**Concurrent Socket Server**

James Quinn Spearman

Chris Foster

Serhii Dubynskyi

CNT4504 – Computer Networks & Distributed Processing

Professor J. Scott Kelly

**Introduction**

The project's purpose is to implement a multi-threaded or, in other words, an Concurrent Socket Server for use in a client-server configuration to examine, analyze, and study an impact the concurrent server has on the efficiency of processing client requests in terms of turn-around time.

The project's goals are the creation of the multi-threaded server and multi-threaded client programs with six operations, collecting data from turn-around time (total and average) for each client request, and analyzing collected data for patterns.

In the remaining sections of the paper, the design of the client and server programs are described, as well as the operations of the client and server programs. Also, turn-around time data from the server is provided, followed by an analysis of the collected data.

**Client-Server Setup and Configuration**

The concurrent server's design, which handles multiple client requests at a time, consists of the Java Server Socket object, for connecting to the client and caching client requests automatically. The server uses a while loop, which has a boolean value of true as its condition to ensure that it is always listening for client requests. When a client request is received, the query gets stored in the form of a string and new thread is created, then the server goes back to listening to client requests. Within each thread it is determined which Linux command is being requested using a case-switch, the requested command is performed using a user-defined function called getCommand, which utilizes the Runtime class to retrieve the output of the requested command, then a string of the command’s output is returned. The server then sends the command's output to the user using the PrintWriter class. The server supports six Linux commands: date and time, uptime, memory use, netstat, current users, and running processes. If an invalid operation is requested, the server replies with "Invalid query." Once the response is sent to the client, the socket is closed within the thread.

The multi-threaded client's design, which allows it to transmit to the server on a specified network port, can spawn multiple client sessions. The client class extends the runnable class, which contains the run function used for spawning multiple threads simultaneously. After the user enters how many clients they would like to spawn, the client program creates the corresponding number of client classes using a for loop, a second for loop then starts all of the threads. When the run function is called, the system time in milliseconds is stored into a start variable, and a socket is created. The query requested by the client is sent to the server using the println method in the PrintWriter class. Once the client starts hearing a response, it uses a while loop to read each line sent by the server individually. It appends each line to a StringBuilder object, which is then converted into a standard string variable. Once the message is fully received, the system time in milliseconds is stored in an end variable. The difference between the start and the end time is calculated and stored in a runtime variable. After the message received from the server is displayed, if all clients have received a response, a user-defined function called displayRuntimes is used to display each client's query runtime individually as a calculation of the total and average runtimes of all of the clients.

One of the most critical design decisions that was made was to make only one class for the client program, and have it created new instances of itself for each client to be spawned. This was done by using static variables for the query, port number, number of clients, total runtime, the array of individual runtimes, and the number of runtimes recorded. This allowed for the data to remain consistent and be tracked across all instances of the client object.

One of the most critical design decisions of the server program was very similar to that of the client program, which was to use only one class for the server, and have the program create a new server instance for every client request.

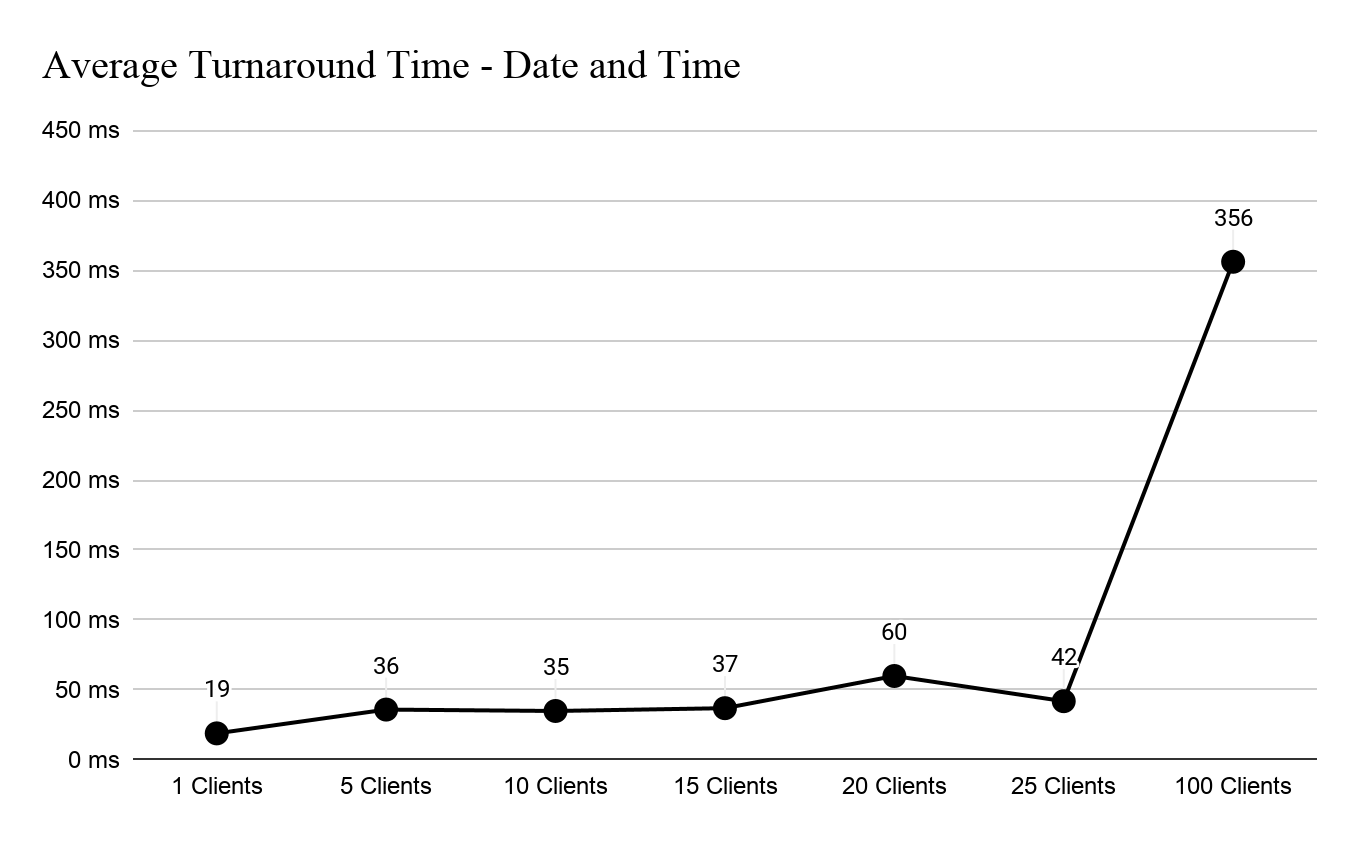
The basic operation of the client program is to prompt the user for an IP address and a port number between 1025 and 4998, followed by a prompt for a command to be requested from the server and a number of clients to spawn. Each client then makes its query to the server, waits for a response, and displays it once a response is received. Then the runtime of each client is displayed, followed by the total and average runtime across all the clients.

The basic operation of the server program is to prompt the user of the server for a port number between 1025 and 4998. Once the port number is entered, the server starts listening on that port. When a query is received from the client, the server passes the string of the client’s request into a new thread and begins listening for client requests again. Each new thread determines which Linux command to execute and sends the output that the command produced back to the client.

**Testing and Data Collection**

To test the concurrent server that we created, we ran each operation one time using 1, 5, 10, 15, 20, 25, and 100 clients, and the average turnaround time for each test was recorded and put into a line graph displayed below. The total turnaround time and the average turnaround time for each set of clients are displayed below each line graph.

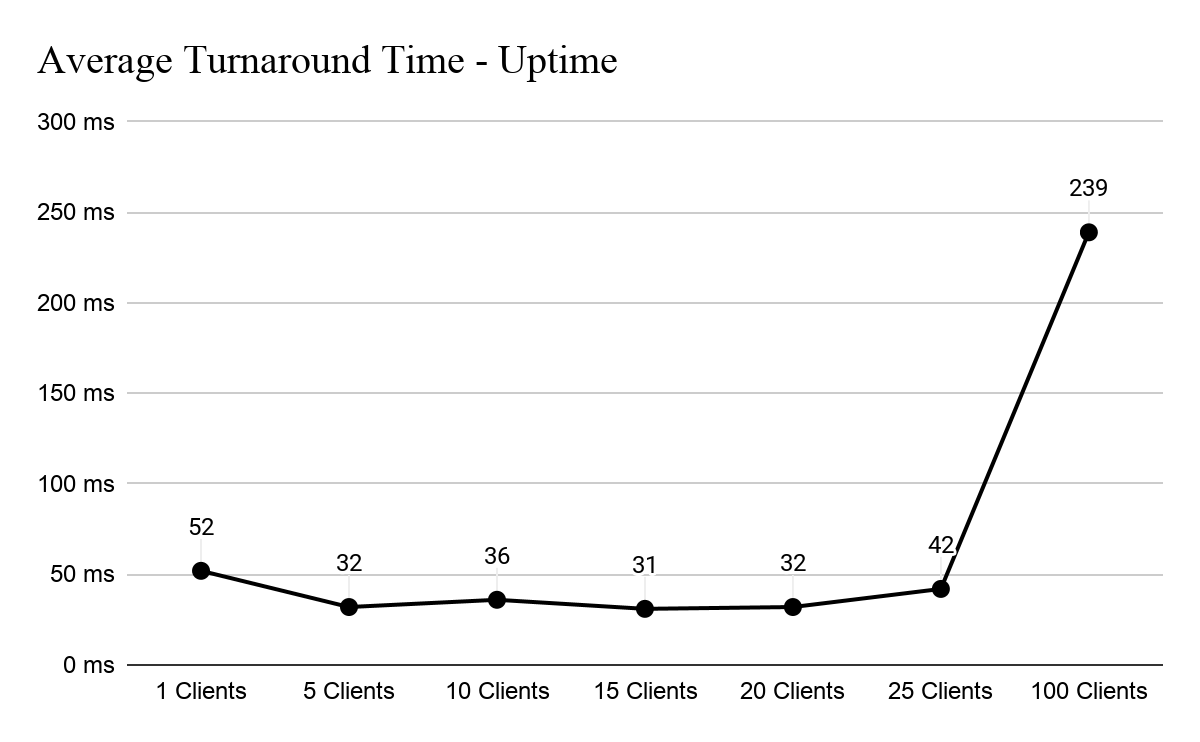
**Concurrent Graphs**



**# Clients 1 5 10 15 20 25 100**

**Total(ms):** 19 183 351 565 1204 1059 35632

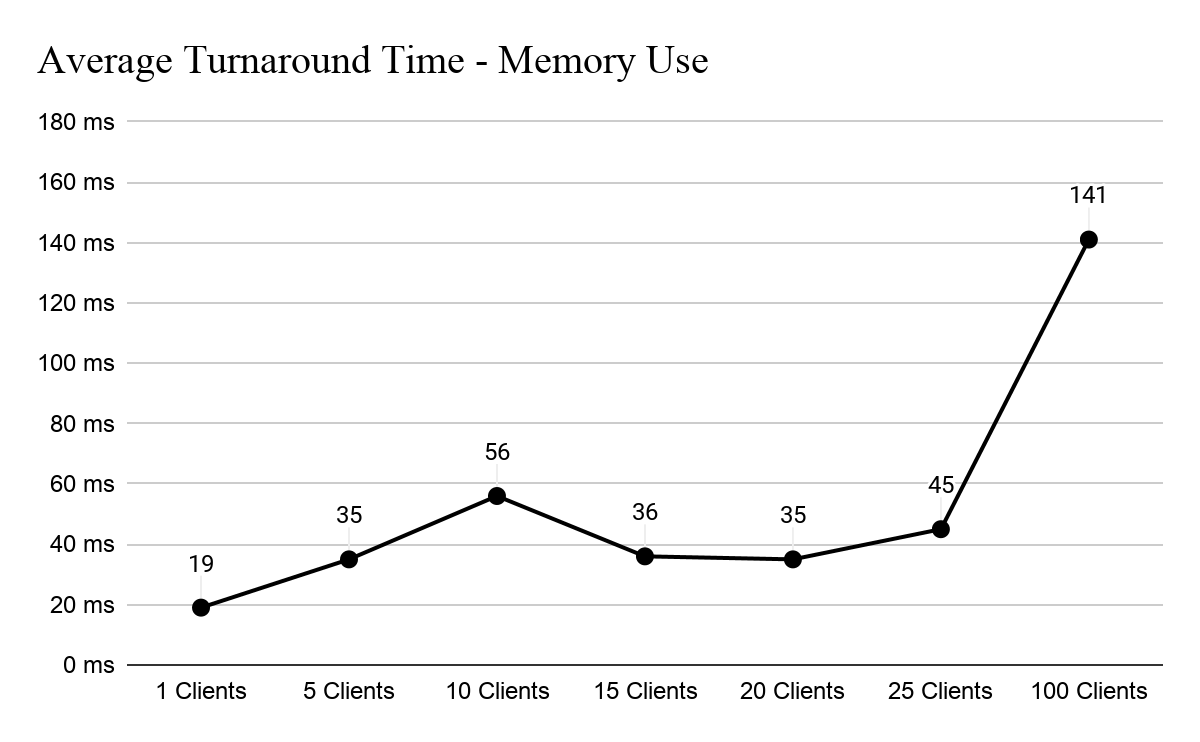
**Average(ms):** 19 36 35 37 60 42 356



**# Clients 1 5 10 15 20 25 100**

**Total(ms):** 52 164 361 470 640 1071 23950

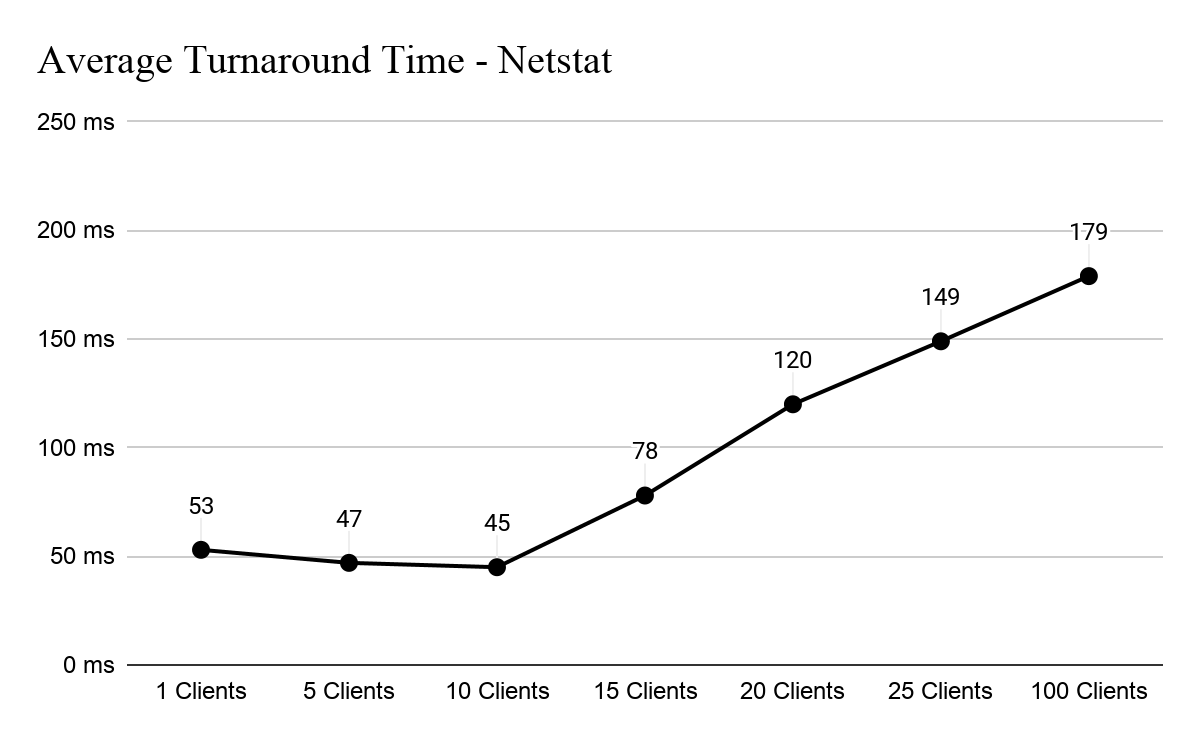
**Average(ms):** 52 32 36 31 32 42 239



**# Clients 1 5 10 15 20 25 100**

**Total(ms):** 19 179 560 551 702 1149 14101

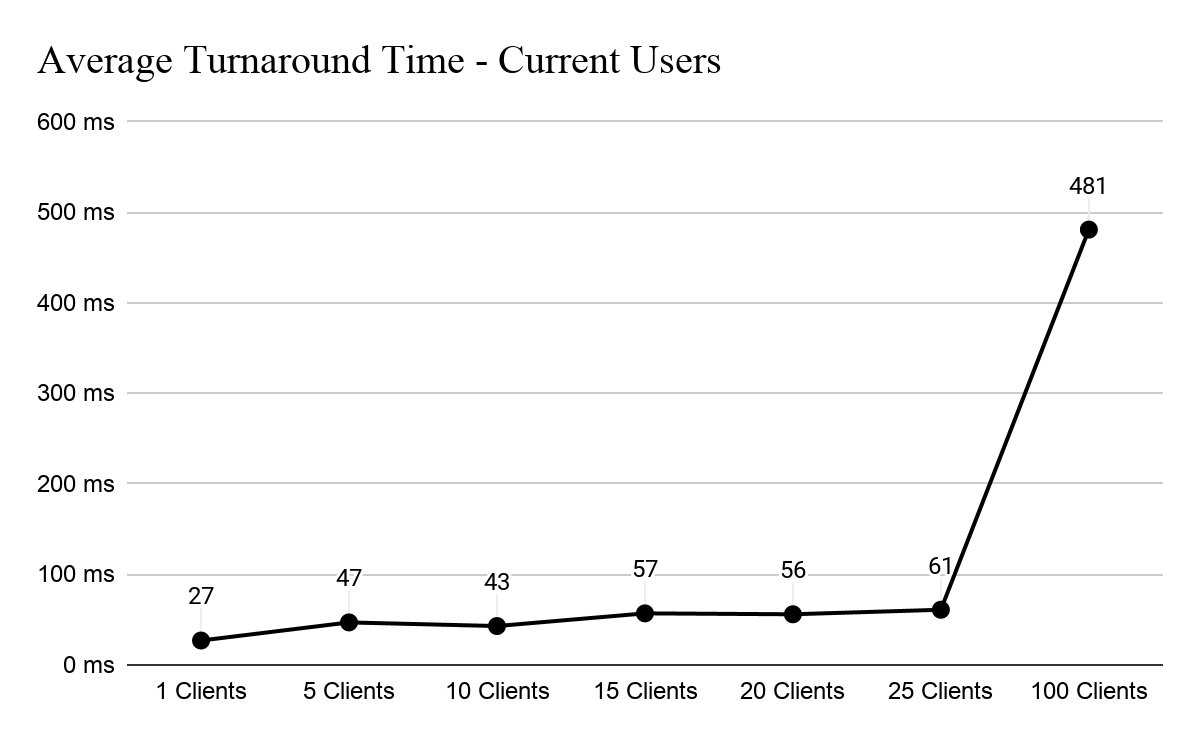
**Average(ms):** 19 35 56 36 35 45 141



**# Clients 1 5 10 15 20 25 100**

**Total(ms):** 53 235 459 1174 2414 3746 179702

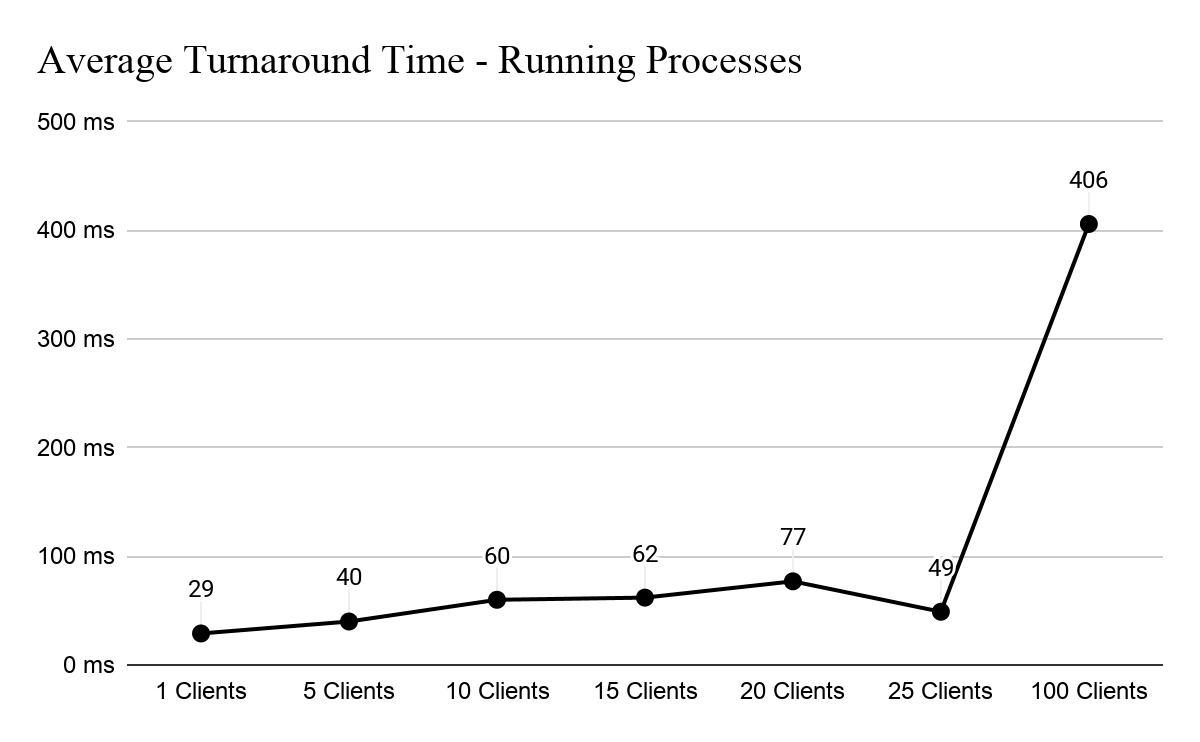
**Average(ms):** 53 47 45 78 120 149 179



**# Clients 1 5 10 15 20 25 100**

**Total(ms):** 27 236 433 865 1133 1532 48106

**Average(ms):** 27 47 43 57 56 61 481

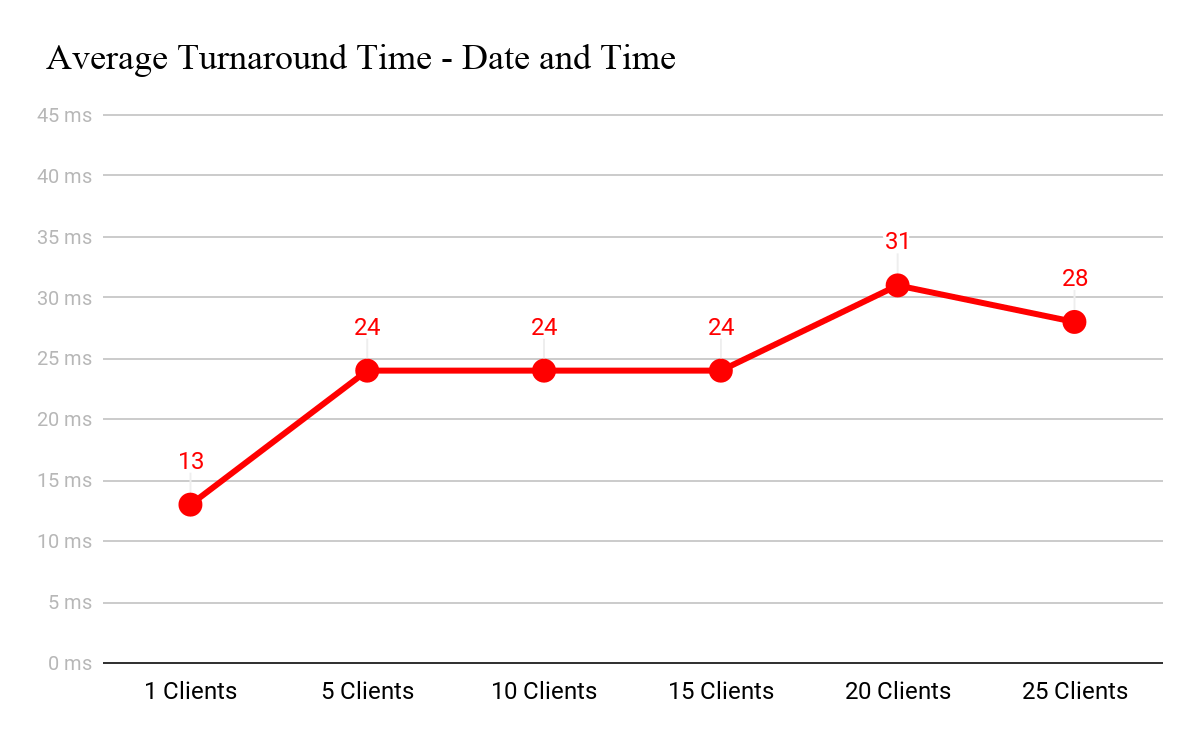


**# Clients 1 5 10 15 20 25 100**

**Total(ms):** 29 202 608 940 1553 1239 40655

**Average(ms):** 29 40 60 62 77 49 406

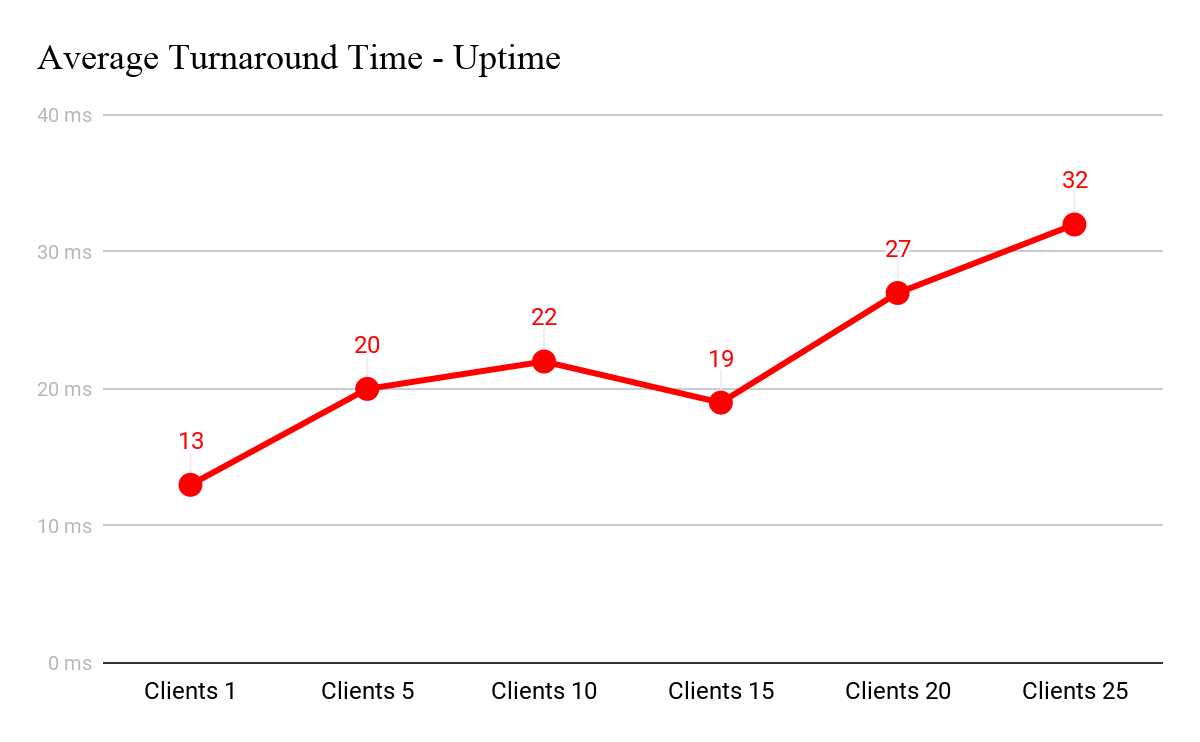
**Iterative Graphs**



**# Clients 1 5 10 15 20 25**

**Total(ms):** 13 122245364620700

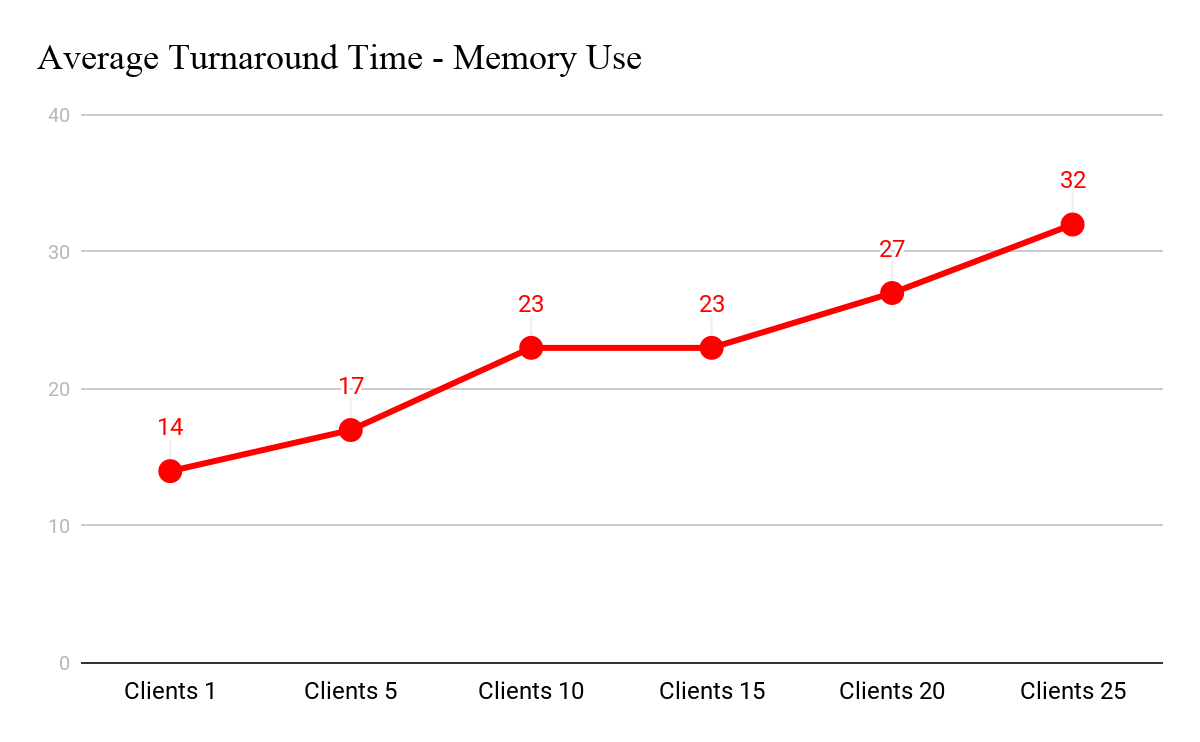
**Average(ms):** 13 2424243128



**# Clients 1 5 10 15 20 25**

**Total(ms):** 13 103220298555818

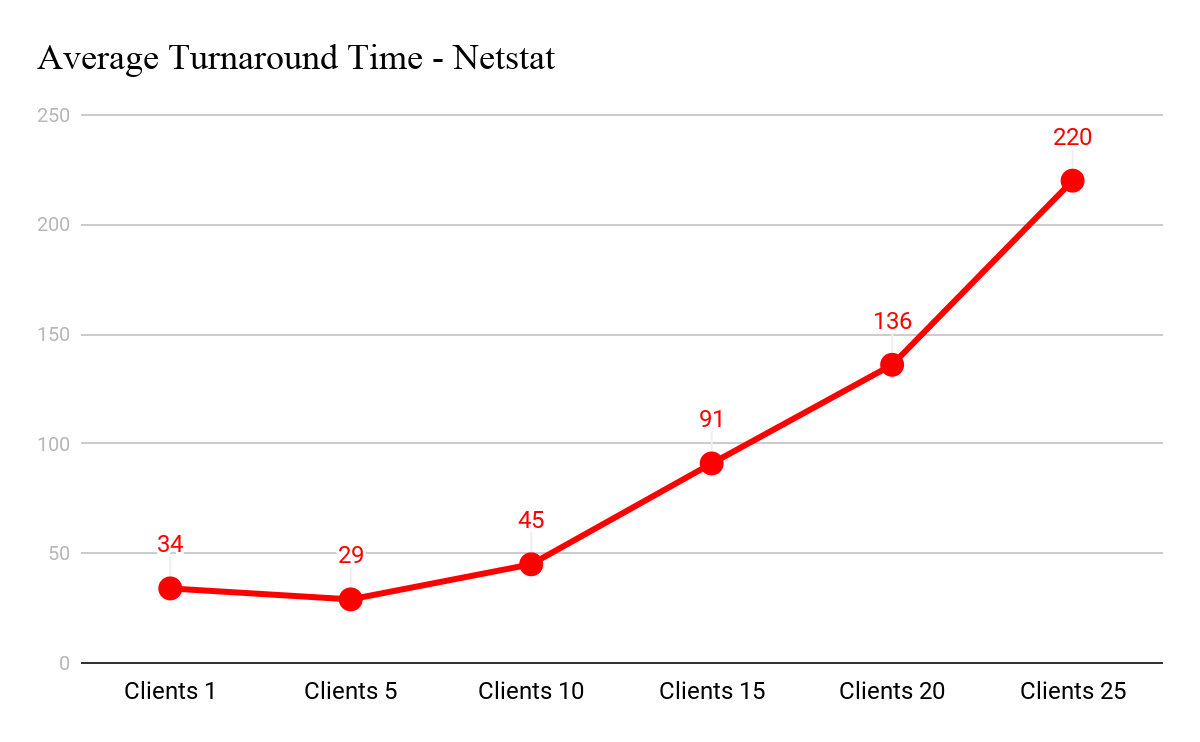
**Average(ms):** 13 2022192732



**# Clients 1 5 10 15 20 25**

**Total(ms):** 14 85234345542800

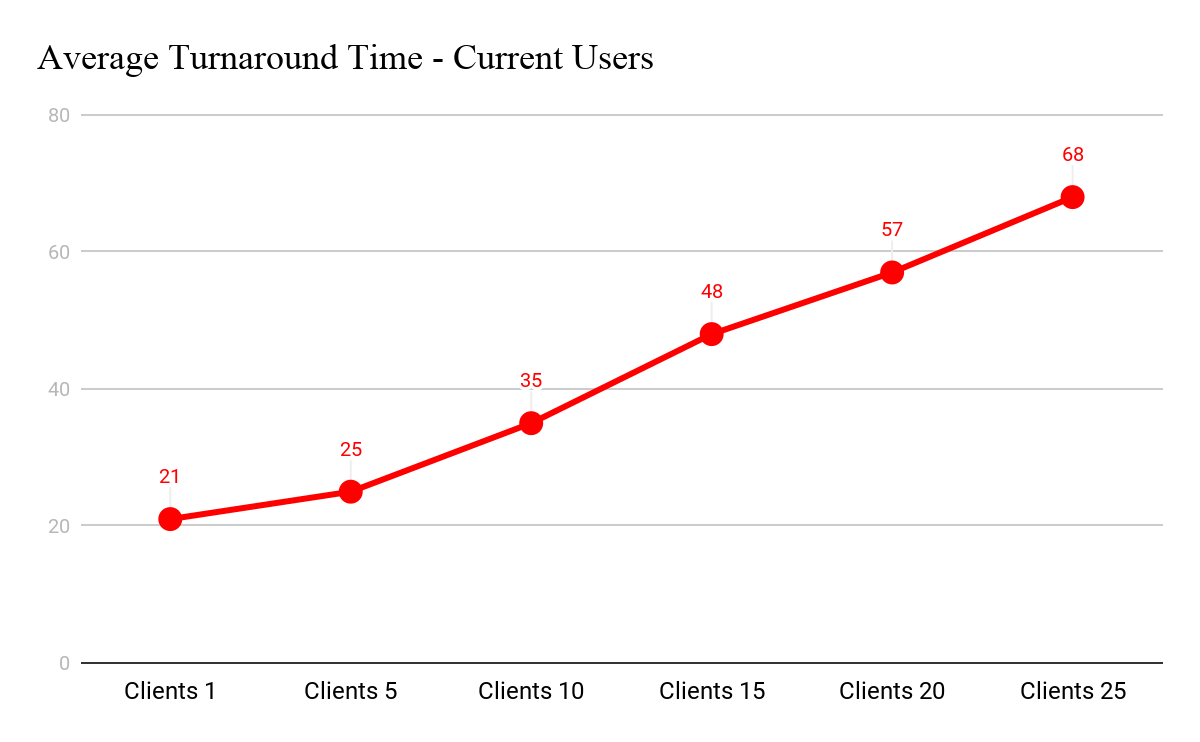
**Average(ms):** 14 1723232732



**# Clients 1 5 10 15 20 25**

**Total(ms):** 34 145450137927355508

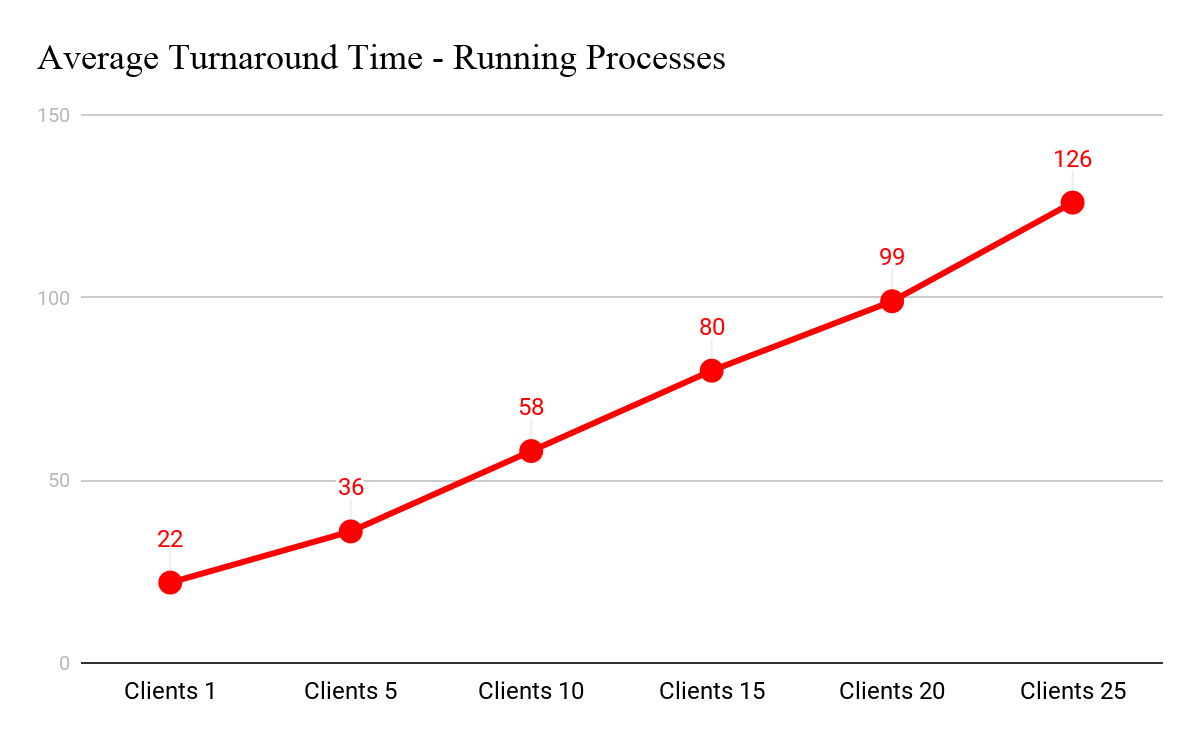
**Average(ms):** 34 294591136220



**# Clients 1 5 10 15 20 25**

**Total(ms):** 21 12935272311411706

**Average(ms):** 21 2535485768



**# Clients 1 5 10 15 20 25**

**Total(ms):** 22 184581120819863169

**Average(ms):** 22 36588099126

**Data Analysis**

The concurrent socket server is similar to the iterative socket server in many ways. The concurrent socket server is similar to the iterative as more clients are added to both servers the individual turnaround time begins to increase. Along with individual increases, this leads to a proportionate increase in the average turnaround time for each command. The concurrent socket server is also similar to the iterative in that some commands are heavier in their turnaround time than others, so commands like netstat will bring in heavier turnaround times than something like date and time. The iterative server can handle a small amount of commands more efficiently, while the concurrent server can handle a large number of commands more efficiently. The iterative server can handle a small amount of client requests more efficiently, while the concurrent server can handle a large number of client requests more efficiently. The situations where you should use iterative servers are where you only need to be handling a small number of client requests simultaneously. Whereas the situations where you should use concurrent servers are those where you will be handling a large number of client requests simultaneously. The iterative server processes both the connection request and the transaction associated with the call itself. Iterative server is quite simple and is appropriate for transactions that do not last long. However, if the transaction takes longer, queues can grow rapidly. Thus, for lengthy transactions, the concurrent server is needed.

**Conclusions**

The concurrent server is well suited for handling a large quantity of clients simultaneously, while the iterative server is able to handle a smaller quantity of clients simultaneously. The iterative and concurrent servers are both very similar in the pattern at which their turnaround time goes from smaller to larger with many more clients. From comparing the averages, concurrent seems to have a higher turnaround time altogether versus the iterative server. Concluded from this was the difference in commands and their turnaround times as well. Also concluded from this is that concurrent servers should be used for many clients versus an iterative server which thrives with less clients rather than more clients.

**Lessons Learned**

A lesson we learned is just how similar the concurrent and iterative servers are to each other. Through our data analysis we were able to determine which server was better in each situation. Most of the programming for the concurrent server was the same as the iterative, the client was the exact same and the concurrent server used the shell of the iterative server. The only major difference was that the code to determine which Linux command was being requested and retrieve its output and send it back to the user was moved to inside a run function, which is called whenever a new thread is created.