Devanshu K., Jonathan S.

March 10, 2020

CSE 160

Professor Cerpa

Project 2 Design Process

Going into this project, we already understood neighbor discovery and flooding due to Project 1. We had implemented neighbor discovery and flooding simultaneously instead of using modules. Instead of changing our old project code into modules, we realized we can implement routing along with neighbor discovery. We were already given that we would be using neighbor discovery again for this lab.

Since neighbor discovery and routing will happen simultaneously, we set up two separate periodic timers for neighbor discovery and routing so that it would give neighbor discovery enough time to gather the necessary information before we send distance vector packets. Next, we implemented a List<RouteNode> for the routing tables. The routes that were entered in the tables were organized in the order they were entered, and each entry in the list contained the following: destination, next hop, cost. We had to take into consideration that there will be some nodes in the network that is either isolated (there are no links connecting to the node) or the node is not active. Therefore, in a real-world scenario, space would be an important because we cannot use the given max for the struct array and as a result, we just append to existing List using the pushback function if there were to be a new destination.

The algorithm behind distance vector routing was trivial. We followed the straight-forward pseudocode provided by the text (*Peterson 3.3.2*). For example, when filling in the routing table for the first time, we need to use neighbor discovery and initialize each table with a cost of one. Next, each node will find its neighbor nodes (one link away) and share their initial tables to each other by sending the entries packet by packet. As more and more nodes keep doing this, this will at one point converge to a full and complete routing table for each node in the network. This was one of the biggest portions in the algorithm.

We then looked into certain conditions where the system would fail, such as if a node goes down or a loop occurs within the system. In order for the system to catch nodes from going down, it will push packets across a complex network with its most ideal number of hops. In order to deal with loops, we were introduced to split horizon with poison reverse. The basic concept behind this idea is to avoid sending to the origin of a specific routing table back to out that specific packet. In order to accommodate this, we simply check to see if the cost exceeds 20 before updating the cost of the destination in the routing table. If it does, then we update our neighbors with the destination cost to be 20. Since the max topology distance in our example is 19 nodes, it was a fair number to choose. If the cost to destination were to be greater than 20, we don’t send the packets to that and we remove the entry from the table.