Distance Vector Routing Algorithms and Simulation

(50 points)

You will be writing a "distributed" set of procedures that implement a distributed asynchronous distance vector routing protocol. A network emulator is provided. Your task is to implement methods that will be called by the emulator to complete the implementation of a distance vector routing protocol for the network shown in Figure 1.

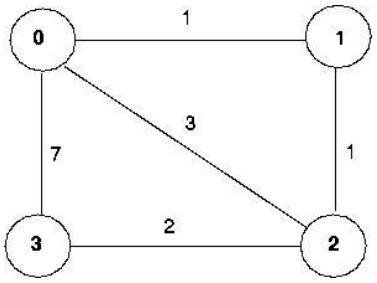

Figure 1: Network topology and link costs for DV routing lab

You may implement this assignment in C or Java. The emulator is provided in both programming languages. The instructions below focus on the Java-based implementation.

Getting Started:

The Java source code that you'll need to get started with this project is provided by the textbook author. Specifically, you'll need to download the following Java class files:

1. Entity.java
2. Entity0.java
3. Entity1.java
4. Entity2.java
5. Entity3.java
6. NetworkSimulator.java
7. Event.java
8. Packet.java
9. EventList.java
10. EventListImpl.java
11. Project.java

What You Need to Implement:

All of the Java source files are provided to you. You are asked to complete methods within those files to implement the *Distance Vector routing protocol* for the network shown in Figure 1. Specifically, you will complete the implementation of the default constructor in `Entity0.java`, `Entity1.java`, `Entity2.java`, and `Entity3.java`.

In Entity0.java -

- `Entity0()`: This constructor will be called once at the beginning of the emulation. The constructor has no arguments. It should initialize the distance table in node 0 to reflect the direct costs of 1, 3, and 7 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 your node 0 methods, the constructor should then send its directly-connected (i.e., one-hop) neighbors (in this case, 1, 2 and 3) the cost of its minimum cost paths to all other network nodes. This minimum cost information is sent to neighboring nodes in a routing packet by calling the `NetworkSimulator` method `toLayer2()`, as described below. The format of the routing packet is also described below.

- `update(Packet p)`. This method will be called when node 0 receives a routing packet that was sent to it by one of its directly connected neighbors. The parameter p is the Packet object that was received. Note that the `update(Packet p)` method is the "heart" of the distance vector algorithm. The values it receives in a routing packet from some other *node i* contain current shortest path costs to all other network nodes. The update method 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, node 0 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 1 and 2 will communicate with each other, but nodes 1 and 3 will communicate with each other.
- The distance table inside each node is the principal data structure used by the distance vector algorithm. You will find it convenient to declare the distance table as a `4-by-4 array of ints`, where `entry [i,j]` in the distance table in node 0 is node 0's currently computed cost to *node i* via direct *neighbor j*. If 0 is not directly connected to j, you can ignore this entry. We will use the convention that the integer value 999 is "infinity".

Similar methods must be defined for nodes 1, 2 and 3 in `Entity1.java`, `Entity2.java`, and `Entity3.java`. Thus, you will write 8 procedures in all: the constructor for each of the four nodes, and the update method for each of the four nodes.

The methods described above are the ones that you will write. The following methods have already been implemented for you in the provided source code and should be called by your objects.

- The `NetworkSimulator` provides the following method that is used emulate sending packets across the network `public static void toLayer2(Packet p)`
- Each of the node classes (`Entity0.java`, `Entity1.java`, `Entity2.java`, `Entity3.java`) includes a method that will pretty print the distance table for that node: `public void printDT()`

What the Emulator Does:

The methods that you implement will send routing packets into the medium. The medium will deliver packets in-order, and without loss to the specified destination. Only directly -connected nodes (i.e., one-hop neighbors) can communicate. The delay between sender and receiver is variable (and unknown).

Your program will run until there is no more routing packets in-transit in the network, at which point the emulator will terminate.

Compiling and Running Your Code:

When you compile your code and run it, you will be asked to a value regarding the simulated network environment: the tracing parameter. Setting a tracing value of 1 or 2 will print out useful information about what is going on inside the emulation (e.g., what is happening to packets and timers). A tracing value of 2 may be helpful to you in debugging your code, as it provides more information. A tracing value

of 0 will turn off tracing. **Do not use** a tracing value greater than 2; these messages relate to debugging the emulator itself and are not meant for your use.

What to Submit:

A `zip` file containing:

- A folder named `src` that includes your implementations of the four node classes: Entity0.java, Entity1.java, Entity2.java, and Entity3.java.
- A project report with sample output:
    - For your sample output, your implementation should print out a message whenever a node's constructor or update method is called. The printout should give the time. In the update methods, you should print a) the identity of the sender of the routing packet that is being passed to your routine, b) whether or not the distance table is updated, c) the contents of the distance table (you can use the `printDT ()` method for this), and a d) description of any messages sent to neighboring nodes as a result of any distance table updates.
    - The sample output should be obtained with a TRACE value of 2. Highlight the final distance table produced in each node.
    - The report should include a screenshot image that shows the running emulator and the result.

Warning:

Unfortunately, this very simple emulation does not check the network topology to govern sending packets. It is possible to use NetworkSimulator's `toLayer2` method and send packets between two hosts that are not connected. Be aware of this; you might find that you've made such a mistake when debugging.

Version of this assignment in C:

An implementation of the emulator is provided in C. You will need to download the following program files:

- ✓ prog3.c (the emulator): http://gaia.cs.umass.edu/kurose/network/prog3.c
- ✓ node0.c: http://gaia.cs.umass.edu/kurose/network/node0.c
- ✓ node1.c: http://gaia.cs.umass.edu/kurose/network/node1.c
- ✓ node2.c: http://gaia.cs.umass.edu/kurose/network/node2.c
- ✓ node3.c: http://gaia.cs.umass.edu/kurose/network/node3.c

You are to write the procedures `rtinit0(), rtinit1(), rtinit2(), rtinit3()` (which are analogous to the constructors in the Java version of this assignment) and `rtupdate0(), rtupdate1(), rtupdate2(), rtupdate3()` (which are analogous to the update methods in the Java version of the assignment).

Note that 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. To compile your routines from the command line: `cc -o dvsim prog3.c node0.c node1.c node2.c node3`, which will generate the executable program called `dvSim`.

This assignment can be completed on any machine supporting C. It makes no use of UNIX features.

Notes:

1. You can work alone or in a team of TWO. Only one submission per team. Team members name should clearly be stated.
2. Submit error free code so that TA can run. Make sure you include instructions so that TA can run easily.
3. Contact TA if you have any question.