Q1. What was the biggest challenge that you encountered in this assignment? [300-350 words]
The biggest problem I encountered in this assignment was designing and implementing the thread pool manager. Initially I had a basic idea of when a task comes in it checks for a worker and if there is one available it polls the worker queue and the worker starts on the task or when a worker thread finishes a task it attempts to poll another task and run it otherwise the worker thread adds itself back to the worker pool. This was the most basic idea of how to implement this design but I ran into problems of deadlock due to synchronizing too many locations which made it so adding a task to a worker thread was an issue. After looking at my design I realized I was synchronizing on a lock for too many tasks in my thread pool manager so I divided the locks and saw a huge increase in performance right away. Another thing that caused me problems was making sure that my reading and writing was not blocking one another on the channel. I didn't come up with a good solution so I made it so once I start reading from the channel I will not read until I am finished writing. Also, I struggled with dealing with software changes needed to fix bugs with not cleaning up certain parts of my code which made it more and more difficult to understand what was actually being done. More comments and actively fighting against the slowly building code obfuscation.

Q2. If you had an opportunity to redesign your implementation, how would you go about doing this and why? [300-350 words]
If I could go back I would redesign my Thread Pool Managing implementation since it was nice at first when I developed my testing but it didn't scale well due to some synchronization problems in the manager. While fixing some of these bugs I got rid of my BlockingLinkedQueue class since it would work better in software design but it was adding a bit of code complexity. Also, I think I should have combined my ThreadPool and ThreadPoolManager class since they did so many similar things and only added to code complexity when attempting to add Workers back to the ThreadPool. This would have added a couple more lines of code but would have made the code complexity much lower. Overall my ThreadPoolManager classes felt a little sloppy because I let fixing bugs get in the way of turning in code that felt clear and concise. I attempted to do a lot of cleanup, but with the time limitations and not writing good enough tests for my ThreadPool initially it was reflected in the multiple late redesigns of my ThreadPool classes. Also, looking back I'm wondering if it would have been better to workout a way to use wait and notify methods instead of an infinite loop checking for the task not being null. It would have used a lot less CPU probably which would have made my programs scale better with multiple clients on one computer.

Q3. How well did your program cope with increases in the number of clients? Did the throughput increase, decrease, or stay steady? What do you think is the primary reason for this? [300-350 words]
Initially my program had a lot of problems scaling well and steady with the throughput and reading and writing through the buffers. A simple redesign to my ThreadPool class made it so that my client scale very well to 100 clients at a message rate of 4. I didn't spend too much time going beyond that since I was having problems starting that many clients due to getting errors

when using the script and not being able to connect from certain cs computers. The primary reason I think my implementation did well is that if there was a task and worker ready it would be started and otherwise the systems would wait for something to be ready. My throughput stayed quite steady at a 95% of the expected value which is what I expected due to how I implemented the read and writing to the client. I wanted to check how much time the Worker threads were being used but it felt like they spent an equal amount of time being active which lead to the load being distributed nicely. Also, the one thing that I'm a little worried about with my implementation is that after I completed a read task I would block that channel until I completed the writing task which means that I couldn't queue up another reading task until completing the writing. Looking back it would have been nice to have a way to queue up the reading tasks so that the channel could read while writing but I'm still not sure if that is possible. If that did happen my throughput probably would have stayed a little closer to 100% expected, but I did notice a slight slow down when a new client was connected which might be because of blocking while iterating through the keys which caused a bit of slowdown.

Q4. Consider the case where the server is required to send each client the number of messages it has received from that particular client so far. It sends this message at fixed intervals of 3 seconds. However, since each client has joined the system at different times, the times at which these messages are sent by the server would be different. For example, if client A joins the system at time T0 it will receive these messages at {T0 +3, T0 +6, T0 + 9, …} and if client B joins the system at time T1 it will receive these messages at {T1 + 3, T1 + 6, T1 + 9, …} How will you change your design so that you can achieve this? [300-400 words]

Looking at this the most obvious way would be to start a thread when a new active connection comes into the server that waits 3 seconds before sending the message to the client. I would of course make the string to have a size of 40 bytes similar to the SHA-1 otherwise it would cause an infinite loop of trying to read out data if it was greater or less than 40 bytes. This would work only if I had a HashMap that kept track of the active connections and stopped them running when the connection was closed. I believe this is very similar to a heartbeat message and I'm not positive how that is handled but would not have been too difficult since I had one class managing all the throughput so instead of increasing the servers overall throughput I would have checked which channel was being read or written to and incremented a tracker for that client through another HashMap then when the HeartBeat thread came time to send the message it would request that number from the ServerMessageTracker, wrap it up, and send to the client. That seems to be a pretty efficient way of dealing with the task of tracking and sending the information depending on when a client connected.

Q5. Consider the overlay that you designed in the previous assignment. This overlay must support 10,000 clients and the requirement is also that the maximum number of hops (a link in the overlay corresponds to a hop) that a packet traverses is not more than 4. Assume that you are upgrading your overlay messaging nodes using the knowledge that you have accrued in the current programming assignment; however, you are still restricted to a maximum of 10 threads in your thread-pool and 100 concurrent

connections. What this means is that your messaging nodes are now servers (with thread pools) to which clients can connect. Also, the messaging nodes will now route packets produced by the clients. Describe how you will configure your overlay to cope with the scenario of managing 10,000 clients. How many messaging nodes will you have? What is the topology that you will use to organize these nodes? [300-400 words]

This question seems a bit oddly worded, but I believe that a good way to manage this is to move from Dijkstra's Algorithm and move to a Minimum Spanning Tree similar to the last Writing Component question, but with the added requirement that we must limit connections to 4. It would require more connections overall I believe but with the requirement of 4 hop maximum I don't see a way around dealing with this. I think the first thing I would do is to use the ThreadPoolManager to deal with the incoming connections and that would speed up the system considerably. I'm a little confused on the 100 concurrent connections but I believe this to mean that of the 10,000 clients they need to have 100 messaging nodes and I would use a topology to split them such like a hash where the first 100 clients would get a messaging node and then the next 100 and so on until we hit all of the concurrent clients. This would allow for them to be split up and deal with the messaging nodes connections.