Iterative 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 socket connection. The client application is able to 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 one request at a time, 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.

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-side software consists of two Java classes, Server and ServerHelper. Server is the primary interface through which the user enters the port the communication with the client will occur, as well as processing all user requests. ServerHelper contains helper functions to aid in the execution of user requests.

## Design Decisions

The receiving and execution of user requests is the primary purpose of the Server class. This currently involves a switch to determine which server command is to be executed. In future iterations of this software, the client should be able to send a string containing the Linux command, significantly reducing the comparative burden on the server software.

The Server relies on two primary helper functions contained in ServerHelper, which is instantiated with the socket created by Server during each iteration of the while loop:

* executeSystemCommand(): This method executes the given command via the system command interface, Runtime.getRuntime().exec(command). The results of this command are buffered in a StringBuilder and returned when the command has completed execution.
* sendMessage(): This method passes the provided message to the client via a buffered writer.

## Processing and Executing Requests

The main server class takes input from the user to define what port the server should be running on and then waits for a client to connect using that port. It then creates a server helper object, buffered reader, and buffered writer. The server class uses the buffered reader to read the input from the client and then checks that input in a switch case for each expected input:

* Date and Time: returns the current date from a call to java.util.date
* Uptime: returns the result of executeSystemCommand from the Linux command “uptime”
* Memory Use: returns the difference Runtime.getRuntime().totalMemory() – Runtime.getRuntime.freeMemory()
* 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 value from the above switch is sent to the client via the sendMessage() method and the results are logged on the sever terminal. In the case that the client request could not be executed or was an invalid request, an error is logged on the server terminal and sent to the client.

This process concludes with the end of transmission message, “END” being sent to the user. The client is responsible for closing the socket at which point, the server will continue listening to accept the next request.

## Lessons Learned

One lesson learned was how to execute Linux commands through the program. The initial approach was to call socket commands to get the information needed. This process was corrected through familiarization with the Linux commands and exposure to the runtime command process within Java.

A slight, but trivial issue was encountered based on the version of Java being run locally and that which is available on the server. Once the project was reverted to the older version of Java and the newer features removed, the issue was 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.

# Date Analysis

As evidenced by the above charts, the netstat command took the longest average compared to all other requests with the CurrPrograms request taking the second longest average. The rest of the requests can be seen grouped closely at the bottom of the graph. This supports the conclusion that the primary cause of individual turnaround time is the amount of data being queried and returned. Both requests return an immense amount of data per request, and therefore the associated time is significant.

When looking at individual requests, it is noteworthy that time average for each request decreases in time from 1 request to 5 requests. However, as the number of requests increases from 5, it increases at an exponential rate leading to the conclusion that increasing the number of clients will greatly increase the turnaround time not only for individuals, but also will greatly increase the average turnaround time average.

# Conclusion

This project was an excellent exposure event to client-server communications and the software associated with said communications. At this point in their academic careers, few students have written asynchronous software and even fewer have implemented iterative sockets. This project gradually brings students up to speed on both and will serve as a solid foundation for continued work in this field.

Concurrent with the programmatic learning, this project aptly demonstrates the impact numerous taxing requests can have on a server. As the data above shows, the increase in demand correlated directly with an increase in overall processing time on the server-side. This is relatively low impact for this project, but the results can be extrapolated to thousands of users, allowing students to better understand the nuances of high demand requests on a server.