Concurrent Socket Server

Adam Abdulhalim & Vincent Bistricky

CNT4504 – Computer Networks

Professor Scott Kelly

**Introduction:**

The purpose of this Concurrent Socket Server project was to give us a means of learning how servers and clients interact with one another and work together in a network. Specifically, how a multithreaded client interacts with a multithreaded server (hence the term, concurrent). This project essentially served as our secondary introduction to network programming, utilizing the java library of java.net. We were able to learn about the many different objects and classes that go into network programming at a base level, like sockets, threads, runnables, writers, and readers. Furthermore, this also served as an introduction to multithreaded servers for us, considering the previous project only utilized an iterative server. This allowed us to dive deeper into the utilization of threads, not only on the client side, but on the server side as well.

The goals going into this project were for us to create both a multithreaded client class and a multithreaded server class, and to have them communicate with one another to transmit and output results from a command menu. To accomplish this, we also had to make a customized thread class that inherits thread, so that we could override the initial run method and have it perform what is necessary to communicate with our server correctly. We also had to repeat this project on the server side, creating a server thread object that inherits thread and is able to perform the proper operations that have been requested on the server side. The overarching final goal of the entire project is to analyze the turn-around time for a concurrent socket server utilizing the six different commands we build in. We then compare the turnaround times of this concurrent socket server to those of our iterative socket server, seeing which has better turnaround time in which scenarios.

In the remaining sections of this report, you will find details regarding how we designed and implemented the concurrent socket server, how we conducted tests to analyze the server’s turn-around time data, and the analysis and conclusions taken from said data. We have prepared many visual aids showcasing the fascinating patterns in the turn-around time data of the server, so look forward to that. Next, we will cover how exactly we configured the concurrent socket server.

**Client-Server Setup and Configuration:**

In terms of design, we did not do anything terribly out of the ordinary for a socket server program. First of all, Java was our language of choice. Considering the plethora of useful classes in java.net, it was a no-brainer. For the client class, when running the file, we collect the IP address of the server we would like to connect to as well as the port that said server will be listening on. Upon starting, we prompt the user with a menu of commands to choose from: date, uptime, memory use, netstat, current users, and running processes, all of which being Linux command line commands (there is also a seventh option to exit the program). Once a command is chosen using a number between 1 and 6, we prompt the user to input how many iterations (threads) of such commands they would like to shoot at the server. For our testing, we sent up to 100 client threads to the server at once.

Speaking of threads, as stated earlier, we created a customized thread class called “RunnableThread” which contains an overridden run method that allows us to give and take with the server properly upon running a new instance of the chosen Linux command. After getting the user input, an array of RunnableThread objects, with the size being the number of instances specified by the user, is created. Two for loops are then ran, the first one starting each thread and thus sending a request to the server, and a second calling the .join method for each thread, which makes it so that each thread waits until the previous thread is completed in order to collect output from the server. You can think of this as a sort of baton pass in a relay race.

When creating our RunnableThread objects we decided to pass them the command choice of the user, the IP address, and the port. These objects also have a separate variable named “turn” which is where we store the turn-around time of each thread. In our run method, we create a new socket using the IP and port we passed, then create an OutputStream and PrintWriter objects so we can write our Linux command choice number to the server. Once the command is processed within the server, we read the output using an InputStream and BufferedReader objects, reading line by line until there is no output left. The socket then closes, and the thread run method is complete.

Almost everything mentioned so far was done the same way as we did in the iterative socket server. Not much was needing of change. However, we did decide to implement a sort of output buffer on the client side since our threads will now be returning separately from each other, given the fact that they are being processed concurrently. To do this, we create a String array with its size corresponding to the amount of client requests being made at any specific point in time. This array is then passed to each client thread, alongside their own “thread #”, which essentially behaves like an index for that specific thread to place their output. This makes it so that all of the client-side output is displayed in the same order as the threads were created, nice and neatly.

On the server side, we specify a port when running the file, and a server socket is created on said port. We then accept any connection coming on said port and start up a new server thread instance. Within the run method of this thread, we take in the input of the command line choice utilizing an InputStream and BufferedReader objects. We run this input through a possible 6 if else statements, each corresponding to the correlated Linux command line in our presented menu. We set a string named “command” to the corresponding menu choice. For example, if input equals 1, we set command to “date”, if input equals 2, we set command to “uptime”, etc. After that, we run a Process object of “Runtime.getRuntime().exec(command)” which essentially runs the specified command that was requested on the server. We then read the output from this command line by line using another BufferedReader, and writing each line to the client thread using a PrintWriter object. Once there is nothing left to write out, we close the socket and end the thread. The process of starting, processing, and ending server threads happens constantly and concurrently until there are no more client request threads to process. We have a while(true) loop in our server class that waits until a socket connection has been accepted and begins a new server thread every time there is a connection.

Throughout this whole process, we are also timing each thread to see what the turn-around time is for each. We get a start time using “System.currentTimeMillis()” right before opening the socket in our run method, then we get an end time at the very end of the run method using the same system method. We subtract the start time from end time and store this value in the “turn” variable of the thread that we mentioned before. In our client class, when each thread is closed in the .join for loop, we add the last ran thread’s turn-around time to a variable named “totalTurn” which stores the total turn-around time of the entire client request, including every thread in that instance. We output the total turn-around time and the average turn-around time per thread (dividing totalTurn by the number of clients specified by user). Utilizing the aforementioned String output buffer, we also output each client request’s individual turnaround time right after said request’s output. The client then loops back to the beginning, resets the turn-around time to 0, and requests a command line menu option once again, and the process repeats until a 7 is inputted to close the program.

**Testing and Data Collection:**

The testing of the Concurrent Server was done by incrementing the number of client requests necessary for the server to process by 5, ranging from 1 request to 25 and a final increment from 25 to 100 for each of the specified client requests. For example, the client would make a single request for the current memory use of the server a single time, the total and average Turn-around-Time (server receiving the request, processing the needed information, and sending the information back to the client) would be recorded, and the test would be repeated with an increasing number of requests for the server to processes concurrently (e.g., the server starts processing the initial request and initiates the following request while finishing the one prior). There is also an output buffer that ensures the final output of all client requests are output in order. This adds on additional time to the Turn-around-Time, which may factor into conclusions regarding said data. Diagram

Description automatically generated

Pictured above is a graph displaying the total and average Turn-around-Time for processing the date and time client request. The function used to create the strongest correlated best fit lines were found to be an exponential function. This is very clearly shown in a strong exponential relationship between the total Turn-around-Time and the client requests. An interesting note is a comparison between the lines of best fit between the concurrent and iterative servers. We see much more clearly defined exponential growth in the Turn-around-Time for both the total and average times. Elaboration on this point will be explained in the Data Analysis section.

Diagram

Description automatically generated with medium confidence

Pictured above is the graph relating total and average Turn-around-Time with client requests for the uptime request. Like previously seen, a nearly perfect exponential relationship is present between both the average and total times. A consistent measure that is seen again is a higher correlation to an exponential fit than in the prior server.

Diagram

Description automatically generated

Pictured above is the graph relating total and average Turn-around-Time with client requests for the memory use request. We can see two perfectly correlated exponential fits for the two times, which is in line with the data being very similar in value to the data found with the uptime request.

Diagram

Description automatically generated with medium confidence

Pictured above is the graph relating total and average Turn-around-Time with client requests for the netstat request. Like before, an almost perfect exponential relationship can be seen with both Turn-around-Times. An important note is the substantially larger data values given from this request, as well as the fact that these larger data values seem to not weaken the correlation to an exponential fit.

Chart

Description automatically generated with medium confidence

Pictured above is the graph relating total and average Turn-around-Time with client requests for the current users request. Once again, two perfect exponential fit correlations can be seen, with similarly large data values to both the memory use and uptime requests.

Diagram

Description automatically generated

Finally, pictured above is the graph relating total and average Turn-around-Time with client requests for the running processes request. Unlike the previous server, the running processes request does not produce the largest data values out of every tested request. The theory behind why this is will be proposed in the Data Analysis section. Regardless, in line with prior testing, this request does produce some of the highest data values overall.

**Data Analysis:**

A few interesting conclusions can be drawn from the results seen from testing all six client requests: some more obvious than others. Most simply, increasing the number of clients increases both the total and average Turn-around-Time for every single request. This is clearly visible in both the graphs and the table of raw data pictured below:

Table

Description automatically generated

The most notable piece of information found in testing how the concurrent server handles requests is the strength of the correlations to exponential fits across all 6 requests. In testing the iterative server, some of the requests with smaller data values seemed to have an equally strong correlation to a linear fit. It was concluded that this was because the data being tested did not encompass a large enough sample size to adequately show a stronger exponential fit. The prior tests stopped at only 25 clients connected, while testing of the concurrent server ranged all the way to 100 clients. This allowed for much larger data to be collected and directly supported the theory that an exponential fit matched the growth model, the data size simply did not showcase this properly.

The piece of information to be elaborated upon is the size of the netstat requests data relative to the running processes requests data. While both data sets were similarly large, we saw a much greater difference in the concurrent server, interestingly favoring the netstat requests (as opposed to favoring the running processes request in the iterative server). After careful deliberation, there seems to be only one likely explanation: when the testing was conducted. Seeing as netstat monitors the incoming and outgoing connections to the server, it would logically follow that the more users connected to the server, the longer the Turn-around-Time due to increased traffic.

Testing of the concurrent server was conducted on December 4th, 2022, the day before the due date for this assignment. Testing of the iterative server was conducted on November 6th, 2022, two days after the due date for the prior assignment. Assuming that the number of students working on their assignments before the due date compared to after would be higher, this would explain the larger data values seen with the netstat request, as each student uses the same server IP.

Consistently, however, the running processes and netstat requests still produce the largest data sets out of all six requests across both versions of the server. The reasoning for this likely has not changed, as both requests require the server to process and return a substantially greater amount of information relative to all other requests. This was true for the iterative server and holds true in testing the concurrent server.

Finally, A raw comparison between the data collected between the iterative and concurrent server tests reveals two important notes. The first being that at larger numbers of requests, a concurrent server is very clearly superior in terms of Turn-around-Time. However, the second note is that although the concurrent server does produce a consistently faster Turn-around-Time, the investment of a concurrent server may not be worth the time/money if the server is only handling a small number of client requests at any given time. The iterative server is only outperformed by a handful of milliseconds, so unless the situation requires the minimum Turn-around-Time possible, an iterative server may be preferred for handling a smaller number of requests.

**Conclusion:**

It is evident given the data collected from the concurrent server that all six client requests show a nearly perfect correlation to an exponential fit. This is most likely due to the increased sample size in client requests and strongly supports the conclusion made at the end of testing the iterative server. We no longer see any graphs that have a linear fit stronger than an exponential fit. The leading theory for the sharp increase in the values returned by the netstat request is that the number of students connected to the server at the time of testing greatly impacted the server’s ability to process the incoming and outgoing connections of each request. Generally, the same conclusions can be made regarding an exponential increase in Turn-around-Time given an increase in client requests, as well as an increase in information to be processed and return drastically increasing both the total and average Turn-around-Time.

**Lessons Learned:**

The main lesson we learned during this server project especially was the necessary usage of an output buffer with concurrent socket servers. Without this buffer, each client thread would have been outputting concurrently with all the other client threads being processed, leaving the output completely jumbled and essentially useless. Utilizing this buffer to store output along with some output loops, we are able to iterate through the entire buffer and print out all of our data in the same order in which it was requested. It was quite simple to implement all things considered, but still fascinating and an essential lesson for future implementations considering we have never worked with such an output buffer previously.