Jack Fitzgerald
3/13/2020
CS455: Introduction to Distributed Systems
Homework 2: Writing Component

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

Getting the thread pool to work with NIO was the most challenging thing I encountered during this assignment. I did not anticipate some of the problems that would arise from letting my thread pool handle the network communications such as accepting client connections and reading data from those clients. When accepting incoming connections, I didn't realize that there would be a significant amount of delay before a worker thread would get the task and execute it. The delay would cause the select() call in the main thread to still think there was an incoming connection trying to be accepted and would create more "accept" tasks to be added to the work queue. Once one of the accept tasks was successfully executed by a worker thread, the rest of the accept tasks would fail. The way to avoid this from happening was to either remove OP_ACCEPT interest from the server socket channel, or attach an object to the server socket channel key that would indicate whether or not the server socket was already trying to accept an incoming connection by using a boolean variable. I chose to remove OP_ACCEPT interest from the server socket channel, and then add it back once it finished accepting the incoming connection. Doing this also avoids a few unnecessary cycles through the select loop in the main thread.

Like accepting incoming connections, I chose to remove OP_READ interest from the client channels when trying to read from them for the same reason of avoiding cycles in the select loop. Reading had the same problem as accepting connections, the delay between when the task was created and when it was picked up by a worker thread would cause many more read tasks to be created. Once the first read task finished, it would read all 8KB at once and cause the other read tasks to get stuck reading nothing until they could get a chance to read from the channel.

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

If I had the opportunity to redesign, I would try to abstract my main thread on the server that did all the NIO to be able to run in multiple threads to load balance the amount of messages trying to be handled by a single thread. There could be one main thread that handled accepting new incoming connections, then there could be other threads that have their own selectors. The main thread could accept clients and then choose a selector thread that has the least load to register the newly accepted client channel with. The selector thread could be picked based off the statistics gathered for the number messages processed in the past 20 seconds, the thread with the least number of messages processed would be chosen. Hopefully this type of load balancing would scale well and increase throughput, testing could be performed to see how badly throughput plummets when the number of messages being processed hits a certain threshold to indicate when load balancing should be taking place.

Another thing I wanted to experiment with was using a priority queue Instead of a FIFO queue to store tasks for the worker threads. I was planning on using a priority queue to give accept tasks the highest priority than any other task. The idea was that if there were too many clients connected and sending messages, then it might take a while for the accept task to be picked up by a worker thread thus making the server appear slow to the client. However, I don't think this would have achieved any significant

improvements and could contribute to a perceived slowdown by clients that are already sending messages. If many clients connect simultaneously, then any of those accept tasks that manage to make it onto the work queue before other reading and writing tasks can be picked up will make the reading and writing appear slower overall. Implementing a priority queue this way would give an illusion that accept tasks execute quicker, while running the risk of slowing down "lower" priority tasks like reading and writing to socket channels.

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?

My program would batch reading tasks, and it would only perform one read task per channel at a time. This typically meant that if the number of clients was less than the batch size, then the throughput would be dependent on the batch timeout. If there was a batch size of 10, and only 1 client was connected sending 4 messages per second, then the batch timeout would need to be 0.25 seconds to keep up with the client and not lose any throughput. If there were at least 10 clients connected, then the throughput would not be dependent on the batch timeout and batches could start filling up as soon as the previous read task finishes and adds read interest back to the channel. Once the total number of clients connected was equal to or greater than the batch size, the throughput increases significantly and stays relatively steady as more clients are added. Adding more clients at this point would slightly decrease throughput, with noticeable decreases at intervals of 100.

For example, with 100 clients sending 4 messages per second, the statistics seemed to be relatively stable for every 20 second interval. But when adding another 100 clients to bring the total count to 200, the statistics began to noticeably fluctuate. At 100 clients sending 4 messages per second the messages processed per second stayed at about 3.99, while with 200 clients the messages processed per second fluctuated between 3.90 to 3.99 which is a noticeable decrease in throughput. I think the primary reason for the decrease in throughput at this stage is because the rate at which batches of tasks are being added to the work queue is faster than the worker threads can process them. Adding more threads to the thread pool along with more CPU cores would allow the server to process more batches in parallel and increase throughput when handling larger amounts of clients concurrently.

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?

To achieve this, I would probably want to use a scheduling algorithm that could be executed in only a single thread. Opening a new thread each for each client to handle the timing would not scale well. The scheduling algorithm would need to be able to identify the starting times for each client and then continually check if at least a multiple of 3 seconds has passed. Another idea is to start a chain of tasks that can be passed off to the thread pool. Once a client has connected, a new task could be created that keeps of track of when it was created with System.currentTimeMillis() call, and when a worker thread picks up the task it makes sure that the difference in time is greater than or equal to 3 seconds. Once that task is completed, it creates another task of the same type and adds it onto the work queue. This

could slow things down though, since in the worst case a worker thread could immediately pick up the task and must wait 3 seconds before sending the receive count. This is approach is probably not much better than making dedicated threads to handle the timing, and it could easily overwhelm the thread pool since there would be a constant stream of the tasks especially when 100s of clients are connected.

Java's Timer object allows multiple TimerTasks to be executed in a single thread, so using the Timer class would be a great solution. The Timer class uses a single background thread to execute all the timer tasks when a client connects a new timer task can be scheduled every three seconds. The timer task can be specified to read and reset the statistics and then create a task that one the worker threads will use to send the receive count back to the client.

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. 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?

If I was restricted to using only 10 threads and 100 concurrent connections for each messaging node, then I would want at least 100 messaging nodes in the overlay so that I can handle up to 10,000 clients. However, it would probably be better to have more than 100 messaging nodes so that they all wouldn't be at maximum capacity when there are 10,000 clients. If there were 128 messaging nodes then a routing table of size 6 could be used, but this means that the maximum number of hops could be 7 in some cases. For example, going from messaging node 64 to node 63 would require 7 hops (64 -> 0 -> 32 -> 48 -> 56 -> 60 -> 62 -> 63). This violates the maximum of 4 hops requirement which means that something else must be done in order to bring the maximum number of hops down to a maximum of 4. Maybe instead of only being to route forwards, routing backwards could be an option, but this would require an algorithm to find all paths to a specific node and also make decisions about the shortest path to take since there would be multiple paths to the same messaging node. A better idea might be to make a few nodes solely responsible for forwarding messages and have them connected to a larger amount of other messaging nodes instead of taking on client connections. Then you could partition the connections in such a way that would allow for only a maximum number of 4 hops to any node. For example, the 128 messaging nodes could be split up in 4 evenly distributed partitions, and each partition has its own master messaging node. If a messaging node needed to send a packet to a node in a different partition, then it would send its packet to the master node for the partition which would then forward the packet to the master node of the destination partition and then finally to the destination node. By restructuring the topology this way, the maximum number of hops would be 3 (source node -> master node of source partition -> master node of destination partition -> destination node).