Josh Keahey
3.14.19

**Homework 2: Writing Component**

**Q1.** The biggest challenge for me for this assignment was deciding how to implement the linked list of linked lists of work units for the thread manager to control. I also talked to others in the class, as well as checked Piazza about using specific data structures over others. Trying to determine if I should use a LinkedBlockingQueue, ConcurrentHashMap or ConcurrentLinkedQueue took more time than I thought it would. I ended up going with the ConcurrentLinkedQueue (which I'll call CLQ from now on), and didn't have many problems with it. At first, the way I structured the work was a CLQ that contained a CLQ of byte arrays. Since I obtained the key from the client in my Server class, I had a hard time figuring out how to send the hashed string back to the original client that sent the byte array in the first place.

Realizing that I could have a CLQ of CLQ that contained an object of work, containing as much information as I wanted it to, took the most time out of anything else in the project, even the implementation of the thread pool. In an object of Work, an extra class that I made, it could have the byte array being sent to the worker thread, but also the key from the client who sent it, so when it became a task, the Work object would already be able to send the hash back to the client. I feel like because of this speed bump, it really helped me see a new and different way to tackle problems in the future when it comes to a server and clients.

A close second as far as challenges go, was trying to determine the smallest possible scope for variables. Could it be static? Private? Package-private? Public? Protected even? I'm obviously hoping I did a good job with them, but I'm sure there is more to learn as far as that goes.

**Q2.** As stated above, making sure all my variables are in tip top shape when it comes to scope, notably the smallest possible. Being able to perfect this will help when going up against new problems in the future.

The biggest thing I would change in the future is the integration of both statistics classes for server and client. I feel like it's one of the biggest things that I didn't complete for the project. I tried to do what I could with the statistics in the server and client classes themselves, but being able to actually create objects that contain the messages sent and received would be a big help displaying the mean, standard deviation, and every other bit of information we needed to show.

The other element I would change is maybe trying to incorporate locks for the queues when it came to compound actions. I used the synchronized keyword to help with that, and I think it did the trick, but I feel like locks could also be useful and would be good to know in the future.

The last element I would not necessarily change, but at least play with, would be the use of the ConcurrentHashMap object. Depending on if I could understand it enough, and if the throughputs of server and client increased, I could use it in the program instead of the CLQ. Being able to have, for the default, 16 threads work on the same object at the same time sounds like a powerful thing to have working for you.

**Q3.** Once the amount of clients went to 100, (or around 100, as for some reason sometimes all clients wouldn't connect) the system did not slow down or lag in the slightest. What was interesting for me, was when I tested with one client, I would print the size of the CLQ that stored the hashed strings. For just one client, the size of the queue steadily grew. It was just by one or two elements, but after a minute or two, the list size was at 18 or 19. As soon as I tested 4 or 5 clients, checking each one showed that the size for each queue in each client was at a steady 7 or 8, and wouldn't get any higher. At 100, even though I couldn't check in depth for all 90+ clients connected, all were within 20 elements, and not increasing in size. This was nice to see, and helped with deciding if what I was doing was correct or not.

At first, I wasn't sure why with one client, the queue size was greater than with more clients connected. I think I have have an explanation to why that is now. What I believe happened was with one client, I was sending messages at every half second (if message rate was 2), and when the work unit queue got to the batch size, it added it to the work queue; a worker thread was dispatched and made to do work in the form of hashing and sending back that hash to the client. While this happened, more message were being received and added to the work unit queue. Maybe the work that was being added to the work unit queue was being added slightly quicker than a worker thread was being assigned, and so there was only ever one worker working at a time. Because of this, the messages sent from the client to the server were a little quicker than the server (or in my case, the Task object) sending back to the client.

When more clients joined however, now every half second had 4 or 5 messages incoming, filling up the queues much quicker than before. The workers would keep noticing that the main work queue wasn't empty, and started doing work on the next work unit queue, completely utilizing all 10 (or however many were given at the command line) worker threads to concurrently do their work and send the hashes back. Each client would be constantly receiving these hashed strings, and could compare and delete from its own continuously. So, to sum up, one client in this setup wouldn't necessarily take full advantage of the thread pool, whereas many clients definitely could. At least that's what it seems like to me.

**Q4.** Without testing this extensively I'm not sure if it would work, but what I would try to do is create something that contained the information (The client key, hostname, port number, etc.) of unique clients and add that client's info to some kind of data structure. I might use the ConcurrentHashMap for this, or maybe the CLQ again. This would probably have to be done in a thread, or maybe even a thread pool so it can send data back quickly if there are 100+ clients connected. Each unit in the data structure is an object from a new class I would create, and as soon as the client is registered, a method would be called that started a timer. I would probably put this in a while true loop. I would need to update the messages in the object every time the server received another message from that client. I would also need to check using synchronized or a lock, if the client has already registered every time I receive a new message so as not to add the same client multiple times in the data structure. Once the timer expires, it uses the key it received upon being created and sends the message to the client with the number of messages received at that time. Since this is in an endless loop in a separate thread, the timer will start over again and repeat this process.

Maybe by playing around with java NIO some more, there could be an easier, or more efficient way of doing this without threads at all. This is just the base idea I would begin with and go from there.

**Q5.** If each messaging node in the overlay had 100 concurrent connections that it could handle, I would have 100 messaging nodes in the overlay. Theoretically, that would be able to handle 10,000 clients connecting and sending messages at roughly the same time. If one server could take care of 100+ clients pretty easily when correctly implemented, multiplied by 100 should be good for this situation. As far as connecting to other nodes, I would first go through every node 1 through 100 and connect each one together so as not to create a partition in the network. So node 1 is connected to 2 is connected to 3 and so on until node 100 is connected back to 1. Then I think each messaging node just needs to be connected to 2 or 3 other nodes at most, bringing the total connected nodes to 4 or 5, for us to send a packet anywhere in the overlay by 4 hops. The thread pools, and use of java NIO could make it easier to send these hundreds of thousands of messages quickly.

Each node would act as the server and client from the first homework assignment, as far as keeping track of the path that the specific message needs to go (like the server), and actually sending the messages to the different nodes (like the client). Since they are all servers, each node could be listening on a thread for incoming clients, and dividing them up between themselves, then use their thread pool to send messages concurrently. If a messaging node has too many clients registered, it could notify that it will slow down if more are added, and the client could be registered on another messaging node instead. This could help find other nodes that are less busy, and keep throughput as high as possible.