Ekin Akyürek
Mehmet Kaan Bozkır
COMP416 Project3: Distance Vector Routing Simulator

PROJECT #3 Report

Project Aims and Problem Description:

The aim of the last project of this course was to create and simulate a network routing, using the Distance Vector Routing Algorithm. But in doing that, we were not allowed to use a centralized approach, which would make our jobs relatively easy, since every node would be able to access the information of their neighbors instantly. We had to choose a decentralized approach, in which nodes would inform their neighbors with a routing packet that included the ID's of source, destination and the values of the current distance vector of the source node. The packet sending would continue until there is no change in the costs of the paths, which suggests that the costs are already minimized.

Overview of the Classes we used in the Simulator:

Despite the fact that we only edited the methods and functions inside of the Entity0,1,2,3 and 4, we believe that all classes have to be explained to understand the problem fully.

Entity Class:

This is an abstract class which overlooks the subclasses **Enitity**0,1,2,3 and 4; for the five nodes of the sample network given to us. It lays out the quantities and the functions used in the subclasses. It includes the definitions of **distanceTable[i][j],** which is a 5*5 matrix showing the distances from node **i** to **j**, **update(***Packet p***)** which is the update function that lies at the heart of the algorithm, **linkCostChangeHandler(***int Link, int newCost***)**, which is a function that changes the cost of a given link, and lastly **printDT(**), which prints out the Distance Table.

Entity<i> Class:

Initialization of the Entitiy <i> Classes:

We first initialize the distance table with infinity distances, to avoid initialization by 0. Later on, we get the distanceVectorof this node and the Distance Table from the node-to-node cost matrix that is given at the Network Simulator class. (**NetworkSimulator.cost[][]** table)

Distance Vector Algorithm runs in this loop:
For all nodes in the topology,
If a node (other than ours) has a "finite" distance to our node,
--in other words if a node is a direct neighbor to our node--
We send a routing packet to inform that node, with our routing information.

Source of packet is this node, destination is the neighbor j, routing information is our distanceVector. Packet is sent to other nodes via method **.toLayer2(Packet p)**

Ekin Akyürek
Mehmet Kaan Bozkır
COMP416 Project3: Distance Vector Routing Simulator

Update (Packet p)

Update function has no output, it is a void function, but it receives routing packets as inputs. When the update function is called with a packet, we update the destination node with the routing information in the given packet. The packet methods used in this function are explained in the explanation of **Packet.java**.

We initialize the new costs as **INF**, get the origin of the packet and set the "**changeHandler**" FLAG to false, again to prevent wrong initialization. If the "**changeHandler**" FLAG becomes true, we will know an update has been used and new packets are sent to the neighbors of the updated node to continue the iteration.

This is the crucial loop of the update function. i represents the **newCost** variable is initially set to be equal to the combination of the cost of getting from the node to source, plus getting from source to node **i**.

Later on, the new value for the distance table is the minimum of either the **newcost** or the already given value at the **distanceTable**.

The new distance vector is saved as the **nextDistanceVector**, for updating purposes.

If any one of the values of the distance vector is updated, the FLAG becomes true, which means a packet has to be sent to direct neighbors of this node. The **distanceVector** is updated with the **nextDistanceVector.**

LinkCostChangeHandler(int whichlink, int newCost)

This part was not long at all, however, it was toiling nevertheless. We tried several different approaches, but none of them quite handled all the cases as we wanted. Finally, we figured out an approach that included bending around the given code by our TA's. Since the changeHandler can not be changed after the link has been altered, the packets weren't sent. We added the line: **if(changeHandler || (linkChange && !IsMyLinkChanged))** so that if there was a link change, the packets would be sent again, and the update function would be invoked. The second part, **!IsMyLinkChanged** is there to prevent looping between the nodes around the changed link.
The rest of the code is explained below:

```
//This is the loop where we find the "direct" neighbors of our node and update our "direct" neighbors.
	for(int i=0; i< NetworkSimulator.NUMENTITIES; i++){
		if( i != thisNode && NetworkSimulator.cost[thisNode][i] != 999 ){
 //Packet source is this node, destination is i, and routing information is the distanceVector.
			Packet packet = new Packet(thisNode,i, distanceVector, true);
			//Packet sent via the .tolayer2(p) method.
			NetworkSimulator.toLayer2(packet);
```

If you examine closely, we construct the Packet as `Packet(thisNode,i, distanceVector, true);` in which the "true" parameter sets the "linkchange" attribute as true. This flags the packet so that we know this packet was constructed due to a link change, so that this special kind of packet will still be able to call the update function. Without this flag, we could not get the system to fully update all nodes correctly, so we figured out this approach.

Ekin Akyürek
Mehmet Kaan Bozkır
COMP416 Project3: Distance Vector Routing Simulator

Event, EventList and EventListImpl Classes

These classes are used in order to activate certain events in a given system time, for instance they are used in activating link cost change events at time **t=10000** and **t=20000**. The **eventList** class even implements some kind of an array list, but we could not master the ingredients of these classes, since it was not a necessity.

Packet.java

This class is used to create a unique packet object, which consists of 2 integer values, the source and the destination node numbers, and an integer array of length 5 that represents the distance vector that is being transferred. It also includes "get" methods to access the source, destination and vector parts of the packet individually.

To correctly implement our new solution, we needed to alter the given Packet class slightly. We added another attribute to the packet object, which is **boolean linkchange**, which indicates if the packet is created due to a linkchange or not.

NetworkSimulator.java Class

Network Simulator class acts as the constructor of the whole project. The 5 node topology is defined and constructed in this class. The **cost[][]** array stores the initial state of the 5 node network. If there is a link between two nodes, the cost of it is stored in the array, if there is no such neighboring connection, then 999 is stored in the array representing an "infinity cost".

The **runSimulator()** function initiates the entities and starts the autonomous data transfer between the nodes. If there is a specific event coming up, such as a link cost change, the function also detects it, starts the event, then deletes the event from the event list.

It also handles the data transfer between nodes via the function toLayer2(Packet p). When this function is called by the entities, the information of the packet is extracted, and the routing information (i.e. distance vector) is sent to the destination node.

We also implemented a function called **displayResults()**, which prints out the total distance table in a easily readable way. We only call this print function after the iterations has been completed. It has no inputs.

Exceptions:

```
try{
            fileOutput = new FileOutputStream("Logs.txt");
            OutPut = new PrintStream(fileOutput);
            System.setOut(OutPut);
      }catch(Exception e){
            System.out.println(e.getMessage());
      }
```

Ekin Akyürek
Mehmet Kaan Bozkır
COMP416 Project3: Distance Vector Routing Simulator

We wrote the logs on a file called Logs.txt, and in doing that, we took care of the file exception with a try-catch mechanism.