**Iterative 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 single-threaded or, in other words, an Iterative Socket Server for use in a client-server configuration to examine, analyze, and study an impact the iterative server has on the efficiency of processing client requests in terms of turn-around time.

The project's goals are the creation of the single-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 iterative server's design, which handles one client request 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. The server determines which Linux command is being requested using a case-switch, performs the requested command using a user-defined function called getCommand, which utilizes the Runtime class to retrieve the output of the requested command, then returns a string of the command's output. 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." After performing any necessary clean-up activities, the server goes back to listening for client requests.

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.

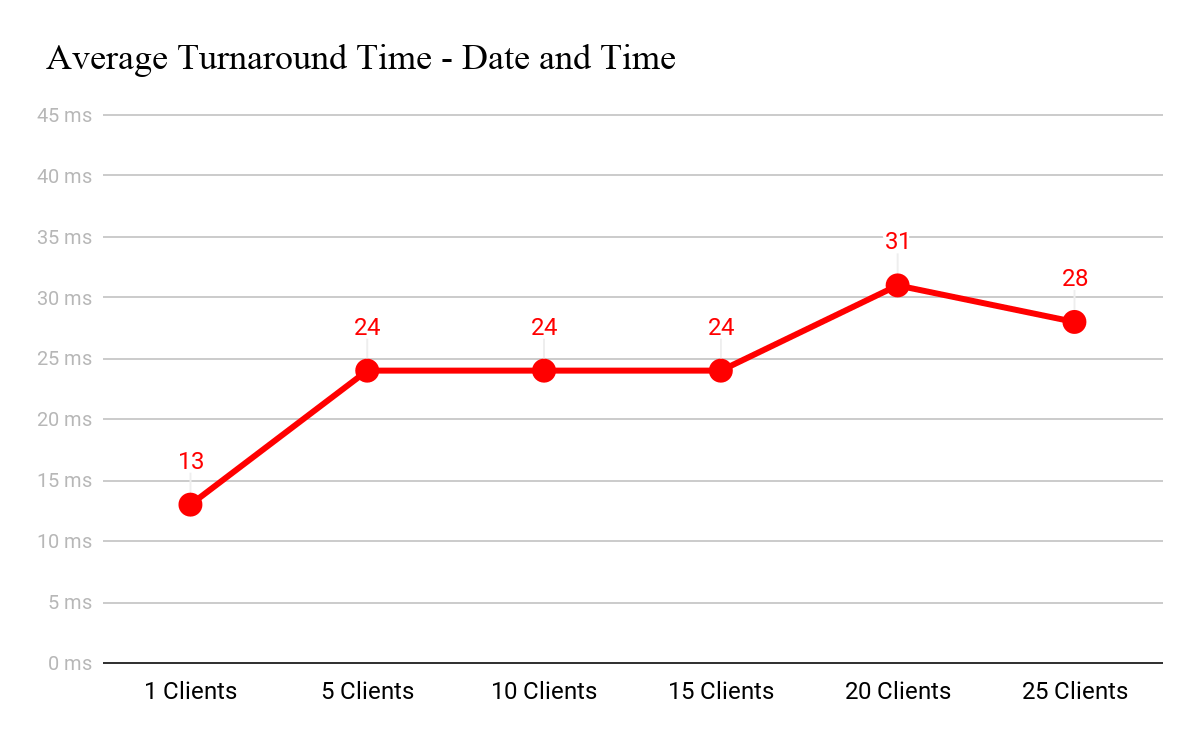
One of the most critical design decisions that was made was to make only one class for the client program, and have it create versions of itself for each client to be spawned. This was done by using static variables for the query, port number, number of clients, total runtime, the array of individual 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.

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 of 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 determines which Linux command to execute, executes it, and sends the output that the command produced back to the client, then continues listening for client requests.

**Testing and Data Collection**

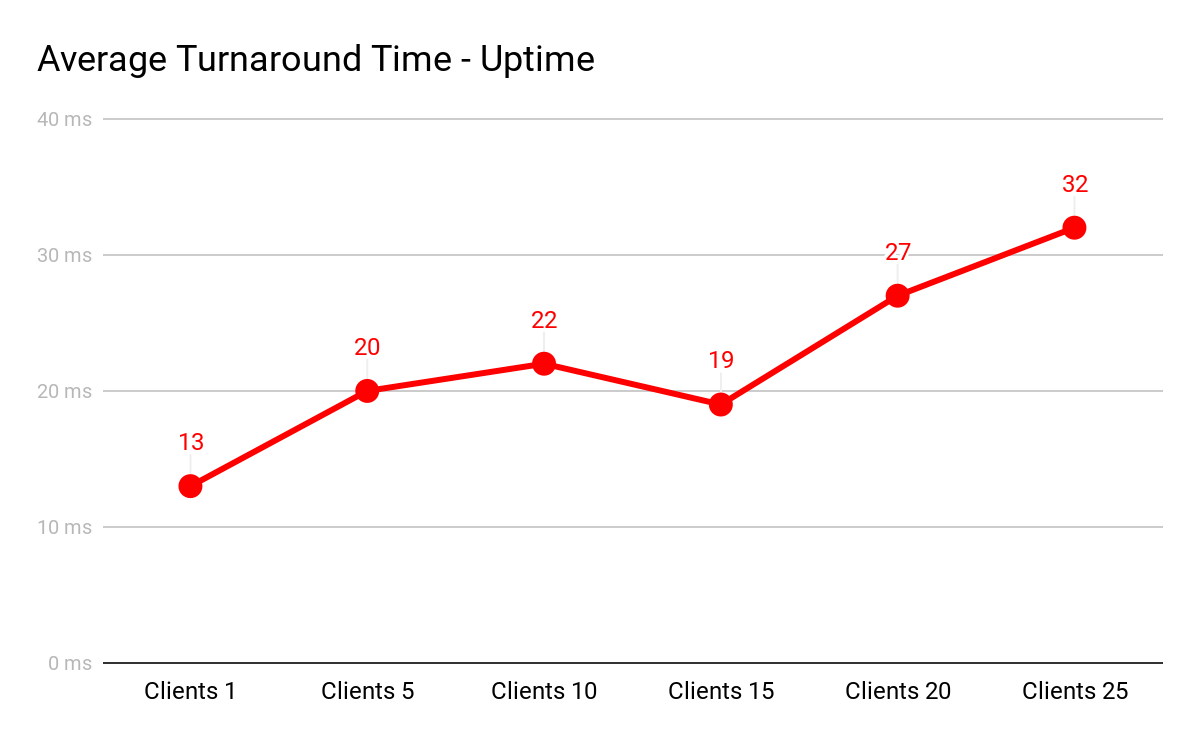
To test the iterative server, we ran each operation one time using 1, 5, 10, 15, 20, and 25 clients, and the average turn-around time for each test was recorded and put into a line graph displayed below. The total turn-around time and the average turn-around time for each client's set is displayed below each line graph.



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

**Total(ms):** 13 122245364620700

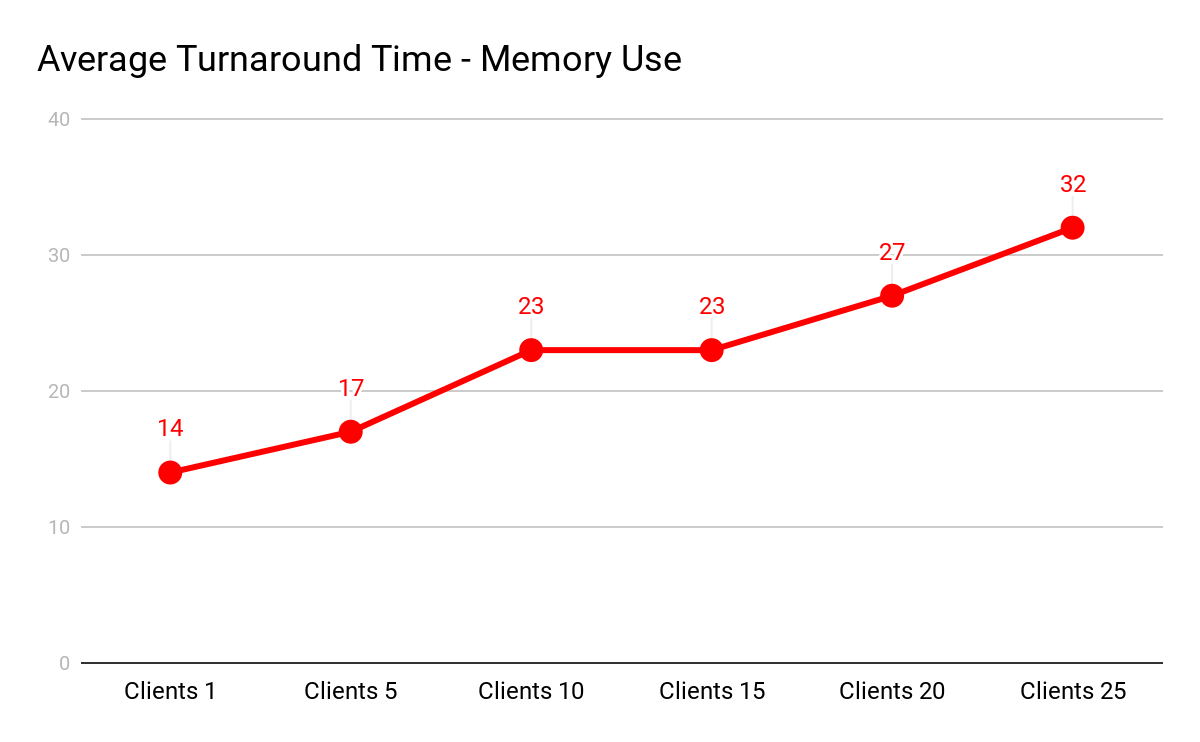
**Average(ms):** 13 2424243128



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

**Total(ms):** 13 103220298555818

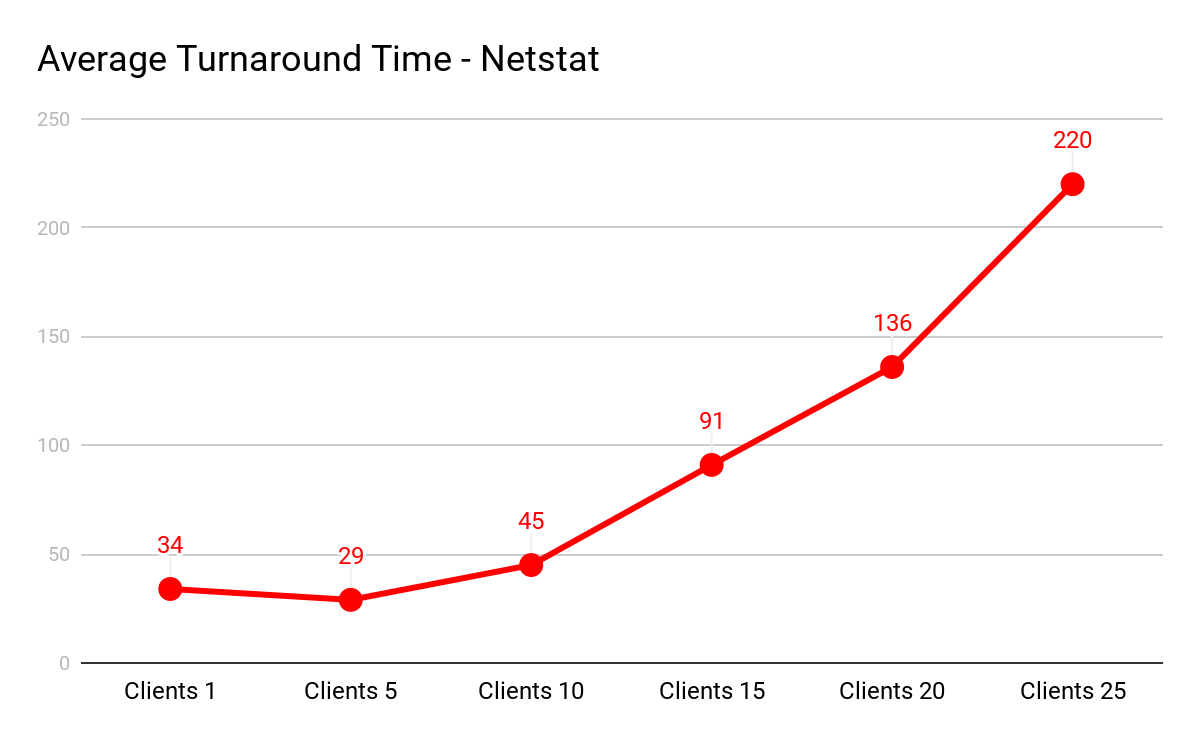
**Average(ms):** 13 2022192732



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

**Total(ms):** 14 85234345542800

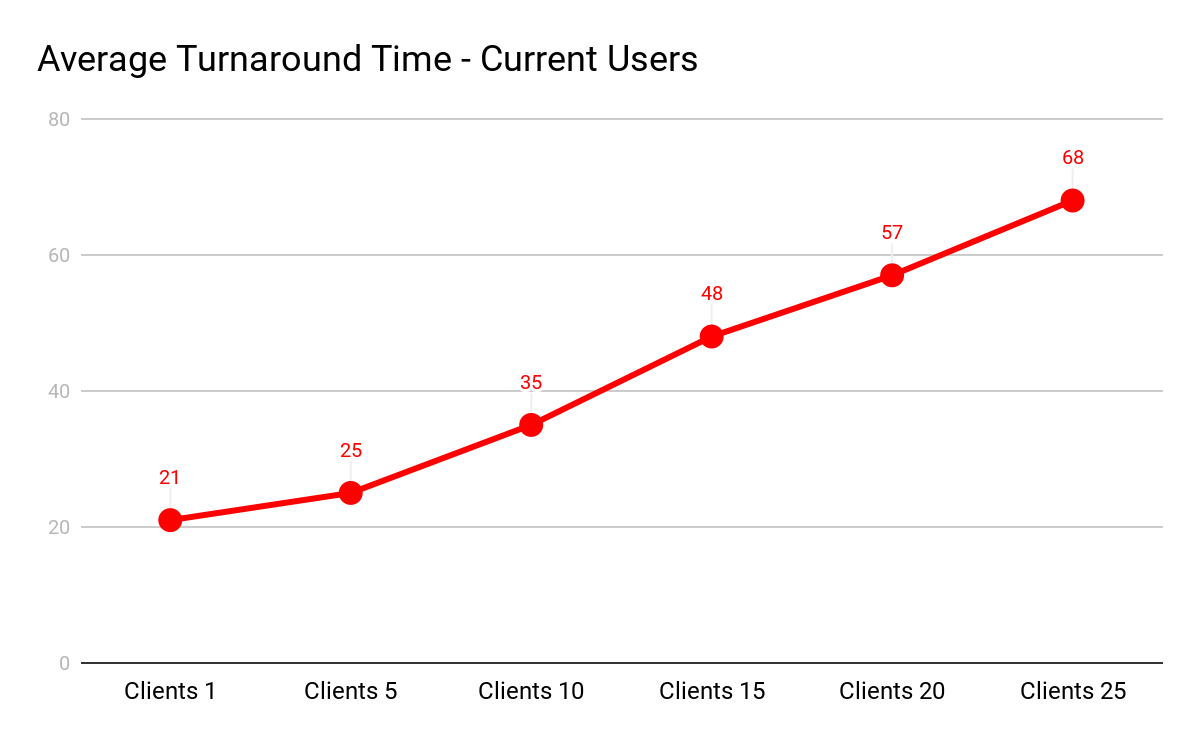
**Average(ms):** 14 1723232732



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

**Total(ms):** 34 145450137927355508

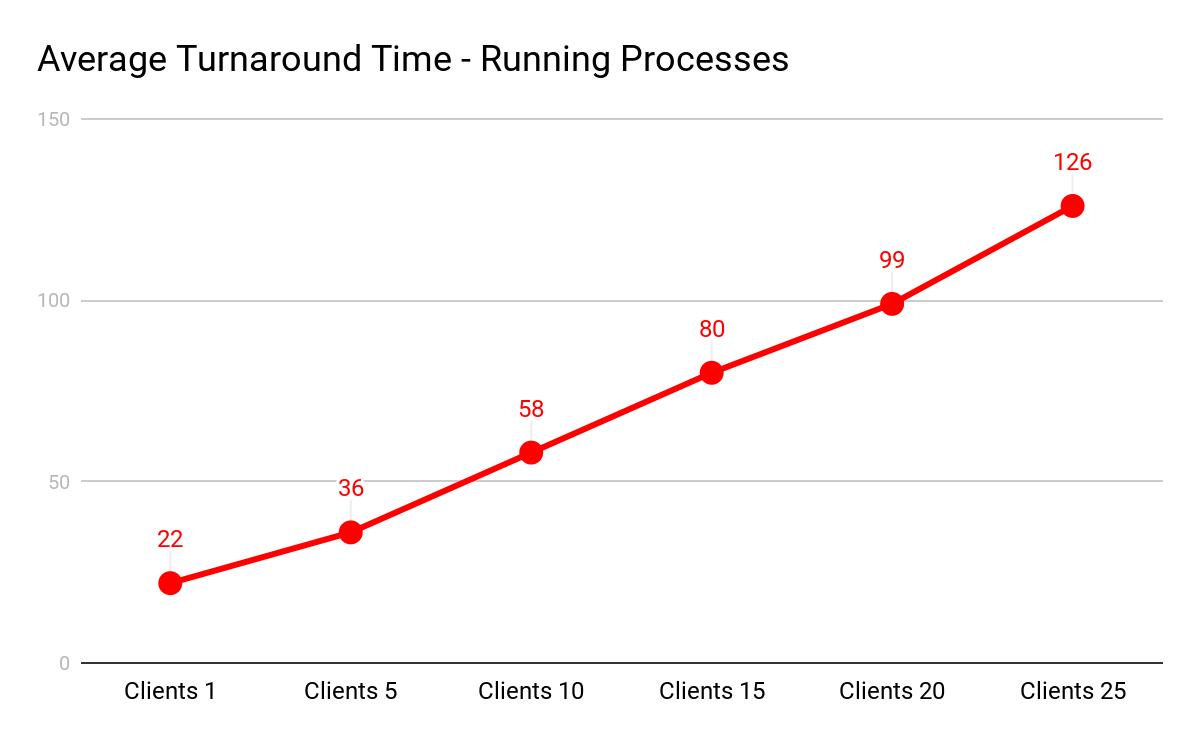
**Average(ms):** 34 294591136220



**# 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 turnaround time was affected by two different variables: how many clients were accessing the server at the same time and the other being which operation was being used. The individual turnaround time for just one client varied depending on what command was performed. Simple commands using only a single client like date and time had little turnaround time around 10 to 15 milliseconds as opposed to more complex operations such as netstat, which had turnaround times in the upper 20’s and lower 30’s. These operations had varying turnaround time averages in correlation to this, as simpler commands had lower averages when given multiple clients and more complex had higher averages when given multiple clients. The effect that these clients have on each of the operations shows that there is a definite increase in average from a low number of clients to a higher one. The effect that increasing the number of clients has on the individual client turnaround time is that the individual turnaround time increases with the number of clients added. The simultaneous increase between the number of clients and the individual turnaround time results in an increase in the average turnaround time. The primary cause for this increase in turnaround time is that the server is trying to process more clients, and as it processes more of them, it creates a longer cycle time and slows down the server.

**Conclusion**

More clients on a server typically results in an increase in turn-around time. This turn-around time can vary between the different commands depending on how complex they are, but overall, as the number of clients increases, the turn-around time for the individual and the average will also increase. In conclusion, the number of clients increased also increases the turn-around time for individual clients and the average turn-around time.

**Lessons Learned**

A problem we had to overcome was the programming of something related to networks because none of us had ever created a server or programmed anything relating to networks, but through creating this we learned of the ability to use Linux commands inside of a java program, how to connect a client to a server, and learned about various java libraries and methods. After creating the program, the data collection came easy as we had already tested the program the same way and the server program handled the varying number of clients well.