Jim Xu

Homework 2: Writing Component

SCALABLE SERVER DESIGN: USING THREAD POOLS TO MANAGE ACTIVE NETWORK CONNECTIONS VERSION 1.1

DUE DATE: Friday March 10th, 2017 @ 5:00 pm Please answer the questions below briefly.

Q1. What was the biggest challenge that you encountered in this assignment? [300-350 words]

ANS: The assignment needs a lot thinking about how to synchronize threads and how to make threads sleep and awake according to the needs. I think my biggest challenge in this assignment is setting up a thread pool and making the threads sleep and awake every time I feed a request. Because I never touched threads safety before, I have no idea on how to implement a thread pool and make it runs as expected. I looked up to the java uilt.concurrent library and find some source code to study, that helps me a lot. Then I tried to build a blocking queue which controls the dequeue action to be a wait-and-notify mechanism. I encountered a lot of problem here and tried different ways to build a brief and thread-safe link-list structure to store my threads, and in dequeue method I integrated thread pool theory, and finally make my blocking queue works like a FIFO thread pool. Another part of thread pool implementation is making the worker thread run whenever there comes a request and sleep when task finished. This part really needs a lot of knowledge on synchronized mechanism. Like how to correctly implement wait-and-notify mechanism to the same object but in different classes; how to design a task object that makes both writertask and readertask can both implements the task class and feed the task object into worker thread. In this part, I read a lot of materials and put lots of tries, making my worker thread accept a task while its self still in waiting state, making my worker thread being notified in thread pool class, retrieving abstract object and make it convert to an object that implements it, correctly making method or block synchronized in order to handle thread-safety problem. I would say that instead of directly using java uilt.concurrent package to implement our assignment, designing a local mechanism implementing thread pool theory is really beneficial for me to get deep understanding of threads and objects.

Q2. If you had an opportunity to redesign your implementation, how would you go about doing this and why? [300-350 words]

ANS: There are some weaknesses of my program for this assignment, although it reaches the request. First of all, my server side could only hold up to 350 active clients even when there are 350+ clients trying to connect it. Second, my server side could only handle up to 1250 messages per second, which is lower than expected. Lastly, I think my client side could have a better structure design like implementing IO instead of using NIO. So, if I have an opportunity to redesign my implementation, I think I will put much more emphyasis on those parts. For the first problem, I still do not have a deep insight on what causes this problem. For the last few

days, I tried to increasingly add connection to server instead of big rush but it ended up with the same problem, I also tried to make each connection has fewer messages sending to the server but it did not work as well. To draw a conclusion from what I have got, I think this problem may have something to do with the way I implement NIO mechanism that limit the number of active clients. For further improvement, I would refer some related solutions and try to redesign my implementation on server side. For the second problem, I also discussed with my friends and they got the same issue, we think this may be caused by the network bandwidth and also with concurrency delay. Considering that, I would address on solving the concurrency delay which is more doable for me. Finally, the client side design could also be improved by replacing NIO with IO mechanism, because each client only communicates with one server in this assignment. However, considering the real world case, clients could not only talk with server but also talk with other clients, so using NIO is indeed a way to do that. For this problem, I think I will address the issue under certain circumstances, if clients only need to connect with server, replacing NIO with IO is more effective; if clients also want to talk with each other, NIO is a way to handle this by using just one thread.

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? [300-350 words]

ANS: I think I did not do well in coping with the increasing number of clients. For my program, as I talked above, I could only holds up to 350 clients and deal with 1250 messages per second (under that, my program works fine). When the clients goes up to more than 350, my server could not handle this connection, however, according to my test results, my server neither abort this connect, instead when I close some clients, those unconnected clients become connected. That basically means, in my program, my server could connect with up to 350 active clients, which works like a pool, when some clients exit the pool by closing connection, other clients who already have connection requests come in. I do not know what wrong with this issue in my program. But for the potential reasons I think it could be due to the way I implemented NIO, or in NIO a Selector could only hold a certain number of registration requests (which could be tested). Also, when clients side sending up to 1250 messages per second to the server side, the throughput of my server fluctuates greatly from 1250 to 2000 messages per second (assuming more than 2000+ messages per second need to be handled). To solve this, I used visualVM to track the threads usages and threads states. From my results, when messages go really high (more than 1250 in my program), my worker threads are awaked in a sudden by my server while all worker threads are not fully loaded yet (usages are always around 15%), that basically means my server thread could not function smoothly with high load and it performs from normal to delay periodically, that makes my throughput fluctuates greatly. I discussed with my friends and we got the same issues, so we tried to find the reason behind it. My friend maximum throughput stayed in 1500 messages per second and he told me he has no fluctuation in throughput. So I think my server side may have a poor structure in concurrency

that make it functionality fluctuate greatly. However, we both come to the issue that throughput is limit in a certain number, like 1250 or 1500. We think it may have something to do with the network bandwidth and tested this in lab machine but still could not draw a reasonable conclusion.

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? [300-400 words]

ANS: I think there are multiple ways to deal with that. The solution jump to my mind is using a link-list structure, where stores the key reference and messages received in last 3 seconds, and also a checktime when the key has been out put last time. For example, each time the server will iterate the key set and match the key interests. When a key goes to the writeable state, it will check the checktime first before it sends back the messages to clients, if current time minus checktime is less than 3 seconds, store the messages into the structure, if the current time minus checktime is greatly than 3 seconds, the writer function sends all the messages store in the structure, refreshes message set as well as resets the checktime into current time. This solution basically realize the request, however, since I don't open a new thread to record the time for each client, the sendback time deviation is unavoidable in this solution. For example, when a key just finishes it iteration, the current time is 3 seconds away from last checktime, however, the key must to wait for next iteration to guarantee this state, in this case, deviation always exists. According to what I have understood, this issue can be alleviated. First of all, key set iteration happens in quite a short time cause the worker thread does the task instead of server thread. In this case, we can assume that a key can be iterated twice in a neglectable time compared to second scale. Second, I can set the check time range into $\pm \beta$ milliseconds (this number can be tested out that causes the least influence on the time deviation), which means in that 2 β range period, a certain key will be "ensured to iterated" (β should not be too large and this does not mean a key will be guaranteed to be iterated while it means this time range is the best average range to alleviate the delay). The reason I favor this design is that it will not invoke any more threads monitoring the time period for each client, it takes advantages of NIO and worker thread pool in my assignment to make it much more cost-effective.

Q5. Consider the overlay that you designed in the previous assignment. This 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? [300-400 words]

ANS: From my point of view, given that there are 10,000 active clients and each messaging node (node act as server) would have 100 concurrent connections, I think I will have 100 messaging nodes to hold these active clients. Further, when each client generates a message sending to its corresponding messaging node, a new overlay should be created to ensure that the maximum number of hops is 4. I think a way to handle this is dividing 100 messaging nodes into 10 groups. Within each group, 10 nodes are fully connected (each node connects with other 9 nodes), which means it takes one step goes from 1 node to another node. Now go to the larger scale, 10 groups (viewing one group as a node) also fully connected, which means it takes one step goes from one group to another group. After this kind of overlay set up, each node, takes at most 3 steps to another node (one step to larger group, one step from group to group, one step to destination node in that group). After the whole stage set up. The process goes like below: 100 concurrent clients connect to one node (server), where 10 threads available to handle all the requests. 100 nodes form the network, where we have 10 nodes as a group and 10 groups. Within each group, nodes are fully connected, within 10 groups, each group are fully connected (choose one node in group represents that group). When a client sends message to the node, the node configures the route that message should travel through. Node sends the message through NIO mechanism, acting as a message coming into another node (server), where that node invokes a thread to deal with the message. The node then decides whether the message arrives its destination or it needs to be transported into another node further. The whole process is well-designed that each node (server) are way below its maximum capacity (we tested in this assignment is 1250 messages/s) which ensures it runs correctly. Also, according to the knowledge from previous assignment, the overlay design ensures that there is no partition and every two nodes are available within 3 steps. This kind of design is pretty cool that combines the primary key points within assignment1 and assignment2.

Thanks for reading.