Iterative Socket Server

Dylan Muraco and James Routzahn

CNT4504 - Computer Networks & Distributed Processing

Professor Scott Kelly

**Introduction:**

The Iterative Socket Server project aimed to design and implement both an iterative, single-threaded server and a multi-threaded client, emphasizing collaboration to strengthen our teamwork skills. Developing these skills is essential for preparing us to work effectively in professional settings, where such collaboration is routine. The responsibilities were divided between two students: Dylan Muraco developed the server, while James Routzahn focused on the client. After completing these parts, Dylan continued with testing and data collection, and James began the project report.

This project tasked us with implementing an iterative, single-threaded server and a multi-threaded client to operate within a client-server configuration, allowing us to examine, analyze, and assess how such a server impacts the efficiency of handling client requests, particularly in terms of average turnaround time. Our primary objective was the development of both the server and client, which demanded the majority of our time and effort. Once these components were completed, we proceeded to test the setup, troubleshoot any issues, and gather critical data to support our analysis. With all data collected, our final goal was to compile a detailed report and conduct a thorough review before submission.

Now to move toward the detailed oriented part of the report over several sections. The first portion will be an overview of the configuration and set up of the server-client relationship. Second, an explanation into how data on the configuration was collected with charts to display the data. Third, an analysis of the data followed by conclusions on the information collected. Finally, the report will conclude with what was learned through the process of this assignment.

**Client-Server Setup and Configuration:**

The configuration of the client-server relationship is that of a single threaded server that processes one request at a time, and a multi-threaded client that will make one type of request a specified number of times. For the server, it will wait for a connection from the client and its request. When the connection has been made and the request comes through, the server will run the designated command before sending the command output back to the client and handling the next connection.

On the multi-threaded client, when started, it begins by requesting an IP address as well as the port number of the server. If the connection is successful, the client displays a list of seven options, six of which send a request to the server, and one that ends the session. If one of the first six options are chosen, the user is then asked how many requests they would like to send to the server. The program then creates a number of threads, in direct correspondence to the number of requests they specified. In a parallel manner, each thread starts a timer, then sends a request to the server, with the user-chosen command. Once the server handles the request and sends back the result, the thread stops their timer. This time noted as ‘Turn-around Time’ is then stored and displayed to the user. After each and every thread gets their request fulfilled, the program display the sum of each thread’s ‘turn-around time,’ and the average ‘turn-around time’ between the threads.

**Testing and Data Collection:**

Each of the allowed 6 commands were tested: ‘date and time,’ ‘uptime,’ ‘memory use,’ ‘netstat,’ ‘current users,’ and ‘running processes.’ Each command was tested with a varying number of requests: 1, 5, 10, 15, 20, and 25. After each test, the results were collected into a spreadsheet before being converted into charts and displayed below.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Date and Time** |  |  |  |  |  |  |  |
| number of clients |  | 1 | 5 | 10 | 15 | 20 | 25 |
| Total Turn-around Time |  | 2.00ms | 13.00ms | 10.00ms | 26.00ms | 46.00ms | 29.00ms |
| Average Turn-around Time |  | 2.00ms | 2.60ms | 1.00ms | 1.73ms | 2.30ms | 1.16ms |
| **Uptime** |  |  |  |  |  |  |  |
| number of clients |  | 1 | 5 | 10 | 15 | 20 | 25 |
| Total Turn-around Time |  | 3.00ms | 9.00ms | 20.00ms | 29.00ms | 38.00ms | 45.00ms |
| Average Turn-around Time |  | 3.00ms | 1.80ms | 2.00ms | 1.93ms | 1.90ms | 1.80ms |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Memory Use** |  |  |  |  |  |  |  |
| number of clients |  | 1 | 5 | 10 | 15 | 20 | 25 |
| Total Turn-around Time |  | 4.00ms | 8.00ms | 24.00ms | 22.00ms | 53.00ms | 49.00ms |
| Average Turn-around Time |  | 4.00ms | 1.60ms | 2.40ms | 1.47ms | 2.65ms | 1.96ms |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Netstat** |  |  |  |  |  |  |  |
| number of clients |  | 1 | 5 | 10 | 15 | 20 | 25 |
| Total Turn-around Time |  | 213.00ms | 564.00ms | 761.00ms | 1193.00ms | 1704.00ms | 1957.00ms |
| Average Turn-around Time |  | 213.00ms | 112.80ms | 76.10ms | 79.53ms | 85.20ms | 78.28ms |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Current Users** |  |  |  |  |  |  |  |
| number of clients |  | 1 | 5 | 10 | 15 | 20 | 25 |
| Total Turn-around Time |  | 2.00ms | 7.00ms | 30.00ms | 26.00ms | 30.00ms | 49.00ms |
| Average Turn-around Time |  | 2.00ms | 1.40ms | 3.00ms | 1.73ms | 1.50ms | 1.96ms |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Running Processes** |  |  |  |  |  |  |  |
| number of clients |  | 1 | 5 | 10 | 15 | 20 | 25 |
| Total Turn-around Time |  | 21.00ms | 76.00ms | 186.00ms | 205.00ms | 292.00ms | 334.00ms |
| Average Turn-around Time |  | 21.00ms | 15.20ms | 18.60ms | 13.67ms | 14.60ms | 13.36ms |

**Data Analysis:**

In our test cases, surprisingly, the increase in client requests came with an improved average ‘turn-around time.’ This can be attributed to the server using its processing power more effectively. As the load on the server increased, it triggered a gain in efficiency due to the increased utilization of its processing power. In more strenuous cases where we pushed the server to its limit, we expected to see performance gains taper off; but with this level of testing that did not occur. However, there are obvious differences in the type of command requested from the server as some took much longer to process than others.

**Conclusion:**

In the data analysis section, some of the basic trends were covered, but the basic conclusion for this client-server configuration is that with the increased load on the server, efficiency gains were made in the processing of each subsequent request, leading to average turnaround times decreasing with the number of client sessions. We also conclude that this will only be possible to a certain point before the number of threads start to trigger a trend in the opposite direction. Another clear theme is the difference in processing time between each type of request with date and time followed closely by current users taking the least amount of time and netstat taking by far the longest as it requires the most processing power per call.

**Lessons Learned:**

Throughout the course of working on this project, much was learned. To start with the creation of a client-server configuration was a big learning opportunity. Teaching ourselves how to create the two ends of the relationship was a new experience that help to grow our understanding in the field of computer networks. Creating the configuration was also followed by the debugging and data collection process. The creation of the client was also a big introduction to the process of multi-threading and how to communicate with a server by spawning multiple different client sessions at once. Through the course of this project, learning to divide and conquer a large project in a process that mimicked a professional project was a good thing to gain experience in, this helped to grow communication and teamwork skills.