Concurrent Socket Server

CNT4504 – Computer Networks & Distributed Processing

David Rojas Gonzalez; Fernando Jimenez Mendez

John Kelly

# Introduction

The purpose of this project is to learn how a client and a server communicate with each other to trade information for various programs. Ultimately, our goal is to create a connection between a client socket and a server socket that is functional (and efficient) enough to exchange information with each other. In the later sections, we will talk about how the client-server programs were set up and configured to complete several tasks as requested by the user. After that, we will discuss how we tested our programs and how the data was collected. Finally, we will go over our data analysis, conclude based on the results that we’ve gathered from our tests, and see what lessons we have learned as we were completing this project.

# Client-Server Setup and Configuration

## Client Program

The client program is composed of four classes, which are: “Client.java” (driver), “cThread.java”, “RequestType.java”, and a “CSVExporter.java” utility. The driver class contains our main method with a static scanning object that is used to request information from the user. The program requests the user to enter the port, the host, the operation of the program and the number of clients to generate. The program first verifies the operation entered by the user using a do-while loop to execute the menu options at least once before asking for the number of clients to generate. Once we have the four fields required to create the client socket, we then use four loops to create the threads, execute the threads, join the threads and calculate the average server response time.

The “cThread” class is used to create a thread for each client in the program. In this class, we have seven fields:

* the client socket,
* the print writer,
* the buffered reader,
* a string variable (used to read responses from the server),
* a long data type to store time consumption on each thread,
* the request type, and
* a static array list used to store the time consumption of all threads.

Our constructor for cThread sets up the client socket, and the print writer and the buffered reader (while handling any error exceptions for the host and the I/O stream) so that it is available for the entire class. Attempting to do this in a specific method would fail since we would need a bigger scope for the buffered reader.  After that, we’ve decided to override the “run” method from the thread class to create the thread whilst sending the server the request type. After this, the run method will read the response from the server, display it, and calculate the total time spent for each thread. Then, we have an enum called “RequestType.java” that stores the command entered by the user, which will then be retrieved by the server program to process the request. The enum will be kept in sync with the client. Finally, we pass the recorded time data to the “CSVExporter.java” utility so we can export it neatly into a readable CSV file.

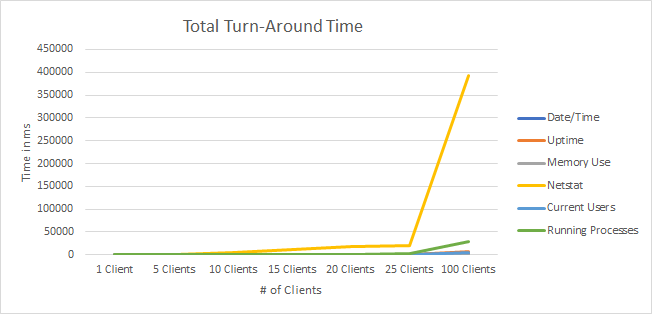
## Server Program

Even though there are many similarities between the iterative socket server and the concurrent socket server, we had to make some changes to make a multithreaded server program. This time, the server program is composed of ten files: “Server.java” (driver), “RequestType.java”, a “RequestHandler.java” interface, a “HandlerThread.java”, and 6 implementations of said interface, one for each RequestType. The driver class will “try and catch” any I/O stream errors while declaring and initializing the server socket object before processing the client requests. After that, it will enter an infinite loop, unless the request type is to quit the program. Similar to the client socket, the server socket is created and the user will have the option to quit the program by pressing CTRL+C. In a try-catch statement, the server will then enter an infinite loop to accept multiple requests from the client, create and start multiple threads (one for each request), establish a connection and store the client’s information by using a buffered reader in the ThreadHandler class. In this program, we have moved the HashMap to the ThreadHandler to make the program simpler. Provided that there are no errors with the file handling, the I/O stream, and the command requested, the driver will call and use the ThreadHandler class with its start() method (extended from the Java built-in thread class) after implementing the multiple handlers to process the commands sent from the client before exiting.  The DateTimeHandler class will handle the date formatting for the server. Lastly, the request type class is the exact same class used in the client program, which is used to collect the command type that was received.

# Testing and Data Collection

The concurrent server was designed and tested on a Windows platform before testing it on the Linux server to find any errors and bugs. After we confirmed that the code ran successfully without encountering any compilation errors, we decided to ­­­­test our program on the Linux server and collect our data the same way we tested the iterative socket server program.

## Total Turn-Around Time

****

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **1 Client** | **5 clients** | **10 clients** | **15 clients** | **20 clients** | **25 clients** | **100 clients** |
| **Date/Time** | 89 | 52 | 150 | 517 | 417 | 164 | 2865 |
| **Uptime** | 34 | 33 | 115 | 212 | 417 | 511 | 6312 |
| **Memory Use** | 24 | 11 | 46 | 71 | 125 | 221 | 1616 |
| **Netstat** | 118 | 1687 | 5239 | 11534 | 17635 | 19402 | 392649 |
| **Current Users** | 2 | 20 | 72 | 136 | 184 | 281 | 5000 |
| **Running Processes** | 17 | 155 | 341 | 896 | 1443 | 2089 | 30000 |

Average Turn-Around Time

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **1 Client** | **5 clients** | **10 clients** | **15 clients** | **20 clients** | **25 clients** | **100 clients** |
| **Date/Time** | 89 | 10.4 | 15 | 34.4667 | 2.85 | 6.56 | 28.65 |
| **Uptime** | 34 | 6.6 | 11.5 | 14.1333 | 20.85 | 20.44 | 63.12 |
| **Memory Use** | 24 | 2.2 | 4.6 | 4.7333 | 6.25 | 8.84 | 16.16 |
| **Netstat** | 118 | 337.4 | 523.9 | 768.933 | 881.75 | 776.08 | 3926.49 |
| **Current Users** | 2 | 4 | 7.2 | 9.0667 | 9.2 | 11.24 | 50 |
| **Running Processes** | 17 | 31 | 34.1 | 59.7333 | 72.15 | 83.56 | 300 |

Similar to the iterative socket server, the listed total turn-around time shows us that it varies for some commands no matter how many clients we request. For example, the “Date/Time” command will still vary in numbers even if there are more clients involved with the request. However, the concurrent socket server has shown larger numbers when tested than those gathered from the iterative server test. Some requests also took a longer time as the number of clients increased. Such requests were: Uptime, netstat (increased tremendously), current users and running processes. However, we can’t say the same on the average turn-around times. The only command processes that have a positive correlation between the average time and the number of clients are: Netstat and running processes. The other commands seem to have varied average turn-around times. Despite the larger data gathered, we still see the same behavior in both versions of the program (iterative and concurrent).

# Lessons Learned

            Throughout the development of this project, we have encountered some problems ranging from thinking of strategic ways of creating the sockets, the number of classes (which was just extended from the iterative socket server), methods to handle every possible user error, and some bugs that needed to be fixed. By extending this program from our previous iterative socket server, we avoided resolving the formatting issues of a string which allowed us to correctly print the server’s response effortlessly. We also needed to create the ThreadHandler class (along with the print writer and the buffered reader in it) to handle multiple client requests. Other than that, many issues were avoided since they were resolved in our previous socket server program.