Iterative Socket Server

Gabriel Pridham and Samuel Schwartz

CNT4504 - Computer Networks & Distributed Processing

Professor Jim Littleton

Introduction

The purpose of this project is to get a better understanding of how a single-threaded server works and understand the efficiency of processing client requests. The goal of this project is to work as a team to create a single-threaded server that will listen for a client request, and when the request is made by the client, the server will execute the command and collect the resulting output. The server will then send the output to the client so that the client is able to determine the turn-around time it took for the server to execute the command. The rest of this paper will go into more detail about the client-server setup and configuration, the testing and data collection, and analysis of the data.

Client-Server Setup and Configuration

The server program is executed with one argument, the port number. When executed, the server stores the time and date that it went live, then it listens for a client connection. When a client has connected, the server waits for the client to give a Linux command to be executed on the server. When the server receives a command, it executes the command and collects the output. The server will then send that output to the client. The server is capable of handling all the Linux server commands, however, we only accepted the following commands: date and time, uptime, memory use, netstat, current users, and running processes. The server accepts the same naming convention as Linux. We keep the names of the commands to be the same names of the Linux server because we figured that it would be easier to just be able to google the commands that the user wants to use, rather then change the naming convention that many people are already familiar with.

The client program is executed with four arguments, the hostname, port number, operation number, and the number of clients. The operation numbers are 1 for the data and time, 2 for uptime, 3 for memory use, 4 for Netstat, 5 for current users, and number 6 for running processes. The client program then creates threads that amount to the number of clients. As each thread is created, the thread grabs the current time and then sends the command to the server to perform the operation. The client then waits for the server to send the output. When the output is sent, the client stores the information and then grabs the time that it the client received the information from the server. The client also stores the received information in many different text files. Then the turn-around time is calculated by subtracting the time that the server received the request from the time that the client received the response. After all threads are executed serially, the client calculates the average turn-around time by getting the sum of the turn-around time of each thread and dividing it by the total number of clients. The client then displays the total turn-around time and the average turn-around time.

If one is to consider the process of the iterative server in a theoretical vacuum one can observe that the average wait time based on the number of clients progresses linearly. If a single client request on the server takes 20ms each time a similar request is made, and the server processes requests one at a time, then if multiple requests were theoretically made at the same time the following will occur:

1 client: 20 / 1 = 20ms average

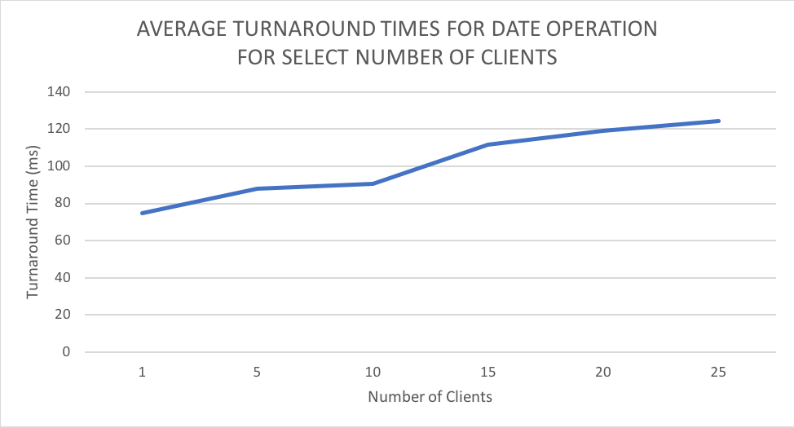
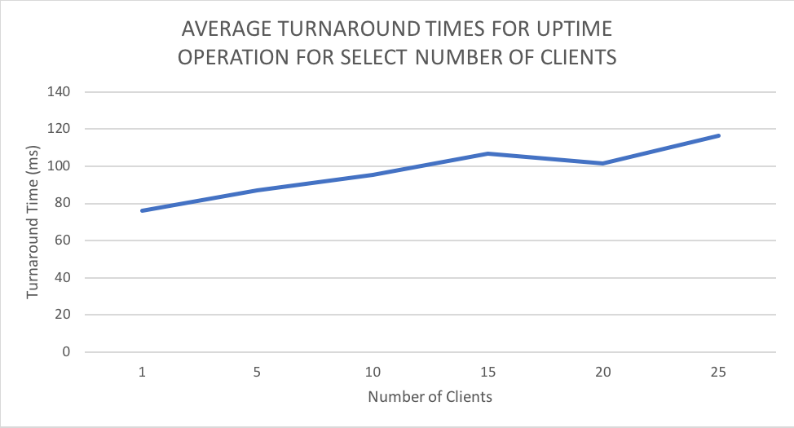
2 clients: 20 + 40 = 60 / 2 = 30ms average (40ms for 2nd client from itself and waiting 20 ms for 1st client)

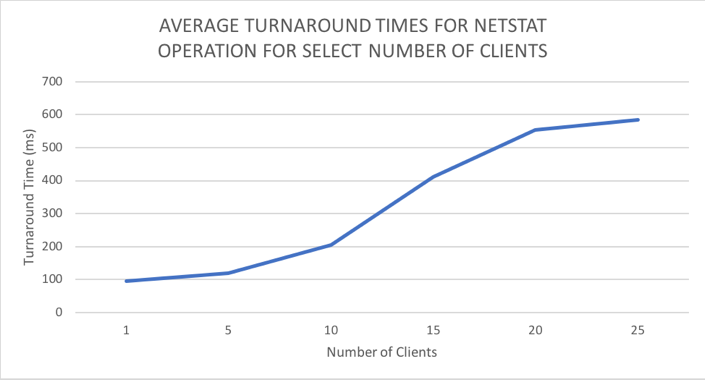
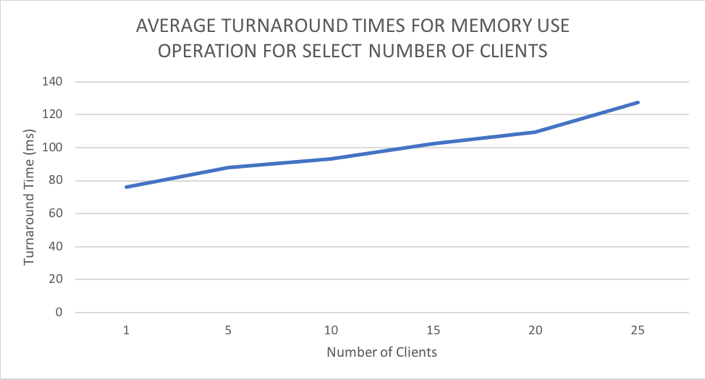
3 clients: 20 + 40 + 60 = 120 / 3 = 40 ms average

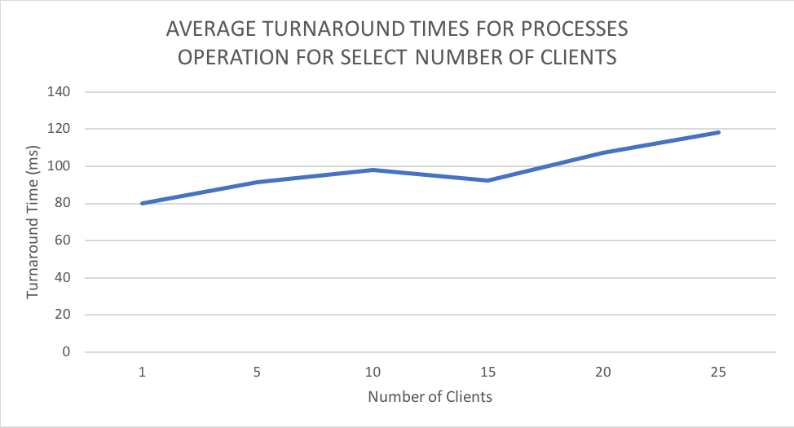
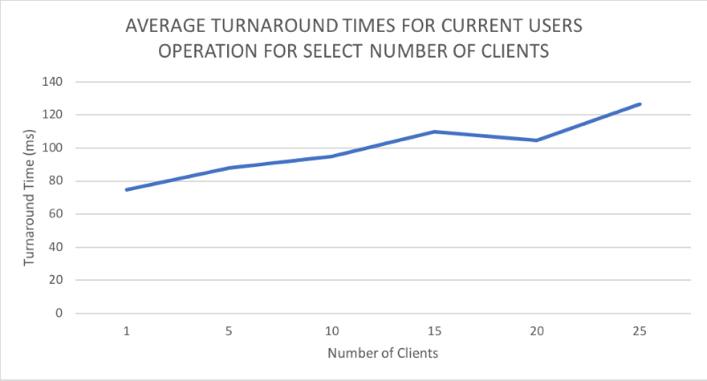
Testing and Data Collection

The iterative server was tested by manually making requests to the server with varying number of clients. We tested with 1, 5, 10, 15, 20, and 25 clients, each test was sampled ten times. We gathered the average turn-around time for each of the varying number of clients and averaged those numbers to create a graph that shows how the time scales with the number of clients for each specific command. The graph scales upwards linearly because it is iterative, and the server only handles the clients one at a time.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Number of Clients** | **Date** | **Uptime** | **Memory Use** | **Netstat** | **Users** | **Processes** |
| 1 | 74.90 | 75.90 | 76.20 | 96.10 | 74.60 | 80.10 |
| 5 | 87.94 | 87.12 | 87.90 | 118.86 | 87.84 | 91.28 |
| 10 | 90.48 | 95.60 | 93.01 | 204.02 | 95.09 | 98.10 |
| 15 | 111.80 | 106.69 | 102.51 | 411.23 | 109.79 | 92.43 |
| 20 | 119.31 | 101.36 | 109.23 | 554.30 | 104.53 | 107.37 |
| 25 | 124.35 | 116.49 | 127.28 | 583.62 | 126.42 | 118.28 |

**Table 1** Turn-around time in milliseconds for select number of clients for the required processes





Data Analysis

Increasing the number of clients increases the Turn-around Time for individuals clients because the threads are executed at nearly the same time and the server only executes the command one at a time, therefore the increase in clients results in a longer time for the later clients to have their request executed.

Increasing the number of clients will cause an increase in the average turn-around time because there is an increase in the individual client turn-around time, which results in the overall average being higher. The increase in clients will generate a high enough difference when it gets to the later clients that it will increase the average of the turn-around time at a linear rate.

The primary cause of the increase in the individual turn-around time and the increase in the average turn-around time is the fact that the server executes the client commands iteratively. The client threads are all spawned at nearly the same time, and each request is executed only after the request before it has finished processing, therefore the later clients will take longer to execute.

Conclusion

In conclusion, this server executes client request iteratively. Therefore, the server is not all that efficient because it can handle multiple client requests, but it executes them one at a time. The server is inefficient and slow because of the iterative execution and can make better use of multiple threads. The data illustrates that the turn-around time for execution increases with more clients because the client request is not being fulfilled it the time that it could be executed. The problem isn’t the fact that the commands are taking longer to execute, but the fact that the client has to wait for another client request to finish before they can have their request fulfilled.

It would be interesting to investigate further with a larger data set for a larger number of clients. As the raw data had some particular outliers which could have been caused by other traffic or processes running on the server while testing.

Lessons Learned

We learned more about working with threads and making two separate programs that can communicate with each other from different servers. We learned how to utilize threads to simulate multiple clients trying to make request to the server at the same time. We learned how to create a server that is capable of listening for a client connection and then taking request from the clients. We learned some new Linux commands. We learned about the Runtime class and the Process class to execute the Linux commands on the server. We learned how to effectively work as a team in getting this project complete in an efficient time and manner. We didn’t have to much problems, but we did have a problem learning how to log into the server. Additionally, we had trouble sending in multiple lines of output from the server to the client to display in a nicely formatted result.