Concurrent Socket Server

CNT4504 - Computer Networks and Distributed Processing - Professor Kelly

Donnie White

Jordan Gallivan

# Introduction

The purpose of this project is to create a software suite that allows for a client application to request and process the response of server commands via a concurrent socket connection. The client application can send several different requests to the server, which in turn returns all necessary data from the client request. These requests can be made numerous times, and to accommodate these requests, multiple threads are created by the client. The server processes each request in a separate thread, returning the results of the request. The goals of these two programs are for them to work seamlessly together, for the server to return accurate data to the client, and to implement a functional user interface on the client-side, all while writing a clean, efficient source code. Additionally, this project will demonstrate the efficiencies and deficiencies of a multi-threaded server when compared to a single threaded server.

This paper will detail the design and design decisions of both the client and the server programs, how both programs were tested, data analysis of the effectiveness of the server in relation to the number of clients connected, as well as the lessons that learned throughout this process.

# Client and Server Design considerations

Both the client and server-side software utilize a primary class through which the user interacts, and a helper class containing methods to process and/or execute requests. The utilization of helper classes aids in the reusability of code throughout, as well as abstracting key components from the user to reduce potential security issues.

The design of the software followed a functional programming paradigm to increase reusability of the code throughout the software. Descriptive function names were used to the max extent to cut down on documentation requirements and increase readability of the code.

In both the client and server software packages, exception handling occurred at the point of the error to prevent propagating of errors and halting of the software. Boolean return values were used throughout to indicate failure conditions, which could then be logged to the user.

# Server-Side Software

The server portion of the project consists of three classes including the main server class, a ServerHelper class, and a ServerThread class. The server class handles creating the socket on the defined port and creating a new thread for each client connection. The ServerThread class handles reading in as well as writing out to and from the client as well as handling each request with the help of the ServerHelper class.

## Design Decisions

The Server class mainly handles creating the socket as well as creating a new thread for each new client request using the defined port. Creating this new thread is crucial to the success of the concurrent implementation because it allows for the server to handle multiple requests almost simultaneously which significantly increases the performance of the server. Since the main server class is now creating a thread for each client connection, the Server class can no longer handle client requests, but instead the requests are now handled within the thread class. The ServerThread class handles all input and output to and from the client as well as utilizes a switch statement to determine which command the client entered. Finally, the ServerThread class also calls on the ServerHelper class to execute the Linux commands using the executeSystemCommand method. This method utilizes the Runtime.getRuntime.exec() command to execute the requested information.

## Processing and Executing Requests

Upon receipt of a client request, the Sever Class spawns a new thread to process the request via ServerThread.

The ServerThread reads in input from the client via a buffered reader. It then processes which case the client input matches based on the known strings being sent that are defined in the client software. These inputs consist of:

* Date and Time: returns the result of executeSystemCommand from the Linux command “date”
* Uptime: returns the result of executeSystemCommand from the Linux command “uptime”
* Memory Use: returns result of executeSystemCommand from the Linux command “free”
* Netstat: returns the result of executeSystemCommand from the Linux command “netstat”
* Current Users: returns the result of executeSystemCommand from the Linux command “who”
* Running Processes: returns the result of executeSystemCommand from the Linux command “ps”
* Default: As a redundancy, we set the default to send a message “NO MATCHES FOUND FOR CLIENT INPUT” back to the client as well as logging it on the server terminal. While the client should catch all unexpected inputs, we opted to implement this redundancy to further prevent unexpected inputs into the server.

The returned values from these commands are then sent to the client via a print writer. If the case if not recognized, it returns the default value, or if the client ends the program, the writer is flushed. After the necessary information has been transmitted, the server the sends an “END” message to the client to signal that the transmission is finished, and the client then closes the socket.

## Lessons Learned

The main lesson learned was that running Java commands over Linux command could significantly impact the time taken to execute the commands. Originally, the java.util.date command was utilized over the Linux date command. After switching to the Linux version, the amount of time taken was significantly reduced. This also held true for the total amount of free memory command since the server originally used java commands to calculate the free by subtracting the used memory from the total memory.

Another lesson learned was that the request handling no longer needing a while loop since each client is being dealt with individually. At first, a while loop was implemented, but was causing errors since the client input was null at the second iteration of the loop. After taking this out, the errors resolved.

# Client-Side Software

The client-side software was implemented utilizing two Java classes. The first class, Client, handles the user interaction to and include: reading in the hostname and port, querying the request and number of requests, validating request parameters, and calling helper functions to execute the requests. The second class, ClientHelper, serves as the communication channel with the server and contains all helper functions, including the threads, to execute a user request.

## Design Decisions

The client-side software provides a scalable package whereby the request options and number of requests are hard coded in the constructor of the ClientHelper in a HashMap and HashSet. The HashMap and Set can be queried in constant time to determine if a request is valid. Additionally, when a request is modified, it only needs to be updated in two places: the HashMap containing the requests and the displayOperations() method.

Further iterations of this software will map the requests directly to their associated Linux commands, reducing the comparative tasking on the server.

## Processing and Executing Requests

When a valid request is received by Client, the sendMessages() method of the ClientHelper is invoked. Within this function the current time is noted, and threads are created for each of the requests made by the client. At the conclusion of these threads’ execution, the current time is compared with the start time and a total run time is displayed to the user as well as the average time per thread.

Each thread created by the sendMessages() method initializes a socket with the server as well as all writers/readers necessary to transmit/receive data from the server. The client request is sent, and the response is buffered in a StringBuilder until the end of transmission keyword, “END” is received. At which point, the results of the request are displayed to the user and the time of execution of that thread is displayed. The thread then flushes the reader and closes it. Closing a buffered reader closes the object it wraps, in this case, the socket.

## Lessons Learned

The biggest lesson learned from the client side was how to implement the socket. The initial iterations of the software employed dependency injection to conserve resources, and a single socket was created and passed to the methods that required communication with the server. This resulted in significant issues and multiple threads were backlogged and not displayed on the client-side. The fix for this was simple and was in keeping with the intent of this project: each thread needs to create its own socket and the server will cache each socket request.

Additionally, there was a tremendous amount of learning to be done on asynchronous programming in Java, specifically with respect to thread pools. Because a total execution time was required, and the software needed to wait until all threads terminated before receiving another request from the client, the threads needed to be pooled and the main thread needed to be halted until all children had terminated. This can easily be done with an ExecutorService. The issue was one of semantics in how the awaitTermination() method of an ExecutorService is correctly implemented. Once this was solved, the main thread halted correctly until all children terminated.

# Testing

Both the client and server-side software packages were tested thoroughly utilizing the server itself. As discussed in the lessons learned for each section above, significant amounts of debugging and refinement were required to get the threads correctly created and to await their termination on the client side.

The first iteration of the project opened a single socket, and all requests were made on that one socket. This was rife with issues and through extensive testing, it was determined that a new socket was to be created for each request which resulted in satisfactory performance.

Final testing was conducted by verifying all requests could be made from the client side, including incorrect or invalid requests. The results of those requests were observed and validated for correctness.

# Data Collection

The following data was collected by running each request/number of requests combination 20 times and averaging them. The initial observation is the first request to the server, regardless of the request took significantly longer than the following requests. To correct for this, the requests were run numerous times to reduce the effect of the initial request time.

# Data Analysis

As suggested by the charts above, the netstat command took the longest average time compared with the other requests with the current programs taking the second longest average. However, both commands have a sudden and dramatic decrease in time from 25 requests to 100 requests, suggesting that as the number of requests grows, a decrease can be expected in time to a certain degree. However, the other requests still have a small increase in time as the number of requests grows, but these are relatively small in comparison to the previously mentioned requests. This supports the conclusion that the larger requests can be expected to eventually take a smaller amount of time when the number of clients are over a certain threshold, but the smaller less intensive requests can be expected to continue to slowly increase.

Looking at individual requests, the concurrent server is overall equal to or less than the iterative server results. This suggests that the concurrent server is overall better than the iterative server, and the difference increases as the number of clients increases. In situations where client requests are few and or spaced out sufficiently, there is less overhead with the iterative socket. In some cases, the iterative socket server is faster for small numbers of requests. However, as the number of requests exceeds 25, the concurrent socket is far more efficient, justifying the overhead.

# Conclusion

Overall, this project was an excellent opportunity to expose the students to the different types of client-server communications, the software that is associated with it, and the difference the software can have on overall performance when compared to each other. Many students have only seen these differences on a theoretical level, but this project allows for those students to be able to see how their design decisions can affect the performance of their software.

This project not only demonstrates how taxing different requests can have on a server, but how software engineers can take certain actions to not only mitigate the difference but improve on the previous versions. While this project may have been small scale, the lessons learned can be translated to even larger scale servers in the field.

When deciding the software suite to deploy on a server the biggest factor is the number of client requests expected and the associated frequency of those requests. When the number of client requests is low, or at a low frequency, an iterative socket software suite requires less memory out performs the concurrent socket server. As the frequency of requests increases past 1 request/ms, the concurrent socket server matches the efficiency of the iterative socket server. Once the frequency exceeds 2 requests/ms, the concurrent socket is the software suite to choose to avoid delays in response.