CS536 Lab4: Distance Vector Routing Algorithm

Due Date: Sunday, Nov 15 2020 (23:59:59 PM), Total points: 140 points

1 Goal

In this lab, we are going to develop Distance Vector (DV) routing algorithm. You will be writing a "distributed' set of procedures that implement DV routing and you will test your code with the topology shown in Figure 1. (Note that your code should work with any topology, theoretically)

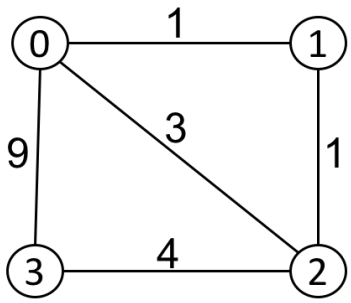

Figure 1: Topology and link costs used in Lab 4 (Part A).

2 Instructions

1. Read Chapter 5.2.2 and the lecture slides carefully. The textbook will help you to understand how DV routing works. We have also went through two examples in class which should also help you to understand when the update should be triggered and how should DVs be updated. If you have questions about the algorithm, please no hesitate to seek for help from your instructor (me) and TA.

2. Please start it early. It is hard for a genius to finish a project in last minutes. No extension will be provided for this lab.

3. You must work individually on this assignment. You will write C code that compiles and operates correctly on the XINU machines (xinu1.cs.purdue.edu, xinu2.cs.purdue.edu, etc.).

2.1 Part A: 100 points

For the basic part of the assignment, you are to write the following routines which will "execute asynchronously" within the emulated environment that has been provided for this assignment.

For each node, you will write two basic routines: **rtinit*()** and **rtupdate*()** (*=0,1,2,3).

- **rtinit*()** This routine will be called only ONCE at the beginning of the emulation. rtinit*() has no arguments. For example, at node 0, rtinit0 should initialize the distance table in node 0 to reflect the direct costs of 1, 3, and 9 to nodes 1, 2 and 3, respectively. In Figure 1, all links are bi-directional and the costs in both directions are identical. After initializing the distance table, and any other data structures needed by node 0, it should then send its directly-connected neighbors (in this case, 1,2,3) its distance vector (i.,e, the cost of it minimum cost paths to all other network nodes at the start). This minimum cost information is sent to neighboring

nodes in a routing packet by calling the routine **send2neighbor()**, as described below. The format of the routing packet is also described below.

- **rtupdate*()** is the 'heart' of the DV algorithm. The values it receives in a routing packet from some other node i contain i's current shortest path costs to all other network nodes. rtupdate*() uses these received values to update its own distance table (as specified by the distance vector algorithm). If its own minimum cost to another node changes as a result of the update, the node informs its directly connected neighbors of this change in minimum cost by sending them a routing packet. Recall that in the distance vector algorithm, only directly connected nodes will exchange routing packets. Thus nodes 0 and 1 will communicate with each other, but nodes 1 and 3 will not communicate with each other.

Let us use c(i,j) to represent the cost of a link from node i to node j. You will find it convenient to declare the cost table as a 1-by-4 array of ints, where entry [i,j] in the distance table is c(i,j). If node i is not directly connected to j, you can ignore this entry. We will use the convention that the integer value 999 or above is *infinity*. For each node, it should have its own cost array.

Let us use d(i,j) to represent the distance vector from node i to node j, namely, the "distance of the minimum-cost path" at the current iteration. You will find it convenient to declare the cost table as a 4-by-4 array of int?s, where entry [i,j] in the distance table is d(i,j). For each node k, the kth row should be updated if triggered; any other rows should be the distance vectors received from its direct neighbors. If node i is not directly connected to node k, you can ignore this row vector. We will use the convention that the integer value 999 or above is *infinity*. For each node, it should have its distance table.

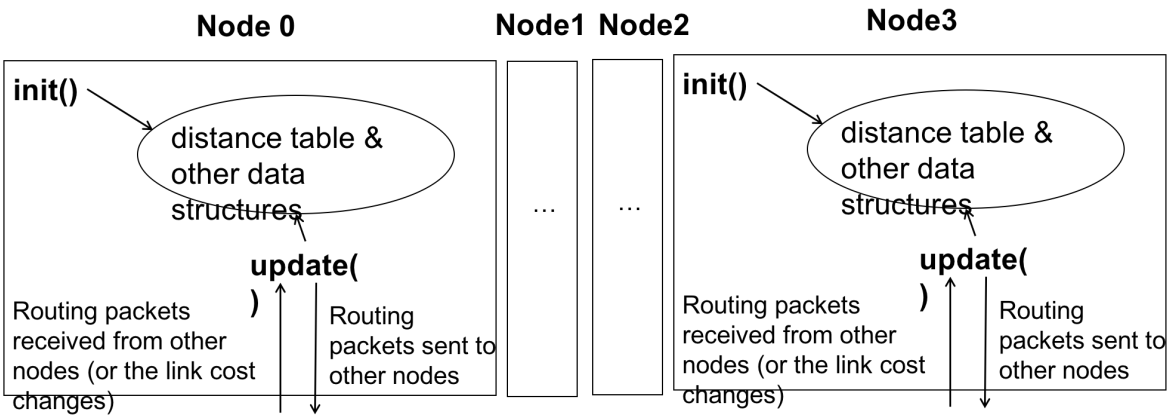

Figure 2: Relationship between procedures within one nodes and across all nodes.

Figure 2.1 provides a conceptual view of the relationship of the procedures inside each node and across all the nodes. As each node needs to implement two core routines, for convenience, you can write 8 procedures in all: rtinit0(), rtinit1(), rtinit2(), rtinit3(), rtupdate0(), rtupdate1(), rtupdate2(), rtupdate3().

In addition, you are going to implement two common routines to support the above functions: **send2neighbor()** and **printdt*()**. The routine of send2neighbor(rcvdpkt) will accept an argument of the packet to receive. The above update(rcvdpkt) will take an argument to receive the packet. In your implementation, you can use rcvdpkt as a pointer to the packet or the packet itself, depending on your choice. In the packet, you need to convey three basic information: (1) sourceid, (2) destid (destination node id) and (3) mincost[4] (min cost to node 0, 1, ..., 3).

printdt*() will print the distance table for the i^{th} node where $* = i$.

You need to run all the routines in the simulated network environment. Your procedures rtinit0(), rtinit1(), rtinit2(), rtinit3(), rtupdate0(), rtupdate1(), rtupdate2(), rtupdate3() send routing packets (whose format is described above) into the medium. The medium will deliver packets in-order, and without loss to the specified destination. Only **directly-connected** nodes can communicate. The delay between is sender and receiver is variable (and unknown). When you compile your procedures and all other procedures together and run the resulting program, you will be asked to specify only one value regarding the simulated network environment:

- Logging. Setting a logging value of 1 or 2 will print out useful information about what is going on inside the emulation (e.g., what's happening to packets and timers). A logging value of 0 will turn this off. A logging

value greater than 2 will display all sorts of odd messages that are for emulator- debugging purposes. A logging value of 2 may be helpful to you in debugging your code.

You should put your procedures for nodes 0 through 3 in files called node0.c, . . . , node3.c. You are NOT allowed to declare any global variables that are visible outside of a given C file (e.g., any global variables you define in node0.c. may only be accessed inside node0.c). This is to force you to abide by the coding conventions that you would have to adopt is you were really running the procedures in four distinct nodes.

You have a main routine which will perform DV algorithms at all the nodes in a distributed, asynchronous way to compute the distance tables for the topology and costs shown in Figure 1. It will first initiate rtinit*() (*=0,1,2,3) for all the nodes and trigger send2neighbor() from them to their neighbors. At each new round, each nodes will check if it receives packets from neighbors and run rtupdate*() according to DV algorithm.

For your sample output, your procedures should print out a message whenever your rtinit*() or rtupdate*() procedures are called, giving the time (available via my global variable clocktime). For rtupdate*(), you should print the identity of the sender of the routing packet that is being passed to your routine, whether or not the distance table is updated, the contents of the distance table (you should use the printdt*() routines defined above to print 4-by-4 array), and a description of any messages sent to neighboring nodes as a result of any distance table updates. A sample output for rtinit*() is as follow (the numbers are imaginary),

```
Initializing Node 1
                via
   D0 |    1     2    3
  ----|-----------------
     1|  32725     3    31
dest 2|    0   1541358144   32725
     3|  32725     0     0

Sending update to neighbor: 0
Distance vector: 0 0 0 0
Sending update to neighbor: 2
Distance vector: 0 0 0 0
```

A sample output for rtupdate*() is as follow (the numbers are imaginary),

```
Table Updated: True
                via
   D0 |    1     2    3
  ----|-----------------
     1|  32725     3    31
dest 2|    0   1541358144   32725
     3|  32725     0     0

Sending update to neighbors: 0 2
Distance vector: 0 0 0 0
```

The sample output should be an output listing with a logging value of 2. Highlight the final distance table produced in each node. Your program will run until there are no more routing packets in-transit in the network, at which point the emulator will terminate.

2.2 Part B: 40 points

You are going to implement the dynamic link cost in the above topology. Here, we consider the cost of the link between 0 and 1 changes. You are to write two procedures to update the link cost for node 0 and node 1. These routines should be defined in the files node0.c and node1.c, respectively. One recommended interface is linkchange0(int linkid, int newcost), and linkchange1(int linkid, int newcost) where linkid denotes the neighboring node id, and newcost is the updated cost. Once they are called, it should also trigger update() to compute the new

distance vector (min-cost to other nodes) in the distance table. If the distance vector is updated, it should send updated routing packets to neighboring nodes.

In Part B, you need to change the cost of the link from 1 to 20 at round 1000 and then change it back to 1 at round 2000. In the sample output, you don't need to print the distance table at each round. You can set the logging value as 1 to print the key update information. Namely, when the distance table is updated at this iteration, you need to display the iteration information, and the ids of nodes performing the updates. At the end, you need to display the distance table when the algorithm converges.

3 Materials to turn in

You submission should zip all the source files, the log file (sample out), a makefile compile the codes, named as "lab4_UID*.zip" You will submit your assignment on blackboard.

Note:

- You do not have to write a lot of details about the code, but just adequately comment your source code, especially when any part of your assignments are incomplete.
- Questions about the assignment should be posted on Piazza and PSO. Note that TA may cover some common issues in PSO.