Amanda Carbonari
Dr. Shrideep Pallickara
CS 455
2/19/2015

Homework 1 – Written Component

1. The biggest challenge I encountered in this assignment was determining how the connections would be stored at the ServerThread, such that the Registry could efficiently and accurately pull the connections made in the server thread and match them to a registering node. The main issue occurs because the registering nodes must send their server port number to the registry. This is required because the registry sends out that port number in the node manifests so the other nodes can establish connections. If the node sends the port that talks to the registry, then the registry does not have anything to send over the node manifest that can be used for the other nodes to connect to the original node. Therefore the primary issue of storage and retrieval of the connections lies in the case where there are multiple nodes from one IP address attempting to register. Since the server port would differ from the port the connection talked to, it is impossible for the registry to determine the exact connection that it should pull. It only knows the IP address. I don't particularly like my solution to the issue, but I could not think of a better one that didn't involve sending extra information over the wire format. I decided that a connection is arbitrary until given a node ID, therefore, the first node from a specific IP address will get the first connection in the list of connections for that IP address. If no more connections exist, obviously some node is attempting to connect under a different IP address, since no more connections exist and I have ensured when I assign a connection to a node they have the same IP address, then the current node attempting to register is doing so under a false IP. It seems more like a "bandage" solution to me, therefore if I were to improve my program outside of the constraints of the assignment, I would add a second value to the registration request wireformat, it would be the server port of the node attempting to register.

2. Considering I wrote the code without a good knowledge of thread safety, there is a lot I would need to do in order to ensure I only have two synchronized blocks in my code. To start off I would probably consider the abstract design, what each class/object needs with regards to global variables and methods (private and public). From there I can determine if I have too many global variables in one particular class. Once I narrow down the global variables, I can then narrow down the number of synchronized I need around those variables. From there I would then turn to the methods. I would develop psuedo code algorithms for each method to determine whether or not they change state variables or need up to date state variables. Once I determine that, I can redesign some of the code to ensure the smallest amount of methods truly need to change state variables or read state variables. I can then see which methods (or parts of methods) need to be synchronized and which variables need to be volatile. Once I make those changes and ensure the program is still working correctly, I can then repeat the process. I would take the improved design and tear it apart to see if I can consolidate the global variables and synchronized methods even more. I challenge I see arising with my current design is the use of the Statistics Collector and Display, currently I use that object to store the information which means it has mutable state that multiple threads would attempt to change. Since I would be restricted in the number of synchronized blocks, I would then need to figure out a way to not store the information, but display it. I could only store the summation variables and print out each statistic as it comes in. This would still have some global variables, but it would reduce the number of synchronized blocks I would need. In summary, my method would be an iterative

analysis of the requirements of the classes to see if I can shuffle duties or global variables to consolidate their locations so only the minimal number of synchronized blocks are needed.

3. To ensure the streaming load for songs is dispersed, the songs themselves must be equally dispersed among the nodes. One way to achieve this is to ensure that the next most popular song is not on the same node as the current popular song, basically we wish to ensure that the popular songs are not all on one node. In order to do that, there must be a way to a) know the popularity of the song before building the hash table b) modify the hash function such that it takes into account the popularity of the song without dumping all the popular songs in one bucket. A simple algorithm to do that is to have a popularity index for the songs 1 – 100 where 1 is the least popular and 100 is the most popular (it is the percentage of people who like the song). Then in descending order of popularity, each song is placed on a node going clockwise. Of course this simple implementation will not work in the sense of a DHT because it does not utilize a hash function. But the idea remains the same, there must be an even distribution of popular songs per node. One hash function that could work, depending on the implementation, could take the artist's name, title of the song, and the popularity index as input to produce a value. This function would need to weigh the popularity index most heavily in the function and would need to involve modulo with the number of nodes in the system. Although I do not have an algorithm explicitly written, it seems as though the best option to ensure even streaming is to ensure that the hash function will evenly distribute the songs based on popularity. That would guarantee that one node does not have all the popular songs and is constantly streaming while another node has all the unpopular songs and sits idle.

4. A distributed system is only as strong as it's weakest link. How I would handle this node depends on what is meant by powerful, does it have a faster connection, better processing power, etc.? If the connection is faster, there isn't much to be done because the other machines would not be able to handle the fast rate of messages the new better machine could produce. One possible option is to put 16x more songs on the new machine, including popular ones. There the machine will be servicing requests within its computational power but it will have a less chance of needing to send the majority of messages off to another node, so it won't clog the network. It will be used to it's full potential but also won't cause the other nodes in the network to be underutilized because the DHT will still have an even distribution of popular songs, the new computer will just contain more than the other computers. Another option to cope with the new machine is to consider it sixteen nodes on one machine rather than one big powerful node. This would maintain the even distribution of songs by popularity and it would also utilize the computer's power. This would occur only if the original set up utilized each node evenly. Then by adding one computer, you would just add x more nodes depending on how much better it is. Since the nodes are already guaranteed to have even utilization, then the new computer will be used x more times than the other nodes. This design would accommodate any new and more powerful computer being added to the system, although it does not really address the need to ensure that the new computer does not clog the network. It does not address this issue because each node in that new machine will have it's own send and receive, whereas if it was just one massive node on the new machine, it would have one send and receive which could slow the traffic on the network enough to not bombard the slower nodes.

5. If a song is three times more popular than the average song, it would be placed with the most unpopular songs. This would ensure that if there is a lot of traffic to that one song, it would go to a node that doesn't also have decently popular songs. In order to implement this, the hash function would have to slightly change to effectively give a similar hash value to extremely

popular songs and extremely unpopular songs. In theory this would maintain the distribution of popularity among nodes. One way to determine this is to assign popularity values where 0 is neutral, anything negative is unpopular and anything positive is popular. Then each node's popularity values should add up close to 0. If this is implemented in the hash function, then the song that is 3x more popular than the average song would only get places with very unpopular songs to even out the amount of requests on the node. Effectively the algorithm would attempt to match up a song with a popularity index of x with a song that has a popularity index similar to -x. This could cause problems with songs close to the neutral index, they would all be clustered in one node and that node could see increased traffic because of it, but that also depends on how the programmer defines the neutral index, it could be considered the average of the popularity percentage or be determined by some other factor.