Christopher Mersman
ASG2 Written Component

1) Two of the bigger challenges that I had encountered during this project included: how I wanted to make my class cohesion as well as how I decided to send data over the wire. At first, I didn't know how I wanted to make my connection class. Since NIO was new to me, I felt that I should start off with a class that just held my connection and make it so that I can read and write only from that connection. That way I would only need to implement the read and write portions of NIO once. However, the other problem was being able to use UDP and TCP. I then had to move to inheritance and make some abstract classes. I ended up creating a factory class called ConnectionFactory that would give me the appropriate connections based on the type of protocol given. From there I had TCPConnection and UDPConnection classes that extend to the abstract class Connection. That way I could just create a Connection from the ConnectionFactory and apply the different protocol based functions with the same names to the objects without knowing which Connection it was.

When it came to sending the data over the wire, I had some mixed thoughts on what I wanted to do. I considered sending chunks of the data over the wire as well as sending my entire message at once. Sending over sections of data, I've learned, can be a little more prone to mishap. If I lose a message while trying to parse the incoming data, then I'll have a big problem because I will lose important information. I decided to set it up so that every message has all the important information needed to complete it. Every message has a header number that is followed by a semicolon. This way I can get the bits, turn it into a String, scan the first part using a delimiter of a semicolon to get the message flag, and then use the scanner to get the rest. There were other methods I considered that got the information by delimiting with different characters, but it seemed to me that Scanner was the most reliable method as well as more familiar. I started by using an explode function that would split everything on semicolons but that only brought my code more problems.

At that point, the question of if I wanted my connections to be saved or to be reconstructed each time came up. I quickly figured out that I wanted to make it so that my connections were always rebuilt. I felt that saving all the connections could end up being a lot of management that would burn me in the end. It would also hold up a lot more memory. Instead of saving the connections, every time that I needed to communicate with another node I would just build the connection, read or write, and then be on my way. This way I could close the connection and let the rest become garbage collected.

One of the more specific problem I encountered was trying to print the Minimum Spanning Tree like the example showed. I tried doing it iteratively but ran into a lot of problems with debugging. I quickly moved to recursion which seemed to be an appropriate solution. I also experienced some confusion over the way I have my data stored. It was a weird process to come up with a way to accomplish this.

2) There are a couple different ways to make sure that the routing reflects the current state of the links. One way to maintain this is to communicate with the registry while sending data and every time a node receives they tell the registry. This way we can tell the registry to update the

weights after everyone has given the ok. However, this could be a bad implementation for the fact that you need to reflect the updates of the waits for different jobs associated with the nodes elsewhere.

The way I had implemented this feature was to send the routing plan along with my data. In other words, I'll have a list of nodes within my message which tells the next node that receives it, to delete everyone it is connected to on the list. This way you are not sended it to the same nodes again. It also prevents nodes from sending data to other nodes that they are notf connected with. Once there no one is left on the list that you are connected to, then that node will not send anything and this will indicate one of the ends of the minimum spanning tree. However, this wouldn't work unless it started out being the minimum spanning tree. If you tried doing this with an entire network graph, then all the nodes would have to use some distributed memory and look onto the same list to check if certain nodes have been hit yet or not. If they had been hit, they would take themselves off of the list.

One other way I thought of that enables me to do this is to allow each new weight update contain a signature. This signature would go with the message that is passed through in order for it to know which mapping of the CDN to use. For example, the first weight list has a signature of 1111, the second has 2222 and the third is 3333. If you've gotten the third weight update but you receive a message that contains 1111, then you will know which graph to use for the specific message.

3) One way for me to change the latency for the communication would be allow my connections to be saved. That way I would not have to build and rebuild my connections every time I want to communicate with a different node. However, when it comes to changing the algorithm, the idea of using a minimum spanning tree would be very efficient from getting from one node to all nodes the fastest. If the goal was instead to send only a message from one node to another single node, then we would have to change the algorithm to a shortest path algorithm. A good example to use here would be Dijkstra's algorithm. This would be able to generate the shortest path to the receiver node making the jumps. Using this algorithm, you would still have to send the routing plan with the message though which could be a large consumption of space within the message, especially if the amount of nodes is vast. Then, in order to ensure that this is faster process, I would start caching the paths so that message would always have the shortest distance to travel between those certain nodes.

If you want to send a message from A to E and the shortest path includes C and F then you now have A's shortest path for E, C, and F. Saving these will reduce the amount of time having to calculate Dijkstra's Algorithm (especially with the use of Hash Tables) even though it is already in polynomial time. If the amount of nodes in the graph become too large however, it may be better to not cache the path and to just calculate the path; therefore, saving so many caches could take up a lot of memory and having to search through a list of caches could make things a little worse as well. Updating the weights constantly could create a couple more problems farther into the project with this implementation.

This system is dependent on how you are wanting to send the appropriate data. If it does not matter that every node in the system gets the message, then the minimum spanning tree would be a good fit or like I will mention later, send the message through the registry. However, if

you want a quick system for node to node communication, then using Dijkstra's algorithm would be appropriate. It would seem that the Dijkstra's algorithm would be the most common example of using a graph like connections. Although, if we related this to something like the entire internet, there would be a problem because the amount of nodes in that system is enormous. There would have to be so much more computation.

4) I can see that the use of an minimum spanning tree could be costly in a very large scaled system. To make it simpler, there are a couple ways I've thought of without using a minimum spanning tree. The first way to do this is by using distributed memory, as I mentioned before. This would allow you to share the memory that everyone has access to. This would show who has received a message and who hasn't. If a node wants to broadcast a message to the whole audience (the rest of the nodes), then it could start by just creating a list in shared memory that has all the nodes (even though it is not connected to all the nodes). For all the nodes that it is connected to, it removes them from that list in shared memory and sends the broadcast to those nodes. Once the receiving nodes get the message they take it and do the same thing. They take out all the nodes that they are connected to within the list and then send the broadcast to those nodes. Once the list is empty, then all the nodes have received the message.

Another way to do this is to pass along a signature before sending the message that tells each node that a message is coming. A node can then tell if it has seen this signature before, so it can write back saying that it has already gotten the message and does not need it. The only problem with this dissemination process is the amount of IO going through the wire to each node. This would allow for a lot of unnecessary communication between nodes which could turn into wasted time. This would be way more efficient than the previous way.

Another way that I think would work is to go through the registry. If the registry has a connection to every node without any weight then the system could be made to have the registry take the message, know who it came from, and then forward it onto the other nodes. This would be a very efficient way to use due to the fact that it would span all connections rather quickly and would allow the message to be received by each node only once. All you would need to send is a flag for which type of message needs to be used, the signature of that node to the registry, and then the complete message.

5) Once I had created my thread pool, it had worked really well coping with big numbers of connections. However, my linked list eventually grew too much and the amount of runnables that grew crashed the system. I had set this system up so that a new runnable is created every time a client node sends. The runnable ReadOrWriteTask then is created and sent to the thread pool where a worker thread takes the runnable, runs it and sends the next read from a client back on to the pool's queue. I was not one hundred percent sure how we were supposed to implement this client design, but in the end, it worked out. The only problem I had was having variable speeds for the clients. If you had a 120 clients that contributed four messages per second, then the system would work perfectly fine. However, if you had 119 clients that contributed four messages per second and 1 client that contributed two messages per second, the system would have trouble and the two messages per second node wouldn't receive any of the correct data.

Also when running all these threads on different machines, my throughput started experiencing problems. These problems were not necessarily on my messaging node side but on my client side. Each of my clients have two threads so that they can listen and send at the same time and when running ten nodes on the same machine, problems would erupt from all the mixing IO. Some clients would not get the correct responses because they would send and receive at different times.

At the beginning of my implementation of the thread pool, I basically started with having each client having its own runnable so that every worker would deal with a client every time. This was a little bit of a problem because for a different client to be taken care of, the system would need to end a previous client. This isn't considered distributing work. It is more of just queueing up each client and only allowing four or five to run (depending on the size of the thread pool). I ended up having to create my own task, like I mentioned before, and have it use the client to read and write once and then move on to the next client's read or write in the list.