Concurrent Socket Server

Ethan Coco

Shariah Haque

David Tovar

CNT4504 - Computer Networks & Distributed Processing

Professor Kelly

Introduction

The purpose of this programming project is to modify the existing single-threaded Iterative Server to implement an *extend Thread* class, thus converting it into a Multi-Threaded Server. With this modification we aimed to test, analyze, and study how a set number of clients - spawned through a similarly multithreaded Client program - affects the former's average turn-around time for processing each of their requests. The overall goal of implementing these server and client programs is to collect data regarding the server's ability to handle increasingly large request queues from clients, determine its level of efficiency, and ultimately determine the effectiveness of the Multi-Threaded Server's new concurrency.

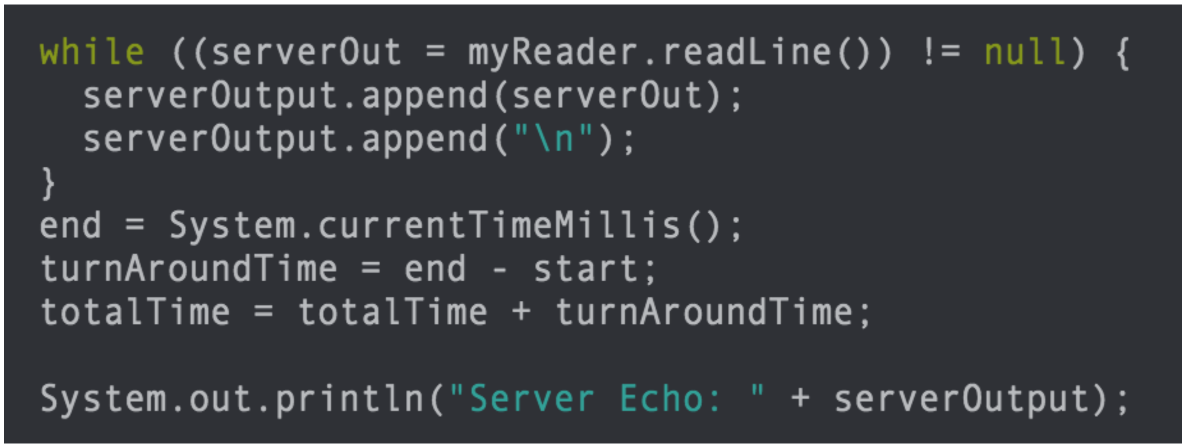
Both server and client programs were previous iterations of an earlier project, implemented via Java with concurrency in mind, and were coded to communicate on a specific IP address and port. Once both programs are set with the correct port and IP address, user input is then accepted on the client's side. The command input for whichever option presented in the menu is then sent to the server's program, evaluated via Linux commands, and is then sent back as a result for the client program to receive and subsequently print out to the user. This output is in the form of an echo, reflecting the processes executing in the Server, and their resulting turn-around times.

All in all, this report will highlight and detail the steps taken to set up the two Client and Server programs and confirm their new parallel communication, as well as walk through the methods we had employed in both testing and collecting the data regarding our new Multi-Threaded Server. We will also report on the changes in individual and average turn-around times for all commands involved as the number of client requests increase, and compare said data to those we had gathered previously in our Iterative Report, in order to determine situations that each type might benefit.

Client-Server Setup and Configuration

Two programs were repurposed for this project: the Concurrent, newly multi-threaded Server, and the existing Multithreaded Client; both coded to be used as console applications for SSH Clients like Bitvise. As stated previously, the Concurrent counterparts to their previous Iterative versions were designed with simplicity in mind, as the Server and Clients are solely meant to be operated via a text terminal or command line interface, and thus possess no real graphical interface (with only the imitation of a GUI; as a menu, presented on the Client's side for ease of readability for the user).

For the purposes of this project, the Client had little to no notable changes to the structure of its code, bar the implementation of print statements detailing "Server Outputs", aiming to echo the processes that were occurring Server-side.



```
while ((serverOut = myReader.readLine()) != null) {
  serverOutput.append(serverOut);
  serverOutput.append("\n");
}
end = System.currentTimeMillis();
turnAroundTime = end - start;
totalTime = totalTime + turnAroundTime;

System.out.println("Server Echo: " + serverOutput);
```

Server Echo: A simple modification to the Client's previous counterpart, appending results from the Multi-threaded Server and printing a message explicitly declaring Server Output. Those running the program are now distinctly aware of echo messages that might otherwise be missed due to the parallel nature of the Client and Server.

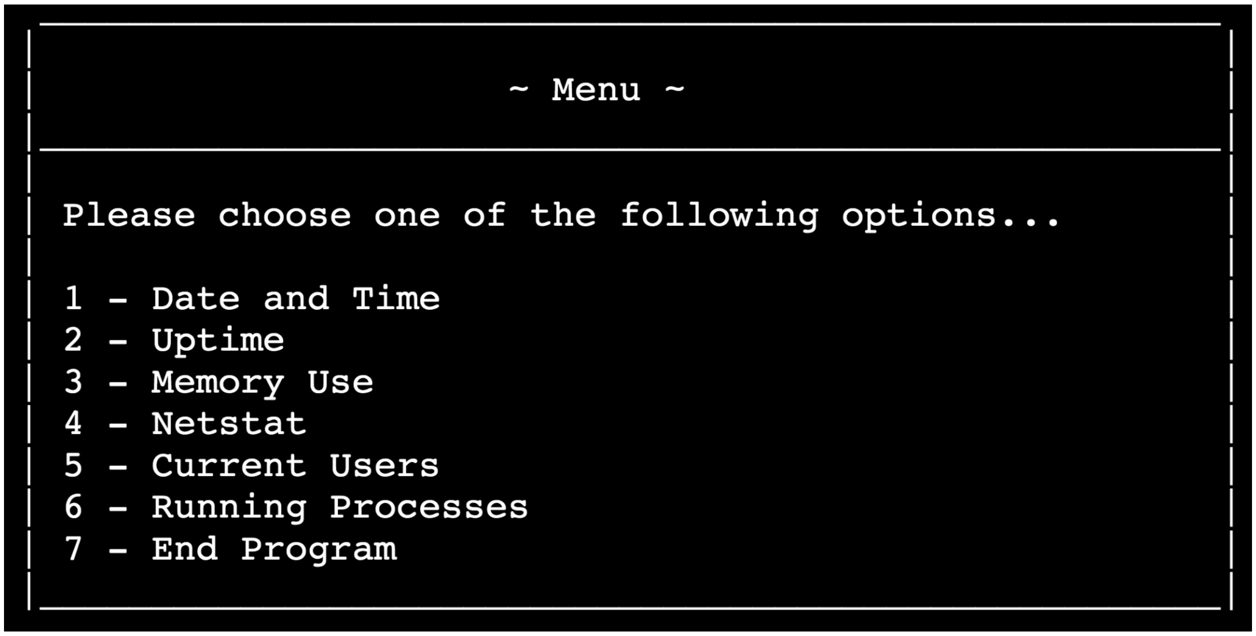
The Server will be running first on the terminal, connected to a predetermined port, in order for the Client to find a socket-accepting server and successfully connect. The Server's default IP and port is that of CNT4504's local machines, and is programmed to accept the Client, its connections, and evaluate its incoming requests in what is now a multi-threaded manner– where each thread creates a unique object. It was redesigned to handle multiple tasks in parallel with a few changes made to the primary code, and will now spawn a new process to service a client connection, tasking itself to then listen for the next client connection.

Along with the Multi-threaded Server, the Multi-threaded Client is programmed to send a number of client requests to the Server based on user input from the command menu (regarding

Data & Time, Uptime, Memory Usage, Netstat, Current Users, and Running Processes respectively via Linux commands), ranging from 1 to 25 clients at one time. Requests made from the Client's menu, numbered 1 to 7, are accepted integers via the nextInt scanner method, and when it comes to closing the connection, a continuing key feature of both the Client and Server's code is that the two programs can simultaneously shut down with only user input on the Client's side. With this simultaneous disconnection, the Server would not have to run idle to wait for another connection (or be forced out of its operation via the Linux Control + C command), and instead can gracefully disconnect its socket when the Client program is terminated.

Testing and Data Collection

For the new Concurrent Server, the prior description of the Server and Client establishing a connection through a socket is set up, and the Server is then prepped for testing. The goal of this testing phase was to keep an eye on how the Server would deal with incoming requests; ideally it should generate multiple threads to accept the numerous requests from multiple clients simultaneously, and confirmation would occur after examining trendlines found in resource-intensive processes (such as Running Processes).



```
 ________________________________________________________
|                                                        |
|                        ~ Menu ~                        |
|________________________________________________________|
|                                                        |
| Please choose one of the following options...          |
|                                                        |
| 1 - Date and Time                                      |
| 2 - Uptime                                             |
| 3 - Memory Use                                         |
| 4 - Netstat                                            |
| 5 - Current Users                                      |
| 6 - Running Processes                                  |
| 7 - End Program                                        |
|________________________________________________________|
```

Client Menu: The user is greeted with a simple menu on the Client's side, and is prompted to select a number for the specified command.

Connection via a server socket is first established, and - on the Client's side - a menu then is displayed to the user, prompting them to choose between Commands 1 to 6 for the Client to perform (Command 7 is not testable, as it is a simultaneous disconnect from both the Client and Server). The chosen command is not sent to the Server until the user also chooses the specific amount of clients to spawn in for the request (from a list of 1, 5, 10, 15, 25 and 100 clients). With a client amount specified, the command is then sent for the Server to evaluate via Linux commands, display on its side, and then return an output to the Client for it to calculate the total and average amount of run time for each action a thread performed. All six server operations were tested. For each command executed, 5 clients were initially spawned in as a "control" group, in order to later determine any relationship between the turn-around time and each operation when upwards to 100 clients were then spawned in to test the Server.

Attached below are tabular data for only the total amount of run time for each operation: Data & Time, Uptime, Memory Use, Netstat, Current Users, and Running Processes. Due to their length, the tables for the individual response times are separately attached from the Concurrent Server report, in the form of excel sheets.

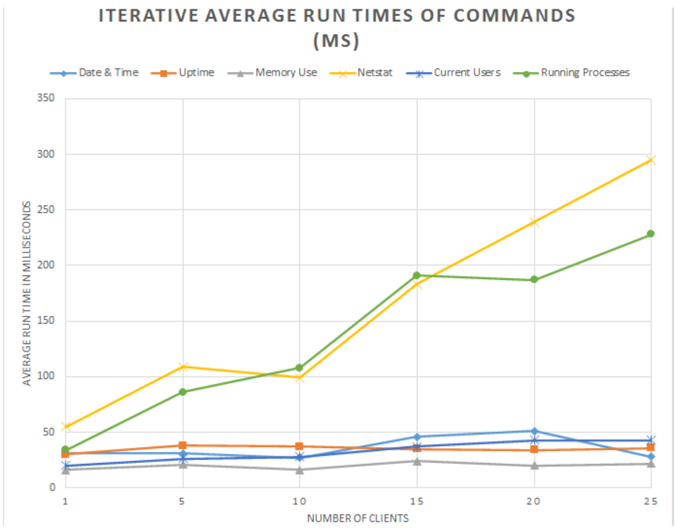
Concurrent Total Response Times (ms)							
	NUMBER OF CLIENTS REQUESTED						
COMMANDS (1-6)	**1**	**5**	**10**	**15**	**20**	**25**	**100**
Date and Time	21	231	451	998	1125	1463	6238
Uptime	27	251	591	646	1068	1608	5094
Memory Use	23	154	337	423	756	855	4692
Netstat	69	255	1073	1643	3197	4814	50666
Current Users	24	160	448	913	1725	2244	6459
Running Processes	16	161	475	1097	1709	3567	24261

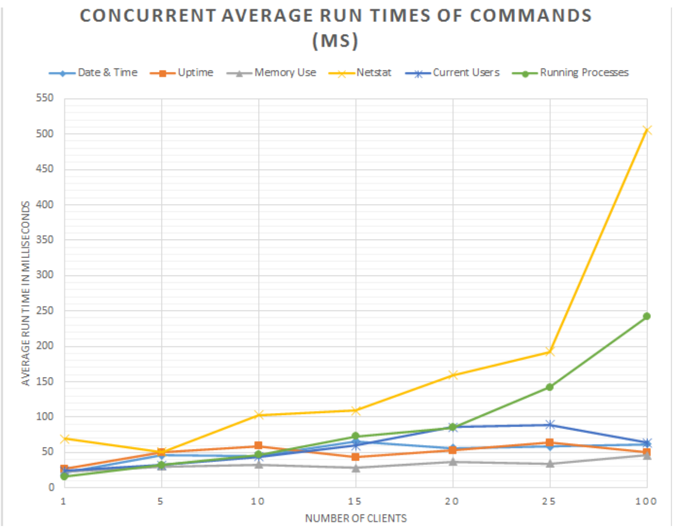
The total concurrent response times for each command are displayed above, in this table, and are meant to directly compare with the previous data gathered from our Iterative Server.

Iterative Total Response Times (ms)							
	NUMBER OF CLIENTS REQUESTED						
COMMANDS (1-6)	**1**	**5**	**10**	**15**	**20**	**25**	**100**
Date and Time	31	157	275	702	1039	717	n/a
Uptime	30	190	370	530	689	900	n/a
Memory Use	16	106	162	365	405	570	n/a
Netstat	55	547	992	2749	4786	7387	n/a
Current Users	20	131	280	562	870	1079	n/a
Running Processes	34	430	1085	2875	3749	5713	n/a

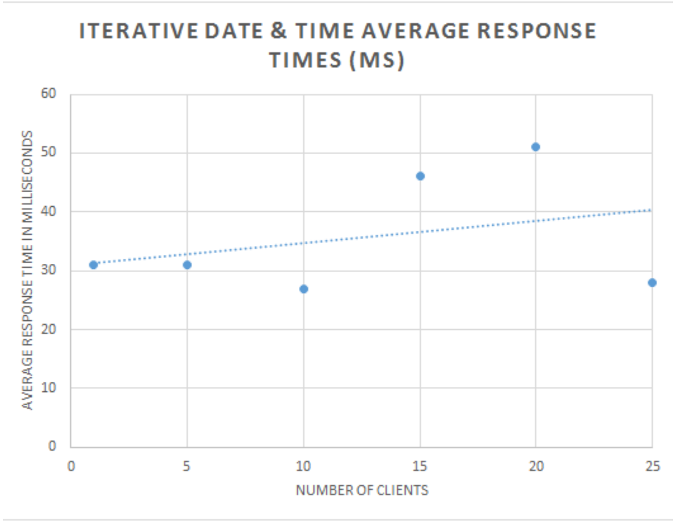
The total iterative response times for each command are displayed above, in this table. Note the intensive Running Processes having a considerably higher Total Response Time, compared to its Concurrent counterpart.

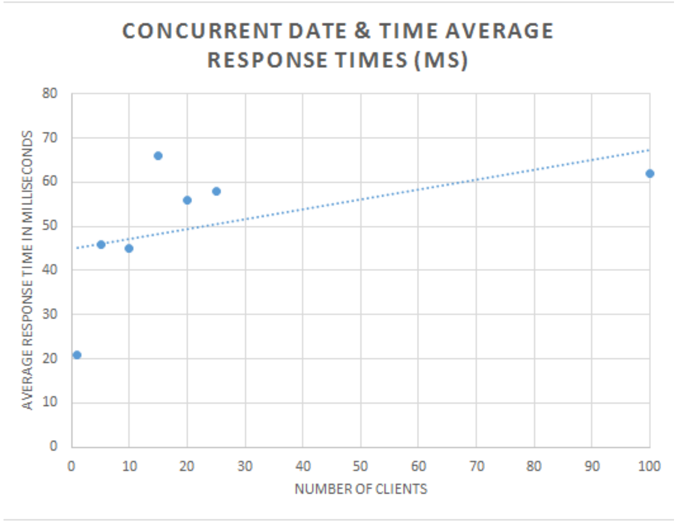
These tables are the results of the total turn-around time for each process, indicating a general upward trend for both individual and total turnaround turn-around times much like the Iterative Server, yet appearing to operate much more efficiently on the typically “consuming” processes.



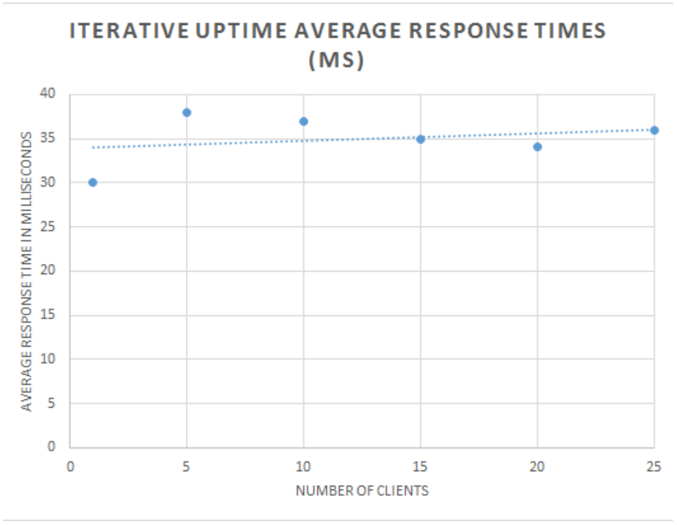


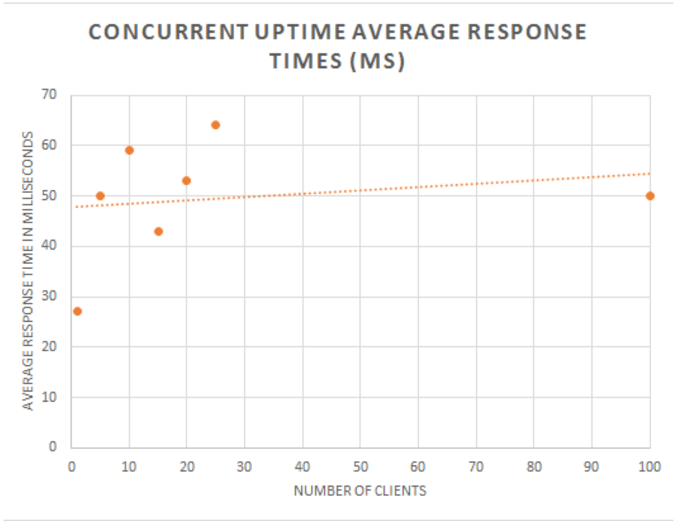
The average Iterative and Concurrent response times for each command are displayed above, respectively, in this line graph. Apart from the steep inclination of Netstat’s 100 client requests, note that the trendline for both Netstat and Running Processes are considerably more “shallow”.



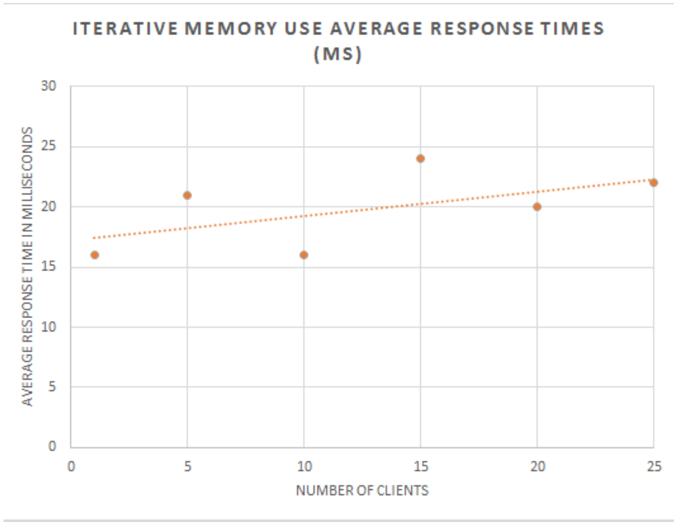


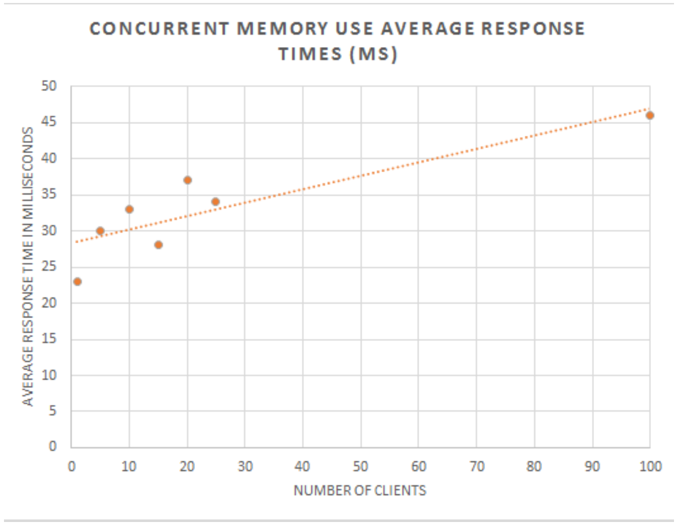
The Iterative and Concurrent average amount of run time for Date & Time, respectively, is catalogued via a scatter plot, above.



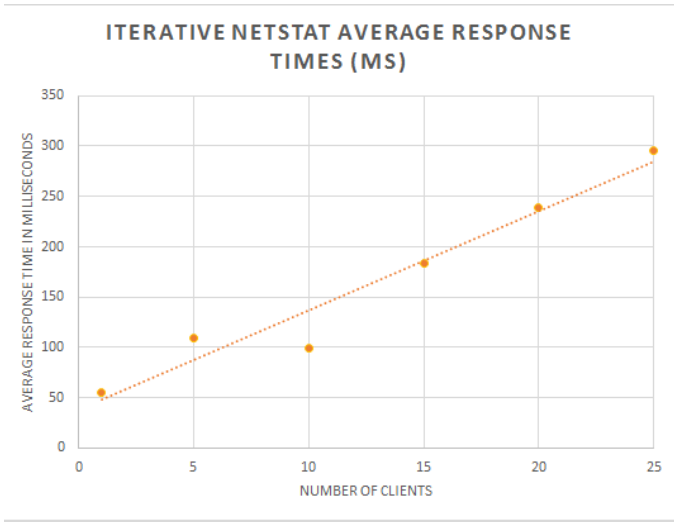


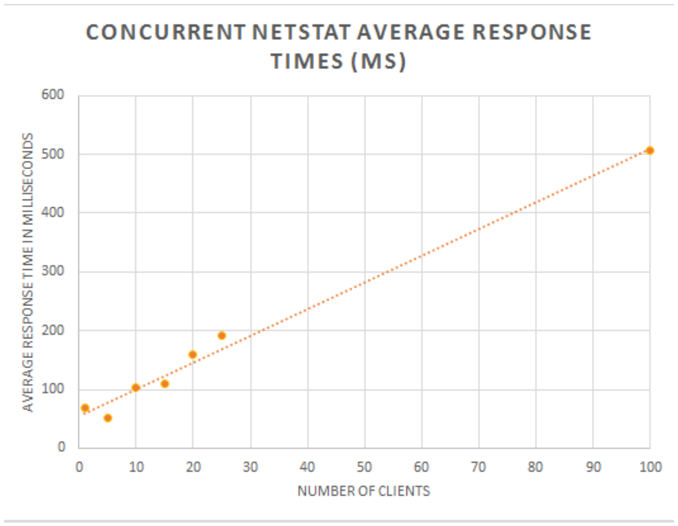
The Iterative and Concurrent average amount of run time, respectively, for Uptime is catalogued via a scatter plot, above.



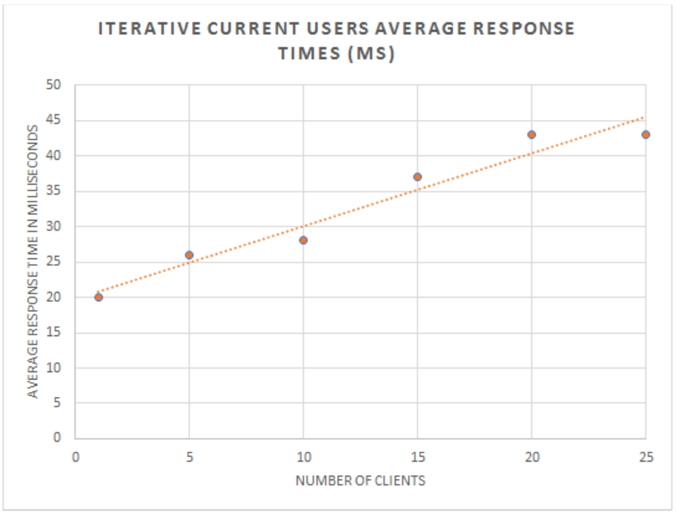


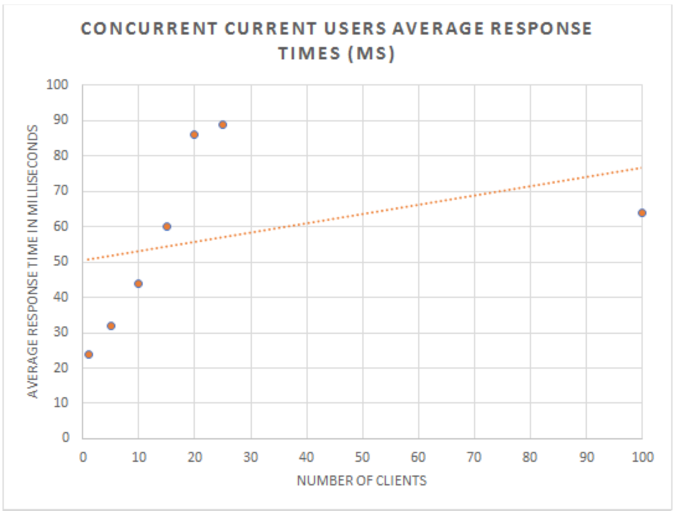
The Iterative and Concurrent average amount of run time, respectively, for Memory Use is catalogued via a scatter plot, above.



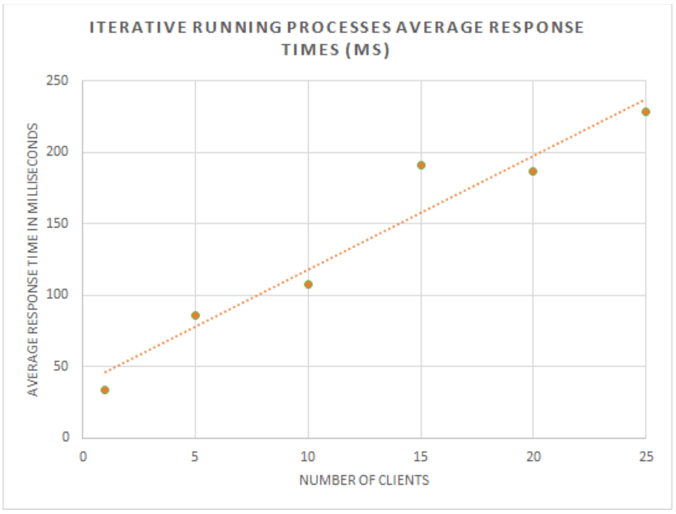


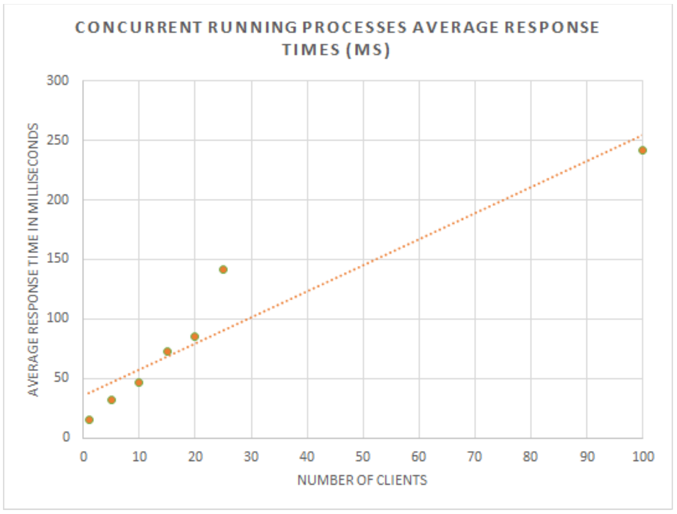
The Iterative and Concurrent average amount of run time, respectively, for Netstat is catalogued via a scatter plot, above.





The Iterative and Concurrent average amount of run time, respectively, for Current Users is catalogued via a scatter plot, above.





The Iterative and Concurrent average amount of run time, respectively, for Running Processes is catalogued via a scatter plot, above.

All Concurrent scatter plots for each command listed indicate a general upward trend for average turn-around times, which is understandable given the conclusion drawn from the previous Iterative Server, yet a select few trendlines exhibit a “shallower” inclination as compared to its Iterative counterparts, suggesting an increase in efficiency.

Also note that while some processes for the Concurrent averages seem to exhibit a higher trendline, the discrepancy is most likely due to the additional request for upwards to 100 clients– a generated amount of requests that was not present for the Iterative Server’s data collection period.

Data Analysis

Initial analysis was focused on the aftereffects of increasing the number of clients on the Concurrent Server, and determining if said increase had the same direct correlation to the turn-around time for each individual client that is generated afterward. The results are definitely of interest, as for the most part the results had mirrored those of the Iterative Server. The accumulated clients on individual turn-around time for the first three operations (Date and Time, Uptime, and Memory Use) was negligible as these processes are relatively not as intensive, and again the turn-around time for individual clients sent through the command line was not greatly affected (as demonstrated with a slight, gradual increase of elapsed time in the tabular data rather than a steep one). No firm patterns can be established, and while the fluctuations *could* be disregarded afterwards, something of interest to note is that the Concurrency of the Server kicks in after approximately 50 client requests (as seen in Date and Time, Uptime and Memory Use).

This could potentially mean that, while the Multi-Threaded Server tends to lag when it comes to producing results of a small amount of less intensive processes, it favors an increase in load as the time it takes to process the requests is now increased, and more favorable than the time it had taken the Server to *create* threads to deal with such minute processing levels like a single client. This idea carries along with the last three operations (Netstat, Current Users, and Running Processes) where the changes were noticeable, and a trendline could be firmly established if the tabular data were converted into scatterplots. There was a direct linear relationship between the number of users generated and individual client turn-around time, but compared to the Iterative results the Concurrent trendline would be noticeably less steep, presumably because the server was able to handle more things at once.

The next area analyzed was if Concurrent average turn-around time was similarly affected when the number of clients generated were increased. Again, much like the Iterative Server results, the Concurrent results were almost completely similar to the individual turn-around times, and it can be noted that the number of clients directly affects the average turn-around time performance. As more clients are requested, an increase of average turn-around time is observed (not as sharply induced as before).

Yet it should be noted that nearly all trend lines for intensive processes were flatter than observed from the Iterative results. As shown from the scatterplots, the commands (Netstat and Running Processes) that had a high increase in both individual and average turn-around times as the number of client requests go up in the Iterative Server were now beginning to take less time to parse through, featuring a flatter trendline than the former Iterative operations. Once again, Memory Use and Uptime's trendlines are level, if a little steeper on the former's part.

Situations that favor the conditions of an Iterative Server - where one request can be processed at a time with the other requests being "shelved" via queuing - would be for connections or exchanges that are quick, and do not last long for queues to build up quickly. And in the case of these exchanges or transactions building up quickly, a Concurrent server would be better suited, as it would be able to handle more load. Because the Server would be able to do multiple things at once, it would be a lot harder to break the server down via overloading.

Conclusion

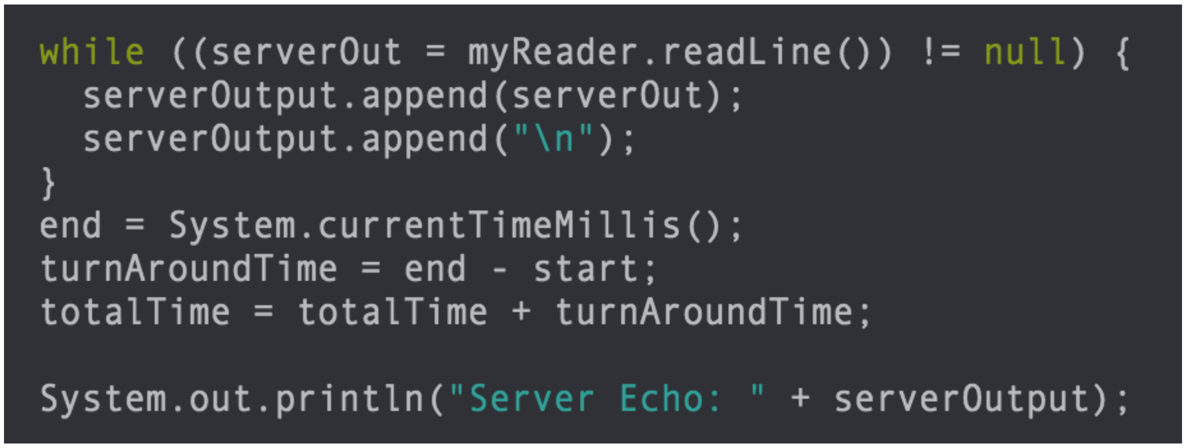
From the data gathered in our line graphs and scatter plots, we can soundly come to the conclusion that Netstat and Running Processes, one of most consuming processes for the Iterative Server, can be handled much more efficiently via a Multi-Threaded Server, as the total turn-around times for 25 clients (4814 ms and 3567 ms) and average turn-around times (192 ms and 142 ms) are significantly less than the Iterative's total turn-around time of 7387 ms and 5713 ms, respectively, and average turn-around times of 295 ms and 228 ms. Current Users and Uptime, however, are still less consuming than the former two, but do not beat the Iterative Server's total turn-around time of 1079 ms and 900 ms. Date & Time and Memory Use, consumers of least amount of resources, also show this trend despite the shallow trendline, as their Iterative totals of 717 ms and 570 ms are still less than those of the Concurrent's 1463 ms and 855 ms.

In conclusion, while a multi-threaded Server dealing with multi-threaded requests can have a quicker response time for extensive processes, there *can* be a cost of concurrency, as receiving a request and spawning a new thread for relatively quick operations takes up time and consumes system resources. So for dealing with minute levels of clients, such as the sole request up to 25 at the most, an Iterative Server would exhibit a lower delay than the Concurrent Server version. If the Iterative Servers were instead making clients wait for service and leaving a performance bottleneck in its wake, then a Concurrent Server would be able to perform more efficiently.

Lessons Learned

To successfully implement a concurrent parallel nature to the Server, our group improved on the multithreading using a different class and run function. To support the multithreading capabilities of the Server, we added a constructor to pass the socket data between functions, extending the Thread class and allowing the Server to differ with the results of its Iterative counterpart in terms of overhead (especially with the two most intensive processes).

However, a point of interest that we had not been entirely expecting was the result of how the Server and Client's parallel nature would affect the order of echoed data. Oftentimes the echoed data playing back within the Client would send out of order, thus creating potential confusion in pinpointing the location of the turnaround times amidst the results of the processes (and this difficulty would increase with processes such as Netstat). In an attempt to combat the parallel natures of the programs, we had initially implemented a "pausing execution with sleep", in hopes of delaying the process and allowing time for the messages to arrive chronologically. In other test iterations, we had briefly entertained the idea of making the data more visually recognizable via a *Thread.getId()* method. However, due to our inherent lack of familiarity with these implementations, we had feared that causing the current thread to suspend execution for a specified period, or potentially delaying the concurrency of the Server in favor of sequential data, would negatively affect the trend lines within our scatter graphs. We had thus elected to go for a more rudimentary "fix": the aforementioned labelling of exactly which material was the echoed data.



```java
while ((serverOut = myReader.readLine()) != null) {
  serverOutput.append(serverOut);
  serverOutput.append("\n");
}
end = System.currentTimeMillis();
turnAroundTime = end - start;
totalTime = totalTime + turnAroundTime;

System.out.println("Server Echo: " + serverOutput);
```

As stated previously, the results from the Multi-threaded Server were appended to a print statement explicitly declaring Server Output, and while the solution is a lot more elementary than a *Thread.getId()* method, this proved to at least show distinctions between the echoed messages and the turn-around calculations independently derived from the Client's side (we would also comment out the echoed data during the testing phase of the Multi-Threaded server for convenience).

An additional lesson learned involves taking care of and dealing with possible resource leaks. This was important during the creation of the multiclient function within the server. Along with this, some additional lessons learned include the importance of multiple functions for readability and function. An additional idea to improve our program originally included adding functions to write and print the server output, yet we focused on the aforementioned aspects, as commenting out portions of the code served to be a suitable substitute for converting print statements for CSVs.