Bruce Darcy
Distributed Systems
Prof. Pallickara

Project 1 Written Component

Q1. What was the biggest challenge that you encountered in this assignment?
Note: The challenges should relate to design decisions and algorithmic choices and not so much to do
with unfamiliarity with programming elements such as sockets, etc.
[300-350 words]

The biggest issue I faced this assignment was properly setting up data structures so that my code would run efficiently. Continuously through this project, I ran into situations where I was unsure whether to use one data structure or another. For example, when I do my dijkstra's shortest path algorithm, I use use leverage both arraylists and hashmaps to simplify different tasks. I designed wrapper classes for both vertices and edges on the graph with specific functionality. My vertex class contains a field containing a pointer the node that the shortest path came from, allowing me to naturally create a linked list from end to start containing the shortest path. I also use a hashmap that lets me easily associate the node identifiers, each of which is the ip address and port of a node combined into one string, to the vertex objects I am working with. These identifiers are something that I used in many places in my code, to allow me to specifically identify a node, even if it is running on the same machine as other nodes, or if two nodes are running on the same port on different machines. I ended up with both a hashmap and a vertex array containing references to the same vertex objects in my shortest path class. Each of them helped me do something faster or better in some part of my code, but it feels redundant and seems like a bad practice in general. I also construct a next jump map right after I run dijkstra's, that quickly can tell me what node I need to send a message to next when I need to relay. In my node classes. I used hashmaps to map identifiers to sockets for faster message sending, alongside the next hop map that maps an identifier destination string to a nexthop destination string. Generally, this assignment made me think a lot more than I had to in the past about how to use data structures properly to make programming easier and my code faster.

Q2. If you had an opportunity to redesign your implementation, how would you go about doing this and why? [300-350 words]

There are few things I would add to my implementation, and a few things I would change. I would add an event queue that all receivers add to, and then I would create worker threads that pull events out of the queue and execute the actions. This would allow my receiver threads to take input more often. I didn't add this at first, because I didn't want my receivers to spend too much time blocking when there is nothing to process, so I made them handle the events as they came in. In hindsight, I think that an event queue and worker threads would have been better. I also would refactor my registry and messaging node classes to do two things: share more functionality, and move functionality into other classes to reduce their size. As it stands, both those classes are very big, and contain a lot of the logic of my program. It would be good to wrap, for example, my next hop map in the routing cache object, which I did not use in this project. It may only be a map, but I could move some functionality there to simplify my program. There are also several things that my registry and messaging node classes share and do often that I could move into my node abstract class. I think it would also be better if instead of constructing my TCP Sender with a socket then giving it the data to send, I would rather make a static method in it that takes a socket and a message and handles everything there. There are a lot of places where I add extra lines to construct the objects first, then use them to send things. I also end up with more local variables in a lot of scopes because of this, and sending a message when I already have the socket and message is not really something that should require an object, especially when that object is immediately discarded in most circumstances.

Q3. Consider the case where the link weights of the overlay are constantly changing and you are routing
UDP packets. How would you make sure that your routing reflects the current state of the links i.e. by
taking alternate routes to destinations. Your response does not need to account for restrictions that
were specified for the programming component.
[300-350 words]

I would periodically rerun my dijkstra's shortest path and update my routing table/map. This would need to happen at a reasonable interval so that the information was still relevant, and such that the next round of dijkstra's doesn't start until the current one has finished. A smart idea would to have one routing thread that continuously reruns dijkstra's. Also, you could do analytics into the rate of change of weights in the network to try and predict where they will be in the future. Maybe this could be done with simple calculus, but this also seems like a good place to try and apply some machine learning. There are probably patterns based on things like time of day and year, types of pathways and other variables that such a model could take into consideration. Even without machine learning, designing a system to consider these things and make prediction in a more hard coded way would be good. If you could get decent predictions using some method, then you could have paths planned before the link weights update, and then compare to see if you still things your predicted paths are good enough to use. Dijkstra's will often take some time to run, so attempting to be ahead of the patterns could be useful. Also, maybe each machine could forward some percentage of packets along a different path to try and reduce congestion before it even happens. Also, perhaps if you keep track of the best few paths for packets to take, then start to look at the total rates of change of each path, you could predict which one would be good to take at a given time. So maybe instead of storing just the shortest path to each vertex from where you are, store K of them, when look at the rates of change when you need to update them. Run dijkstra's completely occasionally to make sure that model stays in sync with the graphs, and make sure that long paths you weren't storing before didn't get a lot shorter while you didn't have them cached in memory.

Q4. Did you really need a routing plan per-message? Please explain. Contrast the efficiency of using a routing plan per-message if the weights are changing constantly, and the routes continually updated? [300-400 words]

I don’t think that a routing per message system will be the most efficient way to design a system. The calculations to find the needed path per message will take more computation time than is efficient to keep message throughput high enough. A routing plan per message could be implemented two way: store the entire plan in the message, or recalculate the whole route at each relay. If you store the entire plan in the message, then you have added a lot of extra data to it, and by the time you get to a hop later in the plan, the weights may have changed and there may be a much better route to take by that point. Having each router calculate a plan for each message, is also very computationally expensive, when handling large amounts of messages. To know that a path is best for a message, you have to check a lot of possible paths, which takes valuable time and would reduce your throughput by a lot. Even though packets would often take better routes with per message planning, because of the speed at which messages can travel, compared to the complexity of finding a path for each message there would probably be much less efficiency through the system as a whole. Even if messages might be able to take a better route, routers are going to need to do so much more computation that the whole system will build up more latency just waiting for computations to finish. A better idea would be to update the routing information every so often, and then store it into a hashmap structure that allows messages to be forwarded quickly, while another thread underneath is looking at the graph structure of the network, and continually updating the next hop map with better paths.

Q5. Let’s say you built a Minimum Spanning Tree (MST) connecting the nodes within the overlay. The link weights were based on a metric that accounted for the throughput of the links and the typical loss rates for UDP packets. Contrast the efficiency of the dissemination scheme where you are constrained to using the MST versus the scheme that entails using Dijkstra’s shortest path. [300-400 words]

A minimum spanning tree would be good because it can make greedy decisions at each step. Packets might take a slightly longer path, but it could be updated to reflect changing weights much faster. Using Kruskal's algorithm, as soon as the minimum spanning tree connects to a node, you immediately know what you can send the packet to the next step because the minimum spanning tree must be optimal, or equivalent to any other minimum spanning tree that might exist. In a network situation where you have precise weights that are very unlikely to have any two take the same value, the tree will most likely be unique. Updating the tree without running the whole algorithm from scratch again may also be possible, if you continually check and compare links that could connect the graph in a different way, and seeing if it reduces the overall cost. Dijkstra’s however would have to be run from scratch each time I believe. The advantage to Dijkstra's however is that the path is guaranteed to give you the absolute shortest path, while minimum spanning tree will give you a pretty good one. With a minimum spanning tree it would also be hard to try to split packets going to the same place along different paths preemptively reduce congestion, but that may not be necessary to make the network work well. If the minimum spanning tree could update fast enough, which algorithmically, it could do pretty well with just a basic scheme of forwarding packets using only that. Because Minimum Spanning Tree could be updated on the fly without rebuilding the data structure, I believe that it would be a better solution mainly because it could react more quickly to network changes than dijkstra’s implementation of routing could. It could also be run pretty easily by every router without a ton of overhead, and optimizations could probably be done to search for more important links to improve than others to improve the system.