Christopher Westerman

Written Component – HW1

Q1: Most of my time spent on assignment one was in the design phase which had a significant number of hurdles. The main issue that came up was how to contain my threads in such a way that they provided a robust interface but reduced the chance of state leaking. Even after spending countless hours designing and trying to solidify my plan, I ended up running into small concurrency bugs within critical sections of my program. Eventually, these issues were resolved by quarantining state into the class while also being more vigilant towards similar internal state based issues. Somewhere near the end of the assignment I had also started to forget minor implementation details. Normally this wouldn't be an issue, however there wasn't significant logging to find both issues in a timely fashion. One of the issues came when my event factory wasn't instantiating new instances of events to be handled but instead would return null. Since there was nothing to check whether the event was null, the event would be simply ignored with no error message or indication that something was amiss. After the previous issue, I ended up spending a considerable amount of time trying to figure out why my messaging nodes wouldn't receive messages. Once again, the error existed in implementation and was absent from my created specification. While attempting to check for incoming messages, my implementation would only check to see whether routing entries had incoming messages. Since receiving messages from entries in the routing table would be impossible, all I had to do was check all my open TCP connections for messages and not limit my search space to my tables. Both of these issues would have been easily and quickly fixed if I had unit testing or mocking in place to narrow down the problem to one section. After a significant amount of time, I found myself being more vigilant towards these silly implementation mistakes.

Q2. After revisiting my code base, it becomes clear that every one of my synchronized keywords aren't needed and my implementation could function without any synchronization outside of the already implemented concurrency data structures. Had all my data structures been implemented without using predefined concurrent data structures, I would have to have a synchronization block surrounding the TCPConnection's send and receive queues. Both of these queues behave the same way as they are accessed by the internal socket and externally to pull or push messages into the queue. Since both are tightly bound into the TCPConnection class, it made sense to make them immutable to prevent any direct access to the outside. To reflect this, my TCPConnection provides a linked blocking queue to both the TCP sender and receiver. When accessing messages, the TCPConnection class provides a receiveMessage and sendMessage to relay messages into the receiver and sender respectively. To provide non-blocking access receiveMessage will internally call LinkedBlockingQueue.poll() which will return null if there wasn't any messages to receive. Similarly, when placing messages into the outgoing queue, the thread will invoke LinkedBlockingQueue.put(). Both functions internally handle the synchronizations and reduce the need to further control the data structure internally. By reducing the chance of releasing references, I can confidently let my thread go on full blast. In addition to those queues, I use a ConcurrentHashMap within my TCPConnectionsCache to manage incoming, broken, and closing connections without running into concurrency issues. Like the previous data structures, the synchronization primitives are used internally to the hash map reducing the need to synchronize on the entire class. Had I not used these data structures, I would have surrounded access with specific locks to prevent concurrent access and redesigned with the internal locking mechanisms in mind. Due to the bottlenecking caused by synchronizing blocks, I tend to avoid them as much as possible.

Q3. The flexibility of hash tables come from their ability to dynamically reassign bucket size and by proxy, distributed hash tables have the same quality. Over time, the table should be able to increase the number of nodes that work on a specific bucket. An example of the previous would be if a song was rising in popularity, it could be transferred to various machines in the overlay and said machine would be able to receive streaming requests for an in demand song. Although it would have an additional overhead due to the rerouting, at least there will be some scalability in the work completion. In general, when the overlay detects the rapidly increasing requests for one item, it increases the number of nodes assisting with that particular task. With this scheme, as long as there isn't a spike in activity, there would still be nodes that would potentially be idling. To further spread out the load between servers, there could also be measures in place to detect the machines that have the highest active usage and force them to route packets to those idling nodes. The ideal condition would be to monitor the most busy and least busy nodes and redirect traffic from most busy to least busy if the delta in their activity grows too large. This addition will do the most to keep each node's activity closer to the average. At the point when there isn't an urgent need to load balance requests, it doesn't seem to be an issue if some nodes are not producing messages, in this case the overhead of communicating with the server and forwarding information to the selected nodes could outweigh the default performance of the overlay. The best solution would most likely have a high enough delta to where it calculates the optimal time in which to start moving around requests.

Q4. Making sure the machine doesn't get underutilized lies in the correspondence between the hash values and the node IDs of each machine. If there is a significant gap between node ID and the machine in question, there would be a higher probability that any particular hash value would land in the space between the hash and the node Id. An example of this would be with the set of node ids [1, 2, 6, 7, 8]. When calculating hashes, a message being sent to 1, 2, 7, or 8 would have to exactly match their node ids where a message bound for node 6 could be hashed with 3, 4, 5, or 6 effectively increasing its hit chance by 400%. Furthermore, the overlay could increase the spacing to adjust the throughput of the node based on performance. According to the analysis done above, the node ID spacing would be 16 times the average spacing between other node ID values. In general, having an even distribution of node IDs will create a system in which each node will have an equal probability of receiving work (assuming the hash was decent) and slightly changing this distribution of node IDs will allow further editing of an individual node's probability of receiving a streaming request. With the additional overhead brought about by load balancing, it would be advantageous to rely on these mathematical principles before moving to what would be exponentially slower given the extra relaying times for message passing.

Q5. Since the primary way to cope with overplaying a song would be to allocate resources, the node that would receive the request could at some point start offloading its work onto nodes randomly picked throughout the network. This node would send its song information over to the selected nodes and it would begin to direct traffic relating to that song over to these nodes. Effectively, the song would have the support of N randomly picked nodes to help balance the load. If at some point the target node was already overloaded or became overloaded, it could tell the original node that it can no longer accept the extra work load and request to be removed from the first node's extra table. The first node would then look for more nodes that it can offload onto. Most likely, to ensure each node has the correct amount of load, an external server could be monitoring throughput and sorting the nodes based on their loads. When this server is contacted, it could return the id of the node that is currently under the lightest load. By maintaining contact with the load balancing server, it could reasonably cope with the rise in the amount of requests. If this approach was adopted it would add the extra overhead of reporting back to the load balancing server periodically although the impact would be negligible given a time window in the order of seconds. Overall, the performance increase of load balancing would be worth the sacrifice of the check.