Josiah May
March 7, 2018
CS-455
Shrideep Pallickara
HW2-WC

Q1. What was the biggest challenge that you encountered in this assignment?

My biggest challenge in this assignment was dealing with the NIO server selection keys. From what was explained to us in class and from what I researched, selector.select() was a blocking call that only returned something when one of the clients channels were ready for I/O operations. This sounded like we could get the selection keys from the selector, preform some task on the key, and then that key would not be selected again until it was ready for another I/O operation. This was not how selector.select() works. My server was making multiple tasks for one message sent from the client. To find the cause of this issue, I created one client that sent a message every one second. For every one task my server was supposed to create, the selection key went through the logic to find its op interest at least twenty times.

Since the selection key op interest was still set to OP_READ the server would create another task for the channel. The extra tasks meant my worker treads were blocking while trying to read an 8kb message that had not been set yet. To fix this problem, I set my selection key's op interest to OP_WRITE. When I was finished reading the message and sending back the response I set the op interest back to OP_WRITE. I also made sure my server only checked if the key's op interest was set to OP_ACCEPT or OP_READ. While this fix did not stop the key from being check multiple times, it did stop extra tasks from being created. This fix is extremely inefficient because when there are hundreds of clients connected to the server, the server repeatedly looping through all those keys even though it has already created a task to handle the I/O operation. There should be a better way to fix this issue, but I could not find it.

Q2. If you had an opportunity to redesign your implementation, how would you go about doing this and why?

When comparing this project to the first project, I think my classes in this project are far more thread safe. Instead of synchronizing most of my methods, I limited their use to only areas they were needed. I also did not let references to my state variables escape their class except for the selection key. I used the selection key to create a channel in the server, tasks, and in the statistics collector as the key for a hash map. This meant none of my classes could be sure of the state of the key when getting attributes from the key. I do not think this was an issue in my program because of how I dealt with the server creating multiple tasks. Since the server skips any selection key it already started a task for, only one worker thread will ever have access to a selection key at a time. To fix any potential issues with concurrency I would make a wrapper class for the key.

This wrapper class would fix an issue I have with my task class as well. Currently every task I create I also allocate memory for a read and write byte buffer. When the task is completed, the JVM needs to garbage collect the task and the allocated byte buffers. This means for every

client I am allocating and deallocating memory for two byte buffers multiple times per second. The wrapper class for the selection key could also hold those byte buffers. After a client connects, the server would make the wrapper class associated to the client's selector key and store it in a hash map. When the client sends a message, the server would make a task with the wrapper class instead of the key. Storing the wrapper class with the byte buffers might use more memory but it would reduce the overhead of garbage collection from the JVM.

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?

I tested my project's throughput thoroughly by adding more clients and increasing the messages sent per second above the expected 2-4 messages per second. After reading the Piazza post that stated, "a good implementation of the system should be able to handle nearly 10,000 messages per second", I tried to reach that threshold. My server has no issues running with 500 clients at 2 messages a second and 100 clients running at 10 messages a second. However, when I tried to increase my throughput from 1000 messages a second to 1500 messages a second, my server's maximum throughput could only reach between 1410-1470 messages per second. This maximum throughput was consistent whether there were 100 clients or 500. I thought there might be to many messages for a thread pool of 10. So, I increased by thread pool to 20, but my maximum throughput stayed between 1410-1470 messages per second.

I was not the only person to run into this maximum throughput issue. I talked to at least 10 other students working in the 120 lab, and all of them had a maximum throughput in the 1400's messages per second range. One student claimed to have reached 40,000 messages per second, but he was only sending 8 byte messages instead of 8 kilobyte messages. To test if the message size was the bottle neck, I changed my message size to 8 byte, and my server could easily handle over 10,000 messages per second.

Another characteristic my server shared with other student's servers was none of our throughputs exactly matched our intended throughput. If I was aiming for a 400 message per second throughput, my server's throughput would be between 398.5 and 399.2. As my intended throughput increased so did this gap in actual throughput. At 1400 messages a second my server was only running at about 1395 messages a second. I did notice that as the number of clients increase my gap would also increase.

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.

Currently my statistics thread stores the number of messages sent per node in a hash map with the selection key as the key for the hash map. It would be inefficient to make another hash map for messages received. So, I would start storing the counts of messages received and messages processed in the wrapper class I described in question two. When a task is added to the queue I would increase the message is received counter, and when it is finished I would increase the messages sent counter. Both counters would be protected by their own lock to keep their states thread safe. Every three seconds the statistics thread would get the number of messages received and reset the counter, and every twenty seconds it would do the same with the messages sent counter.

Since the message counters are now being stored in the wrapper class, my statistics thread would change from storing selection keys in a hash map to storing wrapper classes in three sets. When a client connects to the server, I would get the time it connects in seconds and mod the time by three. The results would be which set the client is assigned to. Every second the statistics thread gets the time and mods it by three. Then it would loop through the set associated with the results, and send the messages received counts to the client.

This method does introduce a synchronization issue. Both the statistics thread and the worker thread could try to write to a socket channel at the same time. To fix this the wrapper class would need to issue a lock before any thread can access the write buffer. I would also need to change how I send and receive messages. Currently, the clients are looking for a 40-byte hash code. I need a way to designate between a message with a hash and one with a counter. I would use a system similar to the first project with the size of the message being sent first. Since it is unlikely for the counter to be 40 digits long, any message length that is not 40 bytes long must be a counter.

Q5. Suppose you are planning to upgrade (or completely redesign) the overlay that you designed in the previous assignment. This new 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.

The biggest challenge of this redesign is making sure a message only takes a maximum of four hops to reach its destination. This means it needs to be at least able to transvers over the entire network in four hops. To accomplish this, I know that the maximum hop distance from the node needs to be $2^{(N-1)} \geq 2500$. So, the size of the overlay must be at least 13 nodes ($2^{(13-1)} = 4096$). However, being able to hop over the network in four hops does not guarantee the message reaches its destination in only four hops. Since the distance between each hop in the overlay is in powers of two, I think I can treat the overlay like a binary search. A binary search is O(log N) and log(10,000) = 4. For this method to work, the last hop in the overlay must be at least half of the total network. The size of the network overlay should be 14 nodes ($2^{(14-1)} = 8192$).

With an overlay with only 14 connections, a messaging node will never come close to reaching the 100 connections limit. The only messaging nodes that should connect to a messaging node are the node 1, 2, 4, …, 4096, 8192 hops behind the node. Therefore, a messaging node should only have connections to the 14 nodes in its overlay, the register, and 14 other messaging nodes connected to it. Which makes the max number of connections for a messaging node to be only 29.

It would be easy to convert the messaging node to an NIO server and implement a task-based system. I would make several changes to my TCPConnectionsCache. The TCPConnection would become a wrapper for the selection key like I discussed before. I would store the TCPConnections in a synchronized collection in the TCPConnectionCache. Since I did not know about those collections in the first project, I made every method in TCPConnectionCache synchronized. This fixed concurrency bugs but tanked my throughput. The messaging nodes were only able to send one message at a time. Instead of synchronizing all the methods, I would let the collections stop concurrent modification issues. Whenever a worker thread needed to read

or write from a socket channel it would get the wrapper class from the collection and grab that objects lock. This prevents multiple threads from accessing the channel at the same time but allows other threads to access other connections.