Anthony Waddell

CSS 432

HW1 – Intro to Network Programming

April 7th, 2018

**Documentation**

**Client.cpp**

To compile: g++ Client.cpp -o filename

The client side of the network connection can all be completed in main. Running the executable takes 6 arguments. To connect to the server socket, the client node must create a socket for itself and then reach out to the server socket. To establish its socket, the client uses AF\_INET which uses IPv4 protocol and SOCK\_STREAM which uses TCP protocol. The third parameter for creating the socket, 0, corresponds to the IP protocol value. After a socket has been established, the client can attempt to connect to the server socket by calling connect. The server sockets address is established earlier we declare the hostent\* structure and assign it to the IP address argument we pass in. In addition to establishing the IP address for the server sockaddr\_in, we also assign it the same address family internet IPv4 protocol and a predetermined port number.

If we can successfully connect to the server socket, then we can start the communication between the two nodes. In Client.cpp, this means we can now create a data buffer, and begin start tracking time. The first scenario to track is multiple writes which calls write() for every number of buffer arguments we pass in from the command line. The second scenario uses a vector and \*iov\_base to fill several buffers (depending on the number of number of buffers passed as an argument) and then write them to a socket descriptor. The third scenario is uses a single write to send an array the size of *nbufs \* bufsize* bytes to a socket descriptor.

When the communication is finished on the client side, we can track the time that it took to send data and read back from the server socket how many times it had to read based on the packets we sent over. Then we stop the timer altogether and use the timevals we created to determine the time it took to send data and to get our read count back. For good hygiene and to end the session we close the socket and exit.

**Server.cpp:**

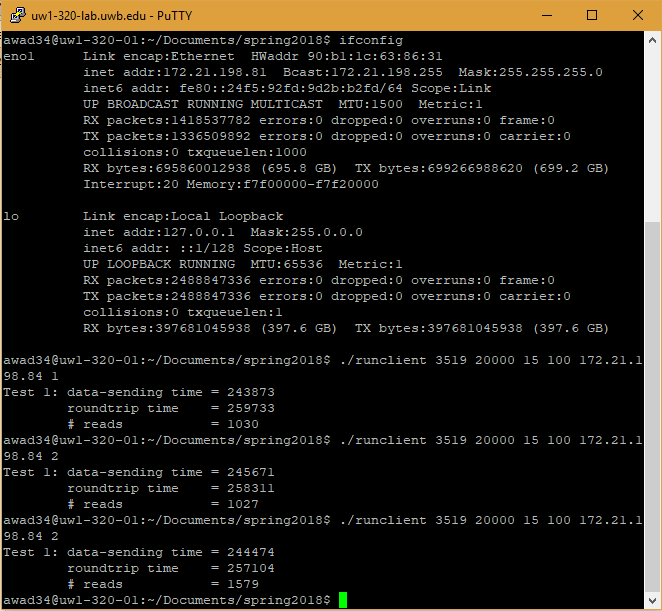
To Compile: g++ -pthread Server.cpp -o filename

The server side of the socket creation for this project benefits from having a separate function that handles what this program is intended to accomplish. Main takes two arguments from terminal: port and repetitions; these arguments are the same for Client.cpp. Server.cpp follows a similar logic flow of Client.cpp, setting up a sockaddr\_in to accept a new connection with the client using the same IPv4 protocol and predetermined port number. After creating a socket we bind it to the IP addres and port number of the client by calling bind(). Once we have bound csocket to the client socket, we call listen to wait for the client’s request to connect.

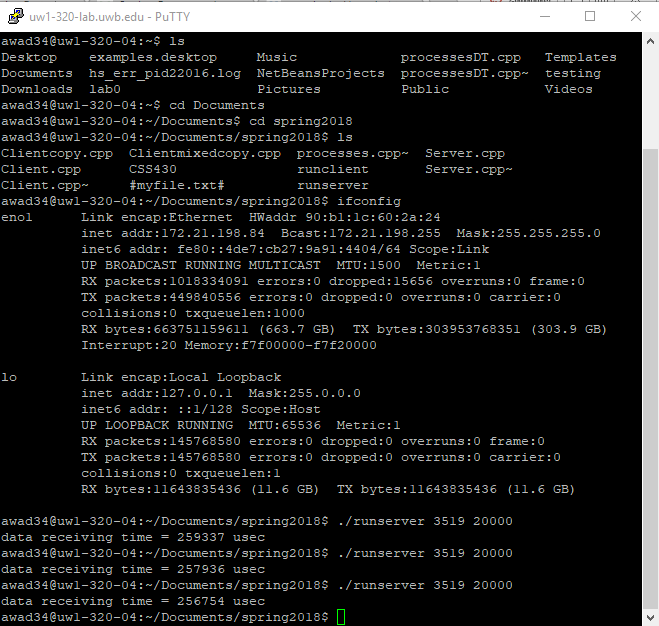
If we can establish a connection, we call accept and create a new socket with a new file descriptor. At this point, we can start reading data from the client and sending acknowledgements back. For this program, we create a new thread that begins by creating a buffer and starting a timer to track execution time. The process is similar to the write process from Client.cpp but instead of writing, it only reads from the what the client sent, into a buffer, and increments the number of times it has to read data based on what the packets sent by the client. When finished , we send the number of times read() was called back to the client, stop the timer to track the receiving time, and close the connection.

**Execution Output**

**Client command view**

****

**Server command view**

****

**Performance Evaluation**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Repetition** | **Nbufs** | **Bufsize** | **Type** | **Data Send time in usecs** | **Round trip time in usec** | **# of reads** | **Data receive time in usecs** |
| 20,000 | 15 | 100 | 1 | 245970 | 262211 | 4925 | 261672 |
| 20,000 | 30 | 50 | 1 | 289971 | 290590 | 5361 | 290086 |
| 20,000 | 60 | 25 | 1 | 638800 | 639138 | 11561 | 638708 |
| 20,000 | 15 | 100 | 2 | 241859 | 259940 | 3356 | 259743 |
| 20,000 | 30 | 50 | 2 | 244987 | 257916 | 2458 | 257628 |
| 20,000 | 60 | 25 | 2 | 242287 | 257583 | 4932 | 257347 |
| 20,000 | 15 | 100 | 3 | 249100 | 262912 | 989 | 262248 |
| 20,000 | 30 | 50 | 3 | 243909 | 261021 | 954 | 259902 |
| 20,000 | 60 | 25 | 3 | 232970 | 260515 | 1136 | 260063 |

**Discussion**

For my performance evaluation, I ran the client and server executables on two separate Linux machines in the lab. For this reason, I believe my results were marginally faster than having ran one or the other executables on WiFi and the other on a lab machine. Overall it seems that writev() performed best in terms of data transmission. If looking only at the number of reads, the single write operation performed best by a large margin against the other two operations. Multi-writes seems to perform the worst of the three operations, although not terribly worse when nbufs was lower and bufsize was higher. With multiple writes and a larger number of buffers however, the multiple write operation performed significantly worse than any of the other test cases; more than doubling every measure that every other test case output. It seems that as the single write operation increases its number of buffers and decreases its buffer size, it gets faster. I think with further testing it would be interesting to graph these results and see when single write becomes faster than writev.

From this assignment and the performance evaluation, it seems that the greatest variation was in the data-sent time and the least amount of variation was in the round-trip time and the data received time. While single and writev operations behaved similarly, it seems that writev experienced performance issues based on the input and output operations associated with it. I am not sure why, but I would guess it would have to do with write having to wait to complete the vector assignment operations before writing to the client. I think the biggest reason to use a thread to service the connection rather than servicing it in the main function would be performance. If everything is happening in main, network operations like what we are accomplishing in this project would slow down and result in more issues. By creating a new thread in Server.cpp however, the server doesn’t have to worry about what happens after it creates the new thread.