Keegan Millard

Dr. Shrideep Pallickara

CS455

March 15, 2019

HW2-WC

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

The biggest challenge I faced in this assignment was designing the ThreadPoolManager class. I decided early on that I wanted to go with a design where the manager is a passive data-structure that the server and worker threads interact with as opposed to a dedicated thread that delegates batches to workers. The way I implemented this choice of design resulted in complex methods with many branches.

My ThreadPoolManager implementation has two linked lists for a worker queue and a overflow batch list. The manager has two important methods: addTask and makeAvailable. “addTask” returns void and is called by server and worker threads with a object that implements a Task interface. “makeAvailable” returns a batch and is called by worker threads when they are created or complete their batches. Diagrams of addTask and a consolidated method called aquireBatch are included at the end of this document.

I implemented the logic for a worker acquiring a batch into three parts over the ThreadPoolManager and Worker classes. Specifically, the worker calls its own waitForBatch method that then sets it batch with the return value from calling makeAvailable in the ThreadPoolManager. It waits for its batch to be not null and then waits for batch time or full batch. If its batch is not full it then calls the removeWorker method in ThreadPoolManager to remove itself. It was challenging to wrap my head around the possible states the system could be in with the design split up as such.

An additional challenge in creating the assignment was working around the requirements of using the thread pool to carry out registration of clients because it presented a scenario where the server could be be deadlocked. This would occur when the server called the method select() on its selector if there was only one client and its registration task was in flight. The solution I used for this problem was to use select(timeout) to prevent the deadlock from lasting eternally.

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

Overall, I am quite pleased with my implementation and believe it turned out to be almost exactly I as envisioned while reading through the HW2-PC document.

However I would make a change to consolidate the synchronization logic around a worker acquiring a batch. I would rename the manager's makeAvailable method to aquireBatch and remove the methods removeWorker and waitForBatch methods from ThreadPoolManager and Worker respectively. The new aquireBatch method could use new private helper methods to delegate the logic of certain states of the system. This I believe would increase readability over having one large method with a heavy branch depth.

Another change I would like to make is to my stats collector, because its locking scheme could be simplified. Initially I had a AtomicInteger associated with every SocketChannel and its instance would be passed around wherever it was needed. When I moved the storage of client statistics to the stats collector I kept using AtomicInteger but also put them in a ConcurrentHashMap with SocketChannel keys. My implementation as such could be changed to use Integers because outside access to elements in the map is synchronized, and the stats collector could synchronize on the map and extract the values at each interval.

One flaw in my program I would change is the inability to create more than one Server, ThreadPoolManager, or StatsCollector because they are singletons. I went with this design because it meant references to these objects would not have to be passed around. I am uncertain of how easy it would be to remedy this issue but adding the feature would improve the ability to reuse the code in other projects.

Another area I could improve is the readability of the timed loops in my program. I used more complicated logic than a simple sleep for interval time and it was effective in timing the program but added around 7 lines to each loop. Maybe adding a layer of abstraction by putting the loop body in a method and creating an external timing class would be an improvement.

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]

My program was able to cope as well as other submissions I have seen on piazza. The program requirements was to handle at least 100 clients at a rate of 4 messages per second for a total throughput of 400 msgs/sec. This was easily achievable with only one worker in my thread pool. The limiting factor for performance seemed to be the network connection between the server and clients. During busy computer lab hours the performance of moderate loads (several hundred clients) would fluctuate and connections would be dropped.

During nighttime hours I conducted a successful test of the server with 850 clients at 7 messages per second for a throughput of 6k msgs/sec. I tested settings further, reaching 27.4k msg/sec but at this level the performance would peak and then decline to a rate around 11k. The bandwidth consumed by just passing the byte arrays themselves at the peak was 1.67Gb/sec, so a network limitation is plausible. I believe my program was able to keep up at a high throughput because of a couple traits in the design.

One, the scope of locks in the program were kept to a minimum. The critical sections that could affect performance were: addTask in ThreadPoolManager, addCount in StatsCollector and registerChannel in Server. In each case care was taken to limit the scope of these sections to the minimum and the data structures used in the sections were efficient for their task.

Secondly, the tasks the program had to perform were highly independent, meaning tasks were well suited to being processed by the thread pool. For example only read and write tasks for a client had to be processed serially and hash tasks from the same client could be processed in batches. Tasks from different clients could be performed at the same time which was the cause of the great concurrency my system could achieve.

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]

The easiest way to add the functionality to my design would be to create a separate thread named StatsSenderThread responsible for queuing the messages to clients. For each client I would store an object containing a socket channel, and the last time a message was sent to the client (set to current time on initialization). A registration task would include a call to a StatsSenderThread create method that would add a clientInfo object to its internal collection (and notify the StatsSenderThread object).

The data structure containing client information would be traversed in order of client arrival time modulo the interval time. A workable choice for this data structure would be a linked list, implementable by adding a “next” field to the client info object. It is necessary to manually iterate over the set of client information, or make a copy of the data structure at every interval because of the concurrent modification of the data structure.

The StatsSenderThread would wait while its client info list is empty and then walk the client info list in order. For each client info object, it would calculate clientTime % 3 - currentTime % 3 and wait that amount if the value is greater than 0. If notified during the wait, it would recalculate its wait time. When the time is right, it would pass a new write task containing the info to be written to the ThreadPoolManager.

The problem with a linked list design is that with a large number of clients it would be inefficient to add client information in the sorted order. Using a skip list data structure would alleviate this issue. An alternative implementation could use a synchronized PriorityQueue to store client info. In this scheme the time a client should be sent data would be stored and the queue would prioritize the earliest time. When a client is serviced, its entry in the queue would be replaced with its last service time + the interval time. This would be easier to implement than the list design.

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?
[300-400 words]

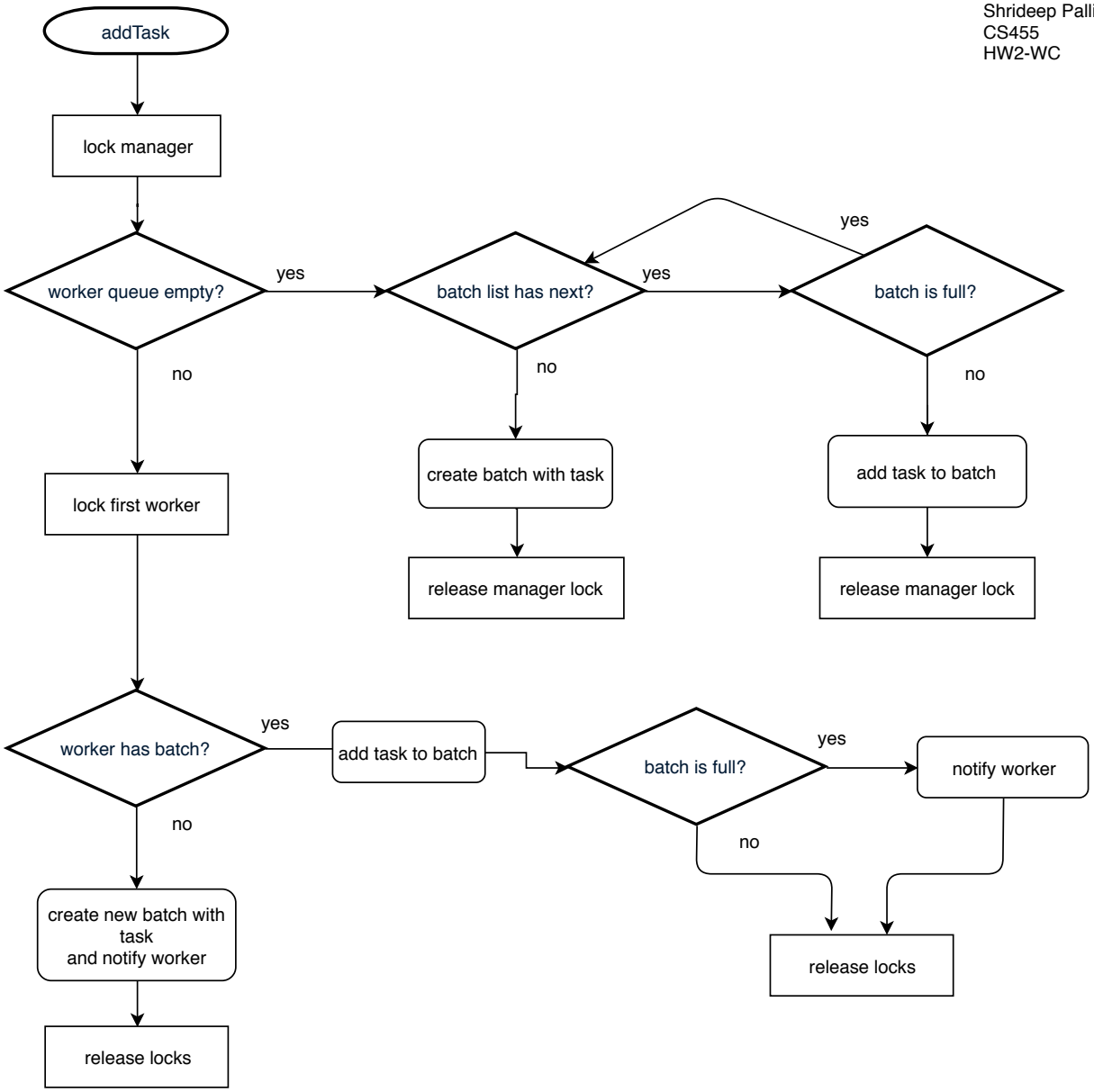
First, the experience I had completing the second assignment leads me to believe that a single messaging node could comfortably support 100 concurrent client connections. This means 100 messaging nodes are needed in total. If each messaging node was connected to each other the total connections per node would be 200, which is within what I think is possible. This means the network topology could be a fully connected graph and packets would not need to make more than one hop. Packet routing plans and route finding would not be necessary when all nodes are connected to each other.

Unfortunately is is hard to say exactly how I would configure the nodes without more information about the requirements of this new project. However, the thread pool and nio server design from the second assignment proved to be very flexible and I think could be easily adapted for a new task.

If clients could connect to any node arbitrarily then it may be important to include a method of load balancing in the system. If a messaging node was nearing its limit of client connections, it could send out requests to several other messaging nodes asking their load level. Excess clients could then be passed off to the node with the lowest load level.

A single Registry node design would probably still work fine with 100 messaging nodes. If the load on the overlay were to fluctuate significantly over time, it may be beneficial to not keep 100 nodes active at all times. Messaging nodes could periodically or upon certain thresholds, report their load to the registry, and the registry could decide a certain load is optimal. Then the registry could give directions to consolidate clients to a set of nodes and shut down others. When load is rising, new nodes could be spooled up to meet dema

Keegan Millard
Shrideep Pallickara
CS455
HW2-WC



Keegan Millard
Shrideep Pallickara
CS455
HW2-WC