Concurrent Socket Server

CNT4504 - Computer Networks & Distributed Processing

Professor Scott Kelly

Matthew Campbell

Matthew Graham

Brett Stennett

The purpose of this project is for the team to learn about concurrent servers and compare them to the iterative server from the previous assignment. In order to learn about the concurrent servers, the team has implemented two programs, a client and a server. The goal of the project is to create a successful implementation of the project given the client’s direction. Each program has different functions it needs to perform. The server will send the correct output to the client based on the commands that are sent as input. There are six operations that the server can do. Knowing this, the server can list the current date and time from the server, server uptime since last started, current memory usage, current connections to the network, a list of users currently logged on, and a list of applications that are running on the server. The server will also do maintenance and delete any unneeded instances. As for the client program, it will send requests about the command to enter, the IP address and the port to connect to, and the amount of clients for the server to handle. Also, it will get the turn-around time for every client request, the average turn-around time for all client requests, and the total turn-around time for all client requests.

**Client-Server Setup and Configuration**

The project included multithreaded client and server programs. Each program was written in Java, using the Socket class for the client program and ServerSocket class for the server program for network endpoint communications. Each program prompts the user for necessary input at runtime.

**Client Program**

With one difference regarding time tracking, the client program in this project is the same as the client program from the Iterative Socket Server project. Upon execution, the client prompts the user for the hostname and port number of the server. Next, the client prompts the user for a command to send to the server, followed by a prompt for the number of identical requests to send. Sending simultaneous requests to the server allows for the measurement of average server response time, which is expected to change as the number of requests changes. For each request, the client program creates a new thread. This architecture supports multiple simultaneous requests that are not hindered by the potential delay of previous requests.

To implement multithreading, the Thread class was extended into a ClientThread class. The ClientThread class creates a new Socket endpoint to attempt connection with the server, then writes the user-input command to the Socket’s output stream and receives the server output from the Socket’s input stream. The last bit of functionality of the ClientThread class is request time tracking. When a Client object is instantiated, its constructor sets its start time field using the nanoTime() function. Then, after the server response is received, the Client object reports the requested output immediately and records the end time using nanoTime(). The nanoTime() function was chosen over comparable methods like currentTimeMillis() due to currentTimeMillis()’s reliance on system time, which can change during program execution. When the client program’s main thread is ready to report the various threads’ runtimes, it calls each Client thread’s getRuntime() function after ensuring that the thread has completed its assigned task. These runtimes can then be averaged to determine the average runtime per client.

Because the server program is multithreaded, the individual times from each thread cannot be summed to calculate the total runtime of the program. For example, if we have 5 ClientThreads that each take 10 ms to complete and they start in 1 ms increments: thread 1 would start at time 0 and run until 10 ms, thread 2 at time 1 until 11 ms, thread 3 at time 2 until 12 ms, thread 4 at time 3 until 13 ms, then thread 5 at time 4 until 14 ms. Thus, the total runtime would be 14ms. However, if the implementation sums each ClientThread’s individual runtime, the program would report a total runtime of 50 ms (10 + 10 + 10 + 10 + 10). To combat this, directly before the group of ClientThreads is instantiated, the program records the start time. Then, immediately following the closure of the final ClientThread, the program records the end time and displays the total elapsed time of all requests.

**Server Program**

Upon execution, the server prompts the user for the port number and tries to create a new ServerSocket instance using the given port number. If the chosen port number is already in use, the program will throw an IOException and terminate. A more robust program could prompt the user for an alternate port number or use another method to choose a different port number, but this was not necessary for this application. Once the ServerSocket is successfully instantiated, the program uses while(true) to purposely loop infinitely. During each loop, the program listens for incoming requests. When a request is received, it is accepted and sent to a new ServerThread instance. When the ServerThread runs, the command is processed from the input stream, where it is then handled by a switch statement.

The switch statement accepts the following commands: date, uptime, netstat, users, memory, and ps. These commands correspond to Linux commands that can be run in a separate process using Java’s Runtime’s exec(command) function. This architecture allows the program to retrieve meta-information about its own execution. Table 1 shows the commands the server accepts and the corresponding Runtime.exec() commands. The program reads the output of the commands from its input stream and send the output to the server’s output stream, sending the response to the client. After the response is sent to the client, the socket is closed.

|  |  |  |
| --- | --- | --- |
| Table 1  *Server and Corresponding Runtime.exec() Commands* | | |
| Server command | Runtime.exec() command | Explanation |
| date | date | Returns the current server time. |
| uptime | uptime | Returns the amount of time the server has been running. |
| netstat | netstat | Returns network connections on the server. |
| users | users | Returns a list of users currently connected to the server. |
| memory | free | Returns the current memory usage of the server. |
| ps | ps -a | Returns the list of programs currently running on the server. |

**Testing and Data Collection**

To test the relationship between number of requests and the server’s response time, the client program included functionality to concurrently send a user-specified number of one particular request using multithreading. The client program tracked each request to report the request’s elapsed time, the average response time of each batch of requests, and the total runtime of each batch of requests. The data was gathered from the client and server programs both running on the class Linux servers.

The date command had an average response time of 67 ms with a single request, 6 ms per request with 5 requests, 10 ms per request with 10 requests, 4 ms per request with 15 requests, 12 ms per request with 20 requests, 6 ms per request with 25 requests, and 54 ms with 100 requests. The uptime command had a response time of 2 ms with a single request, 4 ms per request with 5 requests, 6 ms per request with 10 requests, 11 ms per request with 15 requests, 8 ms per request with 20 requests, 11 ms per request with 25 requests, 116 ms per request with 100 requests. The memory command had a response time of 3 ms with a single request, 5 ms per request with 5 requests, 5 ms per request with 10 requests, 7 ms per request with 15 requests, 9 ms per request with 20 requests, 9 ms per request with 25 requests, and 128 ms per request with 100 requests. The netstat command had a response time of 92 ms with a single request, 221 ms per request with 5 requests, 418 ms per request with 10 requests, 600 ms per request with 15 requests, 819 ms per request with 20 requests, 1207 ms per request with 25 requests, and 2454 ms per request with 100 requests. The users command had a response time of 1 ms with a single request, 2 ms per request with 5 requests, 4 ms per request with 10 requests, 6 ms per request with 15 requests, 6 ms per request with 20 requests, 5 ms per request with 25 requests, and 251 ms per request with 100 requests. Last, the ps command had a response time of 7 ms per request with a single request, 10 ms per request with 5 requests, 19 ms per request with 10 requests, 25 ms per request with 15 requests, 28 ms per request with 20 requests, 33 ms per request with 25 requests, and 479 ms per request with 100 requests. See Table 2 and Figure 1.

The date command had an total response time of 78 ms with a single request, 14 ms per request with 5 requests, 26 ms per request with 10 requests, 17 ms per request with 15 requests, 33 ms per request with 20 requests, 23 ms per request with 25 requests, and 1037 ms with 100 requests. The uptime command had a response time of 2 ms with a single request, 6 ms per request with 5 requests, 10 ms per request with 10 requests, 16 ms per request with 15 requests, 17 ms per request with 20 requests, 22 ms per request with 25

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Table 2  *Number of Requests and Average Response Time for Six Server Commands* | | | | | | |
| Requests | date | uptime | memory | netstat | users | ps |
| 1 | 67 | 2 | 3 | 92 | 1 | 7 |
| 5 | 6 | 4 | 5 | 221 | 2 | 10 |
| 10 | 10 | 6 | 5 | 418 | 4 | 19 |
| 15 | 4 | 11 | 7 | 600 | 6 | 25 |
| 20 | 12 | 8 | 9 | 819 | 6 | 28 |
| 25 | 6 | 11 | 9 | 1207 | 5 | 33 |
| 100 | 54 | 116 | 128 | 2454 | 251 | 479 |

requests, 1065 ms per request with 100 requests. The memory command had a response time of 3 ms with a single request, 9 ms per request with 5 requests, 9 ms per request with 10 requests, 12 ms per request with 15 requests, 20 ms per request with 20 requests, 28 ms per request with 25 requests, and 1060 ms per request with 100 requests. The netstat command had a response time of 92 ms with a single request, 232 ms per request with 5 requests, 442 ms per request with 10 requests, 651 ms per request with 15 requests, 914 ms per request with 20 requests, 1287 ms per request with 25 requests, and 3854 ms per request with 100 requests. The users command had a response time of 2 ms with a single request, 4 ms per request with 5 requests, 7 ms per request with 10 requests, 9 ms per request with 15 requests, 25 ms per request with 20 requests, 14 ms per request with 25 requests, and 1044 ms per request with 100 requests. Last, the ps command had a response time of 7 ms per request with a single request, 14 ms per request with 5 requests, 25 ms per request with 10 requests, 41 ms per request with 15 requests, 49 ms per request with 20 requests, 61 ms per request with 25 requests, and 1141 ms per request with 100 requests. See Table 3 and Figure 2.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Table 3  *Number of Requests and Total Response Time for Six Server Commands* | | | | | | |
| Requests | date | uptime | memory | netstat | users | ps |
| 1 | 78 | 2 | 3 | 92 | 2 | 7 |
| 5 | 14 | 6 | 9 | 232 | 4 | 14 |
| 10 | 26 | 10 | 9 | 442 | 7 | 25 |
| 15 | 17 | 16 | 12 | 651 | 9 | 41 |
| 20 | 33 | 17 | 20 | 914 | 25 | 49 |
| 25 | 23 | 22 | 28 | 1287 | 14 | 61 |
| 100 | 1037 | 1065 | 1060 | 3854 | 1044 | 1141 |

Figure 1

Figure 2

Figure 3

**Data Analysis**

Due to the overlapping nature of multithreading, two mean values can be obtained. First, as calculated during data collection, the average response time per request for each batch can be found by dividing the summation of each request’s response time by the total number of requests. This mean informs about the relative efficiency per request in the batch (see Figure 2). Second, once the total response time for the batch is determined, the average server time per request can be found by dividing the total response time for the entire batch by the number of requests in the batch. This mean informs more about overall efficiency of the server per batch of requests (see Figure 3).

Generally, the results were as expected: increasing the number of simultaneous requests results in greater total and average response times. With exception of the netstat command, the concurrent server ran consistently faster than the iterative version across all metrics: total runtime, average runtime per request, and average server time per request. The faster response of the concurrent server is likely due to the fact that each request can be processed independently and does not rely on the previous request to finish. Thus, processing power can be utilized more efficiently, as the next request in the queue can begin processing while the previous request is waiting for the return value of its command.

The netstat command, however, is a little more involved, as our implementation includes reverse DNS lookup for the connections it detects, further delaying response times. To avoid this, the server could include the “-n” flag with the netstat command to disable hostname lookup. Additional analysis of the netstat results shows that the command consistently runs faster per request under the iterative architecture but runs faster overall under the concurrent architecture. This observation is likely caused by the complex nature of the netstat function and is difficult to determine its specific cause without completely unwrapping netstat, which is outside the scope of this report.

**Conclusion**

The Concurrent Socket Server performed its job much more efficiently than the Iterative Socket Server and was very simple to integrate. Because of the significant difference between the netstat command and other command response times, this command should be analyzed to determine if a different command or variation should be used to report network statistics. Another option would be to place this functionality in a separate program, as not to degrade the performance of other commands that may be received concurrently.

**Lessons Learned**

First, we learned that the statistics used in describing concurrent server performance are nuanced, requiring special consideration. For instance, there are two different ways to describe average request time. First, the average response time per request for each batch can be found by dividing the summation of each request’s response time by the total number of requests. This average tells how long each client can expect to wait for their request to return. Second, once the total response time for the batch is determined, the average server time per request can be found by dividing the total response time for the entire batch by the number of requests in the batch. This average tells how much time the server spends per request.