Todd Worcester
CS455
Spring 2017
HW1-WC

**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.**

The first design challenge I encountered was figuring out how to pass information from a running thread back to the associated node. I first attempted to solve this problem using the java Observer and Observable interfaces. I made each node an Observer and each thread an Observable instance which would notify the Observer when an event came in. This solution worked well, but proved to be not quite flexible enough for my needs. The Observable thread could pass one object reference to the observer, but on a couple of occasions I needed two things passed to the Observer. Perhaps that is indicative of a design flaw, but at the time I could not come up with an elegant solution that worked with Observer/Observable. My next attempt to solve the problem was to pass a reference to the node as a parameter to each thread. This proved to be much more flexible, as I could invoke node methods from a thread as needed.

The second major design challenge I faced was ensuring thread safety. This proved to be a much more difficult challenge to overcome, mostly because I didn't realize it was a problem until I had written nearly all of the code for the assignment. Early on I tested sending and receiving messages only between adjacent nodes. That worked without any problems for large numbers of messages. I foolishly assumed that adding a relay mechanism would not introduce any additional concurrency issues. I could not have been more wrong! As a result of my poor assumption my "finished" program did not work for a large number of rounds and I was reduced to a "guess and check" approach of adding synchronized blocks in different places. This resulted in many hours of frustration, but quite a lot of joy when I finally stumbled on a solution that worked.

**Q2. If you had an opportunity to redesign your implementation, how would you go about doing this and why?**

If I could redesign my implementation I would focus on thread safety as I was writing the code and hopefully avoid having to attempt to fix concurrency bugs after the fact. I would attempt to use techniques that we've discussed in lecture to avoid problems before they manifested themselves. I would reason about which pieces of code could potentially be accessed by multiple threads at once and attempt to compose critical sections which minimize the scope of any needed synchronization.

Specifically, I would re-work message relaying and sending. My code for those two things is contained within two different cases of a switch statement in my messaging node's on event method. It has duplicate code and poorly scoped critical sections, and as a result was a nightmare to debug. Re-composing those two pieces of code in a more well thought out manner would make my implementation much much better.

Although I know it's not really the crux of the assignment, I am not very proud of my Dijkstra implementation. If I could redesign my program I would re-write that and use something elegant like a priority queue rather than a step by step "by the book" algorithm.

**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.**

If link weights are changing while messages are being sent, the registry would need to periodically update the messaging nodes of weight changes and the nodes would then re-calculate their shortest paths. We are routing UDP packets, so we have the flexibility to reroute packets on the fly, there is no need to preserve the order in which the packets were sent.

Therefore, if a packet arrived at a node with a stale route, the messaging node would be able to inject a new routing path into a message in order to get it to the destination most efficiently (maybe even arriving before some previously sent messages).

**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?**

No, it's not necessary for each message to carry an entire routing plan. Just the destination should be enough to get the message where it needs to be. Each messaging node could maintain a mapping of what the next hop is along the shortest path for each destination. When a message arrives, the node looks up where the message should go next based on the destination and then sends it there.

If link weights are constantly changing, there is a high likelihood that the initial routing path assigned to a message will become stale before it reaches its destination, perhaps multiple times. Messages containing routing plans would constantly need to have their plans checked and updated if needed. If instead we used the above scheme, with the messaging nodes each maintaining a table of the best next hop for a given destination, the node's table could be constantly updated and it could quickly route the message in the best direction based on the latest information.

The approach with a table of "best next hops" seems like it would be quite a bit more efficient. There would never be a need to inject any new information into a message, just read the destination and send it on its way. Even just enforcing an embedded routing plan is a somewhat costly operation (at least my implementation of it seemed to be) having to evaluate and insert new routing paths into a message would really hinder efficiency I think.

**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 you have where you are using Dijkstra's shortest path.**

While a minimum spanning tree ensures that the total weight of the network is minimized it does not ensure that the path between any two particular nodes is minimized. Take for example a simple network with the following links:

A -> B 4
B -> C 5
A -> C 6

The minimum spanning tree consists of edges A -> B and B -> C and has a total weight of 9. However, it does not contain the best way to get from A to C (directly with a cost of 6).

If we are constrained to using a MST some messages may not be able to take their most efficient route to the destination. This of course negatively affects efficiency directly, but also may have secondary consequences. A MST artificially constrains the number of routes a message may take. All messages are forced into one path per destination even if there are alternate paths which may be just as good. This may result in an increase in network congestion to no benefit of the messages. It's worth noting that Dijkstra's algorithm also doesn't address this last issue. When given a choice between two equal paths Dijkstra does not attempt to choose the "less traveled" one but there is at least a chance that it will.