Anthony Rezzonico
CS 455
22 February 2019

Homework 1: Writing Component

1) The biggest issue that I faced when I was developing my implementation of the overlay was trying to figure out how I could manipulate the Register Request event to work for both when the Messaging node tries to connect to the Registry and when the Messaging node tries to connect to another peer Messaging node.

When I connect the Messaging node to the Registry, once the server thread accepts the connection, it takes the socket and creates a TCPRegularSocket which is used to help manage the data input and output streams in a class managed socket. This regular socket is added to the Registry's HashMap using the regular sockets Ip:portnumber as the key tied to the actual regular socket. From the receiving thread, it places the event into an event queue thread that is responsible for calling the resolve method on the event. From here the event would add the IP and port number from the register request and would check in another table called ServerToRegular Hashmap to see if the ip:port key was present in the table. If it wasn't in the table then the event would add to the **Registery's** ServerToRegular Map the IP:port of the request to the regular sockets IP:port.

The issue that I had is that I wanted to use the same register request to register the peer nodes in the overlay. In the Register Request I added an integer field that specified the origin the request whether the Messaging node was trying to connect to the Registry or to another peer Messaging node. I was unable to get the nodes send messages to exchange messages with each other and this ultimately was my biggest issue. I had setup everything that would have made sense but did not get to test it in time that allowed me to fix this critical bug.

2) If I had the opportunity to redesign this implementation, their would be some changes that would of made it easier to work with. One of the big things that I realized in the final hours of the project was that I had made too many private static fields within my messaging node and the registry, and I had also included to many public accessors that any of the threads could concurrently access. I had a few methods in the Messaging node that would directly return the data structure and return it to the thread. Instead I should of developed some static helper class that would help the thread interact with the connections. This class could of done thread safe operations that would add the connections and expose data about the connections without directly giving access to the reference to the threads.

Something else that I would of done that could of easily helped would have been to create an entire new event to register the peer nodes to one another. As mentioned above, I ended up struggling getting the Register request to even work and a lot of it is due to the fact that I though I could reuse the Register request event for the peer connections. I saw that the request and the protocol were similar enough that it made sense to incorporate the functionality into the same thing. I ended up spending more time trying to make it work rather just implementing a new event.

I think another thing that I would of done with the implementation would have been to implement the routing cache in a way that all of the messaging nodes would have an instance of the class that would allow them to take the source node, sink node, and the node it is currently at and would direct the message to the next peer node in the path.

3) The biggest focus for solving this kind of a problem is making sure that the potential solution is scale-able and can perform with a large overlay size. One of the first thoughts would have been to have one node completely dedicated to sending an overlay to each node each time that a weight changes. The issue to that though is that node would take on a considerable amount of work to compute the new shortest path for each of the nodes, and by the time that the messaging node receives the the new overlay one of the weights could of easily changed. Another way that this could be done is after the Registry has established all of the link weights for the nodes, the registry then sends out the entire overlay with all of the associated link weights and then each node has to perform Dijkstra's algorithm upon receiving this message. At each node it would then determine the shortest path to a sink node. Since the algorithm is dependent on the current state of the graph, each time a new link weight is adjusted, the algorithm needs to be re ran to determine the best path. That being said, time that weight is modified, the registry would need to send another message to all of the nodes to update the current state of of the overlay. The cache could take as input the node from which the message originated, who the sink node is, and what node currently possesses the message. By allowing each of the Messaging nodes to have a copy of the current overlay with the link weights prevents overhead from the node having to query a routing cache to see who is the next node.

4) I did not include a routing plan in each of the messages so that the node knew who to relay the message to. Had I gotten to this portion of the assignment, I would of implemented a routing cache that would of contained a Map of the origin node to another map that would contain a key of the sink node that would map to a path string that would show the message and what nodes it would visit in transit. With this implementation I could easily derive a set of paths from what the starting node is, and from there I could easily retrieve the path that the message should take. I would then be able to use the index of the current node that the message is at and relay it to the next node.

If the links were dynamically changing in the route, having the routing plan predefined in the message would do no good. If a messaging node *A* were to send message *m* to node *B*, maybe during in transit the weights changed and *m* is no longer going to be forwarded to node *C* but rather now the shortest path is through node *D*. Once node *B* intercepts the message, it will have no knowledge of what the shortest path is since the the weights are dynamically changing the node would need to wait and calculate to determine where the next hop would have to be rending it inefficient. The goal with this is that once the messaging node receives the message, the message needs to go to the next hop in the overlay as soon as possible before all of the link weights update again. Since this is not a minimum spanning tree, each time the link weights update a new shortest path would need to be calculated to maintain a best effort to get the message to the sink node in the shortest path.

5) The issue with using a minimum spanning tree vs. Dijkstra's algorithm is that with the MST implementation of the nodes in the overlay will design a path that connects all of the nodes and obtains the smallest overall weight for the tree. When a link in the minimum spanning tree updates, it is very easy to be able to determine the next path since it does this evaluation at the node level and only happens when it affects a node’s current connection. With Dijkstra’s algorithm, we are guaranteed the shortest path from any node to another node in terms of hops. The place where this differs from the minimum spanning tree if the entire overlay receives a new weight for the graph, an entirely new path must be derived from the updated link weight change. If an implementation of the overlay is done with the minimum spanning tree, we would a higher percentage of losing UDP packets over time as opposed to a Dijkstra’s implementation of the overlay. With this, we would see less of a of a packet loss because the algorithm guarantees that once the message takes route that it will take the shortest amounts of hops in the network.