CS455 HW1-WC

Josh Mau

1. The most challenging part of this assignment for me was determining when all messages have traversed the overlay, and there are no more packets being routed. The issue that occurs at time when there are many messages being sent, is that most of the nodes will complete the sending portion of the task, but messaging will either still be queued on receiver ports, or some messages may still be being forwarded to the appropriate sink. To solve this issue, I came up with two possible solutions. First, I could place a large wait on the Registry server using Thread.sleep(15000) roughly 10-15 seconds, but this kills performance especially on small tasks. The second solution I came up with was to place a small wait on each messenger, although this wait could occur multiple times. Once a messaging node has received the request for task summary from the overlay, the messaging nodes enter a do while loop. Essentially, this loop stores the current values of packets received and packets relayed values and then does a small Thread.sleep(20) of about 20-50ms. When the thread wakes, it rechecks those values, if either has changed it waits again. The reason for checking relayed packet counts changing as well is to avoid sending back data until the entire overlay has finished receiving messages. This solution alone has risk of error as there is a possibility in the time allotted the values could not change even though messages are still being received elsewhere. The solution I decided upon was a combination of the two. Essentially messaging nodes will perform this wait while either of the previously mentioned counters change, then send the data back to the registry. If the numbers don't match the Registry will wait a short amount of time and then resend requests for task summary to all messaging nodes. With this method, I've optimized the task response time far past just doing a 15 second wait. [323 words]

2. The most important parts of the code base that must be synchronized are the counters. This is because multiple threads could be trying to update counters and multiple threads could be performing a task at any given time. There is one set of counters on the registry and one set of counters on the messaging nodes. However, there still needs to be some sort of synchronization in sending messages as the sender thread could still be executing the initiate task request while it's also trying to relay messages. This issue occurred while I was writing this assignment where some of my data would be segments of different byte arrays that were spliced together during transmission. One way around this would be to synchronize both onEvent methods which would solve all the problems but would likely slow performance, but this would guarantee counters would never be in an intermediate state. This would also require the method that sends the requested task not be its own thread. It would have to be the thread executing in the onEvent method from the Registry TCPConnection. I think this would lead to a possible dead lock however or an overflow of the port queues depending on their size and how large the task request is. Another possibility would be to have a class that handles all storing of statistics and a synchronized method to increment counters that both the registry and messaging nodes would have an object of. In this case, there would be one of these objects per statistic you want to

track (ie. Packets sent/received) per node. Any time one of these counters needs incremented, the thread must secure the lock for that object as the increment method would be synchronized. This method must also return the current value after it's incremented so there would be no need for a second synchronized method to get the current values, you could just increment(0) and get the current value. The second synchronized block would be placed on the TCPConnection sender method. [337 words]

3. To ensure even distribution of streaming load across the overlay, I would try to implement some sort of dynamic size of the hash values for a particular node is responsible for. If the set of possible hash values was 1000 per se, and the number of nodes was 100, the network would begin with each node being responsible for a range of 10 hash values. However, partial amount of the range could be passed off to the next node if one machine gets busy. For instance, if a node has connections to *n* other nodes, it could send a request for information about how busy each one is. Upon receiving information back, the node would determine which other node is least busy and pass off a portion of the least active hash values it's responsible for to the other node. Nodes that are not busy may be possibly be responsible for 20-30 hash values (ie. Songs that aren't that popular), while nodes responsible for more popular songs may only be serving 2-3 hash values. There would either need to be some sort of blockchain-like consensus as to which nodes are responsible for what hash values as this will change as the load balance changes, or there would need to be a central hub, like a registry, which coordinates changing responsibilities of hash values. It could be possible to have each range of hash values be assigned to a registry, and then the registry further breaks down responsibilities among its sub-overlay, and they registries would pass off portions of their hash responsibilities to other registries. As the load changes per song, the overlay will balance itself until the most popular songs are only sharing a server with 1-2 other of the most popular songs, while least popular songs will be sharing a server with many other songs. This should result in all or most nodes being evenly active, and no node will be doing the bulk of the work. [329 words]

4. If a node is 16 times more powerful than any other node, this node should have at least 16 times as much traffic as the average standard node in the overlay. Based on the response give above, I would assign each node a suggested max load of 10 for each node, but since this node is 16 times more powerful, its suggested load should be 160. This would mean that a node should not pass off part of its load until its percent load is greater than any of its connected nodes. If it's just total load, the overlay would balance we the more powerful node would have the same as other nodes, it would have to be based off of a percentage. If the most powerful node is at 150 of the hash values, it's almost at the suggested capacity. If it scans its connected nodes and one of the nodes is at ten percent load, this node should pass off one of its responsible hash values to the other node. The nodes should repeat this every so often to check the balance between nodes. Max load could be hard coded in to each node based on the hardware of the machine, or some sort of machine learning algorithm could be applied to determine when the node should start requesting to pass off load to another node. Nodes could also reject this request if they're also at capacity. In this case, the node that denies should check its connected nodes to see if it

can pass off any of its load to another node, and then return a request to the origin node to see if it still needs to pass of load. This should ideally balance the load between all nodes as the more powerful node should, on average, have 16 times the load of and other node, whereas the standard nodes would be at the same percent max capacity as the more powerful node. [328 word]

5. If a song is three times more popular than any other, ideally, with the same reasoning given above, it should have its own server handling this specific hash value. When the server starts it will be sharing a server, but as requests for this song increases, the server will push off other, less popular, songs to other servers to better handle the most frequently requested song. However, if the one server alone cannot handle all the requests, I think it would be possible to split the hash values between two nodes. Instead of passing off segments of the hash values a server is responsible for to another node, the node would simply send some specified message to another node requesting the other node's help in serving this song. Upon the other node approving, the original node would forward some (ideally up to half) of its requests to the other node, splitting the requests in half per node. This should be able to happen multiple times where and number of servers could be forwarding incoming requests to other nodes to help in the large number of requests. This should balance the load as more requests come in, the originating node (already serving 100 percent of its workload to the one song) should request help from another node. That node should accept or pass the request off to another one of its connections until it finds an available node. Once on is found, that node can start accepting requests. If the requests become to large, the new node can then send a request to a third node for assistance, and/or send the load from the other songs it may be serving to another node. If one or more nodes is more powerful than any others, those nodes should not need to request assistance from another node until it reaches a certain threshold of its percent maximum load. [316 words]