Q1.

I had lots of concurrency bugs that I had to fix through the course of the assignment. There were situations where things like synchronization failed me where I didn't think they would. For example, when there is a synchronized block, but something that was supposed to enter later happens to enter earlier than expected. I also focused a lot on making my batch data structure concurrent. I wanted messages from different clients to be able to be added concurrently, so that meant I couldn't just synchronize the whole thing. The other problem with that was that when I needed to be able to add another client to a batch on the fly. I ended up with errors when trying to add another client head to the link list while trying to add messages to a different client in the list. Since I didn't want to stop the parallelism of the whole data structure, I used a semaphore instead. Worker threads would take out one permit from the semaphore, which was initialized to have a very large number of permite, every time they needed to add something to the data structure. Since I had enough permits, all threads could add in parallel. However, when I needed to add another client to the data structure, I would use the semaphore to grab all 1000 of the permits to the data structure. Because it has to grab them all, creating another client would wait for everything else to finish adding, then prevent anything from trying to add and causing problems while it is still adding. I had a problem where even though I synchronized wait notify, the scheduling thread could grab the synchronization lock first and notify before a thread had notified the pool. I had to make sure I was properly adding the thread to the pool inside the synchronized block.

Q2.

There are some relics of different approaches I was trying around my code, and I would like to go back, clean them up, and refactor properly. I would also properly use a semaphore on my counting statistics instead of synchronization to increase parallelism in some parts of my code. As far as conceptual redesigns, there aren't many I would like to do at the moment, I implemented all of the ones I had on my radar. One things I would like to extend the functionality on would be the processing of messages. It would be cool if the server was able to decode messages and create different tasks like the last assignment with messages. I would like to add some neat function of the server implemented as different tasks that messages would get decoded into an run with. There, I could use the implementation I have heard of where you batch selection keys to a selector instead of messages to improve locality. Each read selection key would read the data into the program, transform it into a useful task, then execute whatever that task needs to be executed, then probably send back some data depending on the task. Keeping all the data in caches would make the program run faster. I could also break up some of the functionality of my Thread Pool Manager into one more class, because that one is kind of long and could be refactored. I also might add more precise support for message sending time intervals, maybe down to the nanosecond level. Also, it could be cool to implement

some generalized protocols to allow my server to connect and do tasks for any person out there using said protocol. For example, maybe host part of a could processing network on my machine for some extra cash.

Q3.

My server was able to remain at relatively the same throughput up to around 700 clients. It did decrease slightly with client numbers, but not by very much. I tried 1000, and I was not able to get them all to register quickly after that. I'm not sure if that was because of limitations in my server, or my processes being put on hold in the department as I spawned too many of them. It seems that more clients doesn't add too much overhead per client to my server. With 10 clients, I get around 15,000 messages per second total throughput, and at 200 clients I run about 13500 messages per second. The way I have my batches implemented, more head nodes of each client will be created as more connect, and more socket channels need to registered with Java NIO. I also store the socket channels with the head of my client's message linked list. Storing socket channels is a type of overhead I don't think is very easily mitigatable. You could have each client re-register each time, but that would add a lot more tasks to be processed to the server's workload. However, with a very large number of clients, it may be better to open another socket channel each time, since memory space is so much more valuable than processor time when thinking in terms of what will cause a server to take more times to do tasks. I don't think this is possible with Java NIO however, since accepted keys stick around and remember that they have been accepted when it comes time to pull the socket channel. Data locality of messages and socket channels might start to matter more and more as you increase the number of clients, and data that was stored in caches needs to be moved off cache.

Q4.

To achieve a scheme like this where a per client timer is needed, I would dedicate one thread to all timing events on the server. This timing thread would look through all client timers, as well as the batch timer. Whenever the client timers expired, the thread could add a task to the task queue that tells a worker thread to send a special message with statistics back to the client. The client would now need to be able to differentiate between that message and an incoming hash now, so we would need a marshaling and unmarshaling scheme like the one we used in project 1 to keep track of different messages. Because we are also adding these jobs to the taskqueue that need to send things to the client at specific intervals, I might want to use a priority queue so that I can better guarantee that the message goes out to the client at closer to the correct time. Batch process tasks would also have high priority, so that the server can better respect batch time and batch size. I did not have guaranteed output right when the batch is marked as full in my project because I was

not using a priority queue, things worked in a more best effort fashion, where it would try to get the batch out as soon as possible. Any other timers that may need to be added to the server in the future could also be handled by this timer class, giving the software room to expand and add new features in the future. With this scheme, each client would be able to have different interval for their 3 second updates. Particularly I like this idea, because while having the 3 second timers be different on the clients may not be super important overall to the project, having the implementation gives the ability to do more powerful stuff as the project expands in the future.

Q5.

To achieve this expansion of the previous project with these goals the most important redesign would be setting up the overlay in a way that guarantees that hops will be 4 at most. One way to do that would be to have a link table that works to look at number of hops to get to each other node and minimizes it as the connections are made. It could help to make decisions about which nodes to connect to make sure than the number of hops to reach other nodes goes down as connections are made. Since each messaging node which now resembles our server from project 2 can have at maximum 100 connections, 100 will be our link limit per node. I've been working with the idea of having connections at increasing powers of 2 away from the current node, but I don't think the hops can still be minimized to 4 with that strategy. Another idea might be to make connections at distances in both directions equal to the fibonacci series of numbers. An approach that might work would be to connect to the powers of 2 in each direction then fill the the biggest gaps between nodes in your table with your remaining connections. It takes 14 connections for a node to connect to its neighbors until the halfway point around the a ring of nodes by connecting to the power of 2s. That means it takes only 27 to connect in both direction since we connected to the opposite node. With the remaining 100 connection, I could fill the the biggest gaps that this power of 2 strategy leaves. Or rather connect to the mid points between all of the connections you already made, bringing the total up to 53 connections. This would half the number of hops needed, since each hop is now twice as close node offset wise from what it would have been. You would also need to use the remaining hops after that to connect at least the 5th and 7th nodes. Then even the worst case scenario can reach its destination in 4 hops. This is an interesting property, where we can start with powers of 2, and add connections as they are available to use in between the gaps, to achieve optimal maximum hop length. This same strategy could be abstracted to get a minimum for any problem size.