Max Hayne
HW1-PC
CS455
2.21.20

Q1. I encountered many challenges during the assignment, but the largest one was in dealing with the producer-consumer problem between nodes. After I had finished coding enough classes to begin testing the message passing capabilities of some overlays, I ran into the problem of lockup. When testing overlay networks that only contained one routing entry in each nodes' routing table (every node could only send to the one next to it in ID space), every node would send between 20,000 and 50,000 messages, and then, inexplicably, all traffic would cease. This led me to believe that there was a problem in how locks were being acquired by my messaging nodes during lookup-and-send. And, because my RoutingTable and TCPConnectionsCache utilized vectors to store their entries, I went down the rabbit hole of trying to figure out if having to lock down the entire operation of iterating over a list, and doing something for a particular item in that list, was causing me issues in lock acquisition (this was the exact problem we discussed during the most recent lecture). So, I spent a fair bit of time combing over the 'send', 'relay', and 'receive' invocation cascades to that occurred when a new message arrived. I changed around the synchronizations of a few methods, but it made no difference. It wasn't until I decided to sit down and draw out an example overlay network that I cornered the issue. The crux of the problem was that I was using a single thread to receive and relay new messages on every node. Therefore, if a new message arrived and it needed to be relayed, I would iterate over the TCPConnectionsCache to the node that the message needed to be passed along to, and attempt to send it out of the associated socket. But a send attempt blocks if the receiver's buffer is full, and if the receiver is also waiting to send to a different node whose buffer is full, we get full-blown deadlock. This is when I realized that sending and receiving needed their own independent threads so that nodes could wait to send without preventing themselves from receiving new messages. I solved this issue by adding a thread and a blocking queue to every TCPConnection in my TCPConnectionsCache, and simply adding messages that needed to be relayed to the queue before returning to listen for more incoming messages.

Q2. If I were able to redesign my implementation, I would make a whole host of changes, mostly because I believe that the design choices I made early on held me captive to doing things a particular way in the later stages of the design cycle. In order to achieve the same functionality while only using two synchronized blocks, I'd use one synchronized block for adding entries to the TCPConnectionsCache, and another synchronized block for iterating over that TCPConnectionsCache to create RoutingTables (and to set the status of my network to 'running'). This way, I would prevent overlay nodes from registering while the RoutingTables are being constructed and sent out to registered overlay nodes. I think that all other tasks, ranging from message sending and receiving to incrementing counters, are either read-only operations or

are guaranteed to be accessed by only one thread at a time. For example, the sending of messages from one node to another is done only by one thread, and the queue for the sender is a BlockingQueue which is thread-safe. Atomic variables can be used for the counters, thread-safe hash tables can be used for routing. All parsing of new messages is done by a single thread, using newly created objects which are guaranteed not to be accessed concurrently. At the overlay nodes, no node will check its own routing table while it is in the process of building a new one that has been provided by the registry. So much of what is going on behind the scenes that might be affected by concurrent access has to do with input provided by the user, and at the MessagingNodes, all that is being done is printing out atomic variables, and at the Registry, the only thing that will be modifying important information will by the building of the routing tables, which I've already synchronized.

Q3. Assuming, for this question, that all songs share an identical probability of being streamed, a situation in which one node is getting hammered with streaming requests while other nodes are sitting idle is simply the consequence of bad luck, as, on average, the load will be dispersed more evenly. One strategy for dealing with these less-probable circumstances is to duplicate copies of the same song onto a set of different machines, using some sort of *replication factor*. For instance, a replication factor of three would permit the storage of a song on not one, but three machines, each equally capable of streaming that song to a requesting client. Introducing a replication factor would solve a couple of issues. Firstly, it would decrease the likelihood that a single machine is serving the majority of content in the network. In a worst-case-scenario, where the only songs being served are stored on one machine, using a replication factor would allow that load to instead be equally balanced across as many nodes as the replication factor provides. Expanding this logic to a more likely scenario, where N/3 nodes are serving content, but the nodes serving content are distributed evenly throughout the entire network, introducing a replication factor of three would make it more likely that all N nodes are serving that same content. Secondly, it would allow for more fine-tuned control over the streaming load on machines that hold the same replicated content. If set up efficiently, nodes could actively track the number of clients they are serving using an atomic integer and compare that number to some threshold set by the network manager (or dynamically set based on the capabilities of the machine). If a machine handling a new request observes that it is over its streaming threshold, it could forward that request to another node that replicates the requested song, with appended to the forwarded request its own node identifier and streaming count. When the next node with the replicated content receives this forwarded request, it could check its own streaming count, and either accept the streaming request or pass it along. If the message request has traversed all machines with the replicated content, the final node in the traversal will take on the computational overhead of selecting the least overwhelmed node from the list (that has been appended to the forwarded request) to send the final request back to. Of course, the request could also be dropped if the load on the network is simply too great.

Q4. In order to account for a new, more powerful machine in the network, one solution would be to skew the hashing function in a way such that it is no longer uniform, but instead peaks in probability with the node identifier of that more powerful machine (a logically equivalent way to do this is to leave the hashing function unchanged, but allocate more than an evenly-cut share of the hashes to the more powerful machine). In this particular case, the probability that a song hashes to that machine should be 16x that of the probability that a song hashes to another, less powerful machine. This could be done in one of two ways. The first way would be to shut down the entire streaming service for a set of time in order to perform a complete rehash of all song titles, and to reposition the storage of those songs onto the machine in which their title hashes. Not only would this action be very time intensive and costly to the streaming service, but it wouldn't scale well with a node cluster that is constantly increasing in size and performance. Therefore, a different strategy would be preferred. The second way this could be done would be to (assuming that each node stores songs that hash to a range of different values, e.g. if the network has 16 nodes, and song titles hash to the range 0>=hash>160, node one will store songs that hash to values 0-10) silently populate your new powerful 17th node with half of all of the hashes that the other nodes store, and at a synchronized moment, switch to a new hashing table with updated routing values. Old nodes would then delete their storage for songs that hash to values they no longer serve. For example, extending the example given in parenthesis earlier, the new powerful 17th node would take on hash values 5-10 for the first node, 15-20 for the second node and so on, such that the number of songs that it stores and serves is proportional to its performance relative to the other nodes in the system. Now, out of 17 nodes, the most powerful machine will store and serve half of all possible hash values (directly proportional to its share of the computational horsepower in the network), while each of the other nodes would store and serve their fraction of the remaining hashes.

Q5. It's never just that one song is more popular than the rest, it's that the most popular 10% of songs hold 90% of the streaming traffic for the network (or some other wildly skewed proportion). Under these circumstances, the DHT should actually do a pretty good job at balancing the load, simply because the hashes of the most popular songs are still random, and thus will be distributed evenly between the nodes in the system. Adding a replication factor (as I discussed in my response to Q2) will help to reduce further the likelihood of one node serving all requests for a single, popular track. However, if the number of songs being served is relatively small, and there is in fact only one song (or a few) that are more popular than the rest by a factor of 3x, this strategy will not work well enough. In this particular case, it may be necessary to create a separate hash table that only contains the most popular tracks in the network, and place one on every node in the system. That way, regardless of where the request is coming from, the node serving that request will be capable of streaming that popular song. It could even be set up such that if a request comes in for a song, the hash table for popular songs is first checked, and if the song being requested is in the hash table, that request is forwarded to a random node in the system. This would solve the issue of a particular node receiving the majority of song requests,

which could be the case for any number of unforeseen reasons. Adding a list containing only popular songs would increase the search overhead though, as two hash tables would have to be queried for every request of an unpopular song. Additionally, some method for determining the most popular songs in the network would have to be constructed, perhaps by pairing every song with a ‘stream-count’ (which decays over time) and is collated and analyzed by a master node at the end of each day.