Max Hayne
HW2-WC
CS455
3.13.20

Q1. The largest challenge I encountered during the assignment was the problem of preventing concurrent writes to SocketChannels of the same client. The particular scenario I was worried about was if two threads in the thread pool were dealing concurrently with a write task to the same SelectionKey. With Java NIO, partial writes are possible, and therefore an interleaving of writes to the same SocketChannel is possible. To prevent this scenario, I attached a custom object to every SelectionKey upon registration that holds a semaphore. Whenever a thread in the thread pool attempts to write to a socket associated with a particular SelectionKey, it attempts to obtain the semaphore held by the attached object. If the attempt to grab the semaphore fails, the write task is returned to the taskQueue for another thread to pick up later and try again. The reason this was the biggest challenge was because it wasn't immediately obvious to me that this was a valid solution to the issue, as it almost seemed to negate the benefits of NIO. But the more I thought about it, the more it seemed like the only way to solve concurrent accesses to each SocketChannel. My implementation passively solves the issue of interleaved reads by revoking read interest from a SelectionKey when a read task is added to the taskQueue, and returning the read interest only after the message has been read from the channel.

That's about all there is to talk about for that particular challenge. Another challenge I encountered was keeping track of processed messages for every client connected to the server. Since I began dealing with this problem after I solved the issue described in the previous paragraph, I decided to add an atomic counter to every object attached to each SelectionKey. A TimerTask, called ServerStatisticsReporter, would, every twenty seconds, traverse the KeySet held by the Selector and collect counts for every client, do some calculations, and print findings to the console. What it took me while to realize was that the KeySet being held by the Selector was not thread safe, so in order for the ThreadPoolManager and the ServerStatisticsReporter to access them without throwing a ConcurrentModificationException, the ServerStatisticsReporter had to synchronize on the KeySet before iterating through it.

Q2. Honestly, I feel very confident about my current implementation of this assignment, and I don't think I'd change too much. Thanks to HW1, I felt much less paralyzed at the start of this project, both because I believe HW1 made me a better coder, and because I had a better grasp on the data structures necessary to handle concurrent writes from multiple threads. If I were forced to change something though, I'd probably attempt to minimize concurrent accesses to the KeySet maintained by the Selector on the server side. Currently, my ServerStatisticsReporter takes hold of the Selector's KeySet's lock every time it collects data about how many messages the server has processed for each connected client. This prevents, albeit for a short time, the Selector from

polling for new reads or registrations, because it cannot obtain its KeySet lock. This could probably be done better by maintaining in the ServerStatisticsReporter a List of SelectionKeys, and a LinkedBlockingQueue that threads in the pool can add SelectionKeys to upon registration. Then, when the ServerStatisticsReporter chooses to print statistics about current connections, it won't have to obtain the lock to a shared resource, it will instead iterate through a local list, skipping over SelectionKeys that have become invalid over the last twenty seconds.

While on the topic of statistics, another change I'd make to the server would be to keep permanent records of all connections and their associated message counts in a data structure that would reset every twenty seconds. In this way, the ServerStatisticsReporter wouldn't only report on statistics about clients currently connected to the server, it would report on statistics even for clients that connected at 5 seconds, sent 30 messages, and disconnected at 15 seconds. My current code doesn't handle these types of clients. My implementation was approved for this assignment, but it would be nice to keep track of all clients, permanent or temporary over the past twenty seconds to report on their statistics.

Q3. Overall, my program coped moderately well with increases to the number of clients. On my laptop, I can connect 200 clients, each of whom is sending four messages per second to the server, with no loss in performance. Also on my laptop, while keeping the number of clients at 100, I can increase the number of messages sent per second up to 150, without creating a differential between sent messages and received messages at each client. So yes, throughput increased when the number of clients increased, but only up to a point. The reason it increased was because the server had thread time that was not being utilized. As a back-of-the-envelope calculation: If a server is set up with ten threads in its thread pool, ten batches must be concurrently executing for items to pile up in in the batchQueue. And, if each batch holds ten tasks, 100 tasks must be in-progress for any task-pileup to occur. That's a fair amount of overhead in order to ensure that all threads are working on a batch at one time. And, if the server is finely calibrated to ensure maximum efficiency, responding to 800 messages per second requires 2400 tasks/sec (Read/Hash/Write), and therefore 240 batches/sec, and therefore 24 batches/sec/thread. This may not actually play out in a perfect world, as there may be collisions on writes to channels, or channels' buffers may fill up, blocking writes. As for why performance ever breaks down, it could be for a myriad of reasons, but simply, it is because the server's thread pool cannot complete the number of task requests that are streaming in at one time. Increases in clock speed, or number of cores, or memory speed would increase the throughput capabilities of a server running the same piece of code, but as more connections and tasks are required, the only viable solution would be to increase the number of machines responding to those requests.

Q4. In HW2, I'm using the feature of being able to attach objects to SelectionKeys to keep per-key counters for the number of messages the server has processed for that particular client. In

this way, every twenty seconds I can iterate through the KeySet maintained by the Selector to calculate statistics and print them out to the console. This can be done because the printing of statistics happens so infrequently (every 20 seconds) that holding the lock to the KeySet does not disrupt the regular operations of the server, and the polling of the Selector by the ThreadPoolManager. In handling this new problem, the same type of solution will not work because the frequency with which a thread would have to check the KeySet for "MessageCount" messages to send would be on par with the frequency with which the ThreadPoolManager would call Select(). Therefore, it would be necessary to write another class, let's call it *MessageManager*, which would hold a LinkedList called *clients*, and a LinkedBlockingQueue called *temp*. When the thread pool registers a new client, it will add the SelectionKey and timeOfRegistration of that new client to the *temp* queue. The thread running inside the *MessageManager* will continuously loop on the *clients* queue, and at the end of every loop will check the *temp* queue for newly registered clients to add to the *clients* queue. If ever a SelectionKey inside the *clients* queue becomes invalid, it is removed from the queue. While looping through the *clients* queue, the thread inside the *MessageManager* will compare the current time to the time stored inside the queue with that SelectionKey (perhaps in a tuple), and if the difference is more than three seconds, will add a write task to the *taskQueue* maintained by the ThreadPoolManager, and overwrite the time stored in the *clients* queue with the current time. In this way, the KeySet maintained by the Selector is not bogged down by concurrent accesses, and the current system by which the ThreadPoolManager takes tasks from the taskQueue and batches them into the batchQueue remains the simple centerpiece of the entire operation.

Q5. To reiterate the question, every node in the overlay network is now a server that can serve up to 100 concurrent client connections. On top of all that, messages from clients picked up by a MessagingNode destined for a different MessagingNode must reach that node in four hops or less. First off, in order to solve the problem of guaranteeing that every MessagingNode is within four hops of every other MessagingNode, the same technique as used in the first assignment cannot be reused, as the total number of hops even to get to a MessagingNode that is 127 ID's away in the namespace takes six hops. It becomes apparent that the system of routing used in the last assignment only allows a MessagingNode to reach another node in four hops if the destination node is 15 nodes away in ID space or less (in a worst-case scenario). Therefore, the only way to guarantee that every node can be reached in four hops or less is for every MessagingNode to have four routing table entries (2^0 thru 2^3) spanning the eight nodes succeeding it in ID space, and also a routing table entry for every MessagingeNode in the overlay that is 16 hops away, 24 hops away 32 hops away, 40 hops away (+8) and so on. So, if the overlay network is N nodes in size, every MessagingNodes' routing table will contain (4 + ((N/8)-2)) entries in it. Keeping the overlay network to a size that is a multiple of eight, the lowest possible sized network such that each MessagingNode is connected to the number of MessagingNodes specified above, and all other available connections are used for clients, is 128. In a system of 128 MessagingNodes, each MessagingNode will be connected to 18 other

MessagingNodes (4 + ((128/8) - 2)), leaving 82 connections (100-18) open for clients. And, 82*128 is equal to 10,496 total clients that can be connected to the overlay network at one time. Now, the issue of pairing a new client with a MessagingNode that does not have all of its connections filled is another issue. It would probably be necessary to have a powerful OverlayNodeManager, who keeps track of the number of clients connected to each MessagingNode in the overlay and who, upon request, re-routes registrations of new clients to MessagingNodes that have free slots available.

List of MessagingNodes that node zero would be connected to in an overlay of 128 nodes:
[1, 2, 4, 8, 16, 24, 32, 40, 48, 56, 64, 72, 80, 88, 96, 104, 112, 120]