Juan Benitez
Sigi Lopez
CSE 160

Project 2
Link State Protocol

Design Decisions

Part 1: Receiving Packets

For this project we first decided to build up our project using the implementation we had for flooding. This meant that for the "receive" function we had to modify just a few lines to start receiving link state protocol messages. In us "receive" function we have two sections, the first one is for checking all the message containing protocols such as ping, ping reply and link state. The second section involves the simplest protocol which is neighbor discovery. For the ping, ping reply and link state section we started by checking if we had seen the message before. After learning that the message was new we checked the three main types of messages that we were supposed to receive. The first check was to find out if the message is for this node which meant that there was no other action than to receive the message. The second and third check was to see if the destination was (AM broadcast) or (Node). If the destination is (AM broadcast) it means that it's a link state message that includes neighbors of a node. If the destination of the package received is a (Node) it means that it should be forwarded either through flooding or the link state lookup table. For this project we are specifically using the lookup table that we created with the link state messages to forward our ping messages.

Part 2: Creating Link State Packets

For creating link state packets, we used an array to store the neighbor information of the node trying to send the link state information. The structure was arranged by assigning each index a specific node. If there was a neighbor connection, we would assign that node index the cost and if there was no connection we assigned a value of zero. After creating the specific link state packet, we decided to send it to all of the other nodes using the flooding protocol we previously implemented on project 1.

Part 3: Receiving Link State Packets and Creating the Routing Table

Initially when we receive link state packets from other nodes we store them in a new array which would tell us all of the nodes neighbors and their cost to reach each neighbor. In reality this table gives us enough information to build the topology of the network which in turn allows us to find the shortest path to every node. To calculate this shortest path, we implemented a version of Dijkstra's algorithm that gave us the shortest path to reach every node in the network. After calculating the shortest paths of all nodes, we then created our goal product which is the routing table. The information that this routing table holds is a two-dimensional array of the next node that should receive the message.

Part 4: Timers

Ultimately our link state protocol would not be very efficient if we did not implement timers to prevent the immense amount of information from running continuously. In result we decided to use three timers, the first one was for neighbor discovery, then a second timer for making and sending link state packages to nodes, the third timer was used to recalculate Dijkstra's and update the routing table. Each timer had a different time to run, the shortest timer is neighbor discovery, then a little longer timer would be for link state and longer for Dijkstra's and updating routing table. We mainly decided to assign time in this order to allow neighbor discovery and link state packages propagate through the network first before calculating our routing table. This is very beneficial for updating the routing table since it would prevent useless recalculations of Dijkstra's with old information.

Discussion Questions

1. Why do we use a link for the shortest path computation only if it exists in our database in both directions? What would happen if we used a directed link AB when the link BA does not exist?

   Dijkstra's Algorithm would still compute a possible route since the algorithm only requires the adjacency list of all nodes. Since there is still a possible link to these two nodes it would create a possible route, but it might be different than having a bidirectional link. Thus, bidirectional links give more possible routes for Dijkstra's algorithm.

2. Does your routing algorithm produce symmetric routes (that follow the same path from X to Y in reverse when going from Y to X)? Why or why not?

   Yes, Dijkstra's is symmetric if there are bidirectional links which means that all of their paths are connected both directions. This is not the case for algorithms such as depth first search which only considers the first shortest path first which might not be optimal to the whole graph.

3. What would happen if a node advertised itself as having neighbors, but never forwarded packets? How might you modify your implementation to deal with this case?

   This node would be treated as a dead end which means that other neighbor nodes will have a possible connection with this node, but this node would have an empty routing table. To implement this in our project we would start all nodes with empty routing table. Therefore, if the routing table is not filled in it means that node is a dead end.

4. What happens if link state packets are lost or corrupted?

   For our implementation we first checked if the packet was the correct size which in turn tells us that this packet is clean or corrupted. If the packet is corrupted, we would just drop it since we know we can't trust the information that it is carrying. If they are lost there is no way to know if it has been lost but that does not mean that the other nodes might share the same info that would complete the same graph. Another way to solve this would be to require reply messages from all nodes it would store this info and check it with its final node table and check if a connection is lost then it should be forwarded again. The simplest way to solve this problem would be to wait till the next cycle to receive the lost link state packets and update table.

5. What would happen if a node alternated between advertising and withdrawing a neighbor, every few milliseconds? How might you modify your implementation to deal with this case?

   If this keeps happening with a specific node, I think that the best approach to solve this would be to create a blacklist. The blacklist node would be added if a few withdraw attempts occur.