**Design Process/Analysis**

The initial step in the process of our project 2 was a perfecting our project 1. First off, we needed to add a working timer and finalize the list for our neighbor discovery. We ended up implementing the timer function within the “implementation” section of Node.nc and set it to run our “discoverNeighbors” function every 500mili seconds, constantly keeping all nodes and neighbors up to date. We then input any new neighbor nodes into a list we created after checking whether or not it is a previously unvisited node. If we have gotten to one of the nodes previously visited but from a different path, we check for a difference in path cost because we want to know the routes to each neighbor node with lowest path costs. Once completing this, we could then begin to work on implementing our routing information protocol.

We were given a few parameters to work around early on and these included things like, assuming there would be no node larger than 255 and that path costs will be calculated solely based on hop count. This meant that we needed a much more precise approach versus simply flooding the network and link state routing as in project 1 and this solution was to send the distance vector routing information directly to immediate neighbors. We created a “route” method within our Node.nc and this is where we placed our implementation. We first started off by using our link state algorithm from project 1 as framework of sorts, knowing we could build upon it and create the precision and efficiency we were looking for. We made structures for storing the destination node, the path cost to get there, and the next node we will be hopping to. We then mixed together our link state routing with our previous knowledge of the bellman-ford algorithm to produce our best RIP algorithm.

The way we implemented our routing table was by creating a list of our created LinkState structures. We also created a table for confirmed and tentative routes. The only issue we had with this was getting the routing table to print. We did not know exactly how to perfect the routing table’s structure because we were getting some information at incorrect locations and positions, but then some of our other information was completely missing. Deciphering the data was the first step, finding other pieces of our data that was out of location was step two, and the final step was trying to figure out why certain things just would not print and got lost somewhere along the way. After figuring all this out, perfecting our data and making sure it was all correct was next.

We began by making sure our timer worked and fired off the routing information properly and that our immediate neighbors received the information and could read it properly. First running the destination link state to determine where our destination node was, second running the next link state to determine the next node we need to hop to in order to end up at our destination, and finally we run the cost link state to tell us the cost of the paths we just took. Once we had our packets sending throughout our network, we needed to assure the proper path cost was being achieved and stored for further use. Finally, the nodes were communicating the appropriate path costs, we were then able to check for any shorter paths and make sure that path would be the first choice, but we still needed to keep the longer path information to build the most accurate network model. This checking turned out to be a bit more difficult than expected but we were able to come up with our solution through a bunch of if-else statements within our “route” method.

**Discussion Questions**

1. The pros of using distance vector routing over link state routing are that they send their entire routing table to its directly connected neighbors, it is easier to implement, and its best for small networks. However, link state routing is much more reliable, more efficient on larger networks, and has a faster convergence.
2. For the most part, symmetric routes are created because they are going to be the most efficient routes to get from one point to another based off of lowest cost but there were a couple instances where we had a different routing domain, resulting in slightly different paths.
3. This may cause some problems in creating an accurate routing table and this would create issues but can be easily solved creating an initial structure map for us to have at least a default understanding of where each node was at the beginning. We could also attempt to send packets to those nodes and because of symmetric routes, we can get a finalized version of the routes.
4. If a distance vector packet is lost or corrupted this can result in an infinite loop or incorrect information going into our routing tables and to solve this, we can implement split horizon and poisoned reverse.
5. This could generate incorrect information throughout the routing table because paths changing every time we fire, would disrupt other path costs and could result in a loop. We would need to modify our code and develop a way to highlight constantly changing paths because there should only be a change when a more efficient path cost is discovered, eventually disregarding the ever-changing path.