Jack Fitzgerald
2/19/2020
CS455: Introduction to Distributed Systems
Homework 1: Writing Component

Q1. What was the biggest challenge that you encountered in this assignment?

The biggest challenge I encountered during this assignment was how to effectively manage the socket connections and communications between the Registry node and the MessagingNodes. At first, I thought it would be a good idea to create new sockets every time the Registry needed to communicate with one of its registered MessagingNodes. But I realized that this would create a lot of unnecessary overhead, and the sockets only needed to be established once and a reference to that socket could be stored somewhere. However, I was confused as to how to store the socket references in an effective and clean way.

After lab session 1, Dr. Pallickara suggested using a TCPConnectionCache class that could be used to store TCPConnections. I thought this was a great idea because it offered a clean and effective solution for storing and managing socket references, but I was confused with how to retrieve the correct TCPConnection when I needed to send a message from the Registry to a particular MessagingNode. My main source of confusion was how I would initially go about creating a key that could then be used to retrieve the correct TCPConnection when the Registry node needed to send a message to a specific MessagingNode. Due to my confusion, I opted to store the socket references in my registration table along with the other information associated with a registered MessagingNode (server IP address and port number). I don't think this was a necessarily clean way of doing it, but it made the most sense to me at the time.

I revisited this problem and tried to come up with a solution that used a TCPConnectionCache. I think what I could have done was create a TCPConnection object as soon as the Registry's server socket received a new connection. Then I could pass that TCPConnection object to its own TCPReceiverThread and subsequently to the Registry's onEvent() method that could then add the TCPConnection to the TCPConnectionCache when a MessagingNode successfully registers. The key associated with a TCPConnection could be the node ID of the registered MessagingNode.

Q2. If you had an opportunity to redesign your implementation to ensure that you have exactly two synchronized blocks in your entire codebase, how would you go about doing this?

I would do two main things to redesign my implementation in order to use only two synchronized blocks. I would redesign my Registry to use Java NIO instead of IO, and I would try to redesign as many classes as possible to be thread confined so that synchronization doesn't need to be used. The reason why I would use NIO is because I would not need to synchronize my registration table data structure, and it would essentially be thread confined. The reason why it has to be synchronized without NIO is because multiple MessagingNodes could be trying to register at the same time, and multiple threads would be trying to add new MessagingNodes to the registration table as well as checking to make sure there aren't any duplicate entries in the table. With NIO this synchronization wouldn't need to be done and my RegistrationTable class could be redesigned and confined to only a single thread, thus making synchronization unnecessary in my RegistrationTable class.

I also think I over used synchronization in many of my other classes, for example I synchronized many of my methods in my RoutingTable class which I think is unnecessary. My RoutingTable class could easily be confined to a single thread, and it should never change once it is fully created and initialized by the Registry node. Only one thread creates the RoutingTable, even in the Registry and the MessagingNodes, and after that it should never change. The state of the RoutingTable can essentially be final and it can be thread confined, thus making any synchronization unnecessary.

After redesigning the Registry to use NIO, and making as many classes thread confined, then I think there would be only two things that would need to be synchronized. When my MessagingNodes send data to each other over the sockets output stream, since a MessagingNode could be either sending or relaying data to the same destination, that would need to be synchronized. Then when MessagingNodes receive data from each other, the incrementing of the receive/relay trackers and summations would need to be synchronized as well.

Q3. How do you ensure that the streaming load for songs is dispersed over the DHT? Specifically, you wish to avoid situations where some of your nodes are actively serving content while some are idling away.

In order to avoid Idling on certain machines, I would try to distribute the music content in such a way that bands/albums/songs that are more popular don't get piled up onto one machine and are more spread out across the DHT, and the less popular bands/albums/songs can be less spread out and more of them can be accumulated onto a given machine. The reason for this is that the more popular music content will likely be accessed more often then the less popular music content, so distributing the popular content across more machines could theoretically prevent machines from staying idle for too long. Letting more non-popular music be stored on any given machine could also increase the likelihood that it will not remain idle for too long, since every once in awhile someone might want to access one of the less popular songs. If the less popular songs were more spread out, say one machine only had a few unpopular songs on it, then it would be very unlikely that it would get accessed and it could remain idle for long periods of time. If the machine instead had a relatively large amount of unpopular songs stored on it, then hopefully it would have a lower chance of remaining idle.

To be a little more specific, if the popularity of any given piece of music content is measured in terms of frequency accessed (e.g. every 5 minutes a request is made for that content) then perhaps the music content can be stored on machines in such a way that ensures that any machine is accessed or being used on a specific time interval. For example, let's say we have a set of two pieces of music content, and we measure their frequencies. On average, the first piece is accessed every 5 minutes and the second piece is accessed every 7 minutes. Once the machine starts running, it is accessed at the 5-minute mark, then again at the 7-minute mark, again at the 10-minute mark, and again at the 14-minute mark, etc. If we let the machine run for 15 minutes, then it is accessed on an average of every 3 minutes ((5+2+3+4+1)/5 = 15/5 = 3). Perhaps we could create an algorithm that ensures that the average access time across all nodes in the DHT is minimal.

Q4. Your streaming music service is slowly getting popular and you have procured a machine that is 16 times as powerful as the other (identical and homogeneous) machines in your DHT. How would you account for this new powerful node while ensuring that it is not underutilized?

With the new machine I would ensure that it has the lowest possible average access time out of all the nodes in the DHT, and I would try to place it where people are making the most requests for other pieces of music content from the other machines. I would first try to identify which machine people are making the most search requests from and replace that machine with the more powerful machine. If the more powerful machine is still underutilized after handling the search requests, then I would also try to give it a set of music content that has one of the lowest average access times such that it doesn't become overutilized from handling all of the search requests and streaming. If there aren't any machines that are getting an overabundance of search requests, then I wouldn't worry too much about replacing a machine with the powerful one due to handling search requests. Instead, I would replace the machine with the lowest average access time with the more powerful machine.

If possible, I would also try to rearrange the distribution of the music content in such a way that the powerful machine will have the set of music content that has the lowest possible average access time, along with songs and music content that needs to be streamed for longer periods of time due to the length of the music and possibility of the client wanting to repeat the music again. Perhaps it would be a good idea to put the most popular albums along with the most popular songs that have the highest chance of being repeated on the most powerful machine, since it would then need to frequently create new streams and keep those streams running for the longest period of time. Doing this could have the benefit of taking pressure off some of the less powerful machines that experience a loss in quality of service due to the growing popularity of the streaming service.

Q5. How would you cope with the corner case where a song is 3 times more popular than the average song?

If a song was 3 times more popular than the average song, then it would probably be a good idea to have that song placed on the most powerful machine in the DHT. Especially in a case where the average frequency of being accessed is already very low. For example, say the average is 1 second, then the 3 times more popular song would be accessed every 333 milliseconds. If the song was 2 minutes long, then after 2 minutes the machine that is responsible for streaming that song would constantly have 360 streams ((2m*60s)*3 streams per second) that it would have to handle! This could seriously bog down some of the machines, and in more extreme cases (e.g. the average is 30 milliseconds or something ridiculously low) even the most powerful machine in the DHT could easily get bogged down having potentially thousands of threads simultaneously serving streams.

Another idea could be to split the song up across multiple machines in the DHT, and then have multiple streams setup between the client and the machines that contain the parts that make up that song. Say the average frequency is again 1 second, making the 3 times more popular song accessed every 333 milliseconds, but this time the 2-minute song is split up in 4 even parts that are individually distributed across 4 different machines in the DHT. If each part was 30 seconds long, then each machine would only need to keep their portion of the song/stream alive for approximately 30 seconds. The protocol for handling this might be complicated however, since the whole point is to reduce the amount of time each machine spends streaming the popular song. It would defeat the purpose to keep the streams alive for the entire 2-minute duration of the song, so some sort of buffering protocol would need to be established to ensure that there is a smooth transition between parts of the song. Perhaps 5 – 10 seconds before the next part of the song, the next stream should be setup to ensure that there is enough time to establish a buffer and a smooth transition into the next part. This could obviously introduce more errors as there are now 4 streams, and if say the machine responsible for stream 2 fails then essentially streams 3 and 4 will never be established and therefore the entire song would be rendered un-stream-able!