Stephen Porsche

Q1) The biggest challenge for this assignment was deciding how to send the packets in a clockwise manner. Initially it seemed like an easy task, just do a series of checks to see how far away the sink was, and send to the farthest away node in the routing table without skipping past the sink. However in my first couple of implementations, I found that specifically for sink nodes that were 3 nodes away, the sending node would route the packet counter-clockwise.

My first implementation I tried to implement a routing algorithm based off of nodeID, but I quickly realized that this was flawed, because the node ID doesn't matter, the position of the node in the array was the important key. I wrote a method to grab both the source index, and the sink index from the array of nodes in the overlay. I decided to do some simple math of sink index - source index, and then implement my checks. This worked well when the sink index was greater than the source index. However problems arose when the source index was greater than the sink index. In other words, when I needed to wrap around. In those cases, the packet would either get routed left, or would get routed to a node that was past the sink node. I solved this problem by finding the length of the array, makin it negative, and adding numbers a number corresponding to a hop away (e.g. -(arrays.length + 1 as a check for a node 1 hop away).

This unfortunately does not make itself readily available to scalability. If I wanted to go back and add bigger table sizes, I would need to add extra cases. It would be better for scalability, to add these checks into a loop, that checks for the farthest away possible hop, and if it would go past the sink node, grab the next farthest hop away and repeat the process until I have found a hop that either gets to the sink node, or gets as close as possible. [WC344]

Q2) My synchronized blocks were found on my sendMessage queue, any time I would increment a counter for a packet sent or received, and when threads sent the totals to the Registry to be printed for the total sums. If I were to rewrite this to only use 2 synchronized blocks, I would try a couple different implementations.

First, instead of having a synchronized sendMessage method, I would try to implement a BlockingQueue on both the sender and the receiver, which works in conjunction with Runnable. BlockingQueues are thread safe, performing their functions anatomically with internal locks. {2] If necessary, I would also implement a wait, notify system like in the producer consumer problem, though I do not believe wait() and notify() are necessary for a BlockingQueue. Every message would be sent to the send BlockingQueue. Once the send BlockingQueue has a message to send, it would send the message to the receiver BlockingQueue, which would process the message, and then look for the next message.

The second change that I would make would be to declare my counters (int packetsSent, long sumPacketValsReceived etc) as private final atomicInteger count, or private final atomicLong. Declaring these variables as atomic would allow my to use count.incrementAndGet(); to keep count of the packets sent, and keep the variables thread safe without using synchronized.

Implementing these two changes would leave me with 2 synchronized blocks both in the Registry class. The first for ensuring that all of the MessagingNodes have finished sending their

messages. The second for receiving each MessagingNodes statistics from calling Registry Requests Traffic Summary. In both of these cases, I found the synchronized blocks necessary. In the first case, I would never receive notification from a couple Nodes that they had finished sending without the synchronization. In the second case, my totals were incomplete and different before creating the synchronized block. [WC311]

Q3) I would attempt to maintain an even distribution of songs and bands over the DHT by using a caching algorithm to try to handle skewed requests to the most popular songs/bands. We assume that to start out, the songs and bands are evenly distributed among all of the nodes. However, we cannot assume that the requests for songs and bands will come along in an even manner. Some songs and bands will be more highly requested than others, which means the some nodes would be more heavily queried than others, which will unbalance the load, and bottleneck the entire DHT.

To attempt to solve this problem, I would implement a caching strategy. Caching places copies of a resource on several nodes. [1] To effectively implement this, I would have to determine which song/bands are being requested at a higher rate than average, and change or add a few specifications. First I would need to place the copies on nodes that are on paths to the original. This would give me a high probability that a copy would be hit instead of the original, which helps the other nodes share the load. Second, the more a song or band is requested, the more copies should be made and placed. There is no limit the number of copies that can be be requested, but there should be a maximum number of total copies that can be maintained at any given time. This leads us the the third specification: make sure that space is maintained. Once the maximum number of copies is reached, whenever a new cache copy request arrives, a node should decide which item should be removed to make room for the new request. By adding copies to the DHT in high traffic areas, the load should be able to be kept in a more balanced state, and help ensure nodes are not idling, while a few receive a majority of the traffic. [WC324]

Q4) In order to make use of my shiny new, much more powerful node, I would make use of workload partitioning to make sure that the new node is not being underutilized. Network partitioning would allow me to partition or split the load onto the servers in a manner that is proportional to each servers ability. Theoretically, since my new node is 16 times more powerful, I should be able to put 16 times as much content onto the new node as compared to the other nodes. Assuming an even hashing algorithm and popularity among content, this would keep the bigger node proportionally busy compared to all of the rest of the nodes.

Of course we know that not all bands and songs are queried equally. In addition to partitioning, a majority of the requested copies could be placed on the node in an attempt to help lighten the load on other nodes. However this may not be consistently effective, because if the more powerful node is not on a high traffic path to a popular song, the copies on the new node will not be accessed very often which would lead to underutilization.

Another idea to make use of my fancy new powerful machine would be to move the most popular bands and songs onto said powerful machine. Or at least move popular content, until the node is being contacted 16 times more than the other nodes. This would cause the most

powerful node to bear the brunt of the content and queries, while ensuring that it is constantly utilized. If it becomes overloaded, then copies of the most popular songs can then be distributed among the rest of the DHT.

Workload partitioning, placing copies, moving popular content, or a combination of all three should be an effective way of making sure that the new popular node is not underutilized. [WC311]

Q5) To deal with the corner case of having one song that is three times as the average song, I would implement the caching strategy of adding copies of the song of a proportional amount around the DHT, specifically in high traffic paths leading to the node with the original instance of the song.

Assuming that the routing algorithm is similar to the one used in HW1-PC, I would begin to place copies at nodes of nodes that are 2^0, 2^1, 2^2 hops away from the node that holds the popular song. In other words, any node that may have a direct connection to the node, or the three nodes most likely to direct traffic to the trendy node. This would make it more likely that one of the nodes a hop or 2 in front of the node with the famous song would see the request and play the song, lessening the burden on the overloaded node.

I would then keep a monitor on the song, and performance of the DHT to see if adding 3 copies of the song was enough. If performance has improved or returned to normal levels I would assume the copies had done their job. If performance has not improved, I would add another three copies elsewhere in the ring. If the song were to continue to become more popular, I would find the nodes 1 or 2 hops away from nodes 1 or 2 hops away from the original, to once again spread out more copies and thus increase the likelihood that a less busy node would take the request and even out the load.

This method of adding three copies onto three other nodes should ensure that the load remains balanced, and that the node holding the original copy of the song does not become overwhelmed because of one popular song, and thus bottleneck the rest of the music streaming platform. [WC319]

Works Cited

[1] http://www.dei.unipd.it/~schimdmi/pubs/master_thesis.pdf
[2] https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html