Zach Aubin

CS455, Dist. Systems

Dr. Shrideep Pallickara

#Q1. Big Challenge

For this assignment the biggest challenge was working with a top-level down design approach. In HW1 I built every little link from the bottom up, often redesigning each component of each link over and over again. By now I have enough familiarity with Java and socket programming to feel comfortable partitioning a set of unknowns into reasonable black boxes. This led to a lot of time spent pondering over who will do what while leaving the how to be secondary. I outlined class structures and then stepped back to see if it made sense, or if I could implement the classes at this level. Which are Runnable? I also touched back to the instructions much more frequently to prevent me wasting time learning about things that don't matter. I did this class-design, walk away, redesign a few times over. After a while I started to get worried that I had no concrete progress, but by the time I got going with filling in these boxes it became apparent that I could make up a check-point at any time. In my final iteration of class design I decided that each package would be a layer of "doing something contained as thus" and over-used class generation as a to-do list. For example my server package had a class called "printer" with a comment in it that said server needs to print stats. Later I made the printer just part of the runnable stats tracker class and didn't remove the empty class from the server package until I was cleaning things out for production and submission. Overall it seems important to know what the top-level architecture is, what each package should basically do, what I want each class to do, and then clearly define this in the comments. The total of all class-intention comments should cover the package-intention completely. I've never done this before so it was a pretty big challenge. HW1 showed me that there's never really an end to the things you can try, and if your printf isn't spitting out literal 1's and 0's then you're not debugging hard enough. Avoiding having to do that in HW2 was a decent accomplishment.

#Q2. Redesign

I'd make my classes smaller. I was just about ready to do this as I was cleaning up everything, but I learned last time that changing a working product is a very bad idea. If we had until the end of the month I would do this and secretly build a gui for node deployment. For example, the client nodes have a sender thread, a receiver thread, and a stat tracker thread. The sender and receiver threads live entirely in main(), when they could be their own classes. I'd need to carry objects between things, make them volatile or synchronize their access as necessary, and ensure each variable is being used appropriately. I may employ the use of Singletons in some way. The hashing method is all over the place in my current project. I would change this to a literal hasher class, like I had started out with, to generate/store/return hashes from 8kb msgs. This means I would need to set up how to create and access this object in each of its different forms. I would probably achieve this by taking time aside to make it work in-line as it is now, then split out all classes so that they access the hashing method in some way. Finally I would comment the hasher method out of the working product and replace each method call with new Hasher() etc, so that I could test this item in isolation. At this time I am left to wonder if there exists such a thing as over-modularization, and what constitutes a project size that would make this level of architectural design less work overall. If I were to put each method in a class, would I really be saving work now? Were there more than one server, and had our thread pool been asked to handle a variety of tasks as with HW1's messages, it would make sense to spread things out as such. For the scope of it as it is now, most as is is fine though I would clean up hasher and squeeze my classes down just a smidge. Maybe, just maybe I would make each client run a small thread pool with batch-size = message rate.

#Q3. Cope with Scale, Config@100

We need enough threads to handle this and a decent size for each batch, just leave batch-time at 1 sec as long as nodes are spun up quickly enough. I liked putting max-threads at 8 and batch-size at 8 with a timeout of 1, it saw the least number of bad-hashes being passed. However, I didn't increase past this because it seemed to work just fine. I've heard two threads per core in the CPU is ideal, but I have a stats-tracker thread running as well. I probably could have found a way to identify the number of cores in the current machine's CPU and set it to 2x that if not greater than the max-threads argument. How well did it cope with scaling? Not well at first. I added some synchronization blocks, synchronized access to some places, shifted some things to volatile, modified what exactly my thread pool was doing, and added a counter for bad-messages which showed me that sometimes two hashes come through and there's nothing you can do about it except set up something to parse out each one. Then it worked so that when scaled such that x nodes at y rate yields z server throughput for z=200/s. Synchronizing access to the queue (etc) and setting up nodes to deal with reading multiple messages at once appeared to be the primary issues. I also had to make sure that my queue.poll(timeout) worked as opposed to setting queue.take() inside a timeout block. This involved synchronizing access to the unfinished batch of tasks to be done. All of this had to be done to maintain throughput levels. Without threading (or a task queue) I could handle about 60msgs/sec for some time, but it got gummed up after a while. Then the thread pool allowed me to handle a greater number of msgs/sec, but at the cost of accurate hash matching. Synchronization was the issue, just making sure the right thing is done at the right time by the right doer to the right target.

#Q4. Server: Total uSent @tJoin+3

We plan to have the server report total number of messages received from some certain node every three seconds, starting when the first message is received from that node. We begin by modifying each client node message to send an ID as the message header, in front of the 8kb noise. Server-side we set our "final AtomicLong start_time" whenever that message is received. Since we still only have a few (<1000) nodes, I'd make an empty ArrayList and have each nodeID correspond to the index for that thing's counter. If there is an atomic version of ArrayList then I'd go with that since it's more reliable to use the pre-defined tools. Otherwise, maybe a bunch of AtomicLongs which are named with ID + "_client_counter". Anyway, server would keep track of msgs tx/rx based on ID sent in header of message. It would also keep track of ID + "_timer_start" when the first message is received. Now begs the question, the question we must ask, ask unto ourselves, would this scale? It would be interesting to see how much memory this naive implementation takes up. I would be rather inclined to *not* separate things out into different classes, such that there is an instance of a class which controls for statistic trackers for different client nodes. If I could just jam them all together into one tracker I think I'd use less resources and be in much better shape with regard to the question of scale. However, at what point does this become a mess? It would then become interesting to create both implementations, probably have to figure out what a git branch is before that, and see how the throughput and resource utilization compare between the two. That may, however, be an exercise in futility if there is a better way to understand these things, but I know from spending too much time on HW1 that this would answer a lot of other questions I may never bother to ask.

#Q5. HW(1+2)@10k

First we will need 10k = X * 10threads*(10cnxn/thread) >> nodes, or estimated X=100 messaging nodes with 100 connections each. We set these mNodes up in a ring structure. Hop links are defined with one layer, perhaps two. The outer layer is defined as [id]+2^k, so message goes to next-nearest-w/

o-going-over. This tops out at k=10 and in quick calculations looks like no more than four hops. We have dedicated connections in the overlay from nodeZ to it's 10 in the links as described. Each client node would be registered with a server node until that server node is full, so 90 client nodes will have one messagingServer node. This modifies our original estimate from 100 to 112 as 10k/90=111.11. Thus each client node has a dedicated entry point to the overlay and each messaging node has a dedicated connection to 10-ish other messaging nodes. Dynamic assignment would be handled server-side during Phase 1 of establishing the overlay (pre-setup). We figure out nodeIDx >> messagingServerNodeIDy and ensure no faults or drops, then messagingServerNodeIDy >> messagingServerNodeIDy[1,...,10] (establish overlay).

We now have: 112 messagingServerNodes each with 90 client connections and 10 overlay connections using the same topology as HW1 (dist = 2^k), and 10,000 client nodes whose messagingServerNode entry-point is dynamically assigned pre-overlay-setup. Once established, all connections remain dedicated (client to server, server to server). I wonder what modifications may be required for the thread pool, beyond bringing it to agnosticism - right now it just reads/hashes and responds. The more I think about this the more I think HW2.threadPool would utilize HW1.eventFactory.

Since there are some things to articulate, many in fact as time appears to move by or as we appear to move through it, we may take a moment now to manifest some such ethereal essences from the ineffable by way of this pre-machinistic symbology that is functionally one of the most boring of delayed telepathies. My thoughts as these words are now in your mind. And yet nothing has been said.