<u>The Basic Assignment</u>

The routines you will write. You are to write the following routines that will ``execute'' asynchronously within the emulated environment that you are given for this assignment. We first describe the routines for link state routing that you will write, along with the routines and simulated environment provided to you, then we describe differences for implementing distance vector.

<u>A. Implementing Link State Routing</u>

For the node, you will write the routines:

- **rtinit(int nodename, int [] initial_lkcost)**. This routine will be called once at the beginning of the emulation. It should initialize ***nodename*** and the node's distance to its immediate neighbors ***lkcost****.* You will also need to update the adjacency metric ***graph*** and forwarding table ***costs***. These data structures are explained in detail later. In Figure 1, all links are bi-directional and the costs in both directions are identical. After initializing the data structures needed by your node, the node should then send its link state (cost) information to all the nodes in the network using only its directly connected neighbors. This link-cost information is sent to neighboring nodes in a routing (link state) ***packet*** by calling the routine **NetworkSimulator.tolayer2(*packet*)**, as described below. The format of the routing packet is also described below.

- **rtupdate(Packet rcvdpkt)**. This routine will be called when a node receives a routing (link state) packet that was sent to it by one of its directly connected neighbors. rtupdate() is the ``heart'' of the link state algorithm. The values it receives in a routing packet from some other node *i* contain *i*'s current link costs to its neighbors. rtupdate() uses these received values to update its adjacency metric ***graph***. It then runs Dijkstra's ***shortest path algorithm*** on the *graph* to update its forwarding table ***costs***. Since the received packet is a broadcast (link state) packet, the node forwards the received link-state packet to other neighboring nodes in the network. Note that you need to make sure that flooding stops when all the nodes in the network receive the link-state packet. If you are unsure how to do this, refer to the lecture slides for Chapter 5.

  The forwarding table ***costs*** and adjacency metric ***graph*** are the principal data structures for the link state algorithm. You will find it convenient to declare the ***costs*** as a 4-by-2 array of int's, where entry *costs*[*i*,0] is the minimum cost to reach destination *i*, and entry *costs*[*i*,1] is the **next hop towards** the destination node. We will use the convention that the integer value 9999 is ``infinity.'' If the node doesn't know its cost to node *i*, the entry *costs*[i,0] has integer value 9999 or ``infinity'' and *costs*[i,1] is set to be -1. The adjacency metric ***graph*** should be a 4-by-4 array of int's, where each entry *graph*[*i*,*j*] is the cost of the link from node *i* to its neighboring node *j*. If nodes *i* and *j* are not neighbors, the cost is set to be ``infinity'' or 9999. Note that you will run the shortest path algorithm on the adjacency metric *graph* to generate forwarding table *costs*.

You will be implementing a shortest path algorithm to find the minimum cost path from a source node to all other nodes (potential destinations) in the network. You should implement Dijkstra's algorithm as the shortest path algorithm. You can either implement it in Node.java, or you can use the ShortestPath class for implementing the shortest path algorithm. Your shortest path routine should implement the following:
int[][] dijkstra(int graph[][], int src). The routine takes the adjacency metric *graph* and source node *src* as input, and returns a 2D array *output*. *output* is a 4-by-2 array, where destination *i* can be reached from source node *src* with a minimum cost of *output*[*i*][0] via node *output*[*i*][1]. Note that *output*[*i*][1] is the node immediately before the destination node *i* on the shortest path from *src* to *i*. Note that *output* is different from forwarding table, and you will need information in *output* to update the forwarding table.
Figure 2 provides a conceptual view of the relationship of the procedures inside the node.

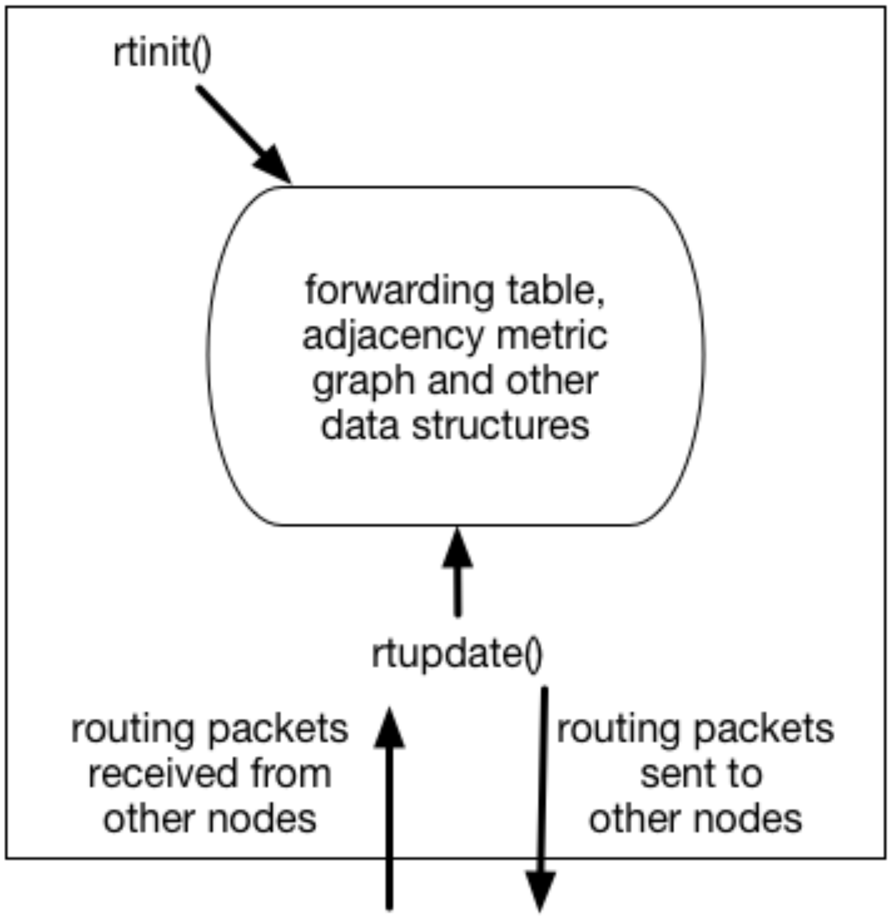
rtinit()
forwarding table, adjacency metric graph and other data structures
rtupdate()
routing packets received from other nodes
routing packets sent to other nodes

Figure 2.

Software Interfaces

The procedures described above are the ones that you will write. You are given the following routines which can be called by your routines:

- **In NetworkSimulator.java:**

public static void tolayer2(Packet packet)

where *packet* is an object of the java class *Packet* with the following constructor:

> **public** Packet(**int** sourceid, **int** destid, **int** nodename, **int**[] mincost, **int** seqNo)

```
int sourceid;      /* id of router sending this pkt, 0, 1, 2, or 3  */
int destid;        /* id of router to which pkt is being sent
int nodename   /*  name of the node who sent this packet */
int[] mincost      /* link costs to neighbor nodes 0 … 3 */
```

```
int seqNo        /*  can be used to distinguish new link state info */
```

- **In Node.java:**

void printdt(): will pretty print the forwarding table for each node. This is the standard output that will be used to grade the assignment.

The Simulated Network Environment
Your procedures rtinit() and rtupdate() 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 a sender and receiver is variable (and unknown).
When you compile your procedures and the procedures given to you together and run the resulting program, you will be asked to specify only two values regarding the simulated network environment:

- **Tracing.** Setting a tracing 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 tracing value of 0 will turn this off.

  A tracing value of 2 may be helpful to you in debugging your code. You should keep in mind that *real* implementers do not have underlying networks that provide such nice information about what is going to happen to their packets!

- **Seed.** Seed of the pseudo-random generator used by the simulator.

B. Implementing Distance Vector Routing
For distance vector, the routines differ mainly in the following:

- **rtinit()**. This routine should initialize the distance table in the node to reflect the direct costs to its neighboring nodes. After initializing the distance table, and any other data structures needed by your node routines, it should then send its directly-connected neighbors the costs of its shortest (minimum-cost) paths to all other network nodes. This minimum-cost information is sent to neighboring nodes in a *routing (distance vector) packet* by calling the routine tolayer2(), as described above. The format of the routing (distance vector) packet is described below.

rtupdate(). This routine will be called when a node receives a routing (distance vector) packet that was sent to it by one of its directly connected neighbors. rtupdate() is the ``heart'' of the distance vector 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 (distance vector) packet.
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 int's ***entry***, where entry [i,j] in the distance table specifies the node's currently computed cost to node *i* via direct neighbor *j*. Note that the routing table at each node (which specifies for each destination the neighbor node via which this node can reach the destination with minimum cost) can be directly derived from the distance table of this node!

In the routing packet for distance vector, **int[] mincost** represents the min cost from the source to each destination node 0 …3.

The Assignment (70 points)

Your procedures will implement a distributed, asynchronous computation of the forwarding tables for the topology shown in Figure 1. You should implement both the ***link state*** algorithm and the ***distance vector*** algorithm.
Do not implement any solution to the count-to-infinity problem in the distance vector algorithm; you should ignore that issue.

Assignment Continued (30 points)

After you are done with the basic assignment, you are to extend your code by writing procedure **linkhandler(int linkid, int newcost)**, which will be called if (and

when) the cost of the link between node 0 and node 1 changes. This routine should be defined in the file *Node.java*. The routine will be passed the name (id) of the neighboring node on the other side of the link whose cost has changed (*linkid*), and the new cost of the link (*newcost*). Note that when a link cost changes, this routine will have to update the neighbor's cost and may (or may not) have to send updated routing packets to neighboring nodes.
In order to complete this part of the assignment, you will need to change the value of the constant LINKCHANGES (line 12 in NetworkSimulator.java) to 1. For your information, the cost of the link will change from 1 to 20 at time 10000 and then change back to 1 at time 20000. Your routine will be invoked at these times.

What to Submit

You should use gsubmit to submit your code under a hw8 folder, and upload your testing document to Gradescope.

Submit a sample output in the testing document, in which your procedures should print out a message whenever your rtinit() or rtupdate() procedures are called. For rtupdate(), you should print the identity of the sender of the routing packet that is being passed to your routine, the contents of the forwarding table (you can use the given pretty-print routines), and a description of any messages sent to neighboring nodes.

The sample output should be an output listing with a TRACE value of 2. Highlight the final forwarding 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.

To compare link state routing and distance vector routing, report on the number of routing messages sent (update this statistic whenever you send routing packets to layer2) and the convergence time (i.e. when the program terminates) for at least five runs (five different random numbers) for the topology shown in Figure 1.
In your documentation, indicate your other team member, if any, along with the division of labor. Only one member needs to gsubmit.