Willis Allstead
CPE 400
December 5, 2018

Technical Report: Project #7

Functionality:

My project and its implementation are based on the code provided for project idea 7, the routing in a multi-hop network using emulation. To get a node running, open a terminal and run "python3 willis-node.py 1 itc(#).txt". The routing strategy that I chose to implement was DV or Distance Vector routing. Essentially in my implementation of the DV routing protocol, each node has 3 tables to dynamically form routing rules in the network. When a node is started up, it knows simply how many nodes there are in the network, which is given by the number of lines in the input document. When it initializes its topology, it assumes the distance to itself is 0. The node assumes the distance to the rest of the nodes in the network is infinity. Periodically, in my code I have arbitrarily chosen every 3 seconds, each node attempts to send its distance vector to its neighboring (linked) nodes. If two nodes are running in the network and they are linked, they will be repeatedly sharing this information between each other. They specifically only share their distance vector within the distance vectors table, but they store the distance vectors of their neighbors in that table as well. When a node receives the distance vector of a neighboring node, it appends that vector into its distance vectors table, and it runs the Bellman-Ford algorithm with the edges in its newly updated distance vectors table.

The Bellman-Ford algorithm is used to find the shortest path from a single source node to all of the other nodes in the network based on weights. In the case of this emulation, a weight of 1 is used for each edge, but this weight could be based on link performance, preference, etc. The output of the Bellman-Ford algorithm is a new distance vector for the current node, and this new distance vector is saved to the node's distance vectors table. Each node also maintains a routing table, that is initialized when the node is started. The routing table is a dictionary with items of the form (destinationNodeID: nextHop). If a message is sent using either UDP or TCP, the message is sent to the "next hop" for a "destination". This routing table is initially populated with the directly linked nodes in the network, since the next hop for these nodes is simply the

linked node itself. Using the distance vectors minimized by the Bellman-Ford algorithm, the routing table is updated to represent the best “next hop” for any running node. This allows a node to be able to successfully send a message to a node for which it is not connected.

The third table that I added to the node object was the alive table. This table is a dictionary holding objects of the form (neighborNodeID: timestamp) where the timestamp is the last time a linked node communicated with this node. The alive table is used to keep track of the nodes that are considered “alive” meaning they can be communicated with. Each time a node sends out its distance vector to its neighbor nodes, it also checks how recently each neighbor node in the alive table has sent out a message to the node. If it has been 3 * 3 seconds, meaning they haven’t communicated in 3 updates, then the node is considered dead. The timestamp is set to 0 to show that the neighbor node is dead, and the distance vector for the current node is updated to show an infinite distance for the dead node. This infinite distance is eventually propagated throughout the network.

To see changes happening in the object of each node, the provided “info” command can be used. Now the info prompt shows the existing NID, Link Table, and Address Table along with the new Distance Vectors table, Routing table, and Alive Table. For example, if a user starts 4 nodes using the provided itc.txt file as the starting data, the nodes will propagate information until the first node outputs the following when the “info” command is used:

NID: 1

Link Table: {1: (2, 3, 0, 0), 2: (1, 4, 0, 0), 3: (1, 4, 0, 0), 4: (2, 3, 0, 0)}

Address Data: {1: (1, '127.0.0.1', 50555), 2: (2, '127.0.0.1', 50556), 3: (3, '127.0.0.1', 50557), 4: (4, '127.0.0.1', 50558)}

Distance Vectors: {1: (0, 1, 1, 2), 2: (1, 0, 2, 1), 3: (1, 2, 0, 1)}

Routing Table: {2: 2, 3: 3, 4: 2}

Alive Table: {2: 1544079408.232595, 3: 1544079407.9335291}

Each distance vector is stored in the distance vectors table in the form (nodeID, <correspondingNodeDistances>). So for node 1, it is 0 hops away from itself, 1 away from node 2, 1 away from node 3, etc.

For sharing distance vectors between neighbors, the send_tcp function was altered to allow for different message types. TCP was chosen to propagate this information since data loss would not be acceptable. A message using TCP can have the type “message”, meaning one that is

supposed to send a text message to a node, or the type “routing”, which is not supposed to be visible to nodes, and is supposed to send the distance vector as an object to neighboring nodes. The way these two message types are differentiated when encoded is by using a leading 0 or 1, respectively. That way, in the TCP listener, if the message received starts with a 0, it knows that the message is purely a distance vector object. If it starts with a 1, the message is printed. In both TCP and UDP messages, the source node ID is encoded in 1 byte binary following the message type bit. This is decoded by the UDP and TCP listeners and used to update the alive table. For example if node 3 receives a message from node 2, it updates node 2 in the alive table with a new timestamp. In the case of TCP, the destination node ID is also encoded in 1 byte of binary, and immediately follows the encoded source node ID in the message. This is decoded by the TCP listener and used to check if the current node accepting a message is the destination node id set by the message. If it is, it simply prints the message, otherwise it updated the source node id to itself and forwards the message to the next hop for the destination node ID.

In testing this code, I recommend running the script as is and repeatedly entering the “info” command for a single node to see how the information changes. The “update_rate” constant that is used for timing the broadcasting of a node’s distance vector is available for altering in the globals of the project. Slowing down the process could be useful.

All functions in this project are either left as-is from the provided code, or the changes have been thoroughly documented in the comments around the code. Non-intuitive parameters have also been explained for each function. The only truly new object in the project is the Graph object. This is used for running the Bellman Ford Algorithm on a set of edges. The only new functions I added were send_distance_vector() and updated_distance_vectors(distance_vector), which are responsible for both sending and making calculations based on distance vectors to and from neighbors respectively.

I have chosen to not add any other notifications for the user of a node than what existed before, because the interface seems to work best if only the important errors are shown. All updates to the data a node stores must be seen through using the “info” command.

Novel Contribution:

To enhance the DV protocol that I implemented, I wanted to create a faster way of creating the best routes for each node. I found that in a network of 6+ nodes, a single node can try to recalculate its distance vector using the Bellman-Ford algorithm many times, and some times the

resulting routes turned out to actually be worse than the previous iteration. I found that by only updating a node distance in the distance vector if it was better than the current value, I could speed up the time it takes for the ideal routes to propagate through the network. I do have concerns about this method though, covered below in the Results section of the paper.

Results & Analysis:

The results of the networks that were run using the supplied itc(#).txt files were mostly very good. As a user creates more nodes, the network dynamically finds the best routes to each node in the network from each node. These best routes propagate to create even better routes until an ideal route for each node to each other node is arrived at. Tested with each text file provided, if all nodes were running, each node correctly calculated the optimal routes for itself. This is how it is meant to work, and I am happy with those results. An example of the output of a node after it has been running for a while in a network is provided above in the functionality section of this paper.

The part of the project that I feel unhappy with is the dynamic removal of nodes from the network. When a node is turned off, within 3 updates its neighbors will know it is dead. They will then attempt to share their updated distance vectors with their remaining neighbors, and the user is left waiting for that information to propagate. This can take a *very* long time, and even one node communicating false information to its neighbors can set off a chain reaction that convinces part of the network that a dead node is not dead. As mentioned in class, bad news travels slowly using DV protocols, especially ones as trivially built as mine, but there are probably other factors at work in this case.

I believe one of the major factors that is leading to bad information traveling slowly and unreliably lies in the way that I calculate distance vectors using the Bellman-Ford algorithm. I think by my method of tossing out values that are worse than previous iterations, nodes struggle to update their distance vectors to show infinite distances for dead nodes. Sometimes when emulating a network, this does not seem to be an issue, but other times it is glaringly obvious. Another possible contributing factor to this issue could be the update rate constant that I have set to 3 seconds. Maybe better results could be had by tuning this value, but from personal testing, I did not find this to be the case.

Another issue with my implementation is it lacks an acceptable solution to the count-to-infinity problem. Through reading, it seems that this problem is rather trivial, but I could not

reproduce the problem in my networks enough to be able to fully solve the problem. With more time, I would have liked to implement all of these fixes to my project.

Overall this project provided a unique learning experience, focused on one small aspect of networking, routing protocols. In writing the code for this program, I naturally ran into problems that led me to the solutions that were found and used back when DV protocols were being created initially. By having to face these problems personally, I know more thoroughly understand why those solutions exist in the first place.