple requests in parallel.

**Conclusion**

To conclude, the concurrent server project provided many valuable insights into the design and implementation of a multi-threaded client-server configuration. Through in-depth testing, data collection, and analysis, several significant conclusions can be drawn. Firstly, we can presume the use of concurrency in the server design led to reduced average turn-around times as concurrent processing allowed the server to parallelize tasks and optimize processes under the stress of multiple clients. We can also draw the conclusion that the concurrent server is more suited for higher loads as the number of clients increased, the server maintained responsiveness and minimized turn-around times.

**Lessons Learned**

There were many valuable lessons learned throughout working on the project. Of which we learned about unpredictability in the testing environment as there are many factors out of our control that could skew test results. For example, when recording our data, there were five other users on the class server, which could put more of a strain on the configuration then initially expected. However, we overcame this unpredictability and were able to generate seemingly accurate test results.