Iterative Socket Server

Adam Abdulhalim & Vincent BistrickY

CNT4504 - Computer Networks & Distributed Processing

Professor: J. Scott Kelly

**Introduction:**

The purpose of this Iterative 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 singularly threaded server (hence the term, iterative). This project essentially served as our 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.

The goals going into this project were for us to create both a multithreaded client class and a single threaded 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. The overarching final goal of the entire project is to analyze the turn-around time for an iterative socket server utilizing the six different commands we build in. In the future we can then compare this to a concurrent socket server with the same purpose and see which has better turn-around time in which scenarios.

In the remaining sections of this report, you will find details regarding how we designed and implemented the iterative 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 iterative 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 25 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.

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 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 go back to listening for new threads, or we accept cached threads.

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). 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 Iterative Server was done by incrementing the number of client requests necessary for the server to process by 5, ranging from 1 request to 25 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 sequentially.

Chart

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. This relationship is less prominent in the average time; however, an exponential relationship was still shown to be the best fit. An important note is how much less the average time increased relative to the total time. The reasoning for this (and likely all future correlations) will be explained and elaborated upon in the Data Analysis section.

A picture containing diagram

Description automatically generated

Pictured above is the graph relating total and average Turn-around-Time with client requests for the uptime request. Like previously seen (except much more prominently here), a strong exponential relationship is present in both the total and average times. This is likely due to an increasing size in values (i.e., larger gaps between values), which will be reinforced in the coming graphs.

Chart

Description automatically generated

Pictured above is the graph relating total and average Turn-around-Time with client requests for the memory use request. Here we can see a very similar relationship to the uptime request graph. This makes sense, as the values for both the total and average times were almost identical for the two requests.

Diagram

Description automatically generated

Pictured above is the graph relating total and average Turn-around-Time with client requests for the netstat request. Like before, a strong exponential relationship can be seen with both Turn-around-Times. Interestingly, however, unlike prior results, the average time shows almost an equally correlated linear fit. This will also be covered in the Data Analysis section.

A picture containing diagram

Description automatically generated

Pictured above is the graph relating total and average Turn-around-Time with client requests for the current users request. The trend of strong, exponential relationships continues with the times for this request. These data values and correlations match closely to the results found with the data and time graph.

A picture containing graphical user interface

Description automatically generated

Finally, pictured above is the graph relating total and average Turn-around-Time with client requests for the running processes request. This brings us back to the highest values recorded across all 6 requests and once again a strong, exponential relationship between both times. However, once again, the best fit line for the average time is also strongly correlated with a linear relationship.

**Data Analysis:**

A few interesting conclusions can be drawn from the results seen from testing all 6 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

What is truly interesting is how exactly these values are increasing. It is clear that consistently across all 6 requests, the strongest relationship between the total Turn-around-Time and the number of client requests is exponential. No other fit has a clearer correlation, and this only becomes more evident the larger the values. This is seen best in the netstat and running processes requests, as they output the most amount of data (in this case, text) to the client. This also seems to be the primary cause of this exponential relationship.

The average time, however, seems to be a different case. The smaller the data size and values, the less linear the relationship between average time and client requests appears to be. As the values increase, a more clear, linear relationship is present between the two, giving almost an equally strong correlation to an exponential fit, as seen in the running processes and netstat graphs. This indicates a couple of different possibilities. The first being, the more data the server needs to process and send back to the client, the less the number of requests matter regarding average time. The second possibility is that the average Turn-around-Time is always linear; the relationship is simply more clear when given larger values. Testing either a larger amount of requests, or the same amount of requests tested with a similar interval may reveal which of these two possibilities is correct.

**Conclusion:**

We personally believe that a linear relationship is simply more evident with larger values. It would be illogical for 6 graphs that have an extremely similarly correlated exponential relationship to have two average time outliers due to an increase in data. Like previously stated, a more thorough testing of the number of client requests may prove this to be true or false, only the data can tell. What is plain to see, however, is that increasing the number of clients in every situation increases the time it takes for the server to process and return the request information. What is also evident is that the more information needed to be processed and returned drastically increases the total and average Turn-around-Time when more client requests are received.

**Lessons Learned:**

The main lesson we learned while coding the client side was that every thread needs its own socket connection with the server. Initially we had only been connecting once per client request, no matter the thread count, and requesting output depending on the number of threads, all on the same single connection. While this still worked and got us correct output, it was going against the entire purpose of the project, which was to be able to request multiple connections with a single-threaded server at once and see how the number of client requests affected the turn-around time. This was a major problem we had to overcome considering we had to reformat a lot of the code to get this to work the way it was intended.

We also learned about a lot of different java objects that we had not worked with previously, such as PrintWriter, Socket, Thread, and Process. The utilization of process to run Linux commands was especially fascinating, as well as creating our own thread object utilizing Thread as a parent class.