Iterative Socket Server

Ethan Coco

Shariah Haque

David Tovar

CNT4504 - Computer Networks & Distributed Processing

Professor Kelly

Introduction

The purpose of this programming project is to create a single-threaded Iterative Server to test, analyze, and study how a set number of clients - spawned through a 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, and ultimately determine its level of efficiency.

Both server and client programs were implemented via Java, 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 in the form of a turn-around time.

All in all, this report will highlight and detail the steps taken to set up the two Client and Server programs, as well as walk through the methods we had employed in both testing and collecting the data regarding our Iterative 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.

Client-Server Setup and Configuration

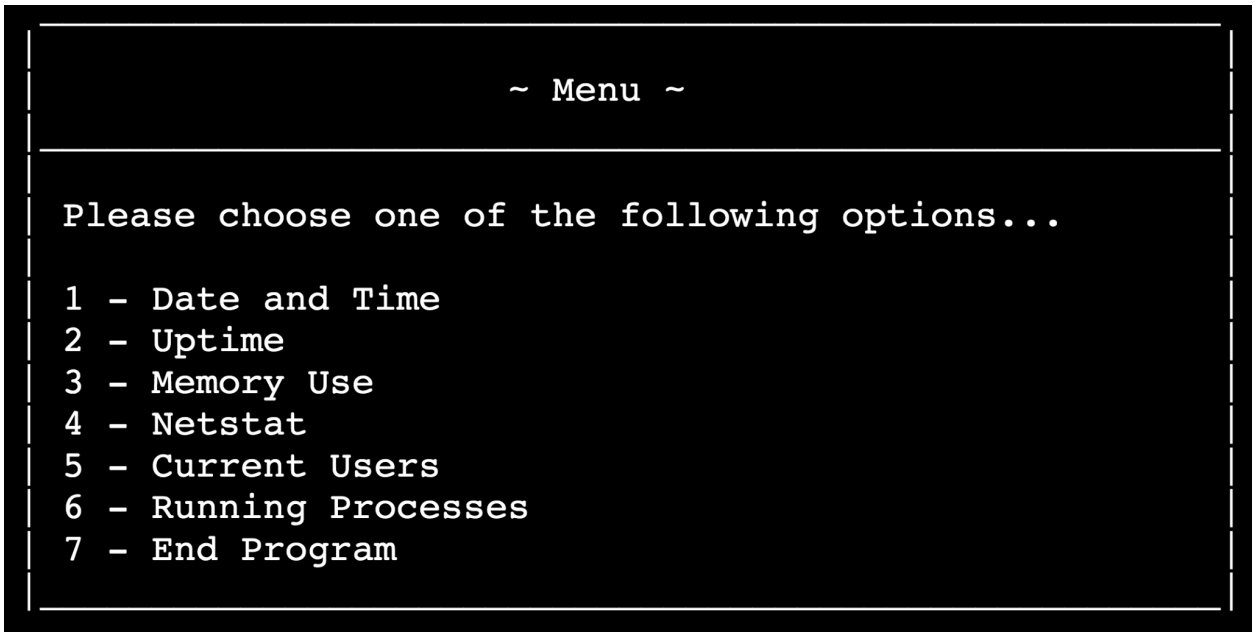
Two programs were created for the purposes of this project: the Iterative, single-threaded Server, and the Multithreaded Client; both coded to be used as console applications for SSH Clients like Bitvise. The programs were designed with simplicity in mind, as the Server and Clients are solely meant to 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).

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 a single-threaded manner– only handling one task at a time and queuing up additional requests if need be. Despite behaving in an iterative manner, the Server could very well be redesigned to handle multiple tasks in parallel with a few changes made to the primary code.

Along with the Iterative 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 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 Iterative 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.



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 and 25 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 25 clients were then spawned in to test the Server.

Attached below are tabular data for the total and average amount of run time for each operation: Data & Time, Uptime, Memory Use, Netstat, Current Users, and Running Processes.

Date & Time:

COMMAND 1: DATE & TIME						
Individual Elapsed Response Times (ms)	**Number of Clients Requested**					
Clients Spawned For Command	**1**	**5**	**10**	**15**	**20**	**25**
Request 1	31	30	28	49	51	32
Request 2	-	29	29	44	54	35
Request 3	-	30	25	48	46	29
Request 4	-	34	26	45	55	36
Request 5	-	34	36	41	51	38
Request 6	-	-	25	46	54	29
Request 7	-	-	27	45	53	30
Request 8	-	-	28	50	53	29
Request 9	-	-	27	48	53	37
Request 10	-	-	24	47	53	25
Request 11	-	-	-	53	45	27
Request 12	-	-	-	48	43	27
Request 13	-	-	-	48	43	26
Request 14	-	-	-	45	55	27
Request 15	-	-	-	45	54	30
Request 16	-	-	-	-	55	26
Request 17	-	-	-	-	49	25
Request 18	-	-	-	-	57	26
Request 19	-	-	-	-	57	23
Request 20	-	-	-	-	58	26

Request 21	-	-	-	-	-	27
Request 22	-	-	-	-	-	29
Request 23	-	-	-	-	-	25
Request 24	-	-	-	-	-	28
Request 25	-	-	-	-	-	25
Total Response Times (ms):	**31**	**157**	**275**	**702**	**1039**	**717**
Average Response Times (ms):	**31**	**31**	**27**	**46**	**51**	**28**

Uptime:

COMMAND 2: UPTIME						
Individual Elapsed Response Times (ms)	**Number of Clients Requested**					
Clients Spawned For Command	**1**	**5**	**10**	**15**	**20**	**25**
Request 1	30	37	37	29	44	38
Request 2	-	37	40	35	36	36
Request 3	-	40	35	35	39	32
Request 4	-	38	35	35	33	42
Request 5	-	38	37	44	39	43
Request 6	-	-	34	40	33	43
Request 7	-	-	36	36	33	30
Request 8	-	-	37	34	32	33
Request 9	-	-	37	32	30	40
Request 10	-	-	42	32	40	34
Request 11	-	-	-	41	40	33

Request 12	-	-	-	33	32	37
Request 13	-	-	-	30	32	36
Request 14	-	-	-	41	33	29
Request 15	-	-	-	33	30	38
Request 16	-	-	-	-	33	41
Request 17	-	-	-	-	32	35
Request 18	-	-	-	-	33	29
Request 19	-	-	-	-	31	32
Request 20	-	-	-	-	34	40
Request 21	-	-	-	-	-	39
Request 22	-	-	-	-	-	29
Request 23	-	-	-	-	-	34
Request 24	-	-	-	-	-	38
Request 25	-	-	-	-	-	39
Total Response Times (ms):	**30**	**190**	**370**	**530**	**689**	**900**
Average Response Times (ms):	**30**	**38**	**37**	**35**	**34**	**36**

Memory Use:

COMMAND 3: MEMORY USE						
Individual Elapsed Response Times (ms)	**Number of Clients Requested**					
Clients Spawned For Command	**1**	**5**	**10**	**15**	**20**	**25**
Request 1	16	18	14	23	17	27
Request 2	-	23	13	32	18	30

Request 3	-	19	14	24	27	18
Request 4	-	19	16	21	15	28
Request 5	-	27	23	22	18	17
Request 6	-	-	16	18	27	18
Request 7	-	-	15	19	15	18
Request 8	-	-	16	28	23	29
Request 9	-	-	21	28	27	30
Request 10	-	-	14	20	20	20
Request 11	-	-	-	21	14	19
Request 12	-	-	-	29	21	28
Request 13	-	-	-	28	22	18
Request 14	-	-	-	22	20	28
Request 15	-	-	-	30	20	28
Request 16	-	-	-	-	21	25
Request 17	-	-	-	-	21	24
Request 18	-	-	-	-	21	20
Request 19	-	-	-	-	22	22
Request 20	-	-	-	-	16	22
Request 21	-	-	-	-	-	22
Request 22	-	-	-	-	-	22
Request 23	-	-	-	-	-	23
Request 24	-	-	-	-	-	16
Request 25	-	-	-	-	-	18
Total Response Times (ms):	**16**	**106**	**162**	**365**	**405**	**570**

Average Response Times (ms):	**16**	**21**	**16**	**24**	**20**	**22**

Netstat:

COMMAND 4: NETSTAT						
Individual Elapsed Response Times (ms)	**Number of Clients Requested**					
Clients Spawned For Command	**1**	**5**	**10**	**15**	**20**	**25**
Request 1	55	49	46	72	48	56
Request 2	-	85	61	59	72	86
Request 3	-	117	74	87	96	101
Request 4	-	137	90	111	120	117
Request 5	-	159	99	132	140	141
Request 6	-	-	106	153	159	161
Request 7	-	-	116	169	178	182
Request 8	-	-	125	189	199	199
Request 9	-	-	133	207	216	219
Request 10	-	-	142	221	233	240
Request 11	-	-	-	241	252	260
Request 12	-	-	-	255	269	279
Request 13	-	-	-	271	290	296
Request 14	-	-	-	284	308	311
Request 15	-	-	-	298	326	330
Request 16	-	-	-	-	342	344
Request 17	-	-	-	-	359	363
Request 18	-	-	-	-	375	390

Request 19	-	-	-	-	394	412
Request 20	-	-	-	-	410	432
Request 21	-	-	-	-	-	450
Request 22	-	-	-	-	-	470
Request 23	-	-	-	-	-	495
Request 24	-	-	-	-	-	518
Request 25	-	-	-	-	-	535
Total Response Times (ms):	**55**	**547**	**992**	**2749**	**4786**	**7387**
Average Response Times (ms):	**55**	**109**	**99**	**183**	**239**	**295**

Current Users:

COMMAND 5: CURRENT USERS						
Individual Elapsed Response Times (ms)	**Number of Clients Requested**					
Clients Spawned For Command	**1**	**5**	**10**	**15**	**20**	**25**
Request 1	20	21	24	25	30	32
Request 2	-	31	18	31	20	30
Request 3	-	20	17	32	25	28
Request 4	-	26	29	20	32	34
Request 5	-	33	20	22	36	23
Request 6	-	-	31	33	29	23
Request 7	-	-	32	36	23	19
Request 8	-	-	35	38	38	36
Request 9	-	-	36	40	40	37

Request 10	-	-	38	42	42	39
Request 11	-	-	-	43	46	40
Request 12	-	-	-	46	48	42
Request 13	-	-	-	50	50	45
Request 14	-	-	-	52	52	45
Request 15	-	-	-	52	54	46
Request 16	-	-	-	-	56	48
Request 17	-	-	-	-	59	49
Request 18	-	-	-	-	60	50
Request 19	-	-	-	-	63	55
Request 20	-	-	-	-	67	56
Request 21	-	-	-	-	-	56
Request 22	-	-	-	-	-	59
Request 23	-	-	-	-	-	62
Request 24	-	-	-	-	-	62
Request 25	-	-	-	-	-	63
Total Response Times (ms):	**20**	**131**	**280**	**562**	**870**	**1079**
Average Response Times (ms):	**20**	**26**	**28**	**37**	**43**	**43**

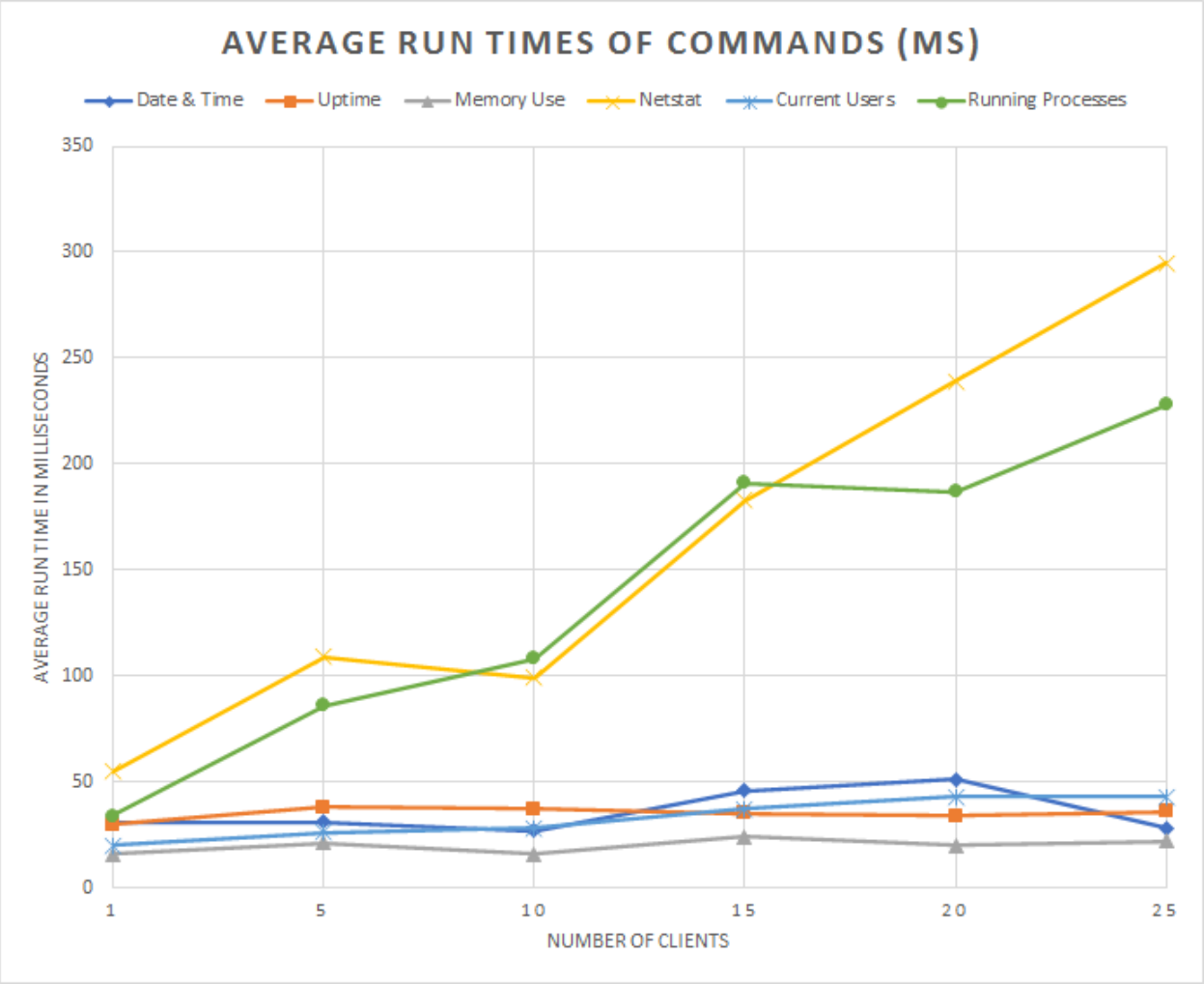
Running Processes:

COMMAND 6: RUNNING PROCESSES						
Individual Elapsed Response Times (ms)	**Number of Clients Requested**					
Clients Spawned For Command	**1**	**5**	**10**	**15**	**20**	**25**

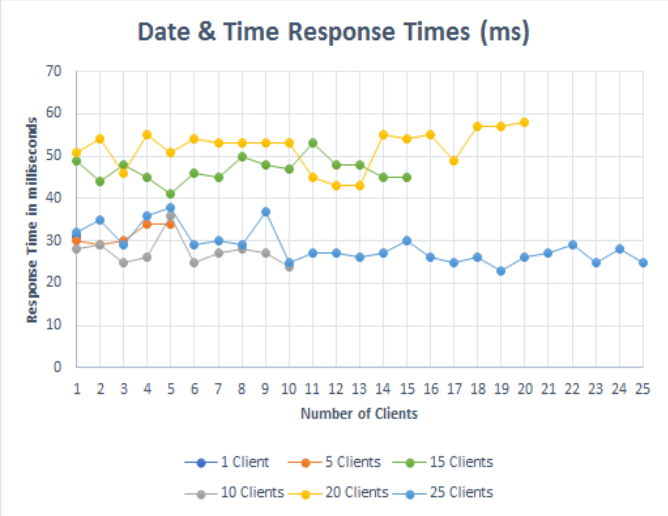
Request 1	34	42	40	48	53	41
Request 2	-	69	67	80	69	60
Request 3	-	86	78	97	84	79
Request 4	-	101	91	122	100	97
Request 5	-	132	105	141	117	115
Request 6	-	-	119	163	130	125
Request 7	-	-	130	176	143	143
Request 8	-	-	141	197	154	158
Request 9	-	-	151	218	167	177
Request 10	-	-	163	237	181	192
Request 11	-	-	-	251	194	205
Request 12	-	-	-	266	211	220
Request 13	-	-	-	280	224	237
Request 14	-	-	-	293	237	252
Request 15	-	-	-	306	250	265
Request 16	-	-	-	-	263	279
Request 17	-	-	-	-	275	293
Request 18	-	-	-	-	287	305
Request 19	-	-	-	-	299	316
Request 20	-	-	-	-	311	327
Request 21	-	-	-	-	-	339
Request 22	-	-	-	-	-	349
Request 23	-	-	-	-	-	361
Request 24	-	-	-	-	-	384
Request 25	-	-	-	-	-	394

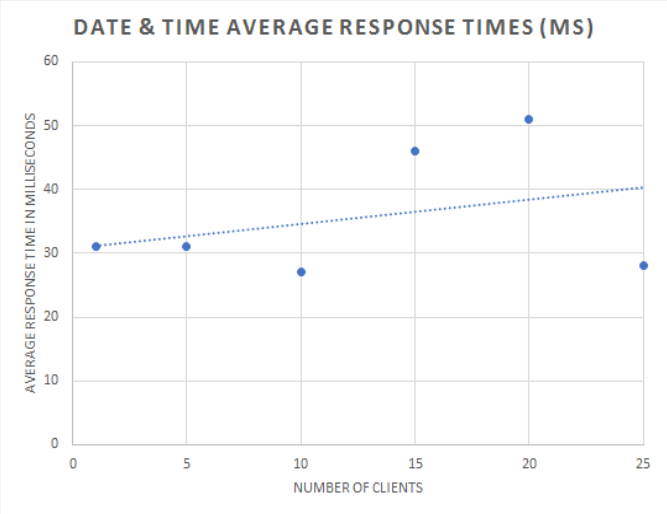
Total Response Times (ms):	**34**	**430**	**1085**	**2875**	**3749**	**5713**
Average Response Times (ms):	**34**	**86**	**108**	**191**	**187**	**228**

These tables are the results of the elapsed turn-around time for each client request, the total turn-around time, and the average turn-around time, indicating a general upward trend for individual and average turn-around times.

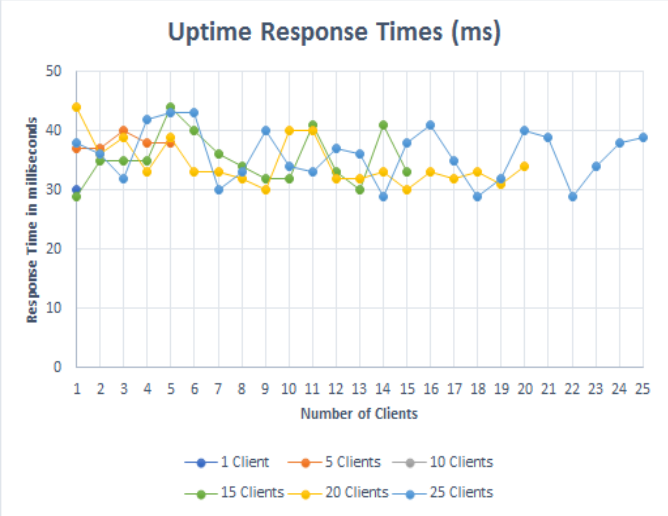


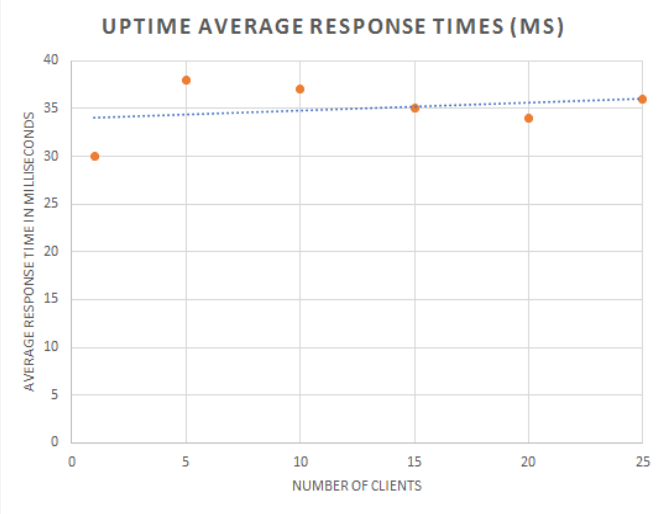
The average response times for each command are displayed above, in this line graph.



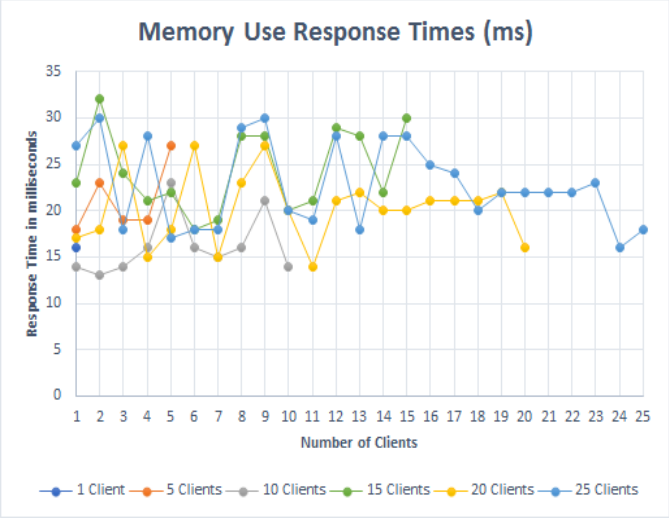


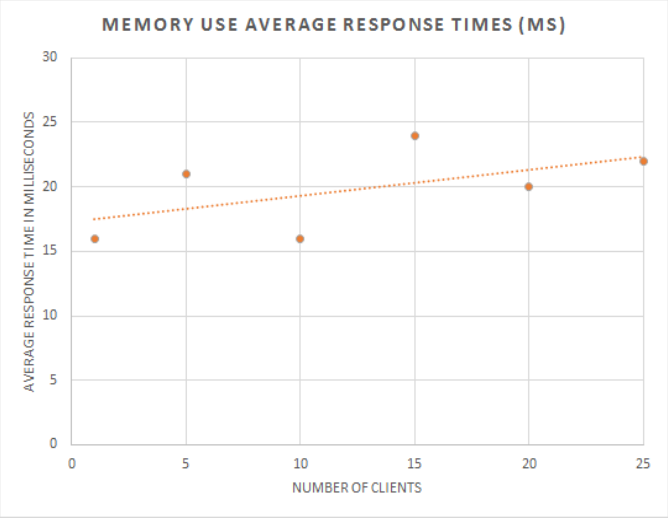
The individual and average amount of run time for Date & Time is catalogued via a scatter plot, above.



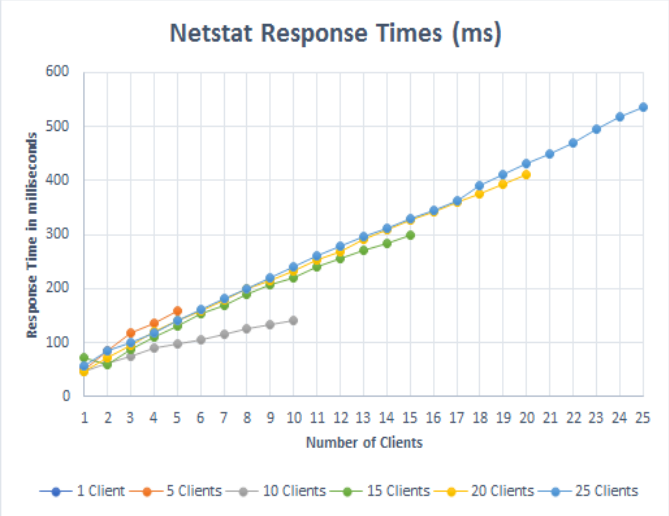


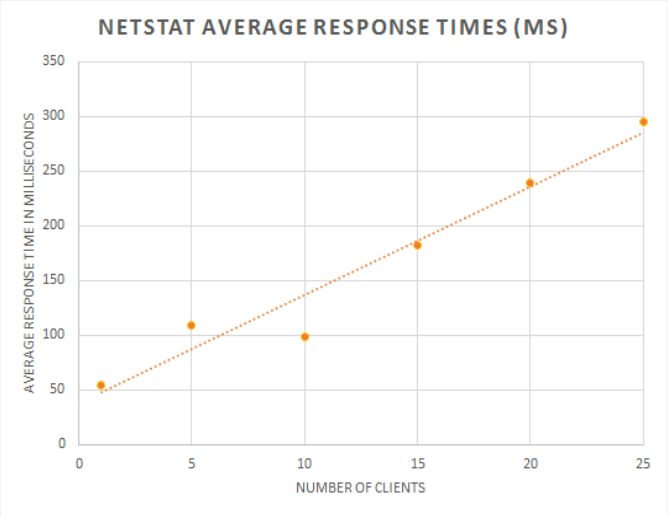
The individual and average amount of run time for Uptime is catalogued via a scatter plot, above.



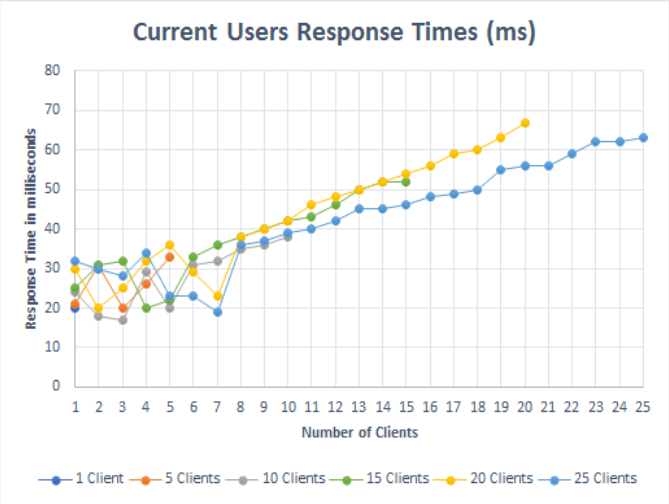


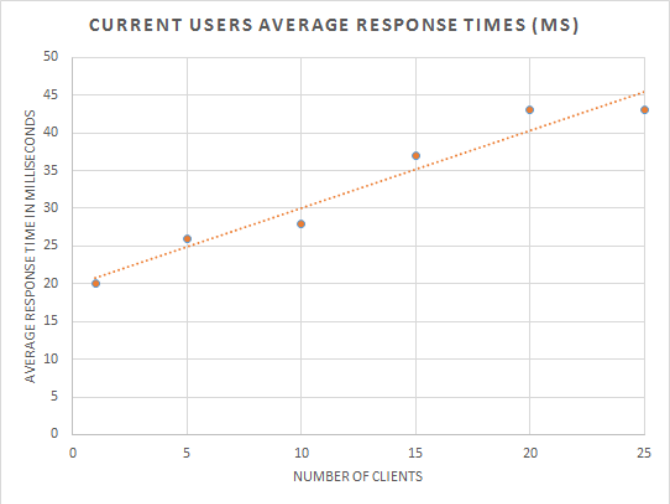
The individual and average amount of run time for Memory Use is catalogued via a scatter plot, above.



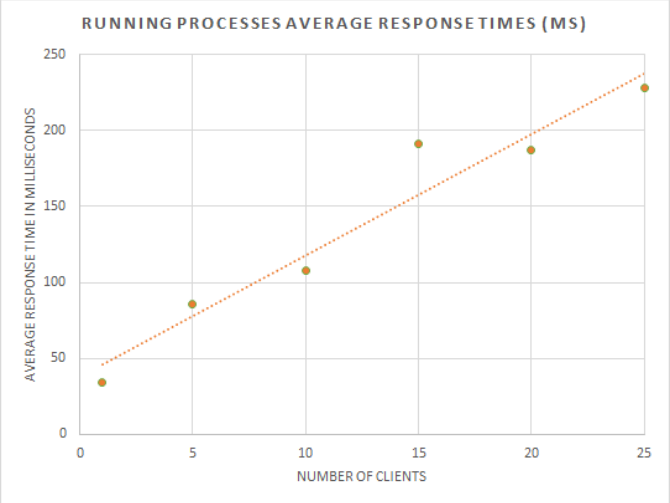


The individual and average amount of run time for Netstat is catalogued via a scatter plot, above.





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



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

All linear and scatter plots for each command listed indicate a general upward trend for individual and average turn-around times, meaning that the number of client requests have a plausible direct effect on the Server's efficiency.

Data Analysis

Initial analysis was focused on the aftereffects of increasing the number of clients, and determining if said increase had a direct correlation to the turn-around time for each individual client that is generated afterward. The result of the accumulated clients on individual turn-around time for the first three operations (Date and Time, Uptime, and Memory Use) was negligible, as the turn-around time for individual clients sent through the command line was not greatly affected (as demonstrated with a slight upward trendline rather than a steep one). A few occurrences of variance would show, as sometimes the elapsed response time of the second client would be a bit higher than the first (as shown in Uptime and Memory Use), but no firm pattern was established, and the fluctuations could be disregarded afterwards. However, for the last three operations (Netstat, Current Users, and Running Processes) the changes were noticeable, and a trendline could be firmly established. There was a direct linear relationship between the number of users generated and individual client turn-around time. For some of the operations, the runtime was more significant than others (like in Netstat and Running Processes), but the relationship between both are still linearly positive. As shown from the scatterplots, the commands Netstat and Running Processes both have a high increase in individual turn-around times as the number of client requests go up. Date and Time - in contrast - stays relatively consistent with normal variation between commands (even dipping in individual turn-around after an uptick of 20 clients), and Memory Use and Uptime's turn-around times - while increasing - consistently feature a flatter trendline than the former two operations.

The next area analyzed was if average turn-around time was affected when the number of clients generated were increased. The 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, a sharp increase of average turn-around time is observed. The only real difference between the severity of the trendlines stemmed from the Current User's operation, where the relationship was still linear but noticeably flatter than both Netstat and Running Processes. The same can be determined for Memory Use and Uptime, as their trendlines are also seemingly level, even if a small increase of average turn-around time as the number of clients accumulate can be noted.

From what had been surmised earlier, the primary cause of the increase of both individual and average turnaround time is due to the nature of an Iterative Server, and the fact that the operations performed are increasingly demanding with additional clients. Only one request can be processed at a time with a single-threaded Server, leading the other requests to be "shelved" via queuing, and thus the requests for a specific command must wait longer to be processed.

Conclusion

From the data gathered in our line graphs and scatter plots, we can soundly come to the conclusion that Netstat and Running Processes are the most consuming processes for the Iterative Server, as they come to a total turn-around time of 7387 ms and 5713 ms, respectively, and average turn-around times of 295 ms and 228 ms. The second highest consumers in the server are Current Users and Uptime, which total at 1079 ms and 900 ms respectively, so while they are ultimately less consuming than the aforementioned two, the turn-around times for each command still increase, if only slightly. Date & Time and Memory Use consumes the least amount of resources, totalling at 717 ms and 570 ms (with even a dip in average turn-around time for Memory Use after 20 client requests). In conclusion, a single-threaded Server dealing with multi-threaded requests has to deal with multiple levels of complexity added to its commands (the increasing amount of clients spawned), which causes more demand on its system. Because the creation of each thread for a client consumes additional memory, the Server subsequently struggles in processing an increasingly lengthening request queue, and its efficiency decreases.

Lessons Learned

Our first encountered problem began when implementing multithreading into the client, as our group initially had little experience with threading. Using the 'Class Extends Thread' method found through the examples shared through the project prompt, we were able to effectively learn how to implement multi-threading. Adding another class to the client meant we would effectively have to communicate freely between classes to achieve the calculations necessary for data collection and threading. To effectively communicate between classes, static variables were shared between classes to allow for change and usage across the client. Although this worked well for the multi-threading class, it eventually created issues when trying to implement new ideas. An example of these issues were when implementing the necessary calculations for data collection, a variable definition was moved and eventually led to our data being calculated wrong. The issue was found and fixed, yet shows the importance of being careful and accurate when communicating between classes.

As most of the problems and lessons learned were resolved in the client, the server created little issues as it was single threaded, and Linux command implementation is highly documented online and in the references given. When encountering issues implementing Linux commands, the references and examples provided in the project prompt and lecture helped us to understand and encode Linux commands seamlessly, to allow focus on multithreading and connection issues.

Another problem we had to overcome included figuring out a "graceful" way to exit both the Server and the Client, and it eventually proved to be a unique challenge. System.exit(0), as it was originally coded as a "disconnection" for the programs, was considered to be a "brute-force" way to disconnect the Client from the Server. The Server also remained open and idle despite the severed connection, as the way it was coded meant that - even while exiting - the Server needed a "spawned Client request" in order to turn itself off. It definitely presented a special problem, and eventually we had coded in a solution that, in order to also turn the Server off when menu option 7 was selected in the Client, spawned a single client without additional user input.

Our final challenge was during the Iterative Server's testing phase. When we were collecting data, one of the issues was the amount of time it took. All the turn-around times were initially recorded on paper but we later realized that copying them to Excel would save time transferring the data to graphs. When transferring data from the terminals, we would need to extract only the specific data we needed at the time, excluding menus and other statements. To overcome this obstacle, we implemented Find and Replace features to extract the proper data. In the future, it would be worth directly printing the results of the turn-around times into a CSV file using Java file implementation. This would make it easier to transfer into Excel and make the data collecting process less daunting.