Concurrent Socket Server

Multithreaded Server & Multithreaded Client in Java

Jonathan Shih

CNT4504 - Computer Networks and Distributed Processing

J. Scott Kelly

July 27, 2021

INTRODUCTION

This programming project expands upon the concepts explored in the Iterative Socket Server project, which focuses on setting up a communications protocol for a single server and client program so that they can interact over a TCP/IP network, by developing a Concurrent Socket Server that is able to more efficiently process incoming requests. In the previous project, a single host program was coded to listen for client messages, process a command, and echo the results back to the client sequentially, while only the client program could generate individual threads to send concurrent requests to the host server.

The goal of this next stage of the project is to expand the host server to also contain multithreading capability. This lets the server program generate additional threads to match each of the multiple concurrent requests from clients. It also allows multiple clients to connect to the same server socket simultaneously via the same IP/Port pairing, make simultaneous requests, and close the socket when finished. This version of a concurrent server program represents a more robust model for server-multiple client communications, as this server should be able to perform queries with faster response times than the iterative version.

CLIENT-SERVER SETUP AND CONFIGURATION

Once again, the Java language was deployed for this project due to its extended library of classes including *ServerSocket, PrintWriter, BufferedReader*, as well as the *try-catch* mechanism to handle input error exceptions. Eclipse was used as the IDE, FortiClient for signing into the UNF VPN, and Bitvise SSH Client was used for file transfer and bringing up the terminal for compiling and testing the code.

Server.java was renamed to MultServer.java to differentiate between the first and second versions of this project. MultServer is launched from the "CNT4505D.ccec.unf.edu" server at IP address 139.62.210.153 and the port number is once again entered by the user as the first argument when executing the program, anywhere in the range between port 1025 to 4998. MultServer contains the main class, and the biggest difference from last time is that the main method now contains a continuous while loop that continues to listen for incoming connections and spins off a new thread every time a new client socket is accepted. So, when accept() is called on a new client socket (*cliSock*), a ServerThread is initialized and run by calling start(). When a new client socket is connected, ServerThread prints "New client connected..." to alert the user of a new thread. ServerThread is a new MultServer subclass that implements the Runnable interface. ServerThread essentially contains the critical code from the original Server.java program from Project 1, in that it attempts to get a BufferedReader input and a PrintWriter output to any connecting client socket, and when an input line is received from the client, the ServerThread handles it using ParseRequest switch method, runs the UNIX command, echoes the reply back to the client, and then signals that it's done transmitting with "EaZy123."

The Client2.java program contains "CNT4505D.ccec.unf.edu" or 139.62.210.153 as its IP address and the port number is taken in as provided by the user in arg[0] when executing the program from the command line interface. Similar to last time, this IP/port number pair is used to form a new Socket with the MultServer. Once the client is connected a loop displays a menu of various UNIX-based commands and prompts the user to enter a command and a number of requests to generate, or exit. There are again a number of error checking methods used to here ensure that the user input is valid. The biggest change this time around is setting the upper limit of requests to be set at 100, instead of capping it at 25. Once the input is validated, the client message is passed to a new RequestThread, which then calls sendRequest to send the command as output to the host server socket. Each RequestThread once again keeps track of the turn-around time of the server to complete each individual request. Client2 checks that all tasks are complete by calling the Boolean function checkAllThreadsComplete() and then prints the total and average turnaround time calculations to the console in milliseconds. Both programs once again run on a loop always listening until told to stop, additional client sessions can connect to the host server socket with the matching paired address and port, and every client can terminate the session and the socket instance with the "exit" command.

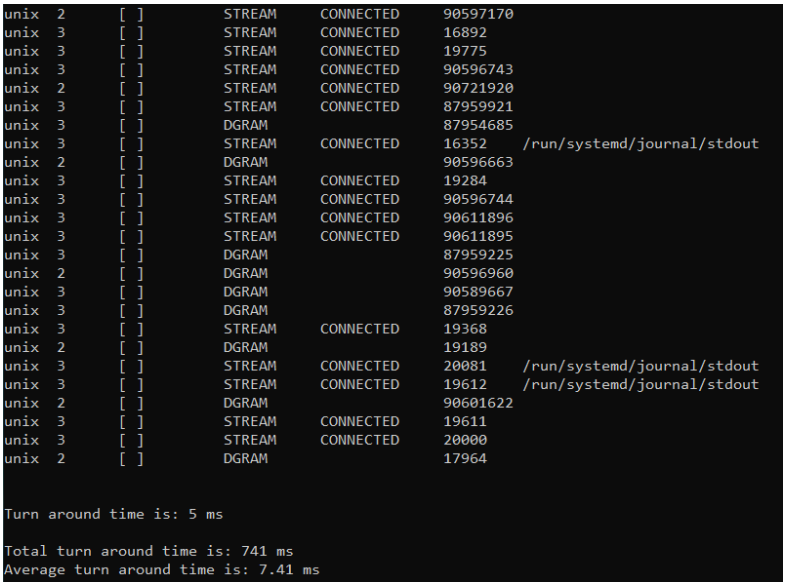
Example output with multiple clients connected and sending multiple requests:

```
n01447401@cisvm-cnt4505c:~$ java MultServer.java 2000
Running server...

New client connected...
Date and time requested...
Tue Jul 27 21:06:52 UTC 2021

New client connected...
Uptime requested...
 21:07:50 up 85 days,  7:24,  4 users,  load average: 0.02, 0.01, 0.02

Uptime requested...
 21:07:51 up 85 days,  7:24,  4 users,  load average: 0.02, 0.01, 0.02

Uptime requested...
 21:07:51 up 85 days,  7:24,  4 users,  load average: 0.02, 0.01, 0.02

Uptime requested...
 21:07:51 up 85 days,  7:24,  4 users,  load average: 0.02, 0.01, 0.02

Uptime requested...
 21:07:51 up 85 days,  7:24,  4 users,  load average: 0.02, 0.01, 0.02
```

TESTING AND DATA COLLECTION

The concurrent server-client program was methodically tested for each of the six commands for each of the number of requests (1, 5, 10, 15, 20, 25, 100) for five trial runs to get an average. Testing was done at around 6PM on a weekday, with approximately 5 to 7 other users currently logged onto the server. The additional traffic on the server likely led to longer turnaround times for certain commands due to increased network latency. This led to some uneven but interesting results, particularly with "Netstat" and "Processes." Overall, the host server exhibited very rapid and consistent response times, showing good performance even when hit with 100 requests.

Example output showing total and average turnaround times after 100 requests:



Test results from Project2 - Concurrent Server:

	Date and Time							Uptime						
Requests	1	5	10	15	20	25	100	1	5	10	15	20	25	100
	1	34.4	39.1	40.1	41.1	41.3	42.7	1	34.5	38.9	40.1	40.9	41.4	42.6
	1	35.4	39	40.1	40.1	41.3	42.7	1	34.8	38.7	40.2	41	41.3	42.6
	1	34.8	38.9	40.1	40.1	41.2	42.6	1	36.4	38.7	40.3	41.1	41.3	42.6
	1	34.2	38.8	40.3	40.3	41.3	42.6	1	34.4	38.8	40.3	40.1	41.4	42.6
	1	35	38.6	40.1	40.1	41.2	42.6	1	34.6	39	40.2	40.9	41.2	42.6
Avg (ms)	**1**	**34.76**	**38.88**	**40.14**	**40.34**	**41.26**	**42.64**	**1**	**34.94**	**38.82**	**40.22**	**40.8**	**41.32**	**42.6**
	Netstat							**Current Users**						
Requests	1	5	10	15	20	25	100	1	5	10	15	20	25	100
	12	13.4	5.4	11.8	7.1	4	4.3	7	39.2	42.9	44.3	45	45.4	46.7
	5	21.8	5.1	6.1	11.3	9.3	4.3	6	39	42.8	44.4	45.1	45.4	46.6
	5	5.2	9.3	5	7.1	4	5	6	39.2	42.9	44.7	45	45.3	46.7
	5	5.4	5.4	5.2	4.1	5.4	5	6	38.8	42.8	44.3	44.9	45.4	46.6
	5	5.2	5.3	5.6	7.5	7.9	5	6	39.2	42.7	44.3	45.2	45.3	46.6
Avg (ms)	**6.4**	**10.2**	**6.1**	**6.74**	**7.42**	**6.12**	**4.72**	**6.2**	**39.08**	**42.82**	**44.4**	**45.04**	**45.36**	**46.64**

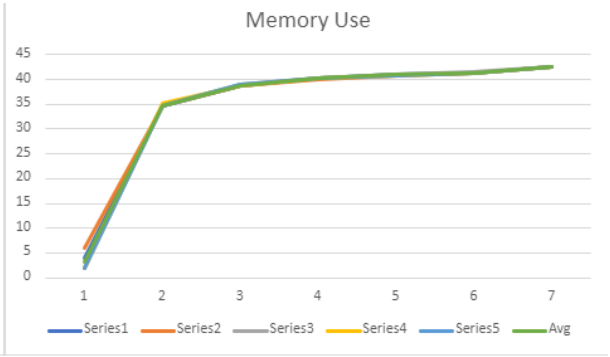
Test results continued:

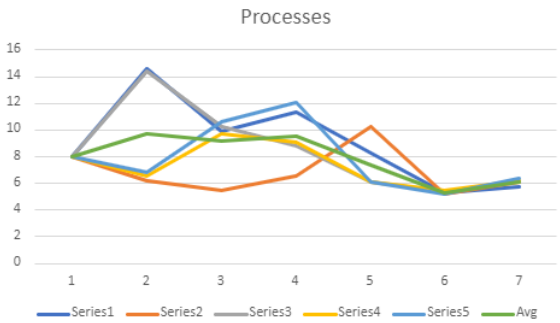
Memory Use						
1	5	10	15	20	25	100
4	34.6	38.8	40.3	41	41.3	42.6
6	34.8	38.8	40.1	40.9	41.4	42.7
2	34.8	38.9	40.3	41	41.5	42.6
2	35.2	38.7	40.3	40.9	41.2	42.6
2	34.6	38.9	40.3	40.9	41.3	42.6
3.2	**34.8**	**38.82**	**40.26**	**40.94**	**41.34**	**42.62**
Processes						
1	5	10	15	20	25	100
8	14.6	9.9	11.3	8.3	5.3	5.7
8	6.2	5.5	6.6	10.3	5.2	6.1
8	14.4	10.3	8.8	6.1	5.4	6.2
8	6.6	9.7	9.1	6.1	5.5	6.2
8	6.8	10.6	12.1	6.1	5.2	6.4
8	**9.72**	**9.2**	**9.58**	**7.38**	**5.32**	**6.12**

Results from Project1 - Iterative Server for contrast:

	Date and Time						Uptime						Memory Use					
Requests	1	5	10	15	20	25	1	5	10	15	20	25	1	5	10	15	20	25
	1	34.6	38.8	39.9	41.1	41.2	2	34.6	39.3	40.2	41.3	41.5	2	34.2	38.6	40.6	40.9	41.6
	1	34.4	39.3	40.3	40.8	41.4	1	34.6	38.8	40.4	41.2	41.3	2	34.4	38.6	40.3	40.9	41.2
	1	34.4	39.1	40.3	40.8	41.5	1	34.6	38.9	40.2	41	41.4	3	34.8	39.1	40.2	41	41.8
	1	34.6	38.5	40.3	40.9	41.3	1	34.8	38.7	40.5	41.2	41.3	2	36.4	38.6	40.3	40.9	41.3
	1	34.8	38.7	40.3	41.2	41.2	1	34.8	39.1	40.7	40.9	41.3	2	34.6	38.9	40.3	41.9	41.3
Avg (ms)	**1**	**34.56**	**38.88**	**40.22**	**40.96**	**41.32**	**1.2**	**34.68**	**38.96**	**40.4**	**41.12**	**41.36**	**2.2**	**34.88**	**38.76**	**40.34**	**41.12**	**41.44**
	Netstat						**Current Users**						**Processes**					
Requests	**1**	**5**	**10**	**15**	**20**	**25**	**1**	**5**	**10**	**15**	**20**	**25**	**1**	**5**	**10**	**15**	**20**	**25**
	4	5	4.5	3.7	3.8	4.1	6	38.5	42.8	44.3	45	45.4	10	5.8	5.6	5.3	5.8	5.2
	5	4.2	3.3	3.1	3.7	4.2	5	38.8	42.7	44.4	45	45.4	9	5.6	5.8	5.3	6.2	5.7
	4	4.4	3.1	3.7	4.2	4	5	38.6	43	44.3	45	45.4	7	5.6	5.6	5.6	6.1	5.5
	6	4.2	3.3	3.4	4.3	3.7	6	38.4	43	44.2	44.9	45.4	6	6	5.1	5.7	6	5.1
	5	4.2	3.1	4.2	4.1	4	5	39	42.8	44.1	44.9	45.4	7	5.4	5	5.3	5.1	5.1
Avg (ms)	**4.8**	**4.4**	**3.46**	**3.62**	**4.02**	**4**	**5.4**	**38.66**	**42.86**	**44.26**	**44.96**	**45.4**	**7.8**	**5.68**	**5.42**	**5.44**	**5.84**	**5.32**

Graphs of turnaround times for each command:





The results are consistent with the first project and show that the concurrent server program is highly efficient because instances of 100 client requests only demonstrate a minor increase in average turnaround time compared to 25 and fewer requests. Similarly, I noticed that the behavior of "Netstat" and "Processes" did not display the same data trends as the others, and this is likely because these system calls operate differently and are more susceptible to volatility due to a number of factors outside of the server-client programming, such as number of users on the server, processes they are running, as well as server background activities. Those reasons compounded with the fact these tests were run during the middle of a school day on a university server is likely what caused the variations in data.

DATA ANALYSIS

- Increasing the number of clients does not affect the turnaround time for individual clients.
- Increasing the number of clients gradually increases the average turnaround time.
- The average turnaround times are faster in the concurrent server than the iterative server.
- You may want to use an iterative server when you are expecting to communicate with a single client or a smaller group of clients with a limited number of transmissions.
- You may want to use a concurrent server when you are expecting multiple clients to connect simultaneously and make multiple coinciding transmissions.

CONCLUSION

TCP networks require a communications protocol that exists on the transport layer that can be considered as computational overhead. For a client sending communications back and forth from a host server, there are a series of "handshakes" that need to take place before and after each connection that causes some latency, or a short delay from end to end. This series of projects observes this concept, as comparing the turnaround times from a single request to

average turnaround times from numerous multiple requests demonstrates the extra overhead incurred form the handshaking protocol between the server and client sockets. Another important concept is that multi-threading processes is an effective solution for servers to meet the demands of providing services for large amounts of clientele and a large number of instantaneous requests.

LESSONS LEARNED

The biggest takeaways are that the Java library provides a simple yet effective framework for basic networking using its ServerSocket class and related classes, that server traffic can have a noticeable impact on latency based on my test results being affected by other users' activity on the same server, and that multi-threaded programming is an elegant solution for handling networks of distributed systems, as creating multiple new instances to handle individual incoming requests greatly increases throughput compared to handling each request iteratively. In the process of writing and testing this program, I learned that it is important to understand the fundamental concepts behind how TCP/IP networks operate within each layer to write more effective networking software and that Java is definitely a practical choice for choosing a programming language for writing programs under the server and client paradigm.