**Concurrent Socket Server**

**Randy Jackson**

**Travis Jones**

**CNT4504 - Computer Networks & Distributed Processing**

**Prof. Littleton**

In this project, we created a multithreaded socket server program in order to investigate how it might respond differently from a single-threaded server. In what situations would a multithreaded server offer better performance than an iterative server? How are turn-around times affected by the complexity of the type of data being requested? In the first part of this paper, we detail how our client and server programs were designed. After that, we explain how the server program was tested and present the results of those tests in terms of the amount of time per client request, the total amount of time for all client requests, and the average time per client request. Based on those results, we offer suggestions for when a multithreaded server might be appropriate, as opposed to an iterative server. Lastly, we discuss what we learned from this project.

**Program Design**

Our Python client program was similar to what we created for the iterative server project, but it differed in creating multiple processes for each concurrent request instead of just threads. We did this because our multithreaded server was getting similar time results to our iterative server. When we investigated further, we determined that Python uses a Global Interpreter Lock (GIL) that prevents multiple threads from executing simultaneously. By using multiple processes, we were able to get around this restriction.

While threads and processes use the join and start functions in the same way, variables are handled differently. New processes don't have access to global variables from an initial process, and they don't return variables in the same way. To access shared variables, we created a separate manager process using the multiprocessing Manager function. Next, we passed the array reference returned by that function to each new process that was created ("return\_dict"). Processes would append their run times to return\_dict, and then those run times were passed back to the initial, main process in order to calculate total and average run times.

The code for our Python based concurrent server is mostly the same as for our iterative server: After setting up a socket server object from the socket library, we bound an initial port to it and used the listen function to create a queue of incoming connections from clients. We used a simple while loop in order to keep the server running. Each time a new client connection is established, a new process is started with the "threaded" function and a reference to that connection object is passed to it.

We initially wrote our concurrent server as multithreaded but ran into the same problems as our client program. To get around Python's GIL, we again modified it to be multi-process. In this case, we didn't need to deal with any return values from the different server processes, since that is all handled by the client. We didn't use the join function because the main process does not need to wait for the other processes to terminate before continuing in its loop.

To run the server, execute "python3 server.py" from command line. Once the server is listening for connections, execute "python3 client.py" in order to start the client. The client specifies 6 different commands it can ask the server to run; enter the number for the associated command, then enter the number of simultaneous requests to run for that command. Lastly, the client program will ask how many tests to execute. This number determines how many times the program should attempt the specified number of simultaneous requests. Results are output to the command line.

**Data Analysis**

For analysis purposes, we first gathered data for all 6 types of client requests when run simultaneously as 25 different threads from the data\_client (output listed in the Appendix). Once we imported the CSV files for that data into Excel, we used a simple formula to average the time for the first client request completed with the first completed from all 10 tests, then did the same for the 2nd client request, the 3rd, and so on. We created 6 different graphs of these results for each request type:

Our results were fairly consistent across test runs, although there was some variation. Each graph ended up showing a linear relationship between the length of time per request and the order in which each request was completed.

Next, we ran our client program with 1, 5, 10, 15, 20, and 25 simultaneous date requests, and then again with netstat requests (output listed in the Appendix). This produced total and average times that became increasingly slow. We observed a parabolic relationship between the number of requests and the total time:

Total and average time increase even more rapidly depending on the complexity of each request. Here, the netstat command had a longer average and total request time than the date command, as shown in the above graphs.

The increasing overhead for additional simultaneous requests probably comes from the number of unaccepted requests added to the server's queue. In our server program, we used the listen function in the Python module socket, with a maximum queue of 30. Having to queue up more simultaneous requests required additional system resources that slowed down the iterative process of accepting those connections.

From our results, we can conclude that the number simultaneous client requests a server might receive plays an important role in determining the time it takes to respond to each client. For our iterative server program, total time increased in a parabolic manner as the number of simultaneous client requests increased. The time for each individual request to be completed, when the number of simultaneous requests was held constant, increased linearly, however. This lends support to the idea that queuing up more not-yet-accepted connections slows the server response time.

From our experience writing the server and client programs, we did learn some useful information. Perhaps most importantly, we learned about setting up and accepting connections between server and client programs. Creating a separate data\_client program to write results to csv files was very helpful for getting results into Excel.

The part we spent the most time on, however, was figuring out how to make our client program truly multithreaded. Initially, we created threads for each request using the start\_new\_thread from the Python thread library in a loop, but we quickly realized that the interpreter would move on to print the total and average request times before all threads were finished executing. To resolve this, we replaced the start\_new\_thread function with the thread.thread function, which used an object reference for each thread. We used the thread.start and thread.join methods in separate loops to start each thread and then force the program to wait for all the request threads to finish.

**Appendix**

**Data for 25 simultaneous requests:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Date** | **Uptime** | **Memory** | **Netstat** | **Users** |
| **Requests completed** | **Time (ms, average of 10)** |  |  |  |  |
| 1 | 3.4484148 | 5.37984371 | 5.09839058 | 9.71877575 | 12.8719807 |
| 2 | 6.44111633 | 8.90512466 | 8.71903896 | 16.6306734 | 23.060751 |
| 3 | 8.69374275 | 12.1742725 | 11.9035721 | 23.6438751 | 33.2750082 |
| 4 | 10.8686924 | 15.4533386 | 15.0940418 | 30.5189848 | 43.7057734 |
| 5 | 13.0665302 | 18.721652 | 18.3026552 | 37.3812914 | 53.9237738 |
| 6 | 15.2700901 | 21.9630003 | 21.4935541 | 44.2591429 | 64.1652822 |
| 7 | 17.483449 | 25.2322197 | 24.7279406 | 51.133728 | 74.4329691 |
| 8 | 19.6404696 | 28.4660816 | 27.8881788 | 58.0355883 | 84.6289396 |
| 9 | 21.8566895 | 31.7804575 | 31.1082125 | 65.2779818 | 94.8453665 |
| 10 | 24.0394354 | 35.1015568 | 34.4076633 | 72.0910311 | 105.061293 |
| 11 | 26.2277842 | 38.3652925 | 37.6370192 | 78.910923 | 115.31055 |
| 12 | 28.431654 | 41.6130781 | 40.8432722 | 85.7488394 | 125.608826 |
| 13 | 30.6055546 | 44.8577166 | 44.0274239 | 92.5773144 | 136.066294 |
| 14 | 32.5889587 | 48.0946541 | 47.2202539 | 99.4180441 | 146.548605 |
| 15 | 34.7580671 | 51.3164759 | 50.4067659 | 106.197572 | 156.752801 |
| 16 | 36.9541168 | 54.5814037 | 53.6037922 | 113.006306 | 166.991591 |
| 17 | 39.1145229 | 57.8407049 | 56.8137884 | 119.799805 | 177.183938 |
| 18 | 41.3155794 | 61.060977 | 59.9809885 | 126.612282 | 187.451577 |
| 19 | 43.4993982 | 64.2636776 | 63.117671 | 133.447886 | 197.58451 |
| 20 | 45.7268953 | 67.4142599 | 66.2397385 | 140.220785 | 207.730842 |
| 21 | 47.8756666 | 70.6403494 | 69.4083452 | 146.975231 | 217.90576 |
| 22 | 50.075984 | 73.8982916 | 72.5929499 | 153.776717 | 228.120804 |
| 23 | 52.2303581 | 77.0422697 | 75.7701635 | 160.457087 | 238.346052 |
| 24 | 54.4252872 | 80.3062439 | 78.9834976 | 167.20283 | 248.596644 |
| 25 | 56.5771341 | 83.6500645 | 82.252264 | 173.995543 | 258.865499 |

|  |  |
| --- | --- |
|  | **Processes** |
| **Requests completed** | **Time (ms, average of 10)** |
| 1 | 14.6744251 |
| 2 | 25.4880905 |
| 3 | 35.2477074 |
| 4 | 44.5972443 |
| 5 | 53.8013697 |
| 6 | 63.0382061 |
| 7 | 72.349143 |
| 8 | 81.6559553 |
| 9 | 90.9705639 |
| 10 | 100.265098 |
| 11 | 109.533453 |
| 12 | 118.769383 |
| 13 | 128.041935 |
| 14 | 137.270212 |
| 15 | 146.512532 |
| 16 | 155.785036 |
| 17 | 165.084791 |
| 18 | 174.306893 |
| 19 | 183.503914 |
| 20 | 192.745733 |
| 21 | 202.878261 |
| 22 | 214.464164 |
| 23 | 225.631166 |
| 24 | 236.399984 |
| 25 | 247.716808 |

**Total and Average Times for Netstat and Date Requests:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Date** |  | **Netstat** |  |
| **Simultaneous Requests** | **Total (ms)** | **Average (ms)** | **Total (ms)** | **Average (ms)** |
| 1 | 2.79402733 | 2.79402733 | 15.8722401 | 15.8722401 |
| 5 | 56.9419861 | 11.3883972 | 113.62505 | 22.7250099 |
| 10 | 139.362097 | 13.9362097 | 400.89345 | 40.089345 |
| 15 | 310.045719 | 20.6697146 | 878.099203 | 58.5399469 |
| 20 | 555.249691 | 27.7624846 | 1535.97045 | 76.7985225 |
| 25 | 916.245699 | 36.649828 | 2416.35132 | 96.6540527 |